

Contents

Documentação do C#

Introdução

Introdução

Tipos

Blocos de construção de programas

Principais áreas de linguagem

Tutoriais

Escolha sua primeira lição

Tutoriais baseados em navegador

Hello world

Números em C#

Loops e branches

Coleções de lista

Trabalhar em seu ambiente local

Configurar seu ambiente

Números em C#

Loops e branches

Coleções de lista

Conceitos básicos

Estrutura do programa

Visão geral

Método Principal

Instruções de nível superior

Sistema de tipos

Visão geral

Namespaces

Classes

Registros

Interfaces

Genéricos

Tipos anônimos

Programação orientada a objeto

Visão geral

Objetos

Herança

Polimorfismo

Técnicas funcionais

Correspondência de padrões

Descartes

Desconstruindo tuplas e outros tipos

Exceções e erros

Visão geral

Usando exceções

Tratamento de exceções

Criando e gerando exceções

Exceções geradas pelo compilador

Estilo de codificação

Nomes de identificadores

Convenções de codificação em C#

Tutoriais

Como exibir argumentos de linha de comando

Introdução às classes

Object-Oriented C #

Herança em C# e .NET

Convertendo tipos

Criar algoritmos controlados por dados com correspondência de padrões

Como manipular uma exceção usando try/catch

Como executar código de limpeza usando finally

Novidades no C#

C# 10

C# 9.0

C# 8.0

[Alterações significativas do compilador](#)

[Histórico de versão do C#](#)

[Relações com a biblioteca do .NET](#)

[Compatibilidade de versões](#)

Tutoriais

[Explorar tipos de registro](#)

[Explore as instruções de nível superior](#)

[Explorar padrões em objetos](#)

[Atualizar interfaces de forma segura com membros de interface padrão](#)

[Criar funcionalidade mixin com métodos de interface padrão](#)

[Explore os índices e os intervalos](#)

[Trabalhar com tipos de referência nula](#)

[Gerar e consumir fluxos assíncronos](#)

[Gravar um manipulador de interpolação de cadeia de caracteres personalizada](#)

Tutoriais

[Explorar a interpolação de cadeia de caracteres – interativo](#)

[Explorar a interpolação de cadeia de caracteres – em seu ambiente](#)

[Cenários avançados de interpolação de cadeia de caracteres](#)

[Aplicativo do Console](#)

[Cliente REST](#)

[Trabalhar com o LINQ](#)

[Usar atributos](#)

Conceitos de C#

[Tipos de referência anuláveis](#)

[Migrações de referência anulável](#)

[Resolver avisos anuláveis](#)

[Métodos](#)

[Propriedades](#)

[Indexadores](#)

[Iterators](#)

[Delegados e eventos](#)

Introdução a Delegados
System.Delegate e a palavra-chave delegado
Delegados Fortemente Tipados
Padrões Comuns para Delegados
Introdução a eventos
Padrões de evento .NET padrão
Padrão de Evento .NET Atualizado
Distinção entre Delegados e Eventos
LINQ (Consulta Integrada à Linguagem)
Visão geral de LINQ
Noções básicas sobre expressões de consulta
LINQ em C#
Escrever consultas LINQ em C#
Consultar uma coleção de objetos
Retornar uma consulta de um método
Armazenar os resultados de uma consulta na memória
Agrupar resultados de consultas
Criar um grupo aninhado
Executar uma subconsulta em uma operação de agrupamento
Agrupar resultados por chaves contíguas
Especificiar filtros predicados dinamicamente em runtime
Executar junções internas
Executar junções agrupadas
Executar junções externas esquerdas
Ordenar os resultados de uma cláusula join
Unir usando chaves compostas
Executar operações de junção personalizadas
Manipular valores nulos em expressões de consulta
Tratar exceções em expressões de consulta
Escrever código eficiente e seguro
Árvores de expressão
Introdução à Árvores de expressão

- [Árvores de Expressão Explicadas](#)
- [Tipos de Framework com suporte a árvores de expressão](#)
- [Executando Expressões](#)
- [Interpretando Expressões](#)
- [Compilando Expressões](#)
- [Traduzindo Expressões](#)
- [Resumo](#)
- [Interoperabilidade nativa](#)
- [Controle de versão](#)
- [Artigos de instruções sobre C#](#)
 - [Índice do artigo](#)
 - [Dividir cadeias de caracteres em substrings](#)
 - [Concatenar cadeias de caracteres](#)
 - [Pesquisar cadeias de caracteres](#)
 - [Modificar o conteúdo de uma cadeia de caracteres](#)
 - [Comparar cadeias de caracteres](#)
 - [Como capturar uma exceção não compatível com CLS](#)
- [O SDK do .NET Compiler Platform \(APIs do Roslyn\)](#)
 - [Visão geral do SDK do .NET Compiler Platform \(APIs do Roslyn\)](#)
 - [Entender o modelo de API do compilador](#)
 - [Trabalhar com sintaxe](#)
 - [Trabalhar com semântica](#)
 - [Trabalhar com um workspace](#)
 - [Explorar código com o visualizador de sintaxe](#)
 - [Geradores de origem](#)
 - [Inícios rápidos](#)
 - [Análise sintática](#)
 - [Análise semântica](#)
 - [Transformação de sintaxe](#)
 - [Tutoriais](#)
 - [Crie seu primeiro analisador e correção de código](#)
 - [Guia de programação em C#](#)
 - [Visão geral](#)

Conceitos de programação

Visão geral

Programação assíncrona

Visão geral

Cenários de programação assíncrona

Modelo de programação assíncrona de tarefa

Tipos de retorno assíncronos

Cancelar tarefas

Cancelar uma lista de tarefas

Cancelar tarefas após um período

Processar tarefas assíncronas conforme elas são concluídas

Acesso a arquivos assíncronos

Atributos

Visão geral

Criando atributos personalizados

Acessando atributos usando reflexão

Como criar uma união do C/C++ usando atributos

Coleções

Covariância e contravariância

Visão geral

Variação em interfaces genéricas

Criar interfaces genéricas variáveis

Usar variação em interfaces para coleções genéricas

Variação em delegações

Usar variação em delegações

Usar variação para delegações genéricas Func e Action

Árvores de expressão

Visão geral

Como executar árvores de expressão

Como modificar árvores de expressão

Como usar árvores de expressão para compilar consultas dinâmicas

Depurar árvores de expressão no Visual Studio

Sintaxe do DebugView

Iterators

LINQ (Consulta Integrada à Linguagem)

Visão geral

Introdução a LINQ em C#

Introdução a consultas LINQ

LINQ e tipos genéricos

Operações de consulta LINQ básicas

Transformações de dados com LINQ

Relacionamentos de tipo em operações de consulta LINQ

Sintaxe de consulta e sintaxe de método em LINQ

funcionalidades do C# que dão suporte a LINQ

Passo a passo: Escrevendo consultas em C# (LINQ)

Visão geral de operadores de consulta padrão

Visão geral

Sintaxe de expressão da consulta para operadores de consulta padrão

Classificação de operadores de consulta padrão por maneira de execução

Classificar dados

Operações de conjunto

Filtrar dados

Operações de quantificador

Operações de projeção

Particionar dados

Operações de junção

Agrupar dados

Operações de geração

Operações de igualdade

Operações de elemento

Converter tipos de dados

Operações de concatenação

Operações de agregação

Objetos LINQ to

[Visão geral](#)

[LINQ e cadeias de caracteres](#)

[Artigos de instruções](#)

[Como contar ocorrências de uma palavra em uma cadeia de caracteres \(LINQ\)](#)

[Como consultar sentenças que contenham um conjunto especificado de palavras \(LINQ\)](#)

[Como consultar caracteres em uma cadeia de caracteres \(LINQ\)](#)

[Como combinar consultas LINQ com expressões regulares](#)

[Como localizar a diferença de conjunto entre duas listas \(LINQ\)](#)

[Como classificar ou filtrar dados de texto por qualquer palavra ou campo \(LINQ\)](#)

[Como reorganizar os campos de um arquivo delimitado \(LINQ\)](#)

[Como combinar e comparar coleções de cadeias de caracteres \(LINQ\)](#)

[Como preencher coleções de objetos por meio de várias fontes \(LINQ\)](#)

[Como dividir um arquivo em vários arquivos usando grupos \(LINQ\)](#)

[Como unir conteúdo por meio de arquivos diferentes \(LINQ\)](#)

[Como computar valores de coluna em um arquivo de texto CSV \(LINQ\)](#)

[LINQ e reflexão](#)

[Como consultar metadados de um assembly com reflexão \(LINQ\)](#)

[LINQ e diretórios de arquivos](#)

[Visão geral](#)

[Como consultar arquivos com um atributo ou nome especificado](#)

[Como agrupar arquivos por extensão \(LINQ\)](#)

[Como consultar o número total de bytes em um conjunto de pastas \(LINQ\)](#)

[Como comparar o conteúdo de duas pastas \(LINQ\)](#)

[Como consultar o maior arquivo ou os arquivos em uma árvore de diretório \(LINQ\)](#)

[Como consultar arquivos duplicados em uma árvore de diretório \(LINQ\)](#)

[Como consultar o conteúdo de arquivos em uma pasta \(LINQ\)](#)

[Como consultar um ArrayList com LINQ](#)

[Como adicionar métodos personalizados para consultas LINQ](#)

[LINQ to ADO.NET \(página do portal\)](#)

[Habilitando uma fonte de dados para consulta LINQ](#)

Suporte do Visual Studio IDE e ferramentas para LINQ

Reflexão

Serialização (C#)

Visão geral

Como gravar dados de objeto em um arquivo XML

Como ler dados de objeto provenientes de um arquivo XML

Passo a passo: persistir um objeto no Visual Studio

Instruções, expressões e operadores

Visão geral

Instruções

Membros aptos para expressão

Igualdade e comparações de igualdade

Comparações de igualdade

Como definir a igualdade de valor para um tipo

Como testar a igualdade de referência (identidade)

Tipos

Conversões cast e conversões de tipo

Conversão boxing e unboxing

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

Usando o tipo dynamic

Passo a passo: criar e usar objetos dinâmicos (C# e Visual Basic)

Classes, estruturas e registros

Polimorfismo

Controle de versão com as palavras-chave override e new

Quando usar as palavras-chave override e new

Como substituir o método ToString

Membros

Visão geral dos membros

Classes e membros de classes abstract e sealed

Classes static e membros de classes static

Modificadores de acesso

Campos

Constantes

Como definir propriedades abstract

Como definir constantes em C#

Propriedades

Visão geral das propriedades

Usando propriedades

Propriedades de interface

Restringindo a acessibilidade ao acessador

Como declarar e usar propriedades de leitura/gravação

Propriedades autoimplementadas

Como implementar uma classe leve com propriedades autoimplementadas

Métodos

Visão geral dos métodos

Funções locais

Ref returns e ref locals

Parâmetros

Passando parâmetros

Passando parâmetros de tipo de valor

Passando parâmetros de tipo de referência

Como saber a diferença entre passar um struct e passar uma referência de classe para um método

Variáveis Locais Tipadas Implicitamente

Como usar matrizes e variáveis locais de tipo implícito em uma expressão de consulta

Métodos de extensão

Como implementar e chamar um método de extensão personalizado

Como criar um novo método para uma enumeração

Argumentos nomeados e opcionais

Como usar argumentos nomeados e opcionais na programação do Office

Construtores

Visão geral dos construtores

- [Usando construtores](#)
- [Construtores de instância](#)
- [Construtores particulares](#)
- [Construtores estáticos](#)
- [Como escrever um construtor de cópia](#)
- [Finalizadores](#)
- [Inicializadores de objeto e coleção](#)
- [Como inicializar objetos usando um inicializador de objeto](#)
- [Como inicializar um dicionário com um inicializador de coleção](#)
- [Tipos aninhados](#)
- [Classes parciais e métodos](#)
- [Como retornar subconjuntos de propriedades de elementos em uma consulta](#)
- [Interfaces](#)
 - [Implementação de interface explícita](#)
 - [Como implementar membros de interface explicitamente](#)
 - [Como implementar membros de duas interfaces explicitamente](#)
- [Delegados](#)
 - [Visão geral](#)
 - [Usando delegados](#)
 - [Delegados com Métodos Nomeados vs. Métodos anônimos](#)
 - [Como combinar delegados \(delegados multicast\) \(Guia de Programação em C#\)](#)
 - [Como declarar e usar um delegado e criar uma instância dele](#)
- [Matrizes](#)
 - [Visão geral](#)
 - [Matrizes unidimensionais](#)
 - [Matrizes multidimensionais](#)
 - [Matrizes denteadas](#)
 - [Usar foreach com matrizes](#)
 - [Passar matrizes como argumentos](#)
 - [Matrizes de tipo implícito](#)
- [Cadeias de caracteres](#)
 - [Programando com cadeias de caracteres](#)

Como determinar se uma cadeia de caracteres representa um valor numérico

Indexadores

Visão geral

Usando indexadores

Indexadores em interfaces

Comparação entre propriedades e indexadores

Eventos

Visão geral

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

Genéricos

Parâmetros de tipo genérico

Restrições a parâmetros de tipo

Classes genéricas

Interfaces genéricas

Métodos genéricos

Genéricos e matrizes

Delegados genéricos

Diferenças entre modelos C++ e genéricos C#

Genéricos em tempo de execução

Genéricos e reflexão

Genéricos e atributos

Sistema de arquivos e o Registro

Visão geral

Como iterar em uma árvore de diretório

Como obter informações sobre arquivos, pastas e unidades

Como criar um arquivo ou uma pasta

Como copiar, excluir e mover arquivos e pastas

Como fornecer uma caixa de diálogo de progresso para operações de arquivo

Como escrever em um arquivo de texto

Como ler de um arquivo de texto

Como ler um arquivo de texto uma linha de cada vez

Como criar uma chave no Registro

Interoperabilidade

Interoperabilidade .NET

Visão geral sobre interoperabilidade

Como acessar objetos de interoperabilidade do Office usando recursos do C#

Como usar propriedades indexadas na programação para interoperabilidade COM

Como usar invocação de plataforma para executar um arquivo WAV

Passo a passo: Programação do Office (C# e Visual Basic)

Exemplo de classe COM

Referência de linguagem

Visão geral

Configurar versão da linguagem

Tipos

Tipos de valor

Visão geral

Tipos numéricos inteiros

tipos inteiros nativos Nint e nuint

Tipos numéricos de ponto flutuante

Conversões numéricas internas

bool

char

Tipos de enumeração

Tipos de estrutura

Tipos de tupla

Tipos de valor anuláveis

Tipos de referência

Recursos dos tipos de referência

Tipos de referência internos

registro

- classe
- interface
- Tipos de referência anuláveis
- void
- var
- Tipos internos
- Tipos não gerenciados
- Valores padrão
- Palavras-chave
- Visão geral
- Modificadores
- Modificadores de acesso
 - Referência rápida
 - Níveis de acessibilidade
 - Domínio de acessibilidade
 - Restrições ao uso de níveis de acessibilidade
 - interno
 - particulares
 - protegidos
 - públicos
 - internos protegidos
 - privado protegido
- abstract
- async
- const
- event
- extern
- in (modificador genérico)
- new (modificador de membro)
- out (modificador genérico)
- override
- readonly

`sealed`

`static`

`unsafe`

`virtual`

`volatile`

Palavras-chave de instrução

Categorias de instrução

Instruções para manipulação de exceções

`throw`

`try-catch`

`try-finally`

`try-catch-finally`

Contexto verificado e não verificado

Visão geral

`checked`

`unchecked`

Instrução `fixed`

Parâmetros de método

Passando parâmetros

`params`

`in` (modificador de parâmetro)

`ref`

`out` (modificador de parâmetro)

Palavras-chave de namespace

`namespace`

`using`

Contextos de uso

Diretiva de uso

Instrução `using`

`extern alias`

Tipo genérico de palavras-chave de restrição

nova restrição

where

Palavras-chave de acesso

base

this

Palavras-chave literais

null

true e false

default

Palavras-chave contextuais

Referência rápida

add

get

init

partial (tipo)

partial (método)

remove

set

when (condição de filtro)

value

yield

Palavras-chave de consulta

Referência rápida

Cláusula from

Cláusula where

Cláusula select

Cláusula group

into

Cláusula orderby

Cláusula join

Cláusula let

ascending

descending

on

equals

by

in

Operadores e expressões

Visão geral

Operadores aritméticos

Operadores lógicos boolianos

Operadores shift e bit a bit

Operadores de igualdade

Operadores de comparação

Operadores de acesso do membro e expressões

Operadores de teste de tipo e expressão de conversão

Operadores de conversões definidas pelo usuário

Operadores relacionados ao ponteiro

Operadores de atribuição

Expressões lambda

Padrões

+ e operadores +=

- e operadores -=

Operador ?:

! Operador do (null-forgiving)

?? e operadores ??=

Operador =>

Operador ::

Operador await

Expressões de valor padrão

operador delegate

Operador is

expressão nameof

Operador new

Operador sizeof

- expressão stackalloc
- expressão switch
- operadores true e false
- expressão with
- Sobrecarga de operador

Instruções

- Instruções de iteração
- Instruções de seleção
- Instruções de atalho
- instrução lock

Caracteres especiais

- Visão geral
- Interpolação de cadeia de caracteres -- \$
- @ – identificador textual

Atributos lidos pelo compilador

- Atributos globais
- Informações do chamador
- Análise estática que permite valor nulo
- Diversos

Código não seguro e ponteiros

Diretivas do pré-processador

Opções do compilador

- Visão geral
- Opções de idioma
- Opções de saída
- Opções de entrada
- Opções de erro e aviso
- Opções de geração de código
- Opções de segurança
- Opções de recursos
- Opções diversas
- Opções Avançadas

Comentários da documentação XML

[Gerar documentação da API](#)

[Marcações recomendadas](#)

[Exemplos](#)

Mensagens do compilador

Especificações

[Especificação de rascunho do C# 6.0](#)

[Sumário detalhado](#)

[Prefácio](#)

[Introdução](#)

[Escopo](#)

[Referências normativas](#)

[Termos e definições](#)

[Acrônimos e abreviações](#)

[Descrição geral](#)

[Conformidade](#)

[Estrutura lexical](#)

[Conceitos básicos](#)

[Tipos](#)

[Variáveis](#)

[Conversões](#)

[Expressões](#)

[Instruções](#)

[Namespaces](#)

[Classes](#)

[Structs](#)

[Matrizes](#)

[Interfaces](#)

[Enums](#)

[Delegados](#)

[Exceções](#)

[Atributos](#)

Código não seguro

Gramática

Problemas de portabilidade

Biblioteca padrão

Comentários de documentação

Bibliografia

Recursos do C# 7.0 – 10.0

Recursos do C# 7.0

Correspondência de padrões

Funções locais

Declarações de variável out

Expressões throw

Literais binários

Separadores de dígito

Tipos de tarefa assíncrona

Recursos do C# 7.1

Método assíncrono principal

Expressões padrão

Inferir nomes de tupla

Correspondência de padrões com genéricos

Recursos do C# 7.2

Referências somente leitura

Segurança de tempo de compilação para tipos semelhantes a ref

Argumentos nomeados que não estejam à direita

Protegido de forma particular

Referência condicional

Separador de dígito à esquerda

Recursos do C# 7.3

Restrições de tipo genérico não gerenciado

A indexação de campos `fixed` não deve exigir fixação independentemente do contexto móvel/imóvel

Declaração `fixed` baseada em padrão

Reatribuição de local de ref

Inicializadores de matriz stackalloc

Atributos direcionados para campos de propriedade autoimplementada

Variáveis de expressão em inicializadores

Igualdade (==) e desigualdade (!=) de tupla

Candidatos de sobrecarga aprimorados

Recursos do C# 8.0

Tipos de referência que permitem valor nulo – proposta

Correspondência de padrões recursiva

Métodos de interface padrão

Fluxos assíncronos

Intervalos

Uso baseado em padrão e declarações de uso

Funções locais estáticas

Atribuição de coalescência nula

Membros da instância ReadOnly

Stackalloc aninhado

Recursos do C# 9.0

Registros

Instruções de nível superior

Tipos de referência que permitem valor nulo – especificação

Melhorias na correspondência de padrões

Setters somente init

Expressões new com tipo de destino

Inicializadores de módulo

Extensão de métodos partial

Funções anônimas static

Expressão condicional com tipo de destino

Tipos de retorno covariantes

Extensão GetEnumerator em loops foreach

Parâmetros discard de lambda

Atributos em funções locais

Inteiros de tamanho nativo

Ponteiros de função

Suprimir a emissão do sinalizador localsinit

Anotações de parâmetro de tipo sem restrição

Recursos do C# 10

Structs de registro

Construtores de struct sem parâmetros

Diretiva de uso global

Namespaces no escopo do arquivo

Padrões de propriedade estendidos

Cadeias de caracteres interpoladas aprimoradas

Cadeias de caracteres interpoladas constantes

Melhorias lambda

Expressão de argumento do chamador

Diretivas #line aprimoradas

Atributos genéricos

Análise de atribuição definida aprimorada

Substituição de AsyncMethodBuilder

Um tour pela linguagem C#

21/01/2022 • 15 minutes to read

O C# (pronuncia-se "See Sharp") é uma linguagem de programação moderna, orientada a objeto e fortemente digitada. 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 fornece uma visão geral dos principais componentes da linguagem no C# 8 e anterior. Se você quiser explorar a linguagem por meio de exemplos interativos, experimente a introdução aos tutoriais [do C#](#).

O C# é uma linguagem de programação orientada a objeto* orientada a componentes. O C# fornece constructos de linguagem para dar suporte direto a esses conceitos, tornando o C# uma linguagem natural na qual criar e usar componentes de software. Desde sua origem, o C# adicionou recursos para dar suporte a novas cargas de trabalho e práticas de design de software emergentes. Em seu núcleo, O C# é uma linguagem * orientada a objeto. Você define tipos e seu comportamento.

Vários recursos C# ajudam a criar aplicativos robustos e duráveis. * [Coleta de lixo](#) _ recupera automaticamente a memória ocupada por objetos não utilizáveis. [Tipos que podem ser](#) anulados 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 são suportadas](#) por técnicas de programação funcional. [A sintaxe LINQ \(Consulta Integrada à Linguagem\)](#) cria um padrão comum para trabalhar com dados de qualquer fonte. O suporte de linguagem [para operações assíncronas](#) fornece sintaxe para a criação de sistemas distribuídos. O C# tem um [sistema de tipos 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 [definidos pelo usuário](#) e [tipos de valor](#). 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 de tipos. 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. Os aspectos do design do C# que foram influenciados diretamente pelas considerações de controle de versão incluem os modificadores e separados, as regras para resolução de sobrecarga de método e suporte para declarações explícitas de membro da `virtual` `override` 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 do assembly.

Quando o programa C# é executado, o assembly é carregado no CLR. O CLR executa a compilação JIT (Just-In-Time) para converter o código IL em instruções de computador nativo. O CLR fornece outros serviços relacionados à coleta automática de lixo, tratamento de exceções e gerenciamento de recursos. O código

executado pelo CLR às vezes é chamado de "código gerenciado". "Código nãomanagedo", é compilado em uma linguagem de computador nativa que tem como destino uma plataforma específica.

A 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 do C# pode interagir com o código que foi gerado nas versões do .NET de F#, Visual Basic, C++. Há mais de 20 outras linguagens em conformidade com CTS. Um único assembly pode conter vários módulos escritos em diferentes linguagens .NET. Os tipos podem fazer referência uns aos outros como se fossem escritos no mesmo idioma.

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

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#:

```
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 `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 chamado `Main` serve como o 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 todos os dados em C#. A declaração de um tipo pode incluir seus membros, tipo base, interfaces que implementa 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 referenciam o mesmo objeto e as operações em uma variável afetem o objeto referenciado pela outra variável. Com tipos de valor, cada uma das variáveis tem sua própria cópia dos dados e não é possível que as operações em um afetem o outro (exceto as variáveis de parâmetro e `ref` `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 um identificador pode ser útil ao interagir com outras linguagens.

Os tipos de valor do C# são divididos ainda mais em tipos *simples*, tipos de *enum*, tipos *de struct*, tipos de valor anulados e tipos de valor de *tupla*. Os tipos de referência do C# são divididos ainda mais em *tipos de classe*, tipos de *interface*, tipos de matriz e tipos *delegados*.

O contorno a seguir fornece uma visão geral do sistema de tipos do C#.

- **Tipos de valor**
 - **Tipos simples**
 - **Integral assinado:** `sbyte` , `short` `int` , `long`
 - **Integral sem assinatura:** `byte` , `ushort` `uint` , `ulong`
 - **Caracteres Unicode:** , que representa uma unidade de código `char` UTF-16
 - **Ponto flutuante binário IEEE:** `float` , `double`
 - **Ponto flutuante decimal de alta precisão:** `decimal`
 - **Booliana:** `bool` , que representa valores booleanas – valores que são `true` ou `false`
 - **Tipos de 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 inteiros. 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:** , que representa uma sequência de unidades de código `string` 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 de tipos do C# são definidas pelo usuário: tipos de classe, tipos de struct, tipos de interface, tipos de enum, tipos delegados e tipos de valor de tupla. Você também pode declarar `record` tipos, `record struct` ou `record class`. Os tipos de registro têm membros sintetizados pelo compilador. Você usa registros principalmente para armazenar valores, com comportamento mínimo 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, ao contrário das classes, os structs são tipos de valor e normalmente não exigem alocação de heap. Os tipos de struct não são suportados pela herança especificada pelo usuário e todos os tipos de struct herdam implicitamente do tipo `object`.
- Um `interface` tipo define um contrato como um conjunto nomeado de membros públicos. Um `class` ou que implementa um deve fornecer `struct` `interface` implementações dos 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 encontrados em algumas outras linguagens. Ao contrário dos ponteiros de função, os delegados são orientados a objetos e são de tipo seguro.

Todos os tipos `, , e` são suportados por genéricos, nos quais `class` eles podem ser `struct` `interface` `delegate` 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 que possam ser usados. Em vez disso, os tipos de matriz são construídos seguindo um nome de tipo entre colchetes. Por exemplo, é uma matriz unidimensional de `,` é uma matriz bidimensional de `e` é uma matriz unidimensional de matrizes unidimensionais, ou uma `int[] int` matriz "desfeita", `int[,] int` de `int[][] int`.

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

O sistema de tipos do C# é unificado de forma que um valor de qualquer tipo possa 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`.

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

O sistema de tipos unificado do C# significa efetivamente que os tipos de valor são `object` tratados como referências "sob demanda". Devido à unificação, as bibliotecas de uso geral que usam o tipo podem ser usadas com todos os tipos derivados de , incluindo tipos de referência `object` `object` 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* *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 C# são compilados, eles são fisicamente empacotados em assemblies. Os assemblies normalmente têm a extensão de arquivo ou , dependendo se eles implementam aplicativos ou `.exe` `.dll` *bibliotecas* *, *respectivamente*.

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

```

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 `Entry` classe contém ainda três membros: uma propriedade chamada `Next`, uma propriedade chamada `Data` e um `Entry` construtor. O `Stack` é uma *classe genérica*. Ele tem um parâmetro de `T` tipo, que é substituído por um tipo concreto quando é usado.

Uma *pilha* é uma coleção "primeiro a entrar – último a sair" (FILO). Novos elementos são adicionados à parte superior da pilha. Quando um elemento é removido, ele é removido da parte superior da pilha. O exemplo anterior declara o `Stack` tipo que define o armazenamento e o comportamento de uma pilha. Você pode declarar uma variável que se refere a uma instância do `Stack` tipo 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 JIT (Just-In-Time) do .NET Common Language Runtime converte o código IL em um assembly em código específico do processador.

Como um assembly é uma unidade de funcionalidade autodescrevente que contém o código e os metadados, não há necessidade de diretivas e arquivos de `#include` header em 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`:

```
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 esse programa, você precisaria *referenciar* o assembly que contém a classe de pilha definida no exemplo anterior.

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 apenas um tipo público, nem exige o nome do arquivo de origem para corresponder a um tipo declarado no arquivo de origem.

Artigos adicionais neste Tour explicam esses blocos organizacionais.

[PRÓXIMO](#)

Tipos e membros

21/01/2022 • 7 minutes to read

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 número 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 leve; é um tipo de pilha alocada que pode implementar interfaces, mas não dá suporte à herança. O C# fornece `record class` e `record struct` tipos que são tipos cuja finalidade é armazenar principalmente os valores de dados.

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* da classe, também conhecida 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`:

```
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 `Point` objetos e armazenam referências a esses objetos em duas variáveis:

```
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 ou é possível desalocar explicitamente objetos em C#.

Parâmetros de tipo

Classes genéricas definem [parâmetros de tipo](#). Parâmetros de tipo são uma lista de nomes de parâmetro 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`:

```
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*. Os 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:

```
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. Siga o nome da classe e os parâmetros de 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`. No primeiro exemplo, a classe base de `Point` é `object`:

```
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 de sua classe base. Uma classe não herda a instância e construtores estáticos e o finalizador. Uma classe derivada pode adicionar novos membros a esses membros que ele herda, mas não pode remover a definição de um membro herdado. No exemplo anterior, `Point3D` herda os `x` Membros e `y` de `Point`, e cada `Point3D` instância 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`:

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

Estruturas

Classes definem tipos que dão suporte a herança e polimorfismo. Eles permitem que você crie comportamentos sofisticados com base em hierarquias de classes derivadas. Por outro lado, os tipos de **struct** são tipos mais simples cujo objetivo principal é armazenar valores de dados. Structs não podem declarar um tipo base; Eles derivam implicitamente de **System.ValueType**. Você não pode derivar outros **struct** tipos de um **struct** tipo. Eles são lacrados implicitamente.

```
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 estruturas. Você define um _interface* para declarar recursos que são compartilhados entre tipos distintos. Por exemplo, a **System.Collections.Generic.IEnumerable<T>** interface define uma maneira consistente de percorrer todos os itens de uma coleção, como uma matriz. Uma interface pode conter métodos, propriedades, eventos e indexadores. Uma interface normalmente não fornece implementações dos membros que ele 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**.

```
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**.

```
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

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

Enumerações

Um tipo de [Enumeração](#) define um conjunto de valores constantes. A seguir, as seguintes `enum` constantes declarações que definem um ou diferente de raiz:

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

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

```
[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:

```
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 anulável_ ou _anuláveis_`. Uma variável anulável pode conter um `null` valor adicional, indicando que não há valor. Tipos de valores anuláveis (structs ou enums) são representados por [System.Nullable<T>](#). Os tipos de referência não anuláveis e anuláveis são representados pelo tipo de referência subjacente. A distinção é representada por metadados lidos pelo compilador e por algumas bibliotecas. O compilador fornece avisos quando referências anuláveis são desreferenciadas sem primeiro verificar seu valor `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 `_int anulável*`, inicializando-o para `null`. Em seguida, ele define o valor como `5`. Ele demonstra o mesmo conceito com uma `* cadeia de caracteres anulável*`. Para obter mais informações, consulte [tipos de valor anulável](#) e [tipos de referência anuláveis](#).

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

Tuplas

O C# oferece suporte a **tuplas**, que fornece 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 os nomes dos membros entre `(` e `)`, conforme mostrado no exemplo a seguir:

```
(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 fornecem 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

21/01/2022 • 25 minutes to read

Os tipos descritos no artigo anterior são criados usando estes blocos de construção: * [Members](#), [Expressions](#) e [Statements](#) *.

Membros

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

- **Constantes**: valores constantes associados à classe
- **Campos**: variáveis que estão 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` : O acesso não é limitado.
- `private` : O acesso é limitado a essa classe.
- `protected` : O acesso é limitado a esta classe ou classes derivadas desta classe.
- `internal` : O acesso é limitado ao assembly atual (`.exe` ou `.dll`).
- `protected internal` : O acesso é limitado a essa classe, classes derivadas dessa classe ou classes dentro do mesmo assembly.
- `private protected` : O acesso é limitado a essa classe ou 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 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 exemplo a seguir, cada instância da `Color` classe tem uma cópia separada dos campos de `R` instância,, `G` e `B`, mas há apenas uma cópia dos `Black` `White` `Red` `Green` campos estáticos,, e `Blue` :

```
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 na 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 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, no número de parâmetros de tipo e no número, nos modificadores e nos tipos de seus parâmetros. A assinatura de um método não inclui o tipo de retorno.

Quando um corpo de método é uma única expressão, o método pode ser definido usando um formato de expressão Compact, conforme mostrado no exemplo a seguir:

```
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. Modificações em um parâmetro de valor não afetam o argumento que foi 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`.

```
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` usando a sintaxe introduzida no C# 7.

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

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {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 `Write` `WriteLine` métodos e da `System.Console` classe são bons exemplos de uso de matriz de parâmetros. Eles são declarados da seguinte maneira.

```
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 de 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

```
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.

```
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 para execução 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.

```
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 incluisse um valor inicial, o compilador relataria um erro para os usos mais recentes do porque não seria `i` `i` definitivamente atribuído a esses pontos no programa.

Um método pode usar instruções `return` para retornar o controle é pelo chamador. Em um método que retorna `void`, as `return` instruções 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 `static` modificador é um *método estático*. Um método estático não opera em uma instância específica e só pode acessar diretamente 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 fazer referência a `this` um método estático.

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

```
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 `Entity` instância 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 o `_serialNo` campo de instância e o `s_nextSerialNo` campo estático.

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 `Entity` classe.

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

Métodos abstratos, virtuais e de substituição

Você usa os métodos `virtual`, `override` e `abstract` para definir o comportamento de uma hierarquia de tipos de classe. Como uma classe pode derivar de uma classe base, essa classe derivada pode 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 `_abstrato*` é um método declarado em uma classe base que `_must` ser substituído em todas as classes derivadas. Na verdade, os métodos abstratos não definem uma implementação na classe base.

As chamadas de método para métodos de instância podem ser resolvidas para a classe base ou implementações de classe derivada. O tipo de uma variável determina seu *tipo de tempo de compilação*. O *tipo de tempo de compilação* é o tipo que o compilador usa para determinar seus membros. No entanto, uma variável pode ser atribuída a uma instância de qualquer tipo derivado de seu *tipo de tempo de compilação*. 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 apresenta 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 `abstract` modificador e é permitido somente em uma classe abstrata. 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 a, mas não relacionado aos tipos de árvore de expressão).

```

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.

```
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`.

```
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 não for possível encontrar uma única melhor correspondência, um erro será relatado. O exemplo a seguir mostra a resolução de sobrecarga em vigor. O comentário para cada invocação no `UsageExample` método mostra qual método é invocado.

```

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)");
    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<string>(string)
        F((double)1);         // Invokes F(double)
        F((object)1);         // Invokes F(object)
        F<int>(1);            // Invokes F<int>(int)
        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 que aumenta. A classe contém vários exemplos dos tipos mais comuns de membros da função.

```

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
    {
        _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 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 sobrepostos 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.

```
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 gravado entre os delimitadores { e } em vez de terminar em 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 computa o valor da propriedade. O acessador set fornece um novo valor para a propriedade. Quando a propriedade é o destino de uma atribuição, ou o operando de ++ 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 código a seguir é um exemplo de uso dessas propriedades:

```
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:

```

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 sobre carregados. 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 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 classe `MyList<T>` declara um membro único de evento chamado `Changed`, que indica que um novo item foi adicionado à lista. 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 a criação de um evento é precisamente equivalente à invocação do delegado representado pelo evento. Não há constructos de linguagem especiais para a criação de 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>`.

```

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 acessadores `e`, que são semelhantes ao acessador de `add`, `remove` uma `set` 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 `MyList<T>` classe declara dois operadores, e `operator ==` `operator !=`. Esses operadores substituídos dão novo significado a expressões que aplicam esses operadores a `MyList` instâncias. Specificamente, os operadores definem a igualdade de duas instâncias como comparando `MyList<T>` cada um dos objetos contidos usando seus `Equals` métodos. O exemplo a seguir usa o operador `==` para comparar duas instâncias `MyList<int>`.

```
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 nãomanagedos. Os finalizadores não podem ter parâmetros, 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, consulte o artigo sobre [finalizadores](#).

O coletor de lixo tem latitude ampla ao decidir quando coletar objetos e executar os finalizadores. Especificamente, o tempo de invocações do finalizador não é determinístico e os finalizadores 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 coalescing 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 e coalescing nulos 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 .

A maioria dos operadores pode ser *sobre carregada*. A sobre carga 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, bit a bit e shift e comparações de igualdade e 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 `if` instruções `switch` e.
 - *Instruções de iteração* são usadas para executar repetidamente uma instrução inserida. Esse grupo contém as `while` instruções , e `do` `for` `foreach` .
 - *Instruções de salto* são usadas para transferir o controle. Esse grupo contém as `break` instruções ,,, e `continue` `goto` `throw` `return` `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.

O seguinte lista os tipos de instruções que podem ser usadas:

- Declaração de variável local.
 - Declaração de constante local.
 - Instrução expression.
 - `if` Declaração.
 - `switch` Declaração.
 - `while` Declaração.
 - `do` Declaração.
 - `for` Declaração.
 - `foreach` Declaração.

- `break` Declaração.
- `continue` Declaração.
- `goto` Declaração.
- `return` Declaração.
- `yield` Declaração.
- `throw` instruções e `try` instruções.
- `checked` Instruções `unchecked` e .
- `lock` Declaração.
- `using` Declaração.

[ANTERIOR](#)

[PRÓXIMO](#)

Principais áreas de linguagem

21/01/2022 • 9 minutes to read

Matrizes, coleções e LINQ

O C# e o .NET fornecem muitos tipos de coleção diferentes. As matrizes têm sintaxe definida pela linguagem. Os tipos de coleção genéricos são listados no [System.Collections.Generic](#) namespace. As coleções especializadas incluem para acessar a memória contínua no quadro de pilha e para acessar a [System.Span<T>](#) memória contínua no heap [System.Memory<T>](#) gerenciado. Todas as coleções, incluindo matrizes, [Span<T>](#) e compartilham um princípio [Memory<T>](#) unificador para iteração. Use a [System.Collections.Generic.IEnumerable<T>](#) interface. 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 várias variáveis acessadas por meio de índices computados. As variáveis contidas em uma matriz, também *chamadas* de elementos da matriz, são do mesmo tipo. 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 operação usando o `new` operador. A operação especifica o comprimento da nova instância de matriz, que é corrigida `new` durante 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, inicializa a `int` matriz e imprime o conteúdo da matriz.

```
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 dá _suporte a matrizes multidimensionais_. O número de dimensões de um tipo de matriz, também conhecido como _ rank do tipo de matriz, é um mais o número de vírgulas entre os colchetes do tipo de matriz. O exemplo a seguir aloca uma matriz unidimensional, bidimensional e tridimensional, respectivamente.

```
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 (10×5) 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 às vezes é chamada de matriz irregular porque os comprimentos das matrizes de elemento não têm que ser todos iguais. O exemplo a seguir aloca uma matriz de

matrizes de `int`:

```
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 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 `new` delimitadores e `{ }` . O exemplo a seguir aloca e inicializa um `int[]` com três elementos.

```
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 forma que o tipo de matriz não precisa ser restado.

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

Ambos os exemplos anteriores são equivalentes ao seguinte código:

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

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

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

A `foreach` instrução usa `IEnumerable<T>` a interface, portanto, pode funcionar com qualquer coleção.

Interpolação de cadeia de caracteres

A *interpolação de cadeia de caracteres* C# permite formatar cadeias de caracteres definindo expressões cujos resultados são colocados em uma cadeia de caracteres de formato. Por exemplo, o exemplo a seguir imprime a temperatura em um determinado dia de um conjunto de dados meteorológicos:

```
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 e substitui o texto entre os colchetes pelo resultado da cadeia de `{ }` string caracteres da expressão. O `:` na primeira expressão especifica a cadeia de `{weatherData.Date:MM-DD-YYYY}` 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 correspondência* de padrões para consultar o estado de um objeto e executar código com base nesse estado. Você pode inspecionar tipos e os valores de propriedades e campos para determinar qual ação deve ser tomada. A `switch` expressão é a expressão primária para correspondência de padrões.

Delegados e expressões lambda

Um *tipo delegado* representa referências a métodos com uma lista de parâmetros e um 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 encontrados em algumas outras linguagens. Ao contrário dos ponteiros de função, os delegados são orientados a objetos e são de tipo seguro.

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

```
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 `Apply` método aplica um determinado aos elementos de um , `Function` `double[]` retornando um com os `double[]` 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 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 exemplo a seguir não cria uma classe:

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

Um delegado não sabe ou se importa com a classe do método referenciado. O método referenciado deve ter os mesmos parâmetros e o 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 `async` modificador a uma declaração de método para declarar que o método é assíncrono. O `await` operador informa ao compilador para aguardar de forma assíncrona um resultado ser final. 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 dá suporte ao padrão awaiter. Esses recursos permitem escrever código que lê como sua contraparte síncrona, mas é executado de forma assíncrona. Por exemplo, o código a seguir baixa o home page para [Documentos da Microsoft](#):

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await client.GetByteArrayAsync("https://docs.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 `async` modificador.
- O corpo do método `await` é o retorno do método `GetByteArrayAsync`.
- O tipo especificado na instrução `return` corresponde ao argumento `type` na declaração do método `Task<T>`. (Um método que retorna um `Task` usaria instruções sem nenhum `return` 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 fornecer links para a documentação associada.

```

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 `Attribute` da classe base 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.

```

[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://docs.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 corrida 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`.

```
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 C# experimentando um dos nossos [tutoriais](#).

[ANTERIOR](#)

Introdução ao C#

21/01/2022 • 2 minutes to read

Bem-vindo aos tutoriais de introdução ao C#. Essas lições começam com o código interativo que você pode executar no navegador. Você pode aprender os conceitos básicos do C# na série de vídeos [C# 101](#) antes de iniciar essas lições interativas.

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 inssiva uma janela de código C# em runnable nas páginas de documentos. Você escreve e executa o código C# no navegador.
- [Na experiência Microsoft Learn](#). Este caminho de aprendizagem contém vários módulos que ensinam os conceitos básicos do C#.
- [No Jupyter no Binder](#). Você pode experimentar o código C# em um jupyter notebook no binder.
- [No 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 [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](#).

Este tutorial presume que você concluiu 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](#).

Este tutorial presume que você concluiu 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](#).

Este tutorial presume que você concluiu as lições listadas acima.

101 Exemplos de Linq

Este exemplo requer a ferramenta global [dotnet-try](#). Depois de instalar a ferramenta e clonar o repo [try-samples](#), você poderá aprender LINQ (Consulta Integrada à Linguagem) por meio de um conjunto de 101 exemplos que você pode executar interativamente. Você pode explorar diferentes maneiras de consultar, explorar e transformar sequências de dados.

Configurar o ambiente local

21/01/2022 • 2 minutes to read

A primeira etapa para executar um tutorial em seu computador é configurar um ambiente de desenvolvimento. Escolha uma das seguintes alternativas:

- Para usar a CLI do .NET e sua escolha de texto ou editor de código, consulte o tutorial do .NET Olá, Mundo [em 10 minutos](#). O tutorial tem instruções para configurar um ambiente de desenvolvimento no Windows, Linux ou macOS.
- Para usar a CLI do .NET e Visual Studio Code, instale o [SDK do .NET](#) e [Visual Studio Code](#).
- Para usar Visual Studio 2019, consulte Tutorial: Criar um aplicativo de [console C# simples no Visual Studio](#).

Fluxo de desenvolvimento de aplicativos básicos

As instruções nesses tutoriais pressuem que você está usando a CLI do .NET para criar, criar e executar aplicativos. Você usará os seguintes comandos:

- `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 Visual Studio 2019 para esses tutoriais, escolherá uma seleção de menu Visual Studio quando um tutorial o direcionar para executar um destes comandos da CLI:

- Arquivo > Novo > Project** cria um aplicativo.
 - O `Console Application` modelo de projeto é recomendado.
 - Você terá a opção de especificar uma estrutura de destino. Os tutoriais abaixo funcionam melhor ao direcionar para o .NET 5 ou superior.
- Build > A Solução de Build** cria o executável.
- Depurar > Iniciar sem depuração** 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 [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.

Esse tutorial pressupõe a conclusão das lições listadas acima.

Manipular números de ponto flutuante e integrais em C#

21/01/2022 • 9 minutes to read

Este tutorial ensina sobre os tipos numéricos em C# de maneira interativa. 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#.

Pré-requisitos

O tutorial espera que você tenha uma máquina configurada para desenvolvimento local. em Windows, Linux ou macOS, você pode usar a CLI do .net para criar, compilar e executar aplicativos. no Mac ou Windows, você pode usar Visual Studio 2019. Para obter instruções de instalação, consulte [configurar seu ambiente local](#).

Explorar a matemática de inteiros

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

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

IMPORTANT

Os modelos C# para .NET 6 usam *instruções de nível superior*. Seu aplicativo pode não corresponder ao código neste artigo, se você já tiver atualizado para as visualizações do .NET 6. Para obter mais informações, consulte o artigo sobre [novos modelos C# gerar 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 código a seguir:

```
using System;

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 uma das operações matemáticas fundamentais com números inteiros. O `int` tipo representa um número **inteiro**, zero, positivo ou inteiro 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`:

```
// 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 escrevendo várias operações matemáticas na mesma linha, se desejar.

Experimente `c = a + b - 12 * 17;`, por exemplo. É permitido misturar variáveis e números constantes.

TIP

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 o contiver mensagens de erro, examine de forma mais detalhada o código de exemplo e o código em sua janela para ver o que corrigir. Esse exercício ajudará você a conhecer a estrutura do código C#.

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

```
using System;  
  
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:

```
//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 de comentários. Como `WorkWithIntegers()` é um método, você precisa comentar apenas 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 após a chamada para `WorkWithIntegers()` e executando `dotnet run`:

```
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:

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

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

```
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 não tiver visto esse comportamento, tente o seguinte código:

```
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 ser semelhante a este:

```
using System;
```

```
// 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 resto usando o operador de **módulo**, o `%` caractere. Tente o seguinte código após a chamada de método para

```
OrderPrecedence() :
```

```
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:

```
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:

```
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 os outros tipos a seguir. 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. A **precisão dupla** é um termo relativo que descreve o número de dígitos binários usados para armazenar o valor. Números de **precisão dupla** têm duas vezes o número de dígitos binários como **uma única precisão**. Em computadores modernos, é mais comum usar precisão dupla do que números de precisão única. Números de **precisão única** são declarados usando a `float` palavra-chave. Vamos explorar. Adicione o seguinte código e veja o resultado:

```
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:

```
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:

```
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:

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

Você sabe que `0.3` repetir não é exatamente o mesmo que `1/3`.

Desafio

Tente outros cálculos com números grandes, números pequenos, multiplicação e divisão usando o `double` tipo. 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. Há um outro tipo para aprender: o `decimal` tipo. O tipo `decimal` tem um intervalo menor, mas precisão maior do que `double`. Vamos dar uma olhada:

```
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:

```
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 assume o `double` tipo.

NOTE

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

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 `System.Math` namespace, é um `double` valor. Por esse motivo, você deve usar `double` valores instead of `decimal` para esse desafio.

Você deve obter uma resposta entre 19 e 20. Você pode verificar sua resposta [examinando o código de exemplo concluído em 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.

Você pode saber mais sobre os números em C# nos seguintes artigos:

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

Saiba mais sobre lógica condicional com instruções branch e loop

21/01/2022 • 10 minutes to read

Este tutorial ensina a escrever código 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#.

Pré-requisitos

O tutorial espera que você tenha uma máquina configurada para desenvolvimento local. em Windows, Linux ou macOS, você pode usar a CLI do .net para criar, compilar e executar aplicativos. no Mac e no Windows, você pode usar Visual Studio 2019. Para obter instruções de instalação, consulte [configurar seu ambiente local](#).

Tome decisões usando a instrução `if`

Crie um diretório chamado *branches-tutorial*. Faça com que o diretório atual e execute o seguinte comando:

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

IMPORTANT

Os modelos C# para .NET 6 usam *instruções de nível superior*. Seu aplicativo pode não corresponder ao código neste artigo, se você já tiver atualizado para as visualizações do .NET 6. Para obter mais informações, consulte o artigo sobre [novos modelos C# gerar 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.

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

```
using System;

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 deve ver a mensagem "A resposta é maior do que 10." impressa no console. Modifique a declaração de `b` para que a soma seja inferior a 10:

```
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.

TIP

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, ele será ignorado. Esse processo de verificação de condições e execução de instruções com base nessas condições é poderoso.

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 `else` ramificação. Adicione as duas últimas linhas no código abaixo (você já deve ter as quatro primeiras):

```
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`.

IMPORTANT

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 deseja que mais de uma instrução faça parte do bloco que é executado condicionalmente. Os programadores em C# normalmente usam essas chaves em todas as cláusulas `if` e `else`. O exemplo a seguir é o mesmo que você criou. Modifique o código acima para coincidir com o código a seguir:

```
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");
}
```

TIP

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 código que você escreveu até agora:

```
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:

```
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 na parte superior do seu programa. Quando você concluiu essas alterações, seu código deve ser semelhante ao seguinte:

```

using System;

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:

```
//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 nenhum código executável de comentários.

Use loops para repetir operações

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

```

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`.

IMPORTANT

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:

```
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:

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

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

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

A parte intermediária é a **condição for**: `index < 10` declara que esse `for` loop continua a ser executado, desde que o valor do contador seja menor que 10.

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

Experimente por conta própria. 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 looping que não é abordada neste tutorial: a `foreach` instrução. A `foreach` instrução repete sua instrução para cada item em uma sequência de itens. Ele é usado com mais frequência com *coleções* e, portanto, é abordado no próximo tutorial.

Loops aninhados criados

Um `while`, `do` ou `for` loop,, ou 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 `for` loop pode gerar as linhas:

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

Outro loop pode gerar as colunas:

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

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

```
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 é incrementado uma vez para cada execução completa do loop interno. Inverta o aninhamento de linha e coluna e veja as alterações por conta própria. 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.

Você pode saber mais sobre esses conceitos nestes artigos:

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

Saiba como gerenciar coleções de dados usando o tipo de lista genérico

21/01/2022 • 6 minutes to read

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

Pré-requisitos

O tutorial espera que você tenha uma máquina configurada para desenvolvimento local. No Windows, Linux ou macOS, você pode usar a CLI do .NET para criar, criar e executar aplicativos. No Mac e Windows, você pode usar Visual Studio 2019. Para obter instruções de instalação, [consulte Configurar seu ambiente local](#).

Um exemplo de lista básica

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

IMPORTANT

Os modelos C# para .NET 6 usam *instruções de nível superior*. Seu aplicativo pode não corresponder ao código neste artigo, se você já tiver atualizado para as visualizações do .NET 6. Para obter mais informações, consulte o artigo sobre [novos modelos C# gerar 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 código existente pelo seguinte:

```
using System;
using System.Collections.Generic;

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 todos os CAPS. 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:

```
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:

```
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 final 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 seguinte código ao final do programa:

```
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 do programa:

```
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 [Sort](#) método classifica todos os itens na lista em sua ordem normal (em ordem alfabética para cadeias de caracteres). Adicione este código à parte inferior do programa:

```
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 esta aparência:

```

using System;
using System.Collections.Generic;

WorkWithString();

void WorkWithString()
{
    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 programa depois de chamar `WorkWithStrings()` :

```

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:

```
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.

TIP

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

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.

Você pode saber mais sobre como trabalhar com o `List` tipo no artigo Conceitos básicos do .NET sobre [coleções](#). Você também aprenderá muitos outros tipos de coleção.

Estrutura geral de um programa em C#

21/01/2022 • 2 minutes to read

Os programas em C# consistem em um ou mais arquivos. Cada arquivo contém zero ou mais namespaces. Um namespace contém tipos como classes, structs, interfaces, enumerações e delegados, ou outros namespaces. O exemplo a seguir é o esqueleto de um programa C# que contém todos esses elementos.

```
// 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`, conforme mostrado no exemplo a seguir:

```
// 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 aprende sobre esses elementos do programa na seção [tipos](#) do guia 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

21/01/2022 • 9 minutes to read

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

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

```
using System;

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

Começando no C# 9, você pode omitir o método e escrever instruções C# como se elas estavam no método , como `Main` `Main` no exemplo a seguir:

```
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 delimitada ou struct não é necessária para ser estática.
- O `Main` pode ter um tipo de retorno `void`, `int` ou, a partir do C# 7.1, `Task` ou `Task<int>`.
- Se e somente se `Main` retornar um ou , a declaração de poderá incluir o `Task` `Task<int>` `Main` `async` modificador . Isso exclui especificamente um `async void Main` método .
- O método `Main` pode ser declarado com ou sem um parâmetro `string[]` que contém os argumentos de linha de comando. Ao usar Visual Studio para criar Windows aplicativos, você pode adicionar o parâmetro manualmente ou usar o método para obter os argumentos de `GetCommandLineArgs()` linha de comando. Os parâmetros são lidos como argumentos de linha de comando indexados por zero. Ao contrário de C e

C++, o nome do programa não é tratado como o primeiro argumento de linha de comando na matriz, mas é o primeiro `args` elemento do `GetCommandLineArgs()` método.

A lista a seguir mostra `Main` assinaturas válidas:

```
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 `public` modificador do acessador. 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 Main()

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

MAIN CÓDIGO DO MÉTODO	MAIN ASSINATURA
Sem uso de <code>args</code> ou <code>await</code>	<code>static int Main()</code>
Usa <code>args</code> , sem uso de <code>await</code>	<code>static int Main(string[] args)</code>
Sem uso de <code>args</code> , usa <code>await</code>	<code>static async Task<int> Main()</code>
Usa <code>args</code> e <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

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

MAIN CÓDIGO DO MÉTODO	MAIN ASSINATURA
Sem uso de <code>args</code> ou <code>await</code>	<code>static void Main()</code>
Usa <code>args</code> , sem uso de <code>await</code>	<code>static void Main(string[] args)</code>
Sem 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, retornar ou permitir que o programa comunique informações de status para outros programas ou scripts que `int` `Task<int>` 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 saber mais sobre elas neste [artigo de início](#).

Crie um novo aplicativo executando `dotnet new console`. Modifique `Main` o método em `Program.cs` da

seguinte forma:

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    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 `ERRORLEVEL` usando de um arquivo em lotes ou do `$LastExitCode` PowerShell.

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

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 digitando `test.ps1` no prompt do PowerShell.

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

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

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

Valores retornados de Async Main

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

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

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

Esse código clichê pode ser substituído por:

```

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

```

Uma vantagem de declarar `Main` como é que o `async` 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();`

NOTE

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:

MAIN CÓDIGO DO MÉTODO	MAIN ASSINATURA
Sem valor de retorno, sem uso de <code>await</code>	<code>static void Main(string[] args)</code>
Valor de retorno, sem uso de <code>await</code>	<code>static int Main(string[] args)</code>
Nenhum valor de retorno, usa <code>await</code>	<code>static async Task Main(string[] args)</code>
Valor de retorno, usa <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

Se os argumentos não são usados, você pode omitir da `args` assinatura do método para um código um pouco mais simples:

MAIN CÓDIGO DO MÉTODO	MAIN ASSINATURA
Sem valor de retorno, sem uso de <code>await</code>	<code>static void Main()</code>
Valor de retorno, sem uso de <code>await</code>	<code>static int Main()</code>
Nenhum valor de retorno, usa <code>await</code>	<code>static async Task Main()</code>
Valor de retorno, usa <code>await</code>	<code>static async Task<int> Main()</code>

NOTE

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

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

TIP

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

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`:

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

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

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

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

```
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 for 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 *fatorial.cs*.

```

using System;

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 **iniciar** ou no menu **iniciar**, abra uma janela do Visual Studio **Prompt de Comando do Desenvolvedor** e, em seguida, navegue até a pasta que contém o arquivo que você criou.
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.

NOTE

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 `Main` métodos

21/01/2022 • 3 minutes to read

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

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

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

Instruções de nível superior permitem escrever programas simples para utilitários pequenos, como Azure Functions e GitHub Actions. Eles também simplificam o aprendizado e a escrita de código para novos programadores em C#.

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

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

```
CS8802 Somente uma unidade de compilação pode ter instruções de nível superior.
```

Um projeto pode ter qualquer número de arquivos de código-fonte adicionais que não têm instruções de nível superior.

Nenhum outro ponto de entrada

Você pode escrever um `Main` método 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 `Main` métodos.

`using` Directivas

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

```
using System.Text;

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

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

Namespace global

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 eles devem vir após as instruções de nível superior. Por exemplo:

```
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 referenciar `args` a variável para acessar os argumentos de linha de comando que foram inseridos. A `args` variável nunca será nula, mas `Length` será zero se nenhum argumento de linha de comando tiver sido fornecido. Por exemplo:

```
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:

```
Console.WriteLine("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

Código de saída para o processo

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

```
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 não é, na verdade, `Main`, é um detalhe de implementação `Main` que seu código não pode referenciar diretamente. A assinatura do método depende se as instruções de nível superior contêm a `await` palavra-chave ou a `return` instrução. A tabela a seguir mostra a aparência da assinatura do método, usando o nome do `Main` método na tabela para sua conveniência.

O CÓDIGO DE NÍVEL SUPERIOR CONTÉM	ASSINATURA <code>MAIN</code> IMPLÍCITA
<code>await</code> e <code>return</code>	<code>static async Task<int> 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> ou <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 de tipos C#

21/01/2022 • 15 minutes to read

O 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 o tipo (valor, referência ou saída) para cada parâmetro de entrada e para o valor de retorno. A biblioteca de classes .NET define tipos numéricos internos e tipos complexos que representam uma ampla variedade de construções. Isso inclui o sistema de arquivos, conexões de rede, coleções e matrizes de objetos e datas. Um programa C# típico usa tipos da biblioteca de classes e dos tipos definidos pelo usuário que modelam os conceitos 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.
- As interfaces que ela implementa.
- Os tipos de operações que são permitidos.

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

```
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;
```

NOTE

Os desenvolvedores de C e C++, observe que, em C#, `bool` não são conversíveis para `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 você declara uma variável ou constante em um programa, deve especificar seu tipo ou usar a `var` palavra-chave para permitir que o compilador inferir o tipo. O exemplo a seguir mostra algumas declarações de variáveis que usam tipos numéricos internos e tipos complexos definidos pelo usuário:

```
// 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:

```
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, você não pode redeclará-la com um novo tipo e não pode atribuir um valor não compatível 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 atribuídos a novas variáveis ou passados como argumentos de método. 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. Elas representam 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 dos tipos internos, consulte [tipos internos](#).

Tipos personalizados

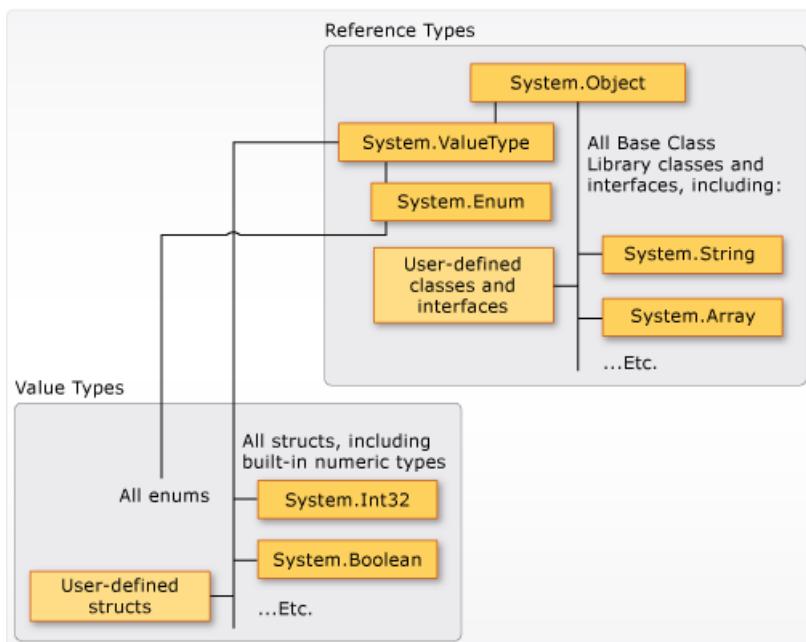
Você usa as `struct` construções, `class`, `interface`, `enum` e `record` para criar seus próprios tipos personalizados. A própria biblioteca de classes do .NET é uma coleção de tipos personalizados que você pode usar em seus próprios aplicativos. Por padrão, os tipos usados com mais frequência na biblioteca de classes estão disponíveis em qualquer programa em C#. Outras se tornam disponíveis somente quando você adiciona explicitamente uma referência de projeto ao assembly que as 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 dois pontos fundamentais sobre o sistema de tipos no .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 `System.Int32` (palavra-chave c#: `int`), derivam, por fim, de um único tipo base, que é `System.Object` (palavra-chave 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 e também seus próprios tipos definidos pelo usuário. Os tipos que você define usando a `struct` palavra-chave são tipos de valor; todos os tipos numéricos internos são `structs`. Os tipos que você define usando a `class` `record` palavra-chave ou 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.



NOTE

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

Uma declaração de classe, estrutura ou registro é como uma planta usada para criar instâncias ou objetos em tempo de execução. Se você definir uma classe, struct ou registro chamado `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 mantém apenas uma referência a essa 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 a struct é atribuída a uma nova variável, ela é copiada. 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.

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. Normalmente, as classes armazenam dados que devem ser modificados depois que um objeto de classe é criado. As structs são mais adequadas para estruturas de dados pequenas. Normalmente, as estruturas armazenam dados que não devem ser modificados após a criação da estrutura. Os tipos de registro são estruturas de dados com membros sintetizados adicionais do compilador. Normalmente, os registros armazenam dados que não devem ser modificados depois que o objeto é criado.

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. Variáveis de tipo de valor contêm diretamente seus valores. A memória de um struct é alocada embutida em qualquer contexto que a variável for declarada. Não há nenhuma alocação de heap separada ou sobrecarga de coleta de lixo para variáveis de tipo de valor. Você pode declarar `record struct` tipos 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:

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

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

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

Os tipos de valor são *lacrados*. Você não pode derivar um tipo de qualquer tipo de valor, por exemplo [System.Int32](#). Você não pode definir uma struct para herdar de qualquer classe definida pelo usuário ou struct porque uma struct só pode herdar de [System.ValueType](#). No entanto, um struct pode implementar uma ou mais interfaces. Você pode converter um tipo struct em qualquer tipo de interface que ele implementa. Essa conversão faz com que uma operação *Boxing* envolva a 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 container para um pequeno conjunto de variáveis relacionadas, conforme mostrado no exemplo a seguir:

```
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 obter mais informações sobre tipos de valor, consulte [tipos de valor](#).

A outra categoria de tipos de valor é `enum`. Uma enum define um conjunto de constantes integrais 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. Ele é definido como mostrado no exemplo a seguir:

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

A `System.IO.FileMode.Create` constante tem um valor de 2. No entanto, o nome é muito mais significativo para as pessoas que leem o código-fonte 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 um `class`, `record`, `delegate`, matriz ou `interface` é um `reference type`.

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

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

Um `interface` não pode ser instaciado diretamente usando o `new` operador. Em vez disso, crie e atribua uma instância de uma classe que implementa a interface. Considere o exemplo a seguir:

```
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 mantém apenas uma referência ao local do objeto. Os tipos no heap gerenciado exigem sobrecarga quando são alocados e quando são 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, ela 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 [System.Array](#) classe. Você declara e usa-os com a sintaxe simplificada fornecida pelo C#, conforme mostrado no exemplo a seguir:

```
// 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, pode herdar de qualquer outra interface ou classe que não esteja definida como [sealed](#). Outras classes podem herdar de sua classe e substituir seus métodos virtuais. Para obter mais informações sobre como criar suas próprias classes, consulte [classes, estruturas 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 "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 inteiros](#) e [tipos numéricos de ponto flutuante](#).

Como os literais são digitados e todos os tipos derivam [System.Object](#), você pode escrever e compilar um código como o código a seguir:

```
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 cria uma instância do tipo. Esses tipos são chamados de *tipos genéricos*. Por exemplo, o tipo .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 especifica o

tipo dos objetos que a lista conterá, por exemplo `string` :

```
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 com rigidez de *tipos*, pois o compilador sabe o tipo específico dos elementos da coleção e pode gerar um erro no momento da compilação se, por exemplo, você tentar adicionar um inteiro ao `stringList` objeto no exemplo anterior. Para obter mais informações, consulte [Genéricos](#).

Tipos implícitos, tipos anônimos e tipos de valor anulável

Você pode digitar implicitamente uma variável local (mas não membros da classe) usando a `var` palavra-chave. 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 passar os limites de método externos. Você pode criar *tipos anônimos* para essa finalidade. Para obter mais informações, consulte [Tipos Anônimos](#).

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

Tipo de tempo de compilação e tipo de tempo de execução

Uma variável pode ter tipos diferentes 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 os mesmos, como no exemplo a seguir:

```
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:

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

Nos dois exemplos anteriores, o tipo de tempo de execução é um `string`. O tipo de tempo de compilação está `object` na primeira linha e `IEnumerable<char>` no segundo.

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 de 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 `is` e `switch` expressões 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

21/01/2022 • 2 minutes to read

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

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

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

```
using System;
```

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

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

IMPORTANT

Os modelos C# para .NET 6 usam *instruções de nível superior*. Seu aplicativo pode não corresponder ao código neste artigo, se você já tiver atualizado para as visualizações do .NET 6. Para obter mais informações, consulte o artigo sobre [novos modelos C# gerar 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.

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:

```
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, conforme

mostrado no exemplo a seguir:

```
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 usando o `.` 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

21/01/2022 • 5 minutes to read

Tipos de referência

Um tipo que é definido como um `class` é um *tipo de referência*. Em 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 `new` operador ou atribua a ela um objeto de um tipo compatível que pode ter sido criado em outro lugar, conforme mostrado no exemplo a seguir:

```
//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. Os tipos no heap gerenciado requerem sobrecarga quando são alocados e quando são recuperados pela funcionalidade de gerenciamento automático de memória do CLR, que é conhecida como *coleta de lixo*. 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 e coleta de lixo](#).

Declarando Classes

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

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

A palavra-chave `class` é precedida pelo nível de acesso. Como `public` é usado neste 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 `new` palavra-chave seguida pelo nome da classe na qual o objeto se baseará, assim:

```
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:

```
Customer object2;
```

Não recomendamos a criação de referências de objeto, como a anterior, que não se refere a um objeto, pois a tentativa de acessar um objeto por meio de uma referência desse tipo falhará em tempo de execução. No entanto, essa referência pode ser feita para fazer referência a um objeto, seja criando um novo objeto ou atribuindo-o a um objeto existente, como este:

```
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.

Herança de classe

As classes dão suporte completo à *herança*, uma característica fundamental da programação orientada a objetos. Quando você cria uma classe, pode herdar de qualquer outra classe que não esteja definida como `sealed`, e 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:

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

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

Uma classe em 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 `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 abstratas e lacradas e membros de classe](#).

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

Exemplo

No exemplo a seguir, é definida uma classe pública que contém uma [propriedade autoimplementada](#), um método e um método especial chamado construtor. Para obter mais informações, consulte os artigos [Propriedades, métodos e construtores](#). As instâncias da classe são então instanciadas com a palavra-chave `new`.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

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 registros

21/01/2022 • 4 minutes to read

Um [registro](#) é uma [classe ou struct que](#) fornece sintaxe e comportamento especiais para trabalhar com modelos de dados.

Quando usar registros

Considere o uso de 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, igualdade de valor significa que duas variáveis de um tipo de registro serão iguais se os tipos corresponderem e todos os valores de propriedade e campo corresponderem. Para outros tipos de referência, como classes, igualdade significa [igualdade de referência](#). Ou seja, duas variáveis de um tipo de classe serão iguais se se referirem 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, [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 mude qualquer propriedade ou valores de campo de um objeto após sua instância. A imutabilidade pode ser útil quando você precisa que um tipo seja thread-safe ou depende de um código hash que permanece 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. [Entity Framework Core](#), por exemplo, não dá suporte à atualização com tipos de entidade imutáveis.

Como os registros diferem de classes e structs

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

- Você pode usar [parâmetros posicionais](#) para criar e criar uma instância de um tipo com propriedades imutáveis.
- Os mesmos métodos e operadores que indicam igualdade ou desigualdade de referência em classes (como `==`), indicam igualdade de valor ou `Object.Equals(Object)` `==` [desigualdade](#) em registros.
- Você pode usar uma [with expressão](#) para criar uma cópia de um objeto imutável com novos valores nas propriedades selecionadas.
- O método de um registro cria uma cadeia de caracteres formatada que mostra o nome do tipo de um objeto e os nomes e valores `ToString` de todas as suas propriedades públicas.
- 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 para igualdade e `ToString`. O compilador sintetiza um `Deconstruct` método para structs de registro posicionais.

Exemplos

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

```
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 exemplo a seguir demonstra a igualdade de valor em registros:

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

```
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 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

21/01/2022 • 4 minutes to read

Uma interface contém definições para um grupo de funcionalidades relacionadas que um não abstrato `class` ou um `struct` deve implementar. Uma interface pode definir `static` métodos, que devem ter uma implementação. A partir do C# 8.0, uma interface pode definir uma implementação padrão para membros. Uma interface não pode declarar dados de instância, como campos, propriedades implementadas automaticamente ou eventos de propriedade.

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 `interface` palavra-chave como mostra o exemplo a seguir.

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

O nome de uma interface deve ser um nome de identificadorC# válido. 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 o `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 abstratas e lacrados](#).

As interfaces podem conter métodos de instância, propriedades, eventos, indexadores ou qualquer combinação desses quatro tipos de membro. As interfaces podem conter construtores, campos, constantes ou operadores estáticos. 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 `private` membro 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.

Quando uma classe ou struct implementa uma interface, a classe ou struct deve fornecer uma implementação para todos os membros que a interface declara, mas não fornece uma implementação padrão para o. 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`.

```
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](#).

As interfaces podem 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 na interface derivada ou em 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 de classe no 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:

- Em versões C# anteriores a 8,0, uma interface é como uma classe base abstrata com apenas membros abstratos. Uma classe ou struct que implementa a 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 [métodos 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

21/01/2022 • 3 minutes to read

Genéricos introduz o conceito de parâmetros de tipo para o .NET, o que possibilita criar classes e métodos que adiam a especificação de um ou mais tipos até que a classe ou o método seja declarado e instanciado pelo código do cliente. Por exemplo, usando um parâmetro de tipo genérico , você pode escrever uma única classe que outro código cliente pode usar sem incorrer no custo ou risco de operações boxing ou de runtime, conforme mostrado `T` aqui:

```
// 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());
        }
    }
}
```

Classes e métodos genéricos combinam reutilização, segurança de tipo e eficiência de uma maneira que suas contrapartes não genéricas não podem. Os genéricos são usados com mais frequência com coleções e com os métodos que operam nelas. O [System.Collections.Generic](#) namespace contém várias classes de coleção baseadas em genéricos. 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 generalizadas e padrões de design que são seguros 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 fornecida pelo .NET em `List<T>` vez de criar sua própria.) O parâmetro `type` é usado em vários locais em que um tipo concreto normalmente seria usado para indicar o tipo do `T` 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 `Node` aninhada. Quando `GenericList<T>` é instanciada com um tipo concreto, por exemplo como um `GenericList<int>` , cada ocorrência de `T` será substituída por `int` .

```

// 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:

```
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");
    }
}
```

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 genéricas no [System.Collections.Generic](#) namespace. As coleções genéricas devem ser usadas sempre que possível em vez de classes como [ArrayList](#) no [System.Collections](#) namespace.
- Você pode criar suas próprias interfaces genéricas, classes, métodos, eventos e delegados.
- Classes genéricas podem ser restrinvidas 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

21/01/2022 • 4 minutes to read

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.

Você cria tipos anônimos usando o `new` operador junto 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`.

```
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);
```

Normalmente, os tipos anônimos são usados na cláusula de uma expressão de consulta para retornar um subconjunto das propriedades de `select` cada objeto na sequência de origem. Para obter mais informações sobre consultas, consulte [LINQ em 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 quantidade menor 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. Você fornece um nome para uma propriedade que está sendo inicializada com uma expressão, conforme mostrado no exemplo anterior. No exemplo a seguir, os nomes das propriedades do tipo anônimo são `Color` e `Price`.

```
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);
}
```

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.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

Tipos anônimos `class` são tipos que derivam diretamente de `object` que não podem ser lançados em nenhum tipo, exceto `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 suportam mutação não destrutiva na forma de [com expressões](#). Isso permite que você crie uma nova instância de um tipo anônimo em que uma ou mais propriedades têm novos valores:

```
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 contém tipos anônimos, como um argumento para um método, você pode declarar o parâmetro como tipo `object`. No entanto, `object` o uso de para tipos anônimos desanimou a finalidade da digitação 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.

Programação orientada a objeto

21/01/2022 • 4 minutes to read

Encapsulamento

Encapsulamento é chamado, ocasionalmente, de primeiro pilar ou princípio da programação orientada a objeto. Uma classe ou struct pode especificar o quanto acessível cada um de seus membros é para codificar fora da classe ou struct. Métodos e variáveis que não se destinam a serem usados de fora da classe ou assembly podem ser ocultos para limitar o potencial de codificação de erros ou explorações mal-intencionadas. Para obter mais informações, consulte [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. Até mesmo o ponto de entrada de um programa, o método `Main`, deve ser declarado dentro de uma classe ou `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, struct ou registro.

- Campos
- Constantes
- Propriedades
- Métodos
- Construtores
- Eventos
- Finalizadores
- Indexadores
- Operadores
- Tipos aninhados

Acessibilidade

Alguns métodos e propriedades devem ser chamados ou acessados do código fora de uma classe ou struct, conhecido como código *do cliente*. Outros métodos e propriedades podem ser usados apenas na classe ou struct em si. É importante limitar a acessibilidade do código para que somente o código do cliente pretendido possa chegar a ele. Especifique o quanto seus tipos e seus membros estão acessíveis ao código do cliente usando os seguintes modificadores de acesso:

- `público`
- `protected`
- `interno`
- `internos protegidos`
- `Privada`
- `private protected`.

A acessibilidade padrão é `private`.

Herança

Classes (mas não structs) dão suporte ao conceito de herança. Uma classe que deriva de outra classe, chamada de *classe base*, contém automaticamente todos os membros públicos, protegidos e internos da classe base, exceto seus construtores e finalizadores. Para obter mais informações, consulte [Herança e Polimorfismo](#).

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.

Interfaces

Classes, structs e registros podem implementar várias interfaces. Implementar 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 `List<T>` classe no `System.Collections.Generic` namespace 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 `new` palavra-chave. 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.

Tipos aninhados

Uma classe, struct ou registro pode ser aninhado dentro de outra classe, struct ou registro.

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.

Inicializadores de objeto

Você pode insinuar e inicializar objetos de classe ou struct, bem como coleções de objetos, atribuindo valores às suas propriedades.

Tipos anônimos

Em situações em que não é conveniente ou necessário criar uma classe nomeada, você usa tipos anônimos. Os tipos anônimos são definidos por seus membros de dados nomeados.

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 pertencesse ao tipo original.

Variáveis Locais Tipadas Implicitamente

Em uma classe ou método struct, você pode usar a digitação implícita para instruir o compilador a determinar o tipo de uma variável no tempo de compilação.

Registros

O C# 9 introduz o tipo `record`, um tipo de referência que você pode criar em vez de uma `record` classe ou um `struct`.

Os registros são classes com comportamento integrado para encapsular dados em tipos imutáveis. O C# 10 introduz o tipo `record struct` de valor. Um registro (ou `record class`) fornece os seguintes `record struct` recursos:

- Sintaxe concisa para criar um tipo de referência com propriedades imutáveis.
- Igualdade de valor. Duas variáveis de um tipo de registro serão iguais se tiverem o mesmo tipo e se, para cada campo, os valores em ambos os registros serão iguais. As classes usam igualdade de referência: duas variáveis de um tipo de classe serão iguais se se referirem ao mesmo objeto.
- Sintaxe concisa para mutação não estruturativa. Uma expressão permite criar uma nova instância de registro que é uma cópia de uma instância existente, mas com valores `with` de propriedade especificados alterados.
- Formatação integrado para exibição. O `ToString` método imprime o nome do tipo de registro e os nomes e valores de propriedades públicas.
- Suporte para hierarquias de herança em classes de registro. As classes de registro são suportadas por herança. Os structs de registro não são suportados por herança.

Para obter mais informações, consulte [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

21/01/2022 • 5 minutes to read

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.

NOTE

Tipos estáticos se comportam de modo diferente do que está descrito aqui. Para obter mais informações, consulte [classes estáticas e membros de classe estática](#).

Instâncias de struct versus 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, ambas as variáveis referem-se ao objeto nesse endereço. Esse ponto é discutido em 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.

```

using System;

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. As instâncias de structs também podem ser criadas usando o `new` operador, mas isso não é necessário, conforme mostrado no exemplo a seguir:

```

using System;

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 junto com o tipo ou o método no qual ele é declarado. 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 determinística um objeto de classe como você pode em C++. Para obter mais informações sobre a coleta de lixo no .NET, consulte [coleta de lixo](#).

NOTE

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 versus 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 você pretende comparar valores, deve 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:

```
// 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 [System.ValueType](#) implementação de `Equals` usa boxing e Reflection em alguns casos. Para obter informações sobre como fornecer um algoritmo de igualdade eficiente que é específico para seu tipo, consulte [como definir a igualdade de valor para um tipo](#). 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 declaradas 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
- novo operador
- Sistema de tipos comum

Herança – derivar tipos para criar um comportamento mais especializado

21/01/2022 • 6 minutes to read

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 `e` e `e` for derivado de `,` `ClassB` `ClassB` `ClassA` `ClassC` herdará os membros declarados em `ClassB` e `ClassA`.

NOTE

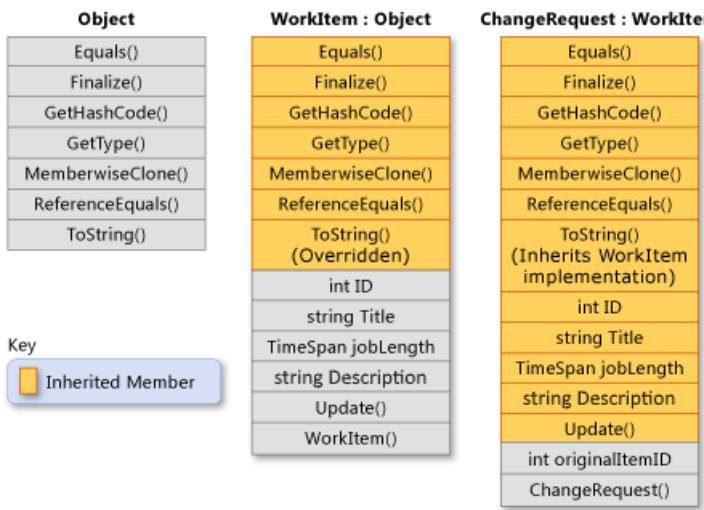
Structs não dão suporte a herança, mas podem implementar interfaces.

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.

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, consulte 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 reimplementá-lo. Você pode adicionar mais membros na classe derivada. 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 cinco 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:

```
// 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 GetNextID 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:

```

// 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 pode usar o método com sua própria `override` implementação. Se uma classe base declarar um membro como `abstract`, esse método deverá ser substituído em qualquer classe não abstrata que `abstract` herda diretamente dessa 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 só poderá ser usada 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 contém um membro abstrato, a própria classe deve ser declarada como abstrata. Classes derivadas que não são abstratas em si devem fornecer a implementação para quaisquer métodos abstratos 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 ou todos esses membros. Uma classe pode implementar várias interfaces, mesmo que ela possa derivar de apenas uma classe base direta.

As interfaces são usadas para definir funcionalidades específicas para classes que não têm necessariamente uma relação "é uma". Por exemplo, a interface pode ser implementada por qualquer classe ou struct para determinar se dois objetos do tipo são equivalentes (no entanto, o tipo `System.IEquatable<T>` define a equivalência). `IEquatable<T>` não implica o mesmo tipo de relação "é uma" existente entre uma classe base e

uma classe derivada (por exemplo, um `Mammal` é um `Animal`). Para obter mais informações, consulte [Interfaces](#).

Impedindo derivação posterior

Uma classe pode impedir que outras classes herdem dela ou de qualquer um de seus membros declarando-se ou o membro como `sealed`.

Ocultação de classe derivada de membros da classe base

Uma classe derivada pode ocultar membros da classe base declarando membros com mesmo nome e assinatura. O modificador pode ser usado para indicar explicitamente que o membro não se destina a ser uma `new` substituição do membro base. O uso de `não` é necessário, mas um `new` aviso do compilador será gerado se não `new` 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

21/01/2022 • 7 minutes to read

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 run-time.
- 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 `List<Shape>` objeto e adicione um , e a `Circle` `Triangle` `Rectangle` ele.

```

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. Embora cada objeto na lista tenha um tipo declarado de , é o tipo de tempo de run (a versão substituído do método em cada classe derivada) que será `shape` 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 do polimorfismo

Membros virtuais

Quando uma classe derivada herda de uma classe base, ela obtém todos os métodos, campos, propriedades e eventos da classe base. O designer da classe derivada tem opções diferentes para o comportamento dos métodos virtuais:

- 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 de 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:

```

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; somente métodos, propriedades, eventos e 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:

```

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 você quiser que sua classe derivada tenha um membro com o mesmo nome de um membro em uma classe base, poderá usar a nova palavra-chave 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:

```

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:

```
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 declarar um membro virtual e a classe derivar de e a classe derivar de , a classe herdará o membro virtual e poderá substituí-lo, independentemente de a classe ter declarado uma substituição para esse membro. O código a seguir mostra um exemplo:

```
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`.

Interromper a herança requer colocar a `sealed` palavra-chave antes `override` da palavra-chave na declaração de membro de classe. O código a seguir mostra um exemplo:

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

No exemplo anterior, o método `DoWork` não é mais virtual para nenhuma classe derivada de . Ele ainda é virtual para instâncias do , mesmo se elas são lançadas para o tipo ou o tipo . Os métodos lacrados podem ser substituídos por classes derivadas usando a `new` palavra-chave , como mostra o exemplo a seguir:

```
public class D : C
{
    public new void DoWork() { }
}
```

Nesse caso, se `DoWork` for chamado usando uma variável do tipo , o novo será `DoWork` chamado. Se uma variável do tipo , ou for usada para acessar uma instância do , uma chamada para seguirá as regras de herança virtual, roteamento dessas chamadas para a implementação de na classe .

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:

```
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](#).

NOTE

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 da correspondência de padrões

21/01/2022 • 7 minutes to read

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" dá suporte à correspondência de padrões para testar uma expressão e declarar condicionalmente uma nova variável `is` para o resultado dessa expressão. A `switch` "expressão" permite que você execute ações com base no primeiro padrão de correspondência de uma expressão. Essas duas expressões suportam um vocabulário rico de [padrões](#).

Este artigo fornece uma visão geral dos cenários em que você pode usar a correspondência de padrões. Essas técnicas podem melhorar a capacidade de leitura e a correção do código. Para 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 que pode ser anulado em seu tipo subjacente durante o teste para `null` usar o exemplo a seguir:

```
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 linguagem torna essa técnica mais segura do que muitas outras. A `number` variável só é acessível e atribuída na parte verdadeira da `if` cláusula. Se você tentar acessá-la em outro lugar, na cláusula ou após o bloco, o `else` `if` compilador emite um erro. Em segundo lugar, como você não está usando o operador `,` esse padrão `==` funciona quando um tipo sobrecarrega o `==` operador. Isso torna ideal verificar valores de referência nulos, adicionando o `not` padrão:

```
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`. O `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 `System.Collections.Generic.IList<T>` interface. Se isso acontecer, ele usará a `ICollection<T>.Count` propriedade nessa lista para encontrar o índice intermediário. O padrão de declaração não corresponderá a um valor, independentemente do tipo de tempo de `null` compilação da variável. O código abaixo protege contra, além de proteger `null` contra um tipo que não implementa `IList`.

```
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 de tempo de runtime específico.

Comparar valores discretos

Você também pode testar uma variável para encontrar uma combinação 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:

```
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 trata qualquer condição de erro em que o valor não corresponder a um dos valores `enum` definidos. Se você omitir esse arm de opção, o compilador avisará que você não lidou com todos os valores de entrada possíveis. Em tempo de operação, a expressão lançará uma exceção se o objeto que está sendo examinado não corresponder a `switch` nenhum dos braços do `comu switch`. Você pode usar constantes numéricas em vez de um conjunto de valores de `enum`. Você também pode usar essa técnica semelhante para valores de cadeia de caracteres constantes que representam os comandos:

```

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 enum. Você usaria esse cenário se seu aplicativo respondesse a comandos de texto em vez de um formato de dados regular. Em todos esses exemplos, o *padrão de descarte* garante que você manipulará cada entrada. O compilador ajuda você a garantir que cada valor de entrada possível seja tratado.

Padrões relacionais

Você pode usar *padrões relacionais* 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:

```

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 padrão lógico conjuntivo para verificar se `and` ambos os padrões relacionais são semelhantes. Você também pode usar um padrão disjuntivo `or` para verificar se qualquer um dos padrões corresponde. Os dois padrões relacionais estão entre parênteses, que você pode usar em torno de qualquer padrão para maior clareza. Os dois últimos comutadores lidam com os casos para o ponto de fusão e o ponto de ress猿ao. Sem esses dois braços, o compilador avisa que sua lógica n猿o abrange 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 n猿o lida com cada valor de entrada. O compilador também emite um aviso se um arm de opção já tiver sido manipulado por um arm de comutador anterior. Isso lhe d谩 liberdade para refactor e reordenar expressões switch. Outra maneira de escrever a mesma expressão pode ser:

```

string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        < 212 => "liquid",
        212 => "liquid / gas transition",
        _ => "gas",
    };

```

A lição principal sobre isso e qualquer outra refa o ou reordena o é que o compilador valida que voc  abordou todas as entradas.

Várias entradas

Todos os padrões que você viu até agora foram verificando uma entrada. Você pode escrever padrões que examinam várias propriedades de um objeto . Considere o seguinte `Order` registro:

```
public record Order(int Items, decimal Cost);
```

O tipo de registro posicional anterior declara dois membros em posições explícitas. Aparecendo primeiro é `Items` o , em seguida, o do `Cost` pedido. Para obter mais informações, consulte [Registros](#).

O código a seguir examina o número de itens e o valor de um pedido para calcular um preço com desconto:

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        (Items: > 10, Cost: > 1000.00m) => 0.10m,
        (Items: > 5, Cost: > 500.00m) => 0.05m,
        Order { 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 em `null` relação a e o final corresponde a qualquer outro valor. Se o tipo definir um método adequado, você poderá omitir os nomes de propriedade do padrão e usar `Order` a [Deconstruct](#) desconstrução para examinar as propriedades:

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        (> 10, > 1000.00m) => 0.10m,
        (> 5, > 50.00m) => 0.05m,
        Order { 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.

Este artigo forneceu um tour dos tipos de código que você pode escrever com correspondência de padrões em C#. Os artigos a seguir mostram mais exemplos de uso de padrões em cenários e o vocabulário completo dos padrões disponíveis para uso.

Confira também

- [Exploração: use a correspondência de padrões para criar seu comportamento de classe para um código melhor](#)
- [Tutorial: Usar a correspondência de padrões para criar algoritmos orientados por tipo e orientados a dados](#)
- [Referência: correspondência de padrões](#)

Descartes – conceitos básicos do C#

21/01/2022 • 9 minutes to read

A partir do C# 7,0, o C# dá suporte a Descartes, que são variáveis de espaço reservado que são intencionalmente não usadas no código do aplicativo. Os descartes são equivalentes a variáveis não atribuídas; Eles não têm um valor. Um descarte se comunica com a intenção do compilador e outros que lêem seu código: você pretende 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 `out` parâmetro para um método ou o destino de uma expressão de correspondência de padrões.

Como há apenas uma única variável de descarte, essa variável talvez nem mesmo seja um armazenamento alocado. Os descartes podem reduzir as alocações de memória. Os descartes tornam a intenção do seu código claro. Eles aprimoraram sua legibilidade e facilidade de 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 descartados. `area` é uma variável previamente declarada definida para o terceiro componente retornado por `GetCityInformation` :

```
(_, _, area) = city.GetCityInformation(cityName);
```

A partir do C# 9,0, você pode usar os 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 gera o erro de compilador CS0301, "o nome '`_`' não existe no contexto atual". Esse erro ocorre porque o `_` não recebe um valor e pode nem mesmo ser atribuído a um local de armazenamento. Se fosse uma variável real, você não poderia descartar mais de um valor, como fazia o exemplo anterior.

Desconstrução de objeto e de tupla

Os descartes são úteis para trabalhar com tuplas quando o código do aplicativo usa alguns elementos de tupla, mas ignora outros. Por exemplo, o método a seguir `QueryCityDataForYears` retorna uma tupla com o nome de uma cidade, sua área, um ano, a população da cidade para esse ano, um segundo ano e a população da cidade para esse 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.

```

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 pode usar os 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.

```

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`. Cada expressão, incluindo `null`, sempre corresponde ao padrão de descarte.

O exemplo a seguir define um `ProvidesFormatInfo` método que usa uma `switch` expressão para determinar se um objeto fornece uma `IFormatProvider` implementação e testa se o objeto é `null`. Ele também usa o padrão de descarte para manipular objetos não nulos de qualquer outro tipo.

```
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 `out` parâmetros

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 `out` argumentos ao chamar qualquer método com um `out` parâmetro.

O 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.

```
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 União nulo](#) para gerar um [System.ArgumentNullException](#) quando o argumento é `null`. O código não precisa do resultado da atribuição e, portanto, é Descartado. A expressão força uma verificação nula. O descarte esclarece sua intenção: o resultado da atribuição não é necessário ou usado.

```
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, pois ela está prestes a ser concluída. Isso torna sua intenção clara: você deseja descartar o `Task` e ignorar todos os erros gerados dessa operação assíncrona.

```
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:

```
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");
```

NOTE

Se você executar qualquer 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 accidental do valor da variável `_` no escopo atribuindo a ela o valor do descarte pretendido. Por exemplo:

```
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:

```
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:

```
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

- [Desconstruindo tuplas e outros tipos](#)
- [is operador](#)
- [switch expressão](#)

Desconstruindo tuplas e outros tipos

21/01/2022 • 11 minutes to read

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 em uma base elemento a elemento é complicado, como mostra o exemplo a seguir. O método retorna uma tupla de três e cada um de seus `QueryCityData` elementos é atribuído a uma variável em uma operação separada.

```
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 campo e propriedade de um objeto pode ser igualmente complicado: você deve atribuir um valor de campo ou propriedade a uma variável em uma base membro a membro.

No C# 7.0 e posterior, você pode recuperar vários elementos de uma tupla ou recuperar vários campos, propriedades e valores computados de um objeto em uma única operação *de desconstrução*. Para desconstruir uma tupla, atribua seus elementos 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:

```
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 as três tuplas retornadas pelo `QueryCityData` método .

```
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 tipo ao desconstruir as três tuplas retornadas pelo `QueryCityData` método .

```
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.

```
public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York City");

    // Do something with the data.
}
```

Isso é complicado e não é recomendado.

- Por fim, você pode desconstruir a tupla em variáveis que já foram declaradas.

```
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.

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

Não é possível especificar um tipo específico fora dos parênteses, mesmo que cada campo na tupla tenha o mesmo tipo. Isso gera o erro do compilador CS8136, o formulário "Desconstrução 'var (...)'" não permite um tipo

específico para 'var'.".

Você deve atribuir cada elemento da tupla a uma variável. Se você omitir elementos, o compilador gerará o erro CS8132, "Não é possível desconstruir uma tupla de elementos 'x' em variáveis 'y'."

Elementos de tupla com descartes

Geralmente, ao desconstruir uma tupla, você está interessado nos valores de apenas alguns elementos.

Começando com o C# 7.0, você pode aproveitar o suporte do C# para *descartes*, que são variáveis somente gravação cujos valores você opta por 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 retorna uma tupla de seis com o nome de uma cidade, sua área, um ano, a população da cidade para esse ano, um segundo ano e a população da cidade para esse `QueryCityDataForYears` 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.

```
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, 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 integrado para desconstruir tipos que não são tuplas diferentes dos tipos `record` 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:

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

Em seguida, você pode desconstruir uma instância da classe `Person` chamada `p` com uma atribuição como o código a seguir:

```
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, meio e sobrenome.
- Um nome, um sobrenome, um nome de cidade e um nome de estado.

```

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 `Deconstruct` métodos com o mesmo número de parâmetros são ambíguos. Você deve ter cuidado para definir `Deconstruct` métodos com números diferentes de parâmetros ou "arity". `Deconstruct` Métodos com o mesmo número de parâmetros não podem ser diferenciados 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 `_` e 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.

```
// 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 indica a acessibilidade da propriedade `access` como um todo. Caso contrário, a acessibilidade dos acessadores get e set é indicada pelas variáveis `getAccess` e `setAccess`.

```
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 `Deconstruct` o método como uma conveniência. Por exemplo, o `System.Collections.Generic.KeyValuePair< TKey, TValue >` tipo fornece essa funcionalidade. Quando você está iterando em um elemento, cada `System.Collections.Generic.Dictionary< TKey, TValue >` elemento é um e pode ser `KeyValuePair< TKey, TValue >` desconstruído. Considere o seguinte exemplo:

```

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 `Deconstruct` um método a tipos de sistema que não têm um. Considere o seguinte método de extensão:

```

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 `Nullable<T>` todos os tipos sejam desconstruídos em uma tupla de `(bool hasValue, T value)`. O exemplo a seguir mostra o código que usa esse método de extensão:

```
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 }
```

record Tipos

Quando você declara [um](#) tipo de registro usando dois ou mais parâmetros posicionais, o compilador cria um método com um parâmetro para cada parâmetro `Deconstruct` `out` posicional na `record` declaração. Para obter mais informações, [consulte Sintaxe posicional](#) para definição de propriedade e Comportamento do [desconstrutor em registros derivados](#).

Confira também

- [Descartes](#)
- [Tipos de tupla](#)

Exceções e manipulação de exceções

21/01/2022 • 2 minutes to read

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. O tratamento de exceções usa as palavras-chave `try`, `catch` e `finally`, e para tentar ações que podem não ser bem-sucedidas, para lidar com falhas quando você decide que é razoável fazer isso e limpar os recursos `try`, `catch`, `finally` posteriormente. As exceções podem ser geradas pelo CLR (Common Language Runtime), pelo .NET ou por 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 uma exceção é lançada, o CLR desenrola a pilha, procurando um método com um bloco para o tipo de exceção específico e executará o primeiro bloco que `catch` 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.

```
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 capturar uma exceção, a menos que você possa lidar com ela 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 `finally` um bloco é executado independentemente de se uma exceção for lançada. Use um bloco `finally` para liberar recursos, por exemplo, para fechar todos os fluxos ou arquivos que foram abertos no bloco `try`.
- Exceções gerenciadas no .NET são implementadas sobre o mecanismo de tratamento de exceção estruturado 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#](#)
- [Jogar](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [Exceções](#)

Usar exceções

21/01/2022 • 3 minutes to read

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 lançadas pelo runtime 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:

```
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:

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

Se a instrução que lança uma exceção não estiver dentro de um bloco ou se o bloco que a inclui não tiver nenhum bloco correspondente, o runtime verificará o método de chamada para uma instrução e `try` `try` `catch` `try` `catch` blocos. 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 que pode manipular a exceção é executada; todas as instruções a seguir, mesmo que sejam `catch` `catch` compatíveis, são ignoradas. Ordenar blocos `catch` do mais específico (ou mais derivado) para o menos específico. Por exemplo:

```

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 (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
    }
}

```

Antes de o bloco `catch` ser executado, o runtime verifica se há blocos `finally`. Os blocos permitem que o programador limpe qualquer estado ambíguo que possa ser deixado de um bloco anulado ou libere recursos externos (como alças de gráficos, conexões de banco de dados ou fluxos de arquivos) sem esperar que o coletor de lixo no `try` runtime finalize os objetos. Por exemplo:

```

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(0xFF);
    }
    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 lançar uma exceção, o código no segundo bloco que tenta reabrir o arquivo falhará se não for chamado e o arquivo `WriteByte()` `try` permanecerá `file.Close()` 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#)

21/01/2022 • 4 minutes to read

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 o código que é executado se uma exceção é lançada ou não no bloco, como a liberação de recursos alocados no `try` bloco. 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`.

```
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.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
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 como o filtro de exceção, a menos que você saiba como lidar com todas as exceções que podem ser lançadas no bloco ou tenha incluído uma instrução throw no final do [Exception](#) `try` `catch` bloco.

Vários `catch` blocos com classes 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 especificar uma classe de exceção correspondente, um bloco que não tem nenhum tipo será selecionado, se um estiver `catch` `catch` presente na instrução. É importante posicionar blocos com as classes de exceção mais específicas `catch` (ou seja, as mais derivadas) primeiro.

Capturar exceções quando as seguintes condições são 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.

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "Parameter index is out of range.", e);
    }
}
```

- Você deseja tratar parcialmente uma exceção antes de passá-la para obter mais tratamento. No exemplo a seguir, um bloco é usado para adicionar uma entrada a um log de erros antes `catch` de relrowing da exceção.

```
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 booliana 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:

```

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "Parameter index cannot be greater than the array size.", e);
    }
}

```

Um filtro de exceção que sempre `false` retorna pode ser usado para examinar todas as exceções, mas não processá-las. Um uso típico é registrar exceções:

```

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 `LogException` método sempre retorna , nenhuma cláusula usando esse filtro de `false` `catch` 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 `finally` bloco sempre é executado, se uma exceção é lançada ou um bloco que corresponde ao tipo de `catch` exceção é 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. Para obter mais informações, consulte a [instrução using](#).

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 não puder abrir o arquivo, o alcance de arquivo ainda terá o valor e o `try null` bloco não tentará `finally` abri-lo. Em vez

disso, se o arquivo for aberto com êxito no `try` bloco, `finally` o bloco fechará o arquivo aberto.

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
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

21/01/2022 • 3 minutes to read

As exceções são usadas para indicar que ocorreu um erro durante a execução do programa. Os objetos de exceção que descrevem um erro são criados e, em seguida, *lançados* com a `throw` palavra-chave. 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:

```
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. Nos casos em que um estado de objeto não permite uma operação, lance uma instância do `InvalidOperationException` ou um objeto com base em uma derivação dessa classe. O código a seguir é um exemplo de um método que gera um `InvalidOperationException` objeto:

```
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`:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index is out of range", nameof(index), ex);
    }
}
```

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 do 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`.

Métodos públicos e protegidos lançam exceções sempre que não podem concluir suas funções pretendidas. A classe de exceção gerada é a exceção mais específica disponível que atende às condições de 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 comum. Use exceções para relatar e manipular condições de erro.
- Exceções não devem ser retornadas como um valor de retorno ou parâmetro em vez de serem geradas.
- Não jogue [System.Exception](#), [System.SystemException](#), [System.NullReferenceException](#) ou [System.IndexOutOfRangeException](#) intencionalmente do seu próprio código-fonte.
- Não crie exceções que possam ser geradas no modo de depuração, mas não no modo de liberação. 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 quatro construtores: um construtor sem parâmetros, um que define a propriedade de mensagem e um que define as propriedades [Message](#) e [InnerException](#). O quarto construtor é usado para serializar a exceção. Novas classes de exceção devem ser serializáveis. Por exemplo:

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }

    public InvalidDepartmentException(string message) : base(message) { }

    public InvalidDepartmentException(string message, Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
                                         System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

Adicione novas propriedades à classe [Exception](#) quando os dados que eles 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

21/01/2022 • 2 minutes to read

Algumas exceções são geradas automaticamente pelo tempo de execução do .NET quando operações básicas falham. Essas exceções e suas condições de erro são listadas na tabela a seguir.

EXCEÇÃO	DESCRIÇÃO
ArithmeticeException	Uma classe base para exceções que ocorrem durante operações aritméticas, tais como DivideByZeroException e OverflowException .
ArrayTypeMismatchException	Gerado quando uma matriz não pode armazenar um 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	Gerado quando uma conversão explícita de um tipo base para uma interface ou para um tipo derivado falha em tempo de execução.
NullReferenceException	Gerado quando é feita uma tentativa de fazer referência a um objeto cujo valor é nulo .
OutOfMemoryException	Lançada quando uma tentativa de alocar memória usando o operador new falha. Essa exceção indica que a memória disponível para o Common Language Runtime foi 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

- [try-catch](#)
- [Experimente-finalmente](#)
- [try – catch-finally](#)

Nomes de identificadores

21/01/2022 • 2 minutes to read

Um **identificador** é o nome que você atribui a um tipo (classe, interface, struct, delegado ou enumerado), membro, variável ou namespace. Os identificadores válidos devem seguir estas regras:

- Os identificadores devem começar com uma letra ou `_`.
- 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 `@` não 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, consulte o [tópico de identificadores na especificação da linguagem C#](#).

Convenções de nomenclatura

Além das regras, há muitas [convenções de nomenclatura](#) de identificador usadas em todas as APIs do .net. Por convenção, os programas C# usam `PascalCase` para nomes de tipo, namespaces e todos os membros públicos. Além disso, as seguintes convenções são comuns:

- 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 não sinalizadores e um substantivo plural para sinalizadores.
- Os identificadores não devem conter dois caracteres consecutivos `_`. Esses nomes são reservados para identificadores gerados pelo compilador.

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 C#](#)
- [Referência do C#](#)
- [Classes](#)
- [Tipos de estrutura](#)
- [Namespaces](#)
- [Interfaces](#)
- [Representantes](#)

Convenções de codificação em C#

21/01/2022 • 11 minutes to read

As convenções de codificação atendem às seguintes finalidades:

- Criam uma aparência consistente para o código, para que os leitores possam se concentrar no conteúdo e não no layout.
- Permitem que os leitores entendam o código com mais rapidez, fazendo suposições com base na experiência anterior.
- Facilitam a cópia, a alteração e a manutenção do código.
- Demonstram as práticas recomendadas do C#.

IMPORTANT

As diretrizes neste artigo são usadas pela Microsoft para desenvolver exemplos e documentação. Eles foram adotados com as [diretrizes de Runtime do .NET](#), [Estilo de Codificação em C#](#). Você pode usá-los ou adaptá-los às suas necessidades.

Os objetivos principais são consistência e capacidade de leitura em seu projeto, equipe, organização ou código-fonte da empresa.

Convenções de nomenclatura

Há várias convenções de nome entre as convenções a considerar ao escrever código C#.

Nos exemplos a seguir, qualquer uma das diretrizes referentes aos elementos marcados também é aplicável ao trabalhar com elementos e , que devem ser visíveis para `public` `protected` `protected internal` chamadores externos.

Pascal Case

Use o pascal casing ("PascalCasing") ao nomear `class` um `record` , ou `struct` .

```
public class DataService
{
}
```

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

```
public struct ValueCoordinate
{
}
```

Ao nomear um `interface` , use o uso de uso de pascal, além de prefixar o nome com um `I` . Isso indica claramente aos consumidores que é um `interface` .

```
public interface IWorkerQueue
{
}
```

Ao nomear membros de tipos, como campos, propriedades, eventos, métodos e funções `public` locais, use pascal casing.

```
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 casing para parâmetros, pois eles são as propriedades públicas do registro.

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

Para obter mais informações sobre registros posicionais, consulte [Sintaxe posicional para definição de propriedade](#).

Caso de camel

Use a caixa de corpo de camel ("camelCasing") ao nomear `private` ou `internal` campos e prefixe-os com `_`.

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

TIP

Ao editar o código C# que segue essas convenções de nomenclatura em um IDE que dá suporte à conclusão da instrução, a digitação mostrará todos os membros com `_` escopo de objeto.

Ao trabalhar com `static` campos que são ou , use o `private` `internal` `s_` prefixo e para thread static use `t_`.

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

Ao escrever parâmetros de método, use a caixa de texto camel.

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

Para obter mais informações sobre convenções de nomen entre C#, consulte [Estilo de codificação em C#](#).

Convenções de nomen adiconais

- Exemplos que não incluem o [uso de diretivas](#), usam qualificações de namespace. Se você sabe que um namespace é importado por padrão em um projeto, não precisa qualificar totalmente os nomes desse namespace. Nomes qualificados podem ser interrompidos após um ponto (.) se forem muito longos para uma única linha, conforme mostrado no exemplo a seguir.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- Você não precisa alterar os nomes dos objetos que foram criados usando as ferramentas Visual Studio designer para fazer com que eles se ajustem a outras diretrizes.

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 (reco 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 devem recuar automaticamente, recue-as uma 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.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Convenções de comentário

- 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.

```
// The following declaration creates a query. It does not run  
// the query.
```

- Não crie blocos formatados de asteriscos em torno de comentários.
 - Verifique se todos os membros públicos têm os comentários XML necessários fornecendo descrições apropriadas sobre seu comportamento.

Diretrizes de linguagem

As seções a seguir descrevem práticas que a equipe de C# segue para preparar exemplos e amostras do código.

Tipos de 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.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- Para anexar cadeias de caracteres em loops, especialmente quando você estiver trabalhando com grandes quantidades de texto, use um `StringBuilder` objeto .

Variáveis locais de tipo implícito

- Use a [digitação implícita](#) para variáveis locais quando o tipo da variável for óbvio do lado direito da atribuição ou quando o tipo exato não for importante.

```
var var1 = "This is clearly a string.";  
var var2 = 27;
```

- Não use var quando o tipo não estiver aparente do lado direito da atribuição. Não suponha que o tipo seja claro em um nome de método. Um tipo de variável será considerado claro se for um new operador ou uma cast explícita.

```
int var3 = Convert.ToInt32(Console.ReadLine());  
int var4 = ExampleClass.ResultSoFar();
```

- Não confie no nome da variável para especificar o tipo da variável. Ele pode não estar correto. No exemplo a seguir, o nome da `inputInt` variável é enganoso. É uma cadeia de caracteres.

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Evite o uso de `var` em vez de `dynamic`. Use `dynamic` quando quiser inferência de tipo de tempo de execução. Para obter mais informações, consulte [Usando o tipo dinâmico \(Guia de Programação em C#\)](#).
 - Use a digitação implícita para determinar o tipo da variável de loop em `for` loops.

O exemplo a seguir usa digitação implícita em uma instrução `for`.

- Não use a digitação implícita para determinar o tipo da variável de loop em `foreach` loops.

O exemplo a seguir usa a digitação explícita em uma `foreach` instrução.

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

NOTE

Tenha cuidado para não alterar accidentalmente um tipo de um elemento da coleção iterável. Por exemplo, é fácil alternar de `System.Linq.IQueryable` para `System.Collections.IEnumerable` em uma `foreach` instrução , o que altera a execução de uma consulta.

Tipos de dados não assinados

Em geral, use `int` em vez de tipos sem assinatura. O uso de `int` é comum em todo o C#, e é mais fácil interagir com outras bibliotecas ao usar `int`.

Matrizes

Use a sintaxe concisa ao inicializar matrizes na linha da declaração. No exemplo a seguir, observe que você não pode usar em `var` vez de `string[]`.

```
string[ ] vowels1 = { "a", "e", "i", "o", "u" };
```

Se você usar a insta instação explícita, poderá usar `var`.

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

Se você especificar um tamanho de matriz, será necessário inicializar os elementos um de cada vez.

```
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Delegados

Use `Func<>` e `Action<>` em vez de definir tipos delegados. Em uma classe, defina o método delegado.

```
public static Action<string> ActionExample1 = x => Console.WriteLine($"x is: {x}");

public static Action<string, string> ActionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x => Convert.ToInt32(x);

public static Func<int, int, int> FuncExample2 = (x, y) => x + y;
```

Chame o método usando a assinatura definida pelo `Func<>` delegado `Action<>` ou .

```
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.

```
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 chame-a. A declaração a seguir mostra a sintaxe condensada.

```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

A declaração a seguir usa a sintaxe completa.

```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

try - catch Instruções using e no tratamento de exceção

- Use uma instrução `try-catch` para a maioria da manipulação de exceções.

```

static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        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, `try` - `finally` a instrução chama `Dispose` apenas no bloco `finally`.

```

Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}

```

Você pode fazer a mesma coisa com uma `using` instrução.

```

using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset2 = font2.GdiCharSet;
}

```

No C# 8 e versões posteriores, use a nova `using` sintaxe que não exige chaves:

```

using Font font3 = new Font("Arial", 10.0f);
byte charset3 = font3.GdiCharSet;

```

Operadores `&&` e `||`

Para evitar exceções e aumentar o desempenho ignorando comparações desnecessárias, use `em vez de` e `em vez de` quando executar comparações, conforme mostrado `&&` & no exemplo a seguir.

```

Console.WriteLine("Enter a dividend:");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("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 `if` instrução causará um erro de tempo de run-time. Mas o `&&` operador de curto-circuito quando a primeira expressão é false. Ou seja, ele não avalia a segunda expressão. O `&` operador avaliará ambos, resultando em um erro em tempo de run time `divisor` quando for 0.

new Operador

- Use uma das formas concisas de instância de objeto, conforme mostrado nas declarações a seguir. O segundo exemplo mostra a sintaxe disponível a partir do C# 9.

```
var instance1 = new ExampleClass();
```

```
ExampleClass instance2 = new();
```

As declarações anteriores são equivalentes à declaração a seguir.

```
ExampleClass instance2 = new ExampleClass();
```

- Use inicializadores de objeto para simplificar a criação de objetos, conforme mostrado no exemplo a seguir.

```
var instance3 = 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.

```
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Manipulação de eventos

Se você estiver definindo um manipulador de eventos que não precisa remover posteriormente, use uma expressão lambda.

```

public Form2()
{
    this.Click += (s, e) =>
    {
        MessageBox.Show(
            ((MouseEventArgs)e).Location.ToString());
    };
}

```

A expressão lambda reduz a definição tradicional a seguir.

```

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 qualificar 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.

```

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.

```

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.

```

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.

```
var seattleCustomers = from customer in customers
    where customer.City == "Seattle"
    select customer.Name;
```

- Alinhe cláusulas de consulta `from` sob a cláusula `where`, conforme mostrado nos exemplos anteriores.
- Use cláusulas antes de outras cláusulas de consulta para garantir que as cláusulas de consulta posteriores operem no conjunto de dados reduzido `where` e filtrado.

```
var seattleCustomers2 = from customer in customers
    where customer.City == "Seattle"
    orderby customer.Name
    select customer;
```

- Use várias `from` cláusulas em vez de uma `join` cláusula para acessar 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.

```
var scoreQuery = from student in students
    from score in student.Scores
    where score > 90
    select new { Last = student.LastName, score };
```

Segurança

Siga as diretrizes em [Diretrizes de codificação segura](#).

Consulte também

- [Diretrizes de codificação de runtime do .NET](#)
- [Convenções de codificação do Visual Basic](#)
- [Diretrizes de codificação segura](#)

Como exibir argumentos de linha de comando

21/01/2022 • 2 minutes to read

Os argumentos fornecidos a um executável na linha de comando podem ser acessados 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"

NOTE

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 para um aplicativo de linha de comando. A saída mostrada é para a primeira entrada da tabela acima.

```
using System;

class CommandLine
{
    static void Main(string[] args)
    {
        // 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]
    */
}
```

Explorar programação orientada a objeto com classes e objetos

21/01/2022 • 9 minutes to read

Neste tutorial, você criará um aplicativo de console e verá os recursos básicos orientados a objeto que fazem parte da linguagem C#.

Pré-requisitos

O tutorial espera que você tenha uma máquina configurada para desenvolvimento local. em Windows, Linux ou macOS, você pode usar a CLI do .net para criar, compilar e executar aplicativos. em Windows, você pode usar Visual Studio 2019. Para obter instruções de instalação, consulte [configurar seu ambiente local](#).

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:

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            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 objeto organiza o código criando 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 como *BankAccount.cs*. Adicione o seguinte código ao seu arquivo *BankAccount.cs*:

```
using System;

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 dentro do `{` e `}` que segue a declaração de classe define o estado e o comportamento da classe. Há cinco **Membros da BankAccount classe**. As três primeiras 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 um novo objeto do `BankAccount` tipo significa definir um **Construtor*** que atribui esses valores. Um *_Construtor** é um membro que tem o mesmo nome que a classe. Ele é usado para inicializar objetos desse tipo de classe. Adicione o seguinte construtor ao `BankAccount` tipo. Coloque o seguinte código acima da declaração de `MakeDeposit`:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Os construtores são chamados quando você cria um objeto usando `new`. Substitua a linha `Console.WriteLine("Hello World!");` em *Program.cs* pelo código a seguir (substitua `<name>` pelo seu nome):

```
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 criou até agora. se você estiver usando Visual Studio, selecione **iniciar sem depurar** no menu **depurar** . Se você estiver usando uma linha de comando, digite `dotnet run` o diretório em que você criou o 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 `BankAccount` classe. Coloque a seguinte linha de código após a chave de abertura `{` no início da `BankAccount` classe:

```
private static int accountNumberSeed = 1234567890;
```

Este 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-os após a linha que diz `this.Balance = initialBalance` :

```
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. Isso apresenta 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. Ao calcular o saldo do histórico de todas as transações, quando for necessário, 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. 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:

```

using System;

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)
        {
            this.Amount = amount;
            this.Date = date;
            this.Notes = note;
        }
    }
}

```

Agora, vamos adicionar um `List<T>` de `Transaction` objetos à classe `BankAccount`. Adicione a seguinte declaração após o Construtor em seu arquivo `BankAccount.cs`:

```
private List<Transaction> allTransactions = new List<Transaction>();
```

A classe `List<T>` exige que você importe um namespace diferente. Adicione o seguinte no início de `BankAccount.cs`:

```
using System.Collections.Generic;
```

Agora, vamos calcular corretamente o `Balance`. O saldo atual pode ser encontrado somando-se os valores de todas as transações. Como o código é atualmente, você só pode obter o saldo inicial da conta, portanto, você precisará atualizar a `Balance` propriedade. Substitua a linha `public decimal Balance { get; }` em `BankAccount.cs` pelo código a seguir:

```

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 aplicarão as duas últimas regras: que o saldo inicial deve ser positivo, e que qualquer saque não pode criar um saldo negativo.

Isso introduz 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 `MakeDeposit` método lançará uma exceção se a quantidade do depósito não for maior que 0. O método lançará uma exceção se o valor de retirada não for maior que 0 ou se a aplicação da retirada `MakeWithdrawal` resulta em um saldo negativo. Adicione o seguinte código após a declaração da `allTransactions` lista:

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(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 `throw` instrução lança uma exceção. A execução do bloco atual é encerrada e o controle transferido para o bloco `catch` da primeira correspondência encontrado 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:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

`DateTime.Now` é uma propriedade que retorna a data e a hora atuais. Teste isso adicionando alguns depósitos e retiradas em seu método, seguindo o `Main` código que cria um novo `BankAccount`:

```
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 capturando condições de erro tentando criar uma conta com um saldo negativo. Adicione o seguinte código após o código anterior que você acabou de adicionar:

```
// 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 as `try` instruções e `catch` para marcar um bloco de código que pode lançar exceções e capturar os erros esperados. Você pode usar a mesma técnica para testar o código que lança uma exceção para um saldo negativo. Adicione o seguinte código ao final do `Main` método :

```
// 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` :

```
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();
}
```

Isso 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`:

```
Console.WriteLine(account.GetAccountHistory());
```

Execute seu programa para ver os resultados.

Próximas etapas

Se você ficou preso, poderá ver a origem deste tutorial [em nosso GitHub repo](#).

Você pode continuar com o tutorial [de programação orientada a objeto](#).

Você pode saber mais sobre esses conceitos nestes artigos:

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

Programação de Object-Oriented (C#)

21/01/2022 • 12 minutes to read

O C# é uma linguagem de programação orientada a objeto. Os quatro princípios básicos da programação orientada a objeto são:

- *Abstração* Modelando os atributos relevantes e as interações de entidades como classes para definir uma representação abstrata de um sistema.
- *Encapsulamento* Ocultar o estado interno e a funcionalidade de um objeto e permitir o acesso apenas 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 `BankAccount` classe forneceu uma abstração para o conceito de uma conta bancária. Você poderia modificar sua implementação sem afetar nenhum código que usava a `BankAccount` classe. As `BankAccount` classes e `Transaction` fornecem encapsulamento dos componentes necessários para descrever esses conceitos no código.

Neste tutorial, você estenderá esse aplicativo para fazer uso de *herança* e *polimorfismo* para adicionar novos recursos. Você também adicionará recursos à `BankAccount` classe, aproveitando as técnicas de *abstração* e de *encapsulamento* aprendidas no tutorial anterior.

Criar diferentes tipos de contas

Depois de criar esse programa, você receberá solicitações para adicionar recursos a ele. Ele funciona muito bem na situação em que há apenas um tipo de conta bancária. Ao longo do tempo, as necessidades mudam e os tipos de conta relacionados são solicitados:

- Uma conta de conquista de interesse que acumula interesse no final de cada mês.
- Uma linha de crédito que pode ter um saldo negativo, mas quando há um saldo, há um encargo de juros a cada mês.
- Uma conta de vale-presente pré-paga que começa com um único depósito e pode ser paga. Ele pode ser recarregado uma vez no início de cada mês.

Todas essas contas diferentes são semelhantes à `BankAccount` classe 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 `BankAccount` classe criada no tutorial anterior. Essas novas classes podem estender a `BankAccount` classe com o comportamento específico necessário para cada tipo:

```
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 `BankAccount` classe. Escreva as implementações para uma funcionalidade nova e diferente em cada uma das *classes derivadas*. Essas classes derivadas já têm todo o comportamento definido na `BankAccount` classe.

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. em [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 de suas classes derivadas são compiladas. 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 da classe base incluído na classe derivada. A inicialização apropriada normalmente ocorre sem nenhum código extra. A `BankAccount` classe declara um construtor público com a seguinte assinatura:

```
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 da classe base. O código a seguir mostra o construtor para o `InterestEarningAccount`:

```
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 da classe base. Você usa a `: base()` sintaxe 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 conquista de interesse:
 - Receberá um crédito de 2% do saldo de término do mês.
- Uma linha de crédito:
 - Pode ter um saldo negativo, mas não deve ser maior em valor absoluto do que o limite de crédito.
 - Incorrerá em um encargo de juros a cada mês em que o saldo do fim do mês não é 0.
 - Incorrerá em uma taxa em cada retirada que passa pelo limite de crédito.
- Uma conta de cartão de presente:
 - Pode ser recarregado com um valor especificado uma vez por mês, no último dia do mês.

Você pode ver que todos esses três tipos de conta têm uma ação que usa locais no final de cada mês. No entanto, cada tipo de conta executa tarefas diferentes. Você usa *polimorfismo* para implementar esse código.

Crie um único `virtual` método na `BankAccount` classe:

```
public virtual void PerformMonthEndTransactions() { }
```

O código anterior mostra como você usa a `virtual` palavra-chave para declarar um método na classe base para a qual uma classe derivada pode fornecer uma implementação diferente. Um `virtual` método é um método em que qualquer classe derivada pode optar por reimplementar. As classes derivadas usam a `override` palavra-chave para definir a nova implementação. Normalmente, você faz referência a isso como "substituindo a implementação da classe base". A `virtual` palavra-chave especifica que as classes derivadas podem substituir o comportamento. Você também pode declarar `abstract` métodos em que classes derivadas devem substituir o comportamento. A classe base não fornece uma implementação para um `abstract` método. Em seguida, você precisa definir a implementação para duas das novas classes que você criou. Comece com

```
InterestEarningAccount :
```

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        var interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Adicione o código a seguir ao `LineOfCreditAccount`. O código nega o saldo para computar um encargo de interesse positivo que é retirado da conta:

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        var interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

A `GiftCardAccount` classe 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:

```
private 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 `monthlyDeposit` valor para que os chamadores possam omitir um `0` para nenhum depósito mensal. Em seguida, substitua o `PerformMonthEndTransactions` método para adicionar o depósito mensal, se ele tiver sido definido como um valor diferente de zero no construtor:

```

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 `Main` método para testar essas alterações para o `GiftCardAccount` e o `InterestEarningAccount` :

```

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 o `LineOfCreditAccount` :

```

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:

```

Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be positive (Parameter 'amount')
at OOPProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String note) in
BankAccount.cs:line 42
at OOPProgramming.BankAccount..ctor(String name, Decimal initialBalance) in BankAccount.cs:line 31
at OOPProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance) in
LineOfCreditAccount.cs:line 9
at OOPProgramming.Program.Main(String[] args) in Program.cs:line 29

```

NOTE

A saída real inclui o caminho completo para a pasta com o projeto. Os nomes das pastas foram omitidos para fins de brevidade. Além disso, dependendo do seu formato de código, os números de linha podem ser um pouco diferentes.

Esse código falha porque o `BankAccount` supõe que o saldo inicial deve ser maior que 0. Outra suposição inclusa na `BankAccount` classe é que o saldo não pode ficar negativo. Em vez disso, qualquer retirada que redesenhe a conta é rejeitada. Ambas as suposições precisam ser alteradas. A linha de conta de crédito começa

em 0 e geralmente terá um saldo negativo. Além disso, se um cliente emprestando muito dinheiro, ele incorrerá em uma taxa. A transação é aceita, ela apenas custa mais. A primeira regra pode ser implementada adicionando um argumento opcional ao `BankAccount` Construtor que especifica o saldo mínimo. O padrão é `0`. A segunda regra requer um mecanismo que permita que as classes derivadas modifiquem o algoritmo padrão. De certa forma, a classe base "pergunta" o tipo derivado o que deve acontecer quando há um superrascunho. O comportamento padrão é rejeitar a transação lançando uma exceção.

Vamos começar adicionando um segundo construtor que inclui um `minimumBalance` parâmetro opcional. Esse novo construtor executa 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 que dois locais mudarão no futuro. Em vez disso, você pode usar *o encadeamento de construtor* para fazer com que um construtor chame outro. O código a seguir mostra os dois construtores e o novo campo adicional:

```
private readonly decimal minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name, initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    this.minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

O código anterior mostra duas novas técnicas. Primeiro, o `minimumBalance` campo é marcado como `readonly` . Isso significa que o valor não pode ser alterado depois que o objeto é construído. Depois que `BankAccount` um é criado, `minimumBalance` o não pode ser alterado. Em segundo lugar, o construtor que usa dois parâmetros usa `: this(name, initialBalance, 0) { }` como sua implementação. A `: this()` expressão 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 `MakeDeposit` chamará 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 seja aberta com um `0` saldo.

Agora que a classe tem um campo somente leitura para o saldo mínimo, a alteração final é alterar o código rígido `BankAccount 0` para no método `minimumBalance MakeWithdrawal` :

```
if (Balance - amount < minimumBalance)
```

Depois de estender a classe , você pode modificar o construtor para chamar o novo construtor base, conforme `BankAccount` mostrado no código a `LineOfCreditAccount` seguir:

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal creditLimit) : base(name,
initialBalance, -creditLimit)
{
```

Observe que o `LineOfCreditAccount` construtor altera o sinal do parâmetro para que ele corresponde ao significado do `creditLimit` `minimumBalance` parâmetro.

Regras de excesso diferentes

O último recurso a ser acrescentado permite que o escuse um valor por passar sobre o limite de crédito em vez `LineOfCreditAccount` de recusá-lo.

Uma técnica é definir uma função virtual na qual você implementa o comportamento necessário. A `BankAccount` classe refatora o `MakeWithdrawal` método em dois métodos. O novo método faz a ação especificada quando a retirada tem o saldo abaixo do mínimo. O método existente `MakeWithdrawal` tem o seguinte código:

```
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:

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    var overdraftTransaction = CheckWithdrawalLimit(Balance - amount < minimumBalance);
    var withdrawal = new Transaction(-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 somente de classes derivadas. Essa declaração impede que outros clientes chamam o método. Também é para `virtual` que as classes derivadas possam alterar o comportamento. O tipo de retorno é um `Transaction?`. A `?` anotação indica que o método pode retornar `null`. Adicione a seguinte implementação no `LineOfCreditAccount` para cobrar uma taxa quando o limite de retirada for excedido:

```
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

A substituição retorna uma transação de valor quando a conta é redesenhada. Se o cancelamento não passar do limite, o método retornará uma `null` transação. Isso indica que não há nenhum valor. Teste essas alterações adicionando o seguinte código `Main` ao método na classe `Program`:

```
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 ficou preso, poderá ver a origem deste tutorial [em nosso GitHub repo](#).

Este tutorial demonstrou muitas das técnicas usadas na Object-Oriented programação:

- Você usou *Abstração* quando definiu classes para cada um dos diferentes tipos de conta. Essas classes descrevem o comportamento desse tipo de conta.
- Você usou *Encapsulamento* ao manter muitos detalhes `private` em cada classe.
- Você usou *Herança* ao aproveitar a implementação já criada na `BankAccount` classe para salvar o código.
- Você usou o *Polimorfismo* quando criou métodos que as classes derivadas poderiam substituir para criar um `virtual` comportamento específico para esse tipo de conta.

Herança em C# e .NET

21/01/2022 • 26 minutes to read

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

Este tutorial pressupõe que você instalou o SDK do .NET. Visite a página de [downloads do .net](#) para baixá-lo. Você também precisa de um editor de código. Este tutorial usa o [Visual Studio Code](#), embora você possa usar qualquer editor de código que quiser.

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.

Você não precisa executar `dotnet restore` porque ele é executado implicitamente por todos os comandos que exigem a ocorrência de uma restauração, como,,, `dotnet new` `dotnet build` `dotnet run` `dotnet test` `dotnet publish` e `dotnet pack`. Para desabilitar a restauração implícita, use a `--no-restore` opção.

O `dotnet restore` comando ainda é útil em determinados cenários em que a restauração explícita faz sentido, como [compilações de integração contínua em Azure DevOps Services](#) ou em sistemas de compilação que precisam controlar explicitamente quando a restauração ocorre.

Para obter informações sobre como gerenciar feeds do NuGet, consulte a [dotnet restore](#) documentação.

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 `A.value` privado fica 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 devido ao seu nível de proteção".

```
using System;

public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return this.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.

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new B();
        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 faz o exemplo a seguir, gera o erro do compilador CS0506: "`<member>` não consegue substituir o membro herdado `<member>`, pois ele não está marcado como virtual, abstrato ou de substituição.

```
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 .

```
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, a tentativa de compilar o código como no exemplo a seguir produz o erro do compilador CS0527: "O tipo 'ValueType' na lista de interfaces não é uma interface". A mensagem de erro indica que, embora você possa definir as interfaces implementadas por um struct, não há suporte para a herança.

```
using System;

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:

```
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.

```

using System;
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 (var 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";
            }
            var 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`.

```
using System;

public class EmptyClass
{}

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new EmptyClass();
        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	<code>Object</code>
struct	<code>ValueType, Object</code>
enum	<code>Enum, ValueType, Object</code>
delegado	<code>MulticastDelegate, Delegate, Object</code>

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.

NOTE

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`.

```
using System;

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.

```
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 tipo, como um livro, uma revista, um jornal, um diário, um artigo, etc. Você também definirá uma `Book` classe derivada 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`. Você pode estender facilmente o exemplo para criar várias classes adicionais que derivam 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 instanciar uma classe marcada com a palavra-chave `abstract` por uma chamada direta para o construtor de classe, o compilador de C# gerará o erro CS0144, "Não é possível criar uma instância da classe abstrata ou interface". Se for feita uma tentativa de instanciar a classe por meio da reflexão, o método de reflexão lançará um `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 sealed <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:

```
using System;

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;

        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;
        this.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:

```

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 `published` privado como `true` quando ele é chamado, e atribui a data

passada para ele como um argumento para o campo `datePublished` privado. 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`.

Object	Publication
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)
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`.

```

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.");
            ulong nISBN = 0;
            if (! UInt64.TryParse(isbn, out nISBN))
                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)
    {
        Book book = obj as Book;
        if (book == null)
            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.

```

using System;

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 da de um 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.

```
using System;

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.

```

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`.

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (var shape in shapes) {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                $"perimeter, {Shape.GetPerimeter(shape)}");
            var rect = shape as Rectangle;
            if (rect != null) {
                Console.WriteLine($"    Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
                continue;
            }
            var sq = shape as Square;
            if (sq != null) {
                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 correspondência de padrões e os operadores `is` e `as`

21/01/2022 • 4 minutes to read

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 de correspondência de padrões `is` :

```

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 { }

class Program
{
    static void Main(string[] args)
    {
        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);
        // Output:
        // I am an animal.
        // SuperNova is not a Mammal
    }

    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");
        }
    }
}

```

O exemplo anterior demonstra alguns recursos da sintaxe de correspondência de padrões. A

`if (a is Mammal m)` instrução 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. Você não pode acessar `m` mais tarde no mesmo método. O exemplo anterior também mostra como usar o `as` operador para converter um objeto em um tipo especificado.

Você também pode usar a mesma sintaxe para testar se um [tipo de valor anulável](#) tem um valor, conforme mostrado no exemplo a seguir:

```

class Program
{
    static void Main(string[] args)
    {
        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(System.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(System.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 `is` padrões do exemplo anterior não são limitados aos tipos de valor anulável. 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 você usa o padrão de tipo em uma `switch` instrução em que a variável pode ser um dos muitos tipos diferentes.

Se você quiser testar se uma variável é um tipo específico, mas não atribuí-la a uma nova variável, você pode usar os `is` operadores e para tipos de `as` referência e tipos de valor anulável. 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:

```
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 { }

class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        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 != 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 != null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
```

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 a correspondência de padrões para criar algoritmos orientados por tipo e orientados a dados

21/01/2022 • 16 minutes to read

O C#7 introduziu recursos básicos de correspondência de padrões. Esses recursos são estendidos em C# 8 a C# 10 com novas expressões e padrões. 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

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 SDK do .NET 6](#).

Este tutorial pressuém que você esteja familiarizado com o C# e o .NET, incluindo 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:

```
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 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 é adequada para resolver:

- 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`.

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    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 `switch expressão` (não igual a uma `switch instrução`) que testa o padrão **de declaração**. 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, `null` o padrão constante detecta quando é passado para esse `null` método. O `null` padrão pode ser o último porque os outros padrões corresponderão apenas a um objeto não nulo do tipo correto.

Você pode testar esse código usando o seguinte código no `Program.cs`:

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            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 teste o que você 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 completos ganham um desconto de R\$ 1,00.

Essas regras podem ser implementadas usando [um padrão de propriedade](#) na mesma expressão switch. Um padrão de propriedade compara um valor da 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:

```
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:

```
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:

```
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:

```

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. Quando terminar, você terá um método muito parecido com o código a seguir:

```

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 se alimenta em um padrão constante. Essa técnica é mostrada no código a seguir:

```

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 `if` e `else` instruções para expressar isso usando o código a seguir:

```

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 o motivo do código. Em vez disso, você usará a correspondência de padrões para esse recurso, mas você o integrará com 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:

DIA	HORA	DIREÇÃO	PREMIUM
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:

```

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:

```

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:

```

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,
    };

```

Você adiciona um privado `enum` para converter cada intervalo de tempo em um valor discreto. Em seguida, o `GetTimeBand` método usa [padrões relacionais](#) `or` [padrões conjuntiva](#), ambos adicionados em C# 9.0. Um padrão relacional permite testar um valor numérico usando `<`, `>`, ou `<=` `>=`. O `or` padrão testa se uma expressão corresponde a um ou mais padrões. Você também pode usar um padrão para garantir que uma expressão corresponde a dois padrões distintos e um padrão para testar se uma `and` expressão não corresponde a um `not` 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:

```

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, 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:

```
(false, _, _) => 1.00m,
```

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:

```
(true, TimeBand.OVERNIGHT, _) => 0.75m,
(true, TimeBand.DAYTIME, _) => 1.5m,
```

O código deverá ser semelhante ao seguinte após essas duas alterações:

```

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:

```

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](#)
- [switch Expressão](#)

Como tratar uma exceção usando try/catch

21/01/2022 • 2 minutes to read

A finalidade de um bloco `try-catch` é capturar e manipular uma exceção gerada pelo código de trabalho.

Algumas exceções podem ser tratadas em um `catch` bloco e o problema é resolvido sem que a exceção seja relançada; no entanto, com mais frequência a única coisa que você pode fazer é verificar se a exceção apropriada foi lançada.

Exemplo

Neste exemplo, `IndexOutOfRangeException` não é a exceção mais apropriada: `ArgumentOutOfRangeException` faz mais sentido para o método porque o erro é causado pelo `index` argumento passado pelo chamador.

```
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`. A instrução `catch` é adicionada logo após para manipular `IndexOutOfRangeException`, se ocorrer. O bloco `catch` manipula o `IndexOutOfRangeException` e gera a exceção `ArgumentException`, que é mais adequada. Para fornecer ao chamador tantas informações quanto possível, considere especificar a exceção original como o `InnerException` da nova exceção. Como a `InnerException` propriedade é `somente leitura`, você deve atribuí-la no construtor da nova exceção.

Como executar código de limpeza usando finally

21/01/2022 • 2 minutes to read

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:

```
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.

```
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 dentro do bloco antes da chamada ou a chamada em si pode falhar, não há garantia de que o arquivo está aberto quando tentamos `try` `OpenWrite()` `OpenWrite()` fechar. O `finally` bloco adiciona uma verificação para garantir que o objeto não seja antes de você chamar o `FileStream` método `null` `Close`. Sem a verificação, o bloco pode lançar seu próprio , mas lançar exceções em blocos deve ser `null` `finally` evitado se for `NullReferenceException` `finally` 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 lançada antes que você possa fechar sua conexão, o uso do bloco será melhor do `finally` que aguardar a coleta de lixo.

Confira também

- [Instrução using](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

Novidades no C# 10

21/01/2022 • 7 minutes to read

O C# 10 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- [Structs de registro](#)
- [Melhorias de tipos de estrutura](#)
- [Manipuladores de cadeia de caracteres interpolados](#)
- [global using Directivas](#)
- [Declaração de namespace no escopo do arquivo](#)
- [Padrões de propriedade estendidos](#)
- [Melhorias em expressões lambda](#)
- [Permitir const cadeias de caracteres interpoladas](#)
- [Os tipos de registro podem selar ToString\(\)](#)
- [Atribuição definida aprimorada](#)
- [Permitir a atribuição e a declaração na mesma desconstrução](#)
- [Permitir AsyncMethodBuilder atributo em métodos](#)
- [Atributo CallerArgumentExpression](#)
- [#line Pragma aprimorado](#)

Recursos adicionais estão disponíveis no modo *de visualização*. Você é incentivado a experimentar esses recursos e fornecer comentários sobre eles. Eles podem mudar antes da versão final. Para usar esses recursos, você deve [definir <LangVersion> como Preview](#) em seu projeto. Leia sobre [atributos genéricos](#) mais adiante neste artigo.

Há suporte para o C# 10 no .NET 6. Para obter mais informações, consulte [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.

Structs de registro

Você pode declarar registros de tipo de valor usando `record struct` as declarações `readonly record struct` ou. Agora você pode esclarecer que um `record` é um tipo de referência com a declaração `record class`.

Melhorias 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 de instância ou propriedade em sua declaração. Para obter mais informações, consulte a [seção Construtores sem parâmetros](#) e inicializadores de campo do artigo [Tipos de estrutura](#).
- Um operand esquerdo da expressão `with` pode ser de qualquer tipo de estrutura ou um tipo anônimo (referência).

Manipulador de cadeia de caracteres interpolado

Você pode criar um tipo que cria a cadeia de caracteres resultante de uma [expressão de cadeia de caracteres interpolada](#). As bibliotecas do .NET usam esse recurso em muitas APIs. Você pode criar um seguindo [este](#)

tutorial.

Diretivas de uso global

Você pode adicionar o modificador a qualquer diretiva using para instruir o compilador que a diretiva se aplica a todos os arquivos de `global` origem na compilação. Normalmente, isso é todos os arquivos de origem em um projeto.

Declaração de namespace no escopo do arquivo

Você pode usar uma nova forma da declaração `namespace` para declarar que todas as declarações a seguir são membros do namespace declarado:

```
namespace MyNamespace;
```

Essa nova sintaxe economiza espaço horizontal e vertical para `namespace` declarações.

Padrões de propriedade estendidos

A partir do C# 10, você pode referenciar propriedades aninhadas ou campos dentro de um padrão de propriedade. Por exemplo, um padrão do formulário

```
{ Prop1.Prop2: pattern }
```

é válido no C# 10 e posterior e equivalente a

```
{ Prop1: { Prop2: pattern } }
```

válido no C# 8.0 e posterior.

Para obter mais informações, consulte a observação [proposta de proposta de recursos Padrões de propriedade estendida](#). Para obter mais informações sobre um padrão de propriedade, consulte a [seção Padrão de propriedade](#) do [artigo Padrões](#).

Melhorias na expressão lambda

O C# 10 inclui muitas melhorias em 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 grupo de métodos.
- Expressões lambda podem declarar um [tipo de retorno](#) quando o compilador não pode inferi-lo.
- [Atributos](#) podem ser aplicados a expressões lambda.

Esses recursos fazem expressões lambda mais semelhantes a métodos e funções locais. Eles facilitam o uso de expressões lambda sem declarar uma variável de um tipo delegado e funcionam de forma mais direta com as novas APIs ASP.NET Core Mínimas.

Cadeias de caracteres interpoladas constantes

No C# 10, as cadeias de caracteres podem ser inicializadas usando interpolação de cadeia de caracteres se todos os espaço reservados são cadeias de `const` caracteres constantes. A interpolação de cadeia de caracteres pode criar cadeias de caracteres constantes mais acessíveis à medida que você cria cadeias de caracteres constantes usadas em seu aplicativo. As expressões de espaço reservado não podem ser constantes numéricas porque

essas constantes são convertidas em cadeias de caracteres em tempo de run. A cultura atual pode afetar sua representação de cadeia de caracteres. Saiba mais na referência de linguagem em [const expressões](#).

Os tipos de registro podem selar `ToString`

No C# 10, você pode adicionar o `sealed` modificador ao substituir `ToString` em um tipo de registro. A vedação `ToString` do método impede que o compilador synthesizing um método para quaisquer tipos de registro `ToString` derivados. Um `sealed ToString` garante que todos os tipos de registro derivados usem `ToString` o método 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 podia atribuir todos os valores a variáveis existentes ou inicializar variáveis recém-declaradas:

```
// Initialization:  
(int x, int y) = point;  
  
// assignment:  
int x1 = 0;  
int y1 = 0;  
(x1, y1) = point;
```

O C# 10 remove essa restrição:

```
int x = 0;  
(x, int y) = point;
```

Atribuição definida aprimorada

Antes do C# 10, havia muitos cenários em que a atribuição definida e análise de estado nulo produzia avisos que eram falsos positivos. Essas comparações geralmente envolvem constantes booleanas, acessando uma variável somente nas instruções ou em uma instrução e expressões `true` `false` de `if` coalização nula. Esses exemplos geraram avisos em versões anteriores do C#, mas não no C# 10:

```
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 definida e análise de estado nulo são mais precisos.

Permitir atributo AsyncMethodBuilder em métodos

No C# 10 e posterior, você pode especificar um construtor de método assíncrono diferente para um único método, além de especificar o tipo de construtor de métodos para todos os métodos que retornam um determinado tipo de tarefa. Um construtor de método 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 [AsyncMethodBuilder](#) sobre no artigo sobre atributos lidos pelo compilador.

Diagnóstico de atributo CallerArgumentExpression

Você pode usar o [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 `false`, a mensagem de exceção conterá a representação de texto do argumento passado para `condition` :

```
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 #line aprimorado

O C# 10 dá suporte a um novo formato para `#line` o pragma. Você provavelmente não usará o novo formato, mas verá seus efeitos. Os aprimoramentos habilitam uma saída mais asseada em DSLs (linguagens específicas do domínio), como o Razor. O mecanismo Razor usa esses aprimoramentos para melhorar a experiência de depuração. Você verá que os depuradores podem realçar sua fonte razor com mais precisão. Para saber mais sobre a nova sintaxe, consulte o artigo sobre [diretivas de pré-processador](#) na referência de linguagem. Você também pode ler a [especificação de recurso](#) para exemplos baseados em Razor.

Atributos genéricos

IMPORTANT

Atributos genéricos é um recurso de visualização. Você deve definir `<LangVersion>` como `Preview` para habilitar esse recurso. Esse recurso pode mudar antes de sua versão final.

Você pode declarar uma [classe genérica](#) cuja classe base é [System.Attribute](#). Isso fornece uma sintaxe mais conveniente para atributos que exigem um [System.Type](#) parâmetro. Anteriormente, você precisaria criar um atributo que aceita um `Type` como seu parâmetro de construtor:

```
public class TypeAttribute : Attribute
{
    public TypeAttribute(Type t) => ParamType = t;

    public Type ParamType { get; }
}
```

E, para aplicar o atributo, use o `typeof` operador :

```
[TypeAttribute(typeof(string))]  
public string Method() => default;
```

Usando esse novo recurso, você pode criar um atributo genérico em vez disso:

```
public class GenericAttribute<T> : Attribute { }
```

Em seguida, especifique o parâmetro de tipo para usar o atributo :

```
[GenericAttribute<string>()]  
public string Method() => default;
```

Você pode aplicar um atributo genérico construído totalmente fechado. Em outras palavras, todos os parâmetros de tipo devem ser especificados. Por exemplo, o seguinte não é permitido:

```
public class GenericType<T>  
{  
    [GenericAttribute<T>()] // Not allowed! generic attributes must be fully closed types.  
    public string Method() => default;  
}
```

Os argumentos de tipo devem atender às mesmas restrições que o `typeof` operador . Tipos que exigem anotações de metadados não são permitidos. Os exemplos incluem o seguinte:

- `dynamic`
- `nint` , `nuint`
- `string?` (ou qualquer tipo de referência que anulável)
- `(int X, int Y)` (ou qualquer outro tipo de tupla usando a sintaxe de tupla C#).

Esses tipos não são representados diretamente nos metadados. Eles incluem anotações que descrevem o tipo.

Em todos os casos, você pode usar o tipo subjacente:

- `object` para `dynamic` .
- `IntPtr` em vez `nint` de ou `nuint` .
- `string` em vez de `string?` .
- `ValueTuple<int, int>` em vez de `(int X, int Y)` .

Novidades do C# 9.0

21/01/2022 • 19 minutes to read

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 interop](#)
 - [Inteiros de tamanho nativo](#)
 - [Ponteiros de função](#)
 - [Suprimir a emissão do sinalizador localsinit](#)
- [Ajustar e concluir recursos](#)
 - [Expressões de tipo `new` de destino](#)
 - [`static` funções anônimas](#)
 - [Expressões condicionais com tipo de destino](#)
 - [Tipos de retorno covariantes](#)
 - [Suporte `GetEnumerator` de extensão para `foreach` loops](#)
 - [Parâmetros `discard` de lambda](#)
 - [Atributos em funções locais](#)
- [Suporte para geradores de código](#)
 - [Inicializadores de módulo](#)
 - [Novos recursos para métodos parciais](#)

Há suporte para o C# 9.0 no .NET 5. Para obter mais informações, consulte [Versão da linguagem C#](#).

Você pode baixar o SDK mais recente do .NET na [página de downloads do .NET](#).

Tipos de registro

O C# 9.0 introduz tipos *de registro*. Use a `record` palavra-chave para definir um tipo de referência que fornece funcionalidades para encapsular dados. Você pode criar tipos de registro com propriedades imutáveis usando parâmetros posicionais ou sintaxe de propriedade padrão:

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

```
public record Person
{
    public string FirstName { get; init; } = default!;
    public string LastName { get; init; } = default!;
};
```

Você também pode criar tipos de registro com propriedades e campos mutáveis:

```
public record Person
{
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
}
```

Embora os registros possam ser mutáveis, eles são destinados principalmente para 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 estruturativa](#)
 - [Formatação integrado 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 são suportados por herança.
- Eles são menos eficientes para determinar a igualdade de valor. Para tipos de valor, `ValueType.Equals` o método usa reflexão para encontrar todos os campos. Para registros, o compilador gera o `Equals` método. Na prática, a implementação da igualdade de valor em 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:

```
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 implementada automaticamente somente por init pública para cada parâmetro posicional fornecido na declaração de registro. Uma [propriedade somente de inicialização](#) só pode ser definida no construtor ou usando um inicializador de propriedade.
- Um construtor primário cujos parâmetros corresponderem aos parâmetros posicionais na declaração de registro.
- Um `Deconstruct` método com um parâmetro para cada parâmetro `out` posicional fornecido na declaração de registro.

Para obter mais informações, [consulte Sintaxe posicional](#) no artigo de referência da linguagem C# sobre registros.

Imutabilidade

Um tipo de registro não é necessariamente imutável. Você pode declarar propriedades `set` com acessadores e campos que não são `readonly`. Mas embora os registros possam ser mutáveis, eles facilitam a criação de modelos de dados imutáveis. As propriedades que você cria usando 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. Ele pode impedir 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 modificando o estado do objeto.

Igualdade de valor

Igualdade de valor significa que duas variáveis de um tipo de registro serão 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 serão iguais se se referirem ao mesmo objeto.

O exemplo a seguir ilustra a igualdade de valor dos tipos de registro:

```
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, você pode substituir manualmente métodos e operadores de igualdade para obter igualdade de valor, mas desenvolver e testar esse código seria demorado e propenso `class` a erros. Ter essa funcionalidade interno impede bugs que resultam em esquecer de atualizar o código de substituição personalizado quando propriedades ou campos são adicionados ou alterados.

Para obter mais informações, [consulte Igualdade de valor](#) no artigo de referência da linguagem C# sobre registros.

Mutação não estruturativa

Se você precisar modificar propriedades imutáveis de uma instância de registro, poderá usar uma expressão para obter mutação não `with` *estruturativa*. Uma expressão faz uma nova instância de registro que é uma cópia de uma instância de registro existente, com propriedades e campos `with` especificados modificados. Use a [sintaxe do inicializador de objeto](#) para especificar os valores a serem alterados, conforme mostrado no exemplo a seguir:

```

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 [Mutação não estruturativa](#) no artigo de referência da linguagem C# sobre registros.

Formatação integrado para exibição

Os tipos de registro têm um método gerado pelo `ToString` compilador que exibe os nomes e valores de propriedades e campos públicos. O `ToString` método retorna uma cadeia de caracteres do seguinte formato:

```
<record type name> { <property name> = <value>, <property name> = <value>, ...}
```

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, é exibida em vez `System.String[]` dos valores de elemento de matriz reais:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

Para obter mais informações, [consulte Formatação nativa](#) no artigo referência da 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:

```

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 run-time deve ser igual. Os tipos das

variáveis que contêm podem ser diferentes. Isso é ilustrado no exemplo de código a seguir:

```
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` `False` retorna, embora ambos sejam `Person` variáveis de tipo. E `student == student2` `True` retorna, embora uma seja uma `Person` variável e outra seja uma `Student` variável.

Todas as propriedades públicas e os campos dos tipos derivados e base são incluídos na `ToString` saída, conforme mostrado no exemplo a seguir:

```
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 da linguagem C# sobre registros.

Setters somente init

Somente setters de inicialização fornecem sintaxe consistente para inicializar membros de um objeto. Os inicializadores de propriedade deixarão claro qual valor está definindo qual propriedade. A desvantagem é que essas propriedades devem ser settable. A partir do C# 9.0, você pode criar acessadores em vez de acessadores `init` `set` para propriedades e indexadores. Os chamadores podem usar a sintaxe do inicializador de propriedade para definir esses valores em expressões de criação, mas essas propriedades são `ReadOnly` quando a construção é concluída. Os setters somente init fornecem uma janela para alterar o estado. Essa janela fecha quando a fase de construção termina. A fase de construção termina com eficiência após toda a inicialização, incluindo inicializadores de propriedade e with-Expressions concluídos.

Você pode declarar `init` somente setters em qualquer tipo que escrever. Por exemplo, a seguinte estrutura define uma estrutura de observação do clima:

```

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 propriedade para definir os valores, enquanto ainda preservam a imutabilidade:

```

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:

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

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

Para obter mais informações, consulte [init \(referência C#\)](#).

Instruções de nível superior

As *instruções de nível superior* removem a cerimônia desnecessária de muitos aplicativos. Considere o canônico "Olá, Mundo!" Program

```

using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}

```

Há apenas uma linha de código que faz qualquer coisa. Com as instruções de nível superior, você pode substituir todo o texto clichê pela `using` diretiva e a única linha que faz o trabalho:

```
using System;

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

Se você quisesse um programa de uma linha, poderia remover a `using` diretiva e usar o nome do tipo totalmente qualificado:

```
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 `Main` método. De certa forma, você pode imaginar que um arquivo contém as instruções que normalmente estaria no `Main` método de uma `Program` classe.

Um dos usos mais comuns para esse recurso é a criação de materiais de ensino. Desenvolvedores de C# iniciantes podem escrever o "Olá, Mundo!" canônico em uma ou duas linhas de código. Nenhuma das cerimônias extras é necessária. No entanto, os desenvolvedores experientes também encontrarão muitos usos para esse recurso. As instruções de nível superior permitem uma experiência semelhante a um script para experimentação semelhante ao que os notebooks Jupyter fornecem. As instruções de nível superior são ótimas para pequenos programas de console e utilitários. [Azure Functions](#) é um caso de uso ideal para instruções de nível superior.

O mais importante é que as instruções de nível superior não limitam o escopo ou a complexidade do aplicativo. Essas instruções podem acessar ou usar qualquer classe .NET. Eles também não limitam o uso de argumentos de linha de comando ou valores de retorno. Instruções de nível superior podem acessar uma matriz de cadeias de caracteres denominadas `args`. Se as instruções de nível superior retornarem um valor inteiro, esse valor se tornará o código de retorno de 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 do C#.

Melhorias na correspondência de padrões

O C# 9 inclui novas melhorias de correspondência de padrões:

- **Padrões de tipo** correspondem a uma variável é um tipo
- **Padrões entre parênteses** impõem ou enfatizam a precedência de combinações de padrões
- `and` **Padrões de conjuntiva** exigem os dois padrões para corresponder
- `or` **Padrões de disjunctive** exigem qualquer padrão para corresponder
- `not` **Padrões negados** exigem que um padrão não corresponda
- Os **padrões relacionais** exigem que a entrada seja menor que, maior que, menor ou igual ou maior ou igual a uma determinada constante.

Esses padrões enriquecem a sintaxe para padrões. Considere estes exemplos:

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

Com parênteses opcionais para tornar claro que `and` tem precedência maior do que `or`:

```
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:

```
if (e is not null)
{
    // ...
}
```

Qualquer um desses padrões pode ser usado em qualquer contexto em que os padrões são permitidos: `is` expressões `switch` de padrão, expressões, padrões aninhados e o padrão do rótulo de uma `switch` instrução `case`.

Para obter mais informações, consulte [padrões \(referência C#\)](#).

Para obter mais informações, consulte as seções [padrões relacionais](#) e [padrões lógicos](#) do artigo [padrões](#).

Desempenho e interoperabilidade

Três novos recursos melhoraram o suporte para a interoperabilidade nativa e bibliotecas de nível baixo que exigem alto desempenho: inteiros de tamanho nativo, ponteiros de função e omissão do `localsinit` sinalizador.

Inteiros de tamanho nativo `nint` e `nuint`, são tipos inteiros. Eles são expressos pelos tipos subjacentes `System.IntPtr` e `System.UIntPtr`. O compilador superfiçies de conversões e operações adicionais para esses tipos como `ints` nativas. Os 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 `ReadOnly` em tempo de execução. Você pode usar valores constantes para `nint` no intervalo `[int.MinValue .. int.MaxValue]`. Você pode usar valores constantes para `nuint` no intervalo `[uint.MinValue .. uint.MaxValue]`. O compilador executa o dobramento constante para todos os operadores unários e binários usando os `System.Int32` `System.UInt32` tipos e. Se o resultado não couber em 32 bits, a operação será executada em tempo de execução e não será considerada uma constante. Os inteiros de tamanho nativo podem aumentar o desempenho em cenários em que a matemática de inteiros é usada extensivamente e precisa ter o desempenho mais rápido possível. Para obter mais informações, consulte [nint](#) e [nuint](#) tipos

Ponteiros de função fornecem uma sintaxe fácil para acessar os opcodes de IL `ldftn` e `calli`. Você pode declarar ponteiros de função usando a nova `delegate*` sintaxe. Um `delegate*` tipo é um tipo de ponteiro. Invocar o `delegate*` tipo usa `calli`, em contraste com um delegado que usa `callvirt` no `Invoke()` método. Sintaticamente, as invocações são idênticas. Invocação de ponteiro de função usa a `managed` Convenção de chamada. Você adiciona a `unmanaged` palavra-chave após a `delegate*` sintaxe para declarar que deseja a `unmanaged` Convenção de chamada. Outras convenções de chamada podem ser especificadas usando atributos na `delegate*` declaração. Para obter mais informações, consulte [tipos de ponteiro e código sem segurança](#).

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

Esses recursos podem melhorar o desempenho em alguns cenários. Eles devem ser usados somente após um

parâmetro de comparação cuidadoso antes e depois da adoção. O código que envolve inteiros de tamanho nativo deve ser testado em várias plataformas de destino com tamanhos de inteiro diferentes. Os outros recursos exigem código não seguro.

Recursos de ajuste e término

Muitos dos outros recursos ajudam a escrever código com mais eficiência. No C# 9.0, você pode omitir o tipo em uma `new` expressão quando o tipo do objeto criado já é conhecido. O uso mais comum está em declarações de campo:

```
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 `ForecastFor()` método com a seguinte assinatura:

```
public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)
```

Você pode chamá-lo da seguinte maneira:

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

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

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

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

Um recurso semelhante melhora a resolução de tipo de destino de [expressões condicionais](#). Com essa alteração, as duas expressões não precisam ter uma conversão implícita de uma para a outra, mas podem ter conversões implícitas em um tipo de destino. Você provavelmente não observará essa alteração. O que você observará é que algumas expressões condicionais que anteriormente exigiam ou não compilam agora apenas funcionam.

A partir do C# 9.0, você pode adicionar o modificador a expressões `static lambda` ou [métodos anônimos](#). Expressões lambda estáticas são análogas às funções locais: um método lambda ou anônimo estático não pode capturar variáveis locais ou `static` estado de instância. O `static` modificador impede a captura acidental de outras variáveis.

Os 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 suportam clonagem virtual ou métodos de fábrica.

Além disso, o `foreach` loop reconhecerá e usará um método de extensão que, de `GetEnumerator` outra forma, satisfaz o `foreach` padrão. Essa alteração significa que é consistente com outras construções baseadas em padrão, como o padrão `foreach` assíncrono e a desconstrução baseada em padrão. Na prática, essa alteração significa que você pode adicionar `foreach` suporte a qualquer tipo. Você deve limitar seu uso a ao enumerar um objeto que 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. Use 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 que [podem ser anuladas](#) a funções locais.

Supporte para geradores de código

Dois recursos finais são suportados por geradores de código C#. Os geradores de código C# são um componente que você pode escrever semelhante a um analisador roslyn ou correção de código. A diferença é que os geradores de código analisam o código e escrevem novos arquivos de código-fonte como parte do processo de compilação. Um gerador de código típico pesquisa o código para atributos ou outras convenções.

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

Os dois recursos adicionados para geradores de código são extensões para *sintaxe de método parcial_*, e *inicializadores de módulo_* **. Primeiro, as alterações em métodos parciais. Antes do C# 9.0, os métodos parciais são, mas não podem especificar um modificador de acesso, ter um retorno e não `private` `void` podem ter `out` parâmetros. Essas restrições significam 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 parciais tenham uma implementação. Geradores de código podem fornecer essa implementação. Para evitar introduzir uma alteração da quebra, o compilador considera qualquer método parcial sem um modificador de acesso para seguir as regras antigas. Se o método parcial incluir o `private` modificador de acesso, as novas regras regem esse método parcial. Para obter mais informações, consulte [método parcial \(Referência de C#\)](#).

O segundo novo recurso para geradores de código são *os inicializadores de módulo*. Os inicializadores de módulo são métodos que têm o `ModuleInitializerAttribute` atributo anexado a eles. Esses métodos serão chamados pelo runtime antes de qualquer outro acesso a campo ou invocação de método em todo o módulo. Um método de inicializador de módulo:

- Deve ser estático
- Deve ser sem parâmetros
- Deve retornar void
- Não deve ser um método genérico
- Não deve estar contido em uma classe genérica
- Deve estar acessível no módulo que o contém

Esse último marcador significa efetivamente que o método e sua classe que o contém devem ser internos ou públicos. O método não pode ser uma função local. Para obter mais informações, consulte [ModuleInitializer atributo](#).

Novidades no C# 8.0

21/01/2022 • 17 minutes to read

O 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](#)
- [Descartável assíncrono](#)
- [Índices e intervalos](#)
- [Atribuição de União nula](#)
- [Tipos construídos não gerenciados](#)
- [Stackalloc em expressões aninhadas](#)
- [Aprimoramento de cadeias de caracteres idênticas interpoladas](#)

O C# 8,0 tem suporte no .NET Core 3. x e .net Standard 2,1. Para obter mais informações, consulte [controle de versão da linguagem C#](#).

O restante deste artigo descreve rapidamente esses recursos. Quando houver artigos detalhados disponíveis, forneceremos links para esses tutoriais e visões gerais. Você pode explorar esses recursos em seu ambiente usando a ferramenta global `dotnet try`:

1. Instale a ferramenta global [dotnet-try](#).
2. Clone o repositório [dotnet/try-samples](#).
3. Defina o diretório atual do subdiretório `csharp8` para o repositório `try-samples`.
4. Execute `dotnet try`.

Membros somente leitura

Você pode aplicar o `readonly` modificador a membros de um struct. Indica que o membro não modifica o estado. É mais granular do que aplicar o modificador `readonly` a uma declaração `struct`. Considere o seguinte struct mutável:

```

public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"{{X}}, {{Y}} is {Distance} from the origin";
}

```

Como a maioria das structs, o `ToString()` método não modifica o estado. É possível indicar isso adicionando o modificador `readonly` à declaração de `ToString()`:

```

public readonly override string ToString() =>
    $"{{X}}, {{Y}} is {Distance} from the origin";

```

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

```

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

```

O compilador avisa quando há a necessidade de criar uma cópia de defesa. A `Distance` propriedade não muda de estado, portanto você pode corrigir esse aviso adicionando o `readonly` modificador à declaração:

```

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

```

Observe que o `readonly` modificador é necessário em uma propriedade somente leitura. O compilador não pressupõe `get` que os acessadores não modifiquem o estado; você deve declarar `readonly` explicitamente. As propriedades implementadas automaticamente são uma exceção; o compilador tratará todos os getters autoimplementados como `readonly`, portanto, aqui não há necessidade de adicionar o `readonly` modificador `X` às `Y` Propriedades e.

O compilador impõe a regra que `readonly` os membros não modificam o estado. O método a seguir não será compilado, a menos que você remova o `readonly` modificador:

```

public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}

```

Esse recurso permite que você especifique sua intenção de design para que o compilador possa impô-la e faça otimizações com base nessa intenção.

Para obter mais informações, consulte a seção [readonly membros da instância](#) do artigo [tipos de estrutura](#).

Métodos de interface padrão

Agora é possível adicionar membros a interfaces e fornecer uma implementação para esses membros. Esse recurso de linguagem permite que os autores de API adicionem métodos a uma interface em versões posteriores sem interromper a fonte ou a compatibilidade binária com implementações existentes dessa interface. As implementações existentes *herdam* a implementação padrão. Esse recurso também permite que o

C# interopere com APIs que direcionam o Android ou o Swift, que dão suporte a recursos semelhantes. Os métodos de interface padrão também habilitam cenários semelhantes a um recurso de linguagem de "características".

Os métodos de interface padrão afetam muitos cenários e elementos de linguagem. Nossa primeiro tutorial aborda [como atualizar uma interface com implementações padrão](#).

Mais padrões em mais partes

Com a **correspondência de padrões**, você recebe ferramentas para fornecer funcionalidades dependentes da forma em tipos de dados relacionados, mas diferentes. O C# 7,0 introduziu a sintaxe para padrões de tipo e padrões constantes usando a `is` expressão e a `switch` instrução. Esses recursos representaram os primeiros passos em direção ao suporte a paradigmas de programação, em que os dados e a funcionalidade vivem separados. À medida que o setor se aproxima mais de microsserviços e de outras arquiteturas baseadas em nuvem, outras ferramentas de linguagem de tornam necessárias.

O C# 8,0 expande esse vocabulário, para que você possa usar mais expressões de padrão em mais partes do seu código. Considere esses recursos quando seus dados e funcionalidades estiverem separados. Considere a correspondência de padrões quando seus algoritmos dependerem de um fato diferente do tipo de runtime de um objeto. Essas técnicas fornecem outra maneira de expressar designs.

Além dos novos padrões em novas partes, o C# 8,0 adiciona **padrões recursivos**. Padrões recursivos são padrões que podem conter outros padrões.

Expressões switch

Geralmente, uma `switch` instrução produz um valor em cada um de seus `case` blocos. As **expressões switch** permitem que você use a sintaxe de expressão mais concisa. Há menos palavras-chave `case` e `break` repetidas, e menos chaves. Por exemplo, considere a enumeração a seguir que lista as cores do arco-íris:

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

Se seu aplicativo definiu um tipo `RGBColor` que é construído a partir dos componentes `R`, `G` e `B`, você poderia converter um valor `Rainbow` em seus valores RGB usando o método a seguir que contém uma expressão de opção:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red      => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange   => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow  => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green   => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue    => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo  => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet  => new RGBColor(0x94, 0x00, 0xD3),
        _                  => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
```

Há vários aprimoramentos de sintaxe aqui:

- A variável vem antes da palavra-chave `switch`. A ordem diferente facilita distinguir visualmente a expressão `switch` da instrução `switch`.
- Os elementos `case` e `:` são substituídos por `=>`. É mais conciso e intuitivo.
- O caso `default` é substituído por um descarte `_`.
- Os corpos são expressões, não instruções.

Compare isso com o código equivalente usando a instrução clássica `switch`:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

Para obter mais informações, consulte [switch expression](#).

Padrões da propriedade

O padrão da propriedade permite que você compare as propriedades do objeto examinado. Considere um site de comércio eletrônico que deve calcular o imposto da venda com base no endereço do comprador. Essa computação não é uma responsabilidade principal de uma `Address` classe. Ele mudará ao longo do tempo, provavelmente com mais frequência do que as alterações de formato de endereço. O valor do imposto depende da propriedade `State` do endereço. O método a seguir usa o padrão de propriedade para calcular o imposto da venda de acordo com o endereço e o preço:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
}
```

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

Para obter mais informações, consulte a seção [padrão de propriedade](#) do artigo [padrões](#).

Padrões de tupla

Alguns algoritmos dependem de várias entradas. Padrões de tupla permitem que você alterne com base em

vários valores, expressadas como uma [tupla](#). O código a seguir mostra uma expressão de comutador para o jogo *pedra, papel, tesoura*:

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
}
```

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

Padrões posicionais

Alguns tipos incluem um método `Deconstruct` que desconstrói suas propriedades em variáveis discretas.

Quando um método `Deconstruct` é acessível, você pode usar **padrões posicionais** para inspecionar as propriedades do objeto e usar essas propriedades para um padrão. Considere a seguinte classe `Point`, que inclui um método `Deconstruct` para criar variáveis discretas para `x` e `y`:

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

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

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

Além disso, considere a seguinte enumeração que representa várias posições de um quadrante:

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

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

```

static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};

```

O padrão de discard na opção anterior encontra a correspondência quando `x` ou `y` é 0, mas não ambos. Uma expressão `switch` deve produzir um valor ou lançar uma exceção. Se não houver correspondência em nenhum dos casos, a expressão `switch` gerará uma exceção. O compilador gerará um aviso para você se você não cobrir todos os casos possíveis em sua expressão de comutador.

Explore técnicas de correspondência de padrões neste [tutorial avançado sobre correspondência de padrões](#). Para obter mais informações sobre um padrão posicional, consulte a seção [padrão posicional](#) do artigo de [padrões](#).

Declarações using

Uma **declaração using** é uma declaração de variável precedida pela palavra-chave `using`. Ele informa ao compilador que a variável que está sendo declarada deve ser descartada ao final do escopo delimitador. Por exemplo, considere o seguinte código que grava um arquivo de texto:

```

static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}

```

No exemplo anterior, o arquivo é descartado quando a chave de fechamento do método é atingida. Esse é o final do escopo no qual `file` é declarado. O código anterior equivale ao código a seguir que usa as [instruções using](#) clássicas:

```

static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}

```

No exemplo anterior, o arquivo é descartado quando a chave de fechamento associada à instrução `using` é atingida.

Em ambos os casos, o compilador gera a chamada para `Dispose()`. O compilador gerará um erro se a expressão na `using` instrução não for descartável.

Funções locais estáticas

Agora você pode adicionar o `static` modificador a **funções locais** para garantir que a função local não Capture (referencie) nenhuma variável do escopo delimitador. Isso gera `CS8421`, "Uma função local estática não pode conter uma referência a <variable>".

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

```

int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}

```

O código a seguir contém uma função local estática. Ela pode ser estática porque não acesse as variáveis no escopo delimitador:

```

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

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

```

Estruturas ref descartáveis

Um `struct` declarado com o `ref` modificador não pode implementar nenhuma interface e, portanto, não pode implementar `IDisposable`. Portanto, para permitir que uma `ref struct` seja descartada, ela deve ter um método `void Dispose()` acessível. Esse recurso também se aplica a `readonly ref struct` declarações.

Tipos de referência anuláveis

Dentro de um contexto de anotação anulável, qualquer variável de um tipo de referência é considerado um **tipo de referência não anulável**. Se você quiser indicar que uma variável pode ser nula, acrescente o nome do tipo com o `?` para declarar a variável como um **tipo de referência anulável**.

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

Os tipos de referência anuláveis não são verificados para garantir que não tenham sido atribuídos ou inicializados como nulos. No entanto, o compilador usa a análise de fluxo para garantir que qualquer variável de um tipo de referência anulável passe por verificação com relação a um nulo antes de ser acessada ou atribuída a um tipo de referência não anulável.

Saiba mais sobre o recurso na visão geral sobre [Tipos de referência anuláveis](#). Experimente-o em um novo aplicativo neste [tutorial de tipos de referência anuláveis](#). Saiba mais sobre as etapas para migrar uma base de código existente para fazer uso de tipos de referência anuláveis no artigo sobre como [atualizar para tipos de referência anuláveis](#).

Fluxos assíncronos

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

1. Ele é declarado com o modificador `async`.
2. Ela retorna um `IAsyncEnumerable<T>`.
3. O método contém instruções `yield return` para retornar elementos sucessivos no fluxo assíncrono.

O consumo de um fluxo assíncrono exige que você adicione a palavra-chave `await` antes da palavra-chave `foreach` quando você enumera os elementos do fluxo. A adição da palavra-chave `await` exige o método que enumera o fluxo assíncrono seja declarado com o modificador `async` e retorne um tipo permitido para um método `async`. Normalmente, isso significa retornar um `Task` ou `Task<TResult>`. Também pode ser `ValueTask` ou `ValueTask<TResult>`. Um método pode consumir e produzir um fluxo assíncrono, o que significa que retornaria um `IAsyncEnumerable<T>`. O código a seguir gera uma sequência de 0 a 19, esperando 100 ms entre a geração de cada número:

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

Você enumeraria a sequência usando a instrução `await foreach`:

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

Experimente você mesmo os fluxos assíncronos em nosso tutorial sobre como [criar e consumir fluxos assíncronos](#). Por padrão, os elementos de fluxo são processados no contexto capturado. Se você quiser desabilitar a captura do contexto, use o [TaskAsyncEnumerableExtensions.ConfigureAwait](#) método de extensão. Para obter mais informações sobre contextos de sincronização e como capturar o contexto atual, consulte o artigo sobre como [consumir o padrão assíncrono baseado em tarefa](#).

Descartável assíncrono

A partir do C# 8.0, a linguagem dá suporte a tipos descartáveis assíncronos que implementam a [System.IAsyncDisposable](#) interface. Você usa a `await using` instrução para trabalhar com um objeto descartável de forma assíncrona. Para obter mais informações, consulte o artigo [implementar um método DisposeAsync](#).

Índices e intervalos

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

Esse suporte a idioma depende de dois novos tipos e de dois novos operadores:

- [System.Index](#) representa um índice em uma sequência.
- O índice do operador end `^`, que especifica que um índice é relativo ao final da sequência.
- [System.Range](#) representa um subintervalo de uma sequência.
- O operador Range `...`, que especifica o início e o término de um intervalo como seus operandos.

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

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

Vamos ver alguns exemplos. Considere a matriz a seguir, anotada com seu índice do íncio e do final:

```
var words = new string[]
{
    "The",           // index from start   index from end
    "quick",         // 0                  ^9
    "brown",         // 1                  ^8
    "fox",           // 2                  ^7
    "jumped",        // 3                  ^6
    "over",          // 4                  ^5
    "the",           // 5                  ^4
    "lazy",          // 6                  ^3
    "dog"            // 7                  ^2
};                  // 8 (or words.Length) ^0
```

Recupere a última palavra com o índice `^1`:

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

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.

```
var quickBrownFox = words[1..4];
```

O código a seguir cria um subintervalo com "lazy" e "dog". Ele inclui `words[2]` e `words[1]`. O índice final `words[0]` não está incluído:

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

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

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

Você também pode declarar intervalos como variáveis:

```
Range phrase = 1..4;
```

Em seguida, o intervalo pode ser usado dentro dos caracteres `[` e `]`:

```
var text = words[phrase];
```

Não apenas as matrizes dão suporte a índices e intervalos. Você também pode usar índices e intervalos com [cadeia de caracteres](#), [Span<T>](#) ou [ReadOnlySpan<T>](#). Para obter mais informações, consulte [suporte de tipo para índices e intervalos](#).

Você pode explorar mais sobre índices e intervalos do tutorial sobre [índices e intervalos](#).

Atribuição de União nula

O C# 8,0 apresenta o operador de atribuição de União nula `??=`. Você pode usar o `??=` operador para atribuir o valor do seu operando à direita para seu operando à esquerda somente se o operando esquerdo for avaliado como `null`.

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

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

Para obter mais informações, consulte [?? e ?? = artigo operadores](#).

Tipos construídos não planejados

No C# 7.3 e anteriores, um tipo construído (um tipo que inclui pelo menos um argumento de tipo) não pode ser um tipo [nãomanaged](#). A partir do C# 8.0, um tipo de valor construído não será managedo se contiver apenas campos de tipos nãomanagedos.

Por exemplo, dada a seguinte definição do tipo `Coords<T>` genérico:

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

O `Coords<int>` tipo é um tipo não planejado no C# 8.0 e posterior. Assim como para qualquer tipo não planejado, você pode criar um ponteiro para uma variável desse tipo ou alocar um bloco de memória na pilha para instâncias desse tipo:

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

Para obter mais informações, consulte [Tipos não planejados](#).

Stackalloc em expressões aninhadas

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

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

Aprimoramento de cadeias de caracteres textual interpoladas

A ordem dos tokens e em cadeias de caracteres textual interpoladas pode ser qualquer: e são cadeias de `$` `@` `$@"..."` `@$..."` caracteres textual interpoladas válidas. Em versões anteriores do C#, `$` o token deve aparecer antes do `@` token.

Saiba mais sobre alterações interruptivas no compilador C#

21/01/2022 • 2 minutes to read

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 em seu repositório de GitHub:

- [Alterações recentes no Roslyn no após o .NET 6 até o .NET 7](#)
- [Alterações recentes em Roslyn em C# 10.0/.NET 6](#)
- [Alterações recentes no Roslyn após o .NET 5](#)
- [Alterações recentes na versão 16.8 do VS2019 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#

21/01/2022 • 14 minutes to read

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](#).

IMPORTANT

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 1.0

quando voltar e olhar, o C# versão 1.0, lançado com Visual Studio .net 2002, ficou muito parecido com o Java. Como parte de suas [metas de design declaradas para ECMA](#), ela buscava ser uma "linguagem simples, moderna, de uso geral e orientada a objeto". No momento, parece que o Java alcançou 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](#)

C# versão 1.2

a versão 1.2 do C# foi 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 2.0

Neste momento, as coisas começam a ficar interessantes. 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. Depois de se acostumarem com a ideia da linguagem OO, os desenvolvedores começaram a sofrer com vários pontos problemáticos graves. E os procuraram de maneira significativa.

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. O uso de genéricos é melhor do que criar um `ListInt` tipo derivado de `ArrayList` ou de conversão 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 3.0

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 implementadas automaticamente](#)
- [Tipos anônimos](#)
- [Expressões de consulta](#)
- [Expressões lambda](#)
- [Árvores de expressão](#)
- [Métodos de extensão](#)

- [Variáveis locais de tipo implícito](#)
- [Métodos parciais](#)
- [Inicializadores de objeto e coleção](#)

Numa retrospectiva, muitos desses recursos parecerem inevitáveis e inseparáveis. Todos eles se encaixam estrategicamente. É pensado que o recurso de Killer da versão C# era a expressão de consulta, também conhecida como LINQ (consulta de Language-Integrated).

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 fez parecer que essa lista de inteiros se tornasse muito mais inteligente.

Levou algum tempo para que as pessoas entendessem e integrassem o conceito, mas isso aconteceu gradualmente. E agora, anos mais tarde, o código é muito mais conciso, simples e funcional.

C# versão 4.0

o C# versão 4,0, lançado com Visual Studio 2010, teria tido um tempo difícil de viver até o status inovador da versão 3,0. Com a versão 3,0, o C# tirou verdadeiramente a linguagem da sombra do Java e a colocou em proeminência. A linguagem foi rapidamente se tornando elegante.

A próxima versão introduziu alguns novos recursos interessantes:

- [Associação dinâmica](#)
- [Argumentos opcionais/nomeados](#)
- [Genérico covariante e contravariante](#)
- [Tipos de interoperabilidade inseridos](#)

Tipos de interoperabilidade inseridos facilitoum o problema 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 5.0

o C# versão 5,0, lançado com Visual Studio 2012, foi 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 de principais recursos:

- [Membros assíncronos](#)
- [Atributos de informações do chamador](#)

Consulte Também

- [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 6.0

Nas versões 3.0 e 5.0, o C# recebeu alguns novos recursos importantes em uma linguagem orientada a objeto. com a versão 6.0, lançada com o Visual Studio 2015, ela desapareceria de um recurso de killer dominante e, em vez disso, liberava muitos recursos menores que tornaram a programação 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# eliminou 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 7.0

a versão 7.0 do C# foi lançada com o Visual Studio 2017. Esta versão tem algumas coisas interessantes e evolutivas na mesma direção que o C# 6.0, mas sem o compilador como um serviço. Aqui estão alguns dos novos recursos:

- Variáveis out
- [Tuplas e desconstrução](#)
- [Correspondência de padrões](#)
- Funções locais
- Membros aptos para expressão expandidos
- [Locais e retornos de ref](#)

Outros recursos incluíam:

- Descartes
- Literais binários e os separadores de dígito
- Expressões throw

Todas essas funcionalidades oferecem novos recursos interessantes 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.

Mas o C# está sendo colocado para um uso cada vez mais amplo. 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 7,1

C# começou a liberar *lançamentos de ponto* com c# 7,1. Esta versão adicionou o elemento de configuração [seleção de versão de idioma](#), três novos recursos de linguagem e novo comportamento de compilador.

Os novos recursos de linguagem nesta versão são:

- `async` `Main` método
 - O ponto de entrada para um aplicativo pode ter o modificador `async`.
- `default` expressões literais
 - 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 tem duas opções `-refout` e `-refonly` essa geração de assembly de referência de controle

C# versão 7,2

O C# 7,2 adicionou vários recursos de linguagem pequena:

- Inicializadores em `stackalloc` matrizes.
- Use `fixed` instruções com qualquer tipo que dê suporte a um padrão.
- Acessar campos fixos sem fixação.
- Reatribua `ref` variáveis locais.
- Declare `readonly struct` tipos, para indicar que uma struct é imutável e deve ser passada como um `in` parâmetro para seus métodos de membro.
- Adicione o `in` modificador nos parâmetros para especificar que um argumento é passado por referência, mas não modificado pelo método chamado.
- Use o `ref readonly` modificador no método retornado para indicar que um método retorna seu valor por referência, mas não permite gravações nesse objeto.
- Declare `ref struct` tipos, para indicar que um tipo struct acessa a memória gerenciada diretamente e sempre deve ser alocado na pilha.
- Use 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.
- **`private protected` modificador de acesso**
 - 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 7,3

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 de compilador também foram adicionadas nesta 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 OSS (software livre) de assemblies.
- `-pathmap` para fornecer um mapeamento para diretórios de origem.

C# versão 8,0

O c# 8,0 é a primeira versão principal do C# que se destina especificamente ao .NET Core. Alguns recursos dependem de novos recursos do CLR, outros em tipos de biblioteca adicionados somente no .NET Core. O 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 União nula
- Tipos construídos não gerenciados
- Stackalloc em expressões aninhadas
- Aprimoramento de cadeias de caracteres idênticas interpoladas

Os membros da interface padrão exigem aprimoramentos no CLR. Esses recursos foram adicionados ao CLR para .NET Core 3,0. Intervalos e índices e fluxos assíncronos exigem novos tipos nas bibliotecas do .NET Core 3,0. Os tipos de referência anuláveis, enquanto implementados no compilador, são muito mais úteis quando as bibliotecas são anotadas para fornecer informações semânticas sobre o estado nulo dos argumentos e valores de retorno. Essas anotações estão sendo adicionadas nas bibliotecas do .NET Core.

C# versão 9

O C# 9 foi lançado com o .NET 5. É a versão de idioma padrão para qualquer assembly que tenha como destino a versão do .NET 5. Ele contém os seguintes recursos novos e aprimorados:

- Grava
- Setters somente init
- Instruções de nível superior
- Melhorias na correspondência de padrões
- Desempenho e interoperabilidade
 - Inteiros de tamanho nativo
 - Ponteiros de função
 - Suprimir emissão do sinalizador localsinit
- Recursos de ajuste e término
 - Expressões com tipo de destino new
 - static funções anônimas
 - Expressões condicionais de tipo de destino
 - Tipos de retorno covariantes
 - GetEnumerator Suporte de extensão para foreach loops
 - Parâmetros discard de lambda
 - Atributos em funções locais
- Suporte para geradores de código
 - Inicializadores de módulo
 - Novos recursos para métodos parciais

O C# 9 continua três dos temas de versões anteriores: removendo a cerimônia, separando dados de algoritmos e fornecendo mais padrões em mais lugares.

As [instruções de nível superior](#) significam que seu programa principal é mais simples de ler. Há menos necessidade de cerimônia: um namespace, uma Program classe e static void Main() todos são desnecessários.

A introdução de records fornece uma sintaxe concisa para tipos de referência que seguem a semântica de valor para igualdade. Você usará esses tipos para definir contêineres de dados que normalmente definem o comportamento mínimo. Setters somente de inicialização fornecem a capacidade de mutação não destrutiva (with expressões) em registros. O 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 [correspondência de padrões](#) foram expandidos de várias maneiras. Os tipos numéricos agora dão suporte a *padrões de intervalo*. Os padrões podem ser combinados `and` usando `or` padrões, e `not`. Os parênteses podem ser adicionados para esclarecer padrões mais complexos.

Outro conjunto de recursos dá suporte à computação de alto desempenho em C#:

- Os `nint` `nuint` tipos e modelam os tipos inteiros de tamanho nativo na CPU de destino.
- Os [ponteiros de função](#) fornecem funcionalidade como um delegado, ao mesmo tempo que evita as alocações necessárias para criar um objeto delegado.
- A `localsinit` instrução pode ser omitida para salvar instruções.

Outro conjunto de melhorias dá suporte a cenários em que os *geradores de código* adicionam funcionalidade:

- [Inicializadores de módulo](#) são métodos que o tempo de execução chama quando um assembly é carregado.
- Os [métodos parciais](#) dão suporte a novos modificadores `accessibly` e tipos de retorno não nulos. Nesses casos, uma implementação deve ser fornecida.

O C# 9 adiciona muitos outros pequenos recursos que melhoraram a produtividade do desenvolvedor, escrevendo e lendo código:

- Expressões de tipo de destino `new`
- `static` funções anônimas
- Expressões condicionais de tipo de destino
- `GetEnumerator()` Suporte de extensão para `foreach` loops
- Expressões lambda podem declarar parâmetros de descarte
- Atributos podem ser aplicados a funções locais

A versão C# 9 continua o trabalho para manter o C# uma linguagem de programação moderna e de uso geral. Os recursos continuam a dar suporte a cargas de trabalho e tipos de aplicativos modernos.

Artigo publicado originalmente no blog do NDepend, cortesia de Erik Dietrich e Patrick Smacchia.

Relações entre os recursos de linguagem e os tipos de bibliotecas

21/01/2022 • 2 minutes to read

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 dessa 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#

21/01/2022 • 2 minutes to read

A compatibilidade é uma meta muito 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.

Mais cuidado pode ser necessário quando você adota novos recursos de linguagem de programação em uma biblioteca. 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 talvez não tenham atualizado as versões ainda. Ao tomar decisões sobre a adoção de novos recursos, você precisará considerar duas variações de compatibilidade: compatível com fonte e binário.

Alterações compatíveis com binários

As alterações em sua biblioteca são **compatíveis com binários** quando sua biblioteca atualizada pode ser usada sem recompilar aplicativos e bibliotecas que o utilizam. Não é necessário recompilar assemblies dependentes, tampouco é necessária qualquer alteração de código-fonte.

Alterações compatíveis com a origem

As alterações na biblioteca são **compatíveis com a origem** quando os aplicativos e as bibliotecas que usam sua biblioteca não exigem alterações no código-fonte, mas a origem deve ser recompilada em relação à nova versão para funcionar corretamente.

Alterações incompatíveis

Se uma alteração não for **compatível com a fonte** nem **compatível com binário**, as alterações no código-fonte junto com a recompilação serão necessárias em bibliotecas e aplicativos dependentes.

Avaliar a biblioteca

Esses conceitos de compatibilidade afetam as declarações públicas e protegidas para sua biblioteca, mas não a respectiva implementação interna. A adoção de quaisquer novos recursos internamente sempre é **compatível com binário**.

as alterações **compatíveis com binários** fornecem uma nova sintaxe que gera o mesmo código compilado para declarações públicas como a sintaxe mais antiga. Por exemplo, alterar um método para um membro de apto para de expressão é uma alteração **compatível com binário** :

Código original:

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

Novo código:

```
public double CalculateSquare(double value) => value * value;
```

as alterações compatíveis com a **origem** introduzem a sintaxe que altera o código compilado para um membro público, mas de forma que seja compatível com os sites de chamada existentes. Por exemplo, a alteração de uma assinatura de método de um parâmetro por valor para um **in** parâmetro by reference é compatível com a fonte, mas não compatível com binário:

Código original:

```
public double CalculateSquare(double value) => value * value;
```

Novo código:

```
public double CalculateSquare(in double value) => value * value;
```

Os artigos [sobre o que há de novo](#) são a observação se a introdução de um recurso que afeta as declarações públicas é compatível com a fonte ou o binário.

Criar tipos de registro

21/01/2022 • 13 minutes to read

O C# 9 introduz *registros*, um novo tipo de referência que você pode criar em vez de classes ou structs. O C# 10 adiciona *structs de registro* para que você possa definir registros como tipos de valor. Os registros são diferentes das classes em que os tipos de registro usam *igualdade baseada em valor*. Duas variáveis de um tipo de registro serão iguais se as definições de tipo de registro são idênticas e se, para cada campo, os valores em ambos os registros são iguais. Duas variáveis de um tipo de classe serão iguais se os objetos referenciados são do mesmo tipo de classe e as variáveis se referem ao mesmo objeto. A igualdade baseada em valor implica outros recursos que você provavelmente deseja nos 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 `record struct` tipos.

Neste tutorial, você aprenderá a:

- 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 [SDK do .NET 6](#).

Características dos registros

Você define um *registro* declarando um tipo com a `record` palavra-chave, em vez da `class` palavra-chave `struct` ou `.` . 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 a semântica de valor, o compilador gera vários métodos para o tipo de registro (para `record class` tipos e `record struct` tipos):

- 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 !=` .
- Os 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 `with` suportam expressões 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 quando você declara registros posicionais:

- Um construtor primário cujos parâmetros corresponderem aos parâmetros posicionais na declaração de registro.
- Propriedades públicas para cada parâmetro de um construtor primário. Essas propriedades são *somente init para* `record class` tipos e `readonly record struct` tipos. Para `record struct` tipos, eles são *de leitura/gravação*.

- Um `Deconstruct` método para extrair propriedades do registro.

Criar dados de temperatura

Dados e estatísticas estão entre os cenários em que você deseja usar registros. Para este tutorial, você criará um aplicativo que calcula dias *de grau* para usos diferentes. Os dias de grau são uma medida de calor (ou falta de calor) em um período de dias, semanas ou meses. Os dias de grau acompanham e preveem o uso de energia. Dias mais quentes significam mais ar-condicionado e dias mais frios significam mais uso de uso de inserção. Os dias de grau ajudam a gerenciar populações de plantas e correlacionam-se ao crescimento da planta à medida que as estações mudam. Os dias de grau ajudam a acompanhar as migrações de animais para as espécies que viajam para corresponder ao clima.

A fórmula é baseada na temperatura média em um determinado dia e em uma temperatura de linha de base. Para calcular o grau de dias 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. Faça um novo aplicativo de console. Crie um novo tipo de registro em um novo arquivo chamado "DailyTemperature.cs":

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp);
```

O código anterior define um *registro posicional*. O registro é um , porque você não pretende herdar dele `DailyTemperature` `readonly record struct` e deve ser imutável. As propriedades e são apenas propriedades de inicialização , o que significa que elas podem ser definidas no `HighTemp` construtor ou usando um `LowTemp` inicializador de propriedade. Se você quiser que os parâmetros posicionais sejam de leitura/gravação, declare um `record struct` em vez de um `readonly record struct` . O `DailyTemperature` tipo também tem um construtor *primário* que tem dois parâmetros que corresponderem às duas propriedades. Use o construtor primário para inicializar um `DailyTemperature` registro. O código a seguir cria e inicializa vários `DailyTemperature` registros. O primeiro usa parâmetros nomeados para esclarecer `HighTemp` o e `LowTemp` o . Os inicializadores restantes usam parâmetros posicionais para inicializar `HighTemp` o e `LowTemp` o :

```
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 aos registros, incluindo registros posicionais. Você precisará calcular a temperatura média de cada dia. Você pode adicionar essa propriedade ao `DailyTemperature` registro:

```
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`:

```
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):

```
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 `ToString` de sintetizada pelo compilador. Se preferir um texto diferente, você poderá escrever sua própria versão do que impede que o `ToString` compilador synthesizing a version for you.

Dias de grau de computação

Para calcular dias de grau, você tira 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 todos os dias em que a temperatura média está abaixo da linha de base. Para medir o frio ao longo do tempo, descarte todos os dias em que a temperatura média está acima da linha de base. Por exemplo, os EUA usam 65F como base para dias de grau de aquecimento e 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 grau de resfriamento e zero dia de grau de aquecimento. Por outro lado, se a temperatura média for 55F, esse dia será de 10 dias de grau de aquecimento e 0 grau de resfriamento dias.

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 levam uma temperatura de linha de base e uma sequência de registros de temperatura diários como argumentos para o construtor primário:

```
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 `DegreeDays` abstrato é a classe base compartilhada para os registros e `HeatingDegreeDays` `CoolingDegreeDays`. As declarações do construtor primário nos registros derivados mostram como gerenciar a inicialização de 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 `Main` método :

```
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:

```
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. Mas este exemplo mostra por que você pode querer 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, os testes de igualdade e `System.IEquatable<T>` desigualdade e `GetHashCode()`. Para essa finalidade, você sintetizará `PrintMembers`. Você também pode declarar seu próprio `ToString`, mas fornece uma opção melhor para `PrintMembers` cenários de herança. Para fornecer sua própria versão de um método sintetizado, a assinatura deve corresponder ao método sintetizado.

O `TempRecords` elemento 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 `PrintMembers` sintetizado. A assinatura depende dos modificadores aplicados à `record` declaração:

- Se um tipo de registro `sealed` for ou um , a assinatura `record struct` será
`private bool PrintMembers(StringBuilder builder);`
- Se um tipo de registro não for e derivar de (ou seja, não declarar um `sealed` `object` 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 do `PrintMembers` . `PrintMembers` adiciona informações sobre cada propriedade em um tipo de registro a uma cadeia de caracteres. O contrato exige que os registros base adicionem seus membros à exibição e presume que membros derivados adicionarão seus membros. Cada tipo de registro sintetiza `ToString` uma substituição semelhante ao exemplo a seguir para `HeatingDegreeDays` :

```

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 `PrintMembers` um método no registro que não `DegreeDays` imprima o tipo da coleção:

```

protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}

```

A assinatura declara um `virtual protected` método para corresponder à versão do compilador. Não se preocupe se os acessadores estão errados; O idioma impõe a assinatura correta. Se você esquecer os modificadores corretos para qualquer método sintetizado, o compilador emite avisos ou erros que ajudam você a obter a assinatura correta.

No C# 10 e posterior, você pode declarar o `ToString` método como em um tipo de `sealed` registro. Isso impede que registros derivados deem uma nova implementação. Os registros derivados ainda conterão a `PrintMembers` substituição. Você faria o selo se não quisesse que ele `ToString` exibisse o tipo de runtime do registro. No exemplo anterior, você perderá as informações sobre onde o registro estava medindo dias de grau de aquecimento ou 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 mais facilmente registros imutáveis. Lembre-se de que você `readonly record struct` declara um para criar um struct de registro imutável. Veja novamente as declarações anteriores para `HeatingDegreeDays` e `CoolingDegreeDays`. Os membros adicionados executam cálculos nos valores do registro, mas não alteram o estado. Registros posicionais facilitam a criação de tipos de referência imutáveis.

A criação de tipos de referência imutáveis significa que você deseja usar a mutação não destrutiva. Você cria novas instâncias de registro semelhantes às instâncias de registro existentes usando `with expressões`. 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 permanece inalterado.

Vamos adicionar alguns recursos ao programa que demonstram `with expressões`. Primeiro, vamos criar um novo registro para calcular os dias de grau crescente usando os mesmos dados. *Os dias de grau crescente* normalmente usam 41F como a linha de base e medem as temperaturas acima da linha de base. Para usar os mesmos dados, você pode criar um novo registro semelhante ao , mas `coolingDegreeDays` com uma temperatura base diferente:

```

// 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 superficiais. A matriz para os dados não é copiada, mas ambos os registros se referem aos mesmos dados. Esse fato é uma vantagem em outro cenário. Para dias de grau crescente, é útil manter o controle do total dos cinco dias anteriores. Você pode criar novos registros com dados de origem diferentes usando `with` expressões. O código a seguir cria uma coleção desses acúmulos e, em seguida, exibe os valores:

```
// 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 `with` expressões para criar cópias de registros. Não especifique nenhuma propriedade entre as chaves da `with` expressão. Isso significa criar uma cópia e não alterar nenhuma propriedade:

```
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 quaisquer outros membros necessários para seus tipos de registro. Você pode criar tipos de registro imutáveis sabendo que nenhum dos membros gerados pelo compilador modificaria o estado. E `with` as expressões facilitam o suporte a mutação não destrutiva.

Os registros adicionam outra maneira de definir tipos. Você usa definições para criar hierarquias orientadas a objeto que se concentram `class` 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 quando deseja igualdade e comparação baseadas em valor, não deseja copiar valores e deseja `record` usar variáveis de referência. Você cria tipos quando deseja que os recursos de registros de um tipo pequeno `record struct` o suficiente para copiar com eficiência.

Você pode saber mais sobre registros no artigo de referência da linguagem C# para o tipo de registro e a especificação de tipo de registro proposta e a especificação [do struct de registro](#).

Tutorial: explorar ideias usando instruções de nível superior para criar código conforme você aprende

21/01/2022 • 8 minutes to read

Neste tutorial, você aprenderá como:

- Conheça as regras que regem o uso de instruções de nível superior.
- Use as instruções de nível superior para explorar os algoritmos.
- Refatorar 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 SDK do [Visual Studio 2022](#) ou do [.net 6](#).

Este tutorial pressupõe que você esteja familiarizado com 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 é semelhante ao seguinte código:

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

O código anterior é o resultado da execução do `dotnet new console` comando 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 remover tudo, exceto duas das linhas deste programa:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

IMPORTANT

Os modelos C# para .NET 6 usam *instruções de nível superior*. Seu aplicativo pode não corresponder ao código neste artigo, se você já tiver atualizado para as visualizações do .NET 6. Para obter mais informações, consulte o artigo sobre [novos modelos C# gerar 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 para cenários de script ou para explorar. Depois de trabalhar com as noções básicas, você pode começar a refatorar o código e criar métodos, classes ou outros assemblies para componentes reutilizáveis que você criou. As instruções de nível superior permitem a rápida experimentação e tutoriais iniciantes. Eles também fornecem um caminho suave de experimentação para programas completos.

As instruções de nível superior são executadas na ordem em que aparecem no arquivo. Instruções de nível superior só podem ser usadas em um arquivo de origem em seu aplicativo. O compilador gerará um erro se você usá-los em mais de um arquivo.

Criar uma máquina de resposta do .NET mágico

Para este tutorial, vamos criar um aplicativo de console que responde 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 de na cerimônia necessária para a estrutura de um programa típico. Em seguida, quando estiver satisfeito com a funcionalidade, você poderá refatorar o aplicativo como desejar.

Um bom ponto de partida é escrever a pergunta de volta ao console. Você pode começar escrevendo o código a seguir:

```
Console.WriteLine(args);
```

Você não declara uma `args` variável. Para o arquivo de origem único que contém as instruções de nível superior, o compilador reconhece `args` para significar os argumentos de linha de comando. O tipo de `args` é a `string[]`, como em todos os programas em C#.

Você pode testar seu código executando o seguinte `dotnet run` comando:

```
dotnet run -- Should I use top level statements in all my programs?
```

Os argumentos após o `--` na linha de comando são passados para o programa. Você pode ver o tipo da `args` variável, pois isso é o que é impresso no console:

```
System.String[]
```

Para gravar a pergunta no console, você precisará enumerar os argumentos e separá-los com um espaço. Substitua a `WriteLine` chamada pelo código a seguir:

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Agora, quando você executar o programa, ele exibirá corretamente a pergunta como uma cadeia de caracteres de argumentos.

Responder com uma resposta aleatória

Depois de ecoar a pergunta, você pode adicionar o código para gerar a resposta aleatória. Comece adicionando uma matriz de possíveis respostas:

```
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 que são afirmativo, cinco que não são confirmadas e cinco que são negativas. Em seguida, adicione o seguinte código para gerar e exibir uma resposta aleatória da matriz:

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Você pode executar o aplicativo novamente para ver os resultados. Você deverá ver algo semelhante à seguinte saída:

```
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 pergunta simulasse pensar sobre a resposta. Você pode fazer isso adicionando um pouco de animação ASCII e pausando enquanto trabalha. Adicione o seguinte código após a linha que ecoa a pergunta:

```
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();
```

Você também precisará adicionar uma `using` instrução à parte superior do arquivo de origem:

```
using System.Threading.Tasks;
```

As `using` instruções devem ser anteriores a qualquer outra instrução no arquivo. Caso contrário, é um erro do compilador. Você pode executar o programa novamente e ver a animação. Isso faz uma melhor experiência. Experimente o comprimento do atraso para corresponder ao seu gosto.

O código anterior cria um conjunto de linhas giratórias separadas por um espaço. A adição da `await` palavra-chave instrui o compilador a gerar o ponto de entrada do programa como um método que tem o `async` modificador e retorna um `System.Threading.Tasks.Task`. Este programa não retorna um valor, portanto, o ponto de entrada do programa retorna um `Task`. Se o seu programa retornar um valor inteiro, você adicionaria uma instrução `return` ao final de suas instruções de nível superior. Essa instrução de retorno especificaria o valor inteiro a ser retornado. Se as instruções de nível superior incluírem uma `await` expressão, o tipo de retorno se tornará `System.Threading.Tasks.Task<TResult>`.

Refatoração para o futuro

Seu programa deve ser semelhante ao seguinte código:

```

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 em espera. Esse trecho de código pode se tornar um método:

Você pode começar criando uma função local em seu arquivo. Substitua a animação atual pelo código a seguir:

```

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 dentro do método Main. Isso ainda não pode ser reutilizado. Portanto, extraia esse código em uma classe. Crie um novo arquivo chamado *Utilities.cs* e adicione o seguinte código:

```

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 que tem instruções de nível superior também pode conter namespaces e tipos no final do arquivo, após as instruções de nível superior. Mas, para este tutorial, você coloca o método de animação em um arquivo separado para torná-lo mais prontamente reutilizável.

Por fim, você pode limpar o código de animação para remover alguma duplicação:

```

foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.WriteLine(s);
    await Task.Delay(50);
    Console.WriteLine("\b\b\b");
}

```

Agora você tem um aplicativo completo e refatorou as partes reutilizáveis para uso posterior. Você pode chamar o novo método de utilitário de suas instruções de nível superior, conforme mostrado abaixo na versão concluída do programa principal:

```

using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.WriteLine(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 para `Utilities.ShowConsoleAnimation` e adiciona uma instrução adicional `using`.

Resumo

As instruções de nível superior facilitam a criação de programas simples para serem usados para explorar novos algoritmos. Você pode experimentar algoritmos experimentando trechos de código diferentes. Depois de aprender o que funciona, você pode refatorar o código para ser mais passível de manutenção.

Instruções de nível superior simplificam programas baseados em aplicativos de console. Isso inclui o Azure Functions, ações de GitHub e outros utilitários pequenos. Para obter mais informações, consulte [Instruções de nível superior \(guia de programação C#\)](#).

Usar a correspondência de padrões para criar seu comportamento de classe para um código melhor

21/01/2022 • 11 minutes to read

Os recursos de correspondência de padrões em C# fornecem sintaxe para expressar seus algoritmos. Você pode usar essas técnicas para implementar o comportamento em suas 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:

- Expresse suas classes orientadas a objeto usando padrões de dados.
- Implemente esses padrões usando os recursos de correspondência de padrões do C#.
- Aproveite o diagnóstico do compilador para validar sua implementação.

Pré-requisitos

Você precisará configurar seu 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 [SDK do .NET 5](#).

Criar uma simulação de um bloqueio de canal

Neste tutorial, você criará uma classe C# que simula um bloqueio de canal. Resumidamente, um bloqueio de canal é um dispositivo que gera e reduz a redução à medida que eles viajam entre dois trechos de água em níveis diferentes. Um bloqueio tem dois portões e algum mecanismo para alterar o nível da água.

Em sua operação normal, um barco entra em uma das portas enquanto o nível da água no bloqueio corresponde ao nível da água no lado em que o barco entra. Uma vez no bloqueio, o nível da água é alterado para corresponder ao nível da água em que o barco sairá do bloqueio. Depois que o nível da água corresponde a esse lado, o portão no lado de saída é aberto. 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 uma porta, o nível da água no bloqueio deve corresponder ao nível da água fora da porta que está sendo aberta.

Você pode criar uma classe C# para modelar esse comportamento. Uma `CanalLock` classe dá suporte a comandos para abrir ou fechar qualquer porta. Ele teria outros comandos para aumentar ou diminuir a água. A classe também deve dar suporte a propriedades para ler o estado atual dos portões e do nível da água. Seus métodos implementam as medidas de segurança.

Definir um classe

Você criará um aplicativo de console para testar sua `CanalLock` classe. Crie um novo projeto de console para o .NET 5 usando o Visual Studio ou a CLI do .NET. Em seguida, adicione uma nova classe e nomee-a `CanalLock` como . Em seguida, projete sua API pública, mas deixe os métodos não implementados:

```

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 em seu método para orientá-lo ao criar `Main` uma primeira implementação da classe :

```

// 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}");
Console.WriteLine(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 `CanalLock` classe . O código a seguir implementa os métodos da classe sem se preocupar com as regras de segurança. Você adicionará testes de segurança mais tarde:

```

// 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 os conceitos básicos. Agora, escreva um teste para a primeira condição de falha. No final dos testes anteriores, os dois portões são fechados e o nível da água é definido como baixo. Adicione um teste para tentar abrir a porta superior:

```

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 falha porque a porta é aberta. Como uma primeira implementação, você pode corrigi-la com o seguinte código:

```

// 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");
}

```

Seus testes são aprovados. Mas, à medida que adicionar mais testes, você adicionará cada vez mais `if` cláusulas e testará propriedades diferentes. Em breve, esses métodos serão muito complicados à medida que você adicionar mais condicionais.

Implementar os comandos com padrões

Uma 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 da porta, o nível da água e a nova configuração:

NOVA CONFIGURAÇÃO	ESTADO DO PORTÃO	NÍVEL DA ÁGUA	RESULTADO
Fechado	Fechado	Alto	Fechado
Fechado	Fechado	Baixo	Fechado
Fechado	Abrir	Alto	Abrir
Fechadas	Abrir	Baixa	Fechadas
Abrir	Fechadas	Alto	Abrir
Abrir	Fechadas	Baixo	Fechado (Erro)
Abrir	Abrir	Alto	Abrir
Abrir	Abrir	Baixa	Fechado (Erro)

A quarta e a última linhas da tabela têm o texto como inválido. O código que você está adicionando agora deve garantir que a porta d'água alta nunca seja aberta quando a água estiver baixa. Esses estados podem ser

codificados como uma única expressão switch (lembre-se de `false` que 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. Seus testes são aprovados, validando o código. A tabela completa mostra as possíveis combinações de entradas e resultados. Isso significa que você e outros desenvolvedores podem rapidamente observar a tabela e ver que você acobertou todas as entradas possíveis. 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 switch não abrange todas as entradas possíveis. O motivo desse aviso é que uma das entradas é um `enum` tipo. O compilador interpreta "todas as entradas possíveis" como todas as entradas do tipo subjacente, normalmente um `int`. Essa `switch` expressão verifica apenas os valores declarados no `enum`. Para remover o aviso, você pode adicionar um padrão de descarte catch-all para o último ARM da expressão. Essa condição gera uma exceção, porque ela indica entrada inválida:

```
_ => throw new InvalidOperationException("Invalid internal state"),
```

O braço de comutador anterior deve ser o último em sua `switch` expressão, pois ele corresponde a todas as entradas. Experimente movê-lo anteriormente na ordem. Isso causa um erro do compilador *CS8510* para código inacessível em um padrão. A estrutura natural de expressões switch permite que o compilador gere erros e avisos para possíveis erros. O compilador "rede de segurança" facilita a criação de código correto em menos iterações e a liberdade de combinar os braços de switch com caracteres curinga. O compilador emitirá erros se sua combinação resultar em braços inacessíveis que você não esperava e avisos se você remover um ARM necessário.

A primeira alteração é combinar todos os braços em que o comando é para fechar a porta; Isso é sempre permitido. Adicione o seguinte código como o primeiro ARM em sua expressão de comutador:

```
(false, _, _) => false,
```

Depois de adicionar o braço de comutador anterior, você obterá quatro erros de compilador, um em cada um dos braços em que o comando é `false`. Esses braços já estão cobertos pelo ARM recém-adicionado. Você pode remover essas quatro linhas com segurança. Você pretendia esse novo braço de comutador para substituir essas condições.

Em seguida, você pode simplificar os quatro braços em que o comando é para abrir o portão. Em ambos os casos em que o nível de água é alto, o portão pode ser aberto. (Em um, ele já está aberto.) Um caso em que o nível de água está 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:

```
(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 passam. Esta é a versão final do `SetHighGate` método:

```
// 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 `SetLowGate` métodos e por `SetWaterLevel` conta própria. Comece adicionando o código a seguir para testar operações inválidas nesses métodos:

```

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 canal entra em um estado inválido. Tente implementar os métodos restantes por conta própria. O método para definir o portão inferior deve ser semelhante ao método para definir o portão superior. O método que altera o nível de água tem verificações diferentes, mas deve seguir uma estrutura semelhante. Talvez você ache útil usar o mesmo processo para o método que define o nível de água. Comece com todas as quatro entradas: o estado de ambos os Gates, o estado atual do nível de água e o novo nível de água solicitado. A expressão switch deve começar com:

```

CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
{
    // elided
};

```

Você terá 16 braços de comutação total para preencher. Em seguida, teste e simplifique.

Você fez métodos como este?

```

// 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"),
    };
}

```

Seus testes devem passar e o bloqueio canal deve funcionar com segurança.

Resumo

Neste tutorial, você aprendeu a usar a correspondência de padrões para verificar o estado interno de um objeto antes de aplicar qualquer alteração a esse estado. Você pode verificar combinações de propriedades. Depois de criar tabelas para qualquer uma dessas transições, você testará seu código e, em seguida, simplificará a legibilidade e a capacidade de manutenção. Essas refatoridades iniciais podem sugerir refatoração adicional que validam o estado interno ou gerenciam outras alterações de API. Este tutorial combinou classes e objetos com uma abordagem mais orientada a dados, baseada em padrões, para implementar essas classes.

Tutorial: Atualizar interfaces com métodos de interface padrão no C# 8.0

21/01/2022 • 6 minutes to read

Desde o C# 8.0 no .NET Core 3.0, é possível definir uma implementação em que você declara 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 precisará configurar seu computador para executar o .NET Core, incluindo o compilador C# 8.0. O compilador C# 8.0 está disponível a partir [do Visual Studio 2019 versão 16.3](#) ou do [SDK do .NET Core 3.0](#).

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:

```
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:

```
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 lançadas! Esta é uma alteração da falha!" O C# 8.0 adiciona *implementações de interface padrão* para interfaces de atualização. 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. Isso o torna um cenário perfeito para métodos de interface padrão. Você pode adicionar um método à `ICustomer` interface 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 :

```
// 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:

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))  
{  
    Reminders =  
    {  
        { new DateTime(2010, 08, 12), "child's birthday" },  
        { new DateTime(2012, 11, 15), "anniversary" }  
    }  
};  
  
SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);  
c.AddOrder(o);  
  
o = new SampleOrder(new DateTime(2013, 7, 4), 25m);  
c.AddOrder(o);  
  
// Check the discount:  
ICustomer theCustomer = c;  
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Observe a seguinte parte do teste:

```
// 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

Esse é um bom início. Porém, a implementação padrão é restritiva demais. 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:

```
// 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 muitas novas funcionalidades de linguagem mostradas nesse pequeno fragmento de código. Agora as interfaces podem incluir membros estáticos, incluindo campos e métodos. Modificadores de acesso diferentes também estão habilitados. Os campos adicionais são particulares, 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:

```
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 obtém 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:

```
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":

```
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: Misturar funcionalidades ao criar classes usando interfaces com métodos de interface padrão

21/01/2022 • 9 minutes to read

Desde o C# 8.0 no .NET Core 3.0, é possível definir uma implementação em que você declara um membro de uma interface. Esse recurso 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 para 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 substituem algumas ou todas as implementações padrão.

Pré-requisitos

Você precisará configurar seu computador para executar o .NET Core, incluindo o compilador C# 8.0. O compilador C# 8.0 está disponível a partir [do Visual Studio 2019 versão 16.3](#)ou do [SDK do .NET Core 3.0](#) ou posterior.

Limitações dos métodos de extensão

Uma maneira de implementar o comportamento que aparece como parte de uma interface é definir 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 fornecem a implementação para que qualquer [Enumerable](#) sequência seja a origem de uma consulta LINQ.

Os métodos de extensão são resolvidos em tempo de compilação, usando o tipo declarado da variável. 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 corresponde à interface , as chamadas de método resolvem para o método de extensão. Outra preocupação com os métodos de extensão é que esses métodos são acessíveis sempre que a classe que contém os métodos de extensão está acessível. As classes não podem declarar se devem ou não fornecer recursos declarados em métodos de extensão.

A partir do C# 8.0, 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. Em um sentido, essa técnica parece semelhante a como você pode usar métodos [de extensão](#).

Neste artigo, você aprenderá como as implementações de interface padrão permitem novos cenários.

Criar o aplicativo

Considere um aplicativo de automação residencial. Você provavelmente tem muitos tipos diferentes de luzes e indicadores que podem ser usados em toda a casa. Cada luz deve dar suporte a APIs para a ligar e desligar e

relatar o estado atual. Algumas luzes e indicadores podem dar suporte a outros recursos, como:

- Ligue a luz e, em seguida, desligue-a após um temporizador.
- Piscar a luz por um período de tempo.

Alguns desses recursos estendidos podem ser emulados em dispositivos que suportam o conjunto mínimo. Isso indica o fornecimento de uma implementação padrão. Para os dispositivos que têm mais recursos integrados, o software do dispositivo usaria as funcionalidades nativas. Para outras luzes, eles podem optar por implementar a interface e usar a implementação padrão.

Os membros da interface padrão são uma solução melhor para esse cenário do que os métodos de extensão. Os autores de classe podem controlar quais interfaces eles optam por 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:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Uma luminária de sobrecarga básica pode implementar essa interface, conforme mostrado no código a seguir:

```
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 dispositivos IoT, mas emula essas atividades escrevendo mensagens no console. Você pode explorar o código sem automatizar sua casa.

Em seguida, vamos definir a interface para uma luz que pode ser desligada automaticamente após um tempoout:

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Você pode adicionar uma implementação básica à luz de sobrecarga, mas uma solução melhor é modificar essa definição de interface para fornecer uma `virtual` implementação padrão:

```

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.");
    }
}

```

Ao adicionar essa alteração, `OverheadLight` a classe pode implementar a função de temporizador declarando suporte para a interface :

```

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 código a seguir:

```

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 de substituindo métodos de classe virtual, a declaração de `TurnOnFor` na classe não usa a `HalogenLight` `override` palavra-chave .

Combinar e combinar funcionalidades

As vantagens dos métodos de interface padrão ficam mais claras à medida que você introduz funcionalidades mais avançadas. O uso de interfaces permite combinar e combinar funcionalidades. Ele também permite que cada autor de classe escolha entre a implementação padrão e uma implementação personalizada. Vamos adicionar uma interface com uma implementação padrão para uma luz piscando:

```

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 de sobrecarga pode adicionar funcionalidades de temporizador e piscar usando a implementação padrão:

```

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, o `LEDLight` dá suporte à função de temporizador e à função blink diretamente. Esse estilo claro implementa as `ITimerLight` `IBlinkingLight` interfaces e substitui o `Blink` método :

```

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 a funções de piscar e temporizador diretamente:

```

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 o `IBlinkingLight` à lista de suas interfaces com suporte.

Detectar os tipos de luz usando a correspondência de padrões

Em seguida, vamos escrever um código de teste. Você pode usar o recurso de correspondência de padrões do C# para determinar os recursos de uma luz examinando quais interfaces ele dá suporte. O método a seguir exercícios as funcionalidades com suporte de cada luz:

```

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 em `Main` seu método cria cada tipo de luz na sequência e testa essa luz:

```
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

Este cenário mostra uma interface base sem nenhuma implementação. Adicionar um método à `ILight` interface introduz novas complexidades. As regras de linguagem que regem 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 mostrar como isso altera seu uso. Cada luz de indicador pode relatar seu status de energia como um valor enumerado:

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

A implementação padrão não assume nenhuma potência:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Essas alterações são compiladas corretamente, mesmo que o declare suporte para a interface e ambas `ExtraFancyLight` `ILight` as interfaces derivadas, `ITimerLight` e `IBlinkingLight`. Há apenas uma implementação "mais próxima" declarada na `ILight` interface. Qualquer classe que declarasse uma substituição se tornaria a implementação "mais próxima". Você viu exemplos nas classes anteriores que overrode os membros de outras interfaces derivadas.

Evite a substituição do mesmo método em várias interfaces derivadas. Isso cria uma chamada de método

ambígua sempre que uma classe implementa ambas as interfaces derivadas. O compilador não pode escolher um único método melhor, portanto, emite um erro. Por exemplo, se e implementarem uma substituição de , o `IIBlinkingLight` precisará fornecer uma substituição mais `ITimerLight` `PowerStatus` `OverheadLight` 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 sua própria interface; várias interfaces são herdadas apenas por classes.

Este exemplo mostra um cenário em que você pode definir recursos discretos que podem ser mistos em classes. Você declara qualquer conjunto de funcionalidades com suporte declarando quais interfaces uma classe dá suporte. O uso de métodos de interface padrão virtuais permite que as classes usem ou definam uma implementação diferente para qualquer ou todos os métodos de interface. Essa funcionalidade de linguagem fornece novas maneiras de modelar os sistemas do mundo real que você está criando. Os métodos de interface padrão fornecem uma maneira mais clara de expressar classes relacionadas que podem combinar e corresponder a diferentes recursos usando implementações virtuais desses recursos.

Índices e intervalos

21/01/2022 • 6 minutes to read

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.
- Compreenda as decisões de design para o início e o fim de cada sequência.
- Conheça cenários para os tipos [Index](#) e [Range](#).

Supporte a idioma para intervalos e índices

Esse suporte a idioma depende de dois novos tipos e de dois novos operadores:

- [System.Index](#) representa um índice em uma sequência.
- O índice do operador end `^`, que especifica que um índice é relativo ao final de uma sequência.
- [System.Range](#) representa um subintervalo de uma sequência.
- O operador Range `...`, que especifica o início e o término 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]` faz. Para qualquer número `n`, o índice `^n` é o mesmo que `sequence[sequence.Length - n]`.

```
string[] words = new string[]
{
    // index from start      index from end
    "The",        // 0           ^9
    "quick",      // 1           ^8
    "brown",      // 2           ^7
    "fox",        // 3           ^6
    "jumped",     // 4           ^5
    "over",       // 5           ^4
    "the",        // 6           ^3
    "lazy",       // 7           ^2
    "dog"         // 8           ^1
};                // 9 (or words.Length) ^0
```

Você pode recuperar a última palavra com o índice `^1`. Adicione o código a seguir abaixo da inicialização:

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

Um intervalo especifica o *íncio* e o *final* de um intervalo. Os intervalos são exclusivos, o que significa que 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. Adicione o seguinte código ao mesmo método. Copie e cole-o na parte inferior da janela interativa.

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.WriteLine($"< {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:

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

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

```
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($"< {word} >");
foreach (var word in firstPhrase)
    Console.WriteLine($"< {word} >");
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 `]`:

```
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:

```

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]}, numbers[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 {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($"\\t{start_x[0]}..{start_x[^1]} is the same as {zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..{z_zero[^1]}");

```

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 fornece um [indexador](#) com um [Index Range](#) parâmetro ou dá suporte explicitamente a índices ou intervalos, respectivamente. Um indexador que usa um único [Range](#) parâmetro pode retornar um tipo de sequência diferente, como [System.Span<T>](#).

IMPORTANT

O desempenho do código usando o operador Range depende do tipo do operando de sequência.

A complexidade de tempo do operador de intervalo depende do tipo de sequência. Por exemplo, se a sequência for uma `string` matriz ou, 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 faz referência ao mesmo armazenamento de backup, o que significa que não há cópia e a operação é $O(1)$.

Além da complexidade do tempo, isso causa alocações e cópias extras, afetando o desempenho. Em código sensível ao desempenho, considere usar `Span<T>` ou `Memory<T>` como o tipo de sequência, já que o operador de intervalo não é alocado para eles.

Um tipo pode ser [contabilizado](#) se tiver uma propriedade chamada `Length` ou `Count` com um getter acessível e um tipo de retorno de `int`. Um tipo com contagem que não dá suporte explicitamente a índices ou intervalos pode fornecer um suporte implícito para eles. Para obter mais informações, consulte as seções [suporte ao índice implícito](#) e [suporte de intervalo implícito](#) da [Nota de proposta de recurso](#). Intervalos usando o suporte de intervalo implícito retornam o mesmo tipo de sequência que a sequência de origem.

Por exemplo, os seguintes tipos .NET oferecem suporte a índices e intervalos: [String](#), [Span<T>](#) e [ReadOnlySpan<T>](#). O [List<T>](#) dá suporte a índices, mas não dá suporte a intervalos.

[Array](#) tem um comportamento mais nuanceado. 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 chamadas de matriz de matrizes, oferecem suporte a intervalos e indexadores. O exemplo a seguir mostra como iterar uma subseção retangular de uma matriz denteadada. Ele itera a seção no centro, excluindo as primeiras e últimas três linhas, e a primeira e a última duas colunas de cada linha selecionada:

```
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 Range para [Array](#) aloca uma matriz para armazenar os elementos retornados.

Cenários para índices e intervalos

Você geralmente usará intervalos e índices quando desejar analisar uma parte de uma sequência maior. A nova sintaxe fica mais clara na leitura exata do que parte da seqüência 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:

```
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 + 10..^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();
```

Tutorial: Expressar sua intenção de design mais claramente com tipos de referência que permitem valor nulo e tipos que não permitem valor nulo

21/01/2022 • 11 minutes to read

O C# 8.0 introduz tipos de referência que podem ser anulados, que complementam os tipos de referência da mesma maneira que os tipos de valor que podem ser anulados 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 seu computador para executar o .NET Core, incluindo o compilador C# 8.0. O compilador C# 8.0 está disponível [com o Visual Studio 2019](#) ou [o .NET Core 3.0](#).

Este tutorial pressupõe que você esteja familiarizado com o C# e o .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 estar `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 que permite valor nulo **habilitado**. Abra o arquivo `.csproj` e adicione `Nullable` um elemento ao elemento `PropertyGroup`. Defina seu valor como `enable`. Você deve optar pelo recurso de **tipos de referência que permitem valor nulo**, mesmo em projetos C# 8.0. 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 ajude a encontrar problemas em que o código existente pode não ter verificações nulas adequadas, ela pode não refletir com precisão sua intenção de design original:

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

Antes do .NET 6, novos projetos não incluem o `Nullable` elemento. Começando com o .NET 6, novos projetos incluem `<Nullable>enable</Nullable>` o elemento 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 você tiver programado em C#, poderá estar tão acostumado a tipos de referência que permitem valores que podem ter perdido outras oportunidades para declarar instâncias não `null` anuladas:

- 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 que não pode ser anulado como o padrão para referências evita erros comuns que podem levar a `NullReferenceException`. Uma lição deste tutorial é que você tomou decisões sobre quais variáveis podem ou não ser `null`. O idioma não forneceu sintaxe para expressar essas decisões. Agora ele já fornece.

O aplicativo que você criará faz as seguintes etapas:

1. Cria uma pesquisa e adiciona perguntas a ela.
2. Cria um conjunto pseudo-aleatório de respondentes para a pesquisa.
3. Entra em contato com os respondentes até que o tamanho da pesquisa concluída atinja o número da meta.
4. Grava estatísticas importantes nas respostas da pesquisa.

Criar a pesquisa com tipos de referência que podem ser anulados e não anulados

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 `public SurveyQuestion` classe:

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

O compilador interpreta cada declaração de variável de tipo de referência como um tipo de referência que não permite valor nulo para o código em um contexto de anotação que permite valor nulo habilitado. Para ver seu primeiro aviso, adicione propriedades ao texto da pergunta e tipo de pergunta, conforme mostrado no código a seguir:

```
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:

```
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:

```
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`.

Alternar para `Program.cs` em seu editor e substituir o conteúdo de `Main` por linhas de código a seguir:

```
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 todo o projeto está em um contexto de anotação que permite valor nulo habilitado, você obterá avisos quando passar para qualquer método esperando um tipo de referência que não permite valor `null` nulo.

Experimente adicionar a seguinte linha a `Main`:

```
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. Habilite o suporte para tipos que permitem valor nulo. Adicione uma propriedade `Id` e um construtor para inicializá-la, conforme mostrado no código a seguir:

```
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:

```
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`:

```

private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                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 você alguma vez desreferenciar sem verificar o `surveyResponses` `null` valor primeiro, você obterá 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`:

```
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`:

```
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`:

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No answer";
```

Como `surveyResponses` é um tipo de referência anulável, verificações nulas são necessárias antes de fazer referência a ela. O `Answer` método retorna uma cadeia de caracteres não anulável, portanto, precisamos abordar o caso de uma resposta ausente usando o operador de União nula.

Em seguida, adicione esses três membros com corpo de expressão à classe `SurveyRun`:

```
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`:

```
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 usar Entity Framework:

[Conceitos básicos de Entity Framework Core: trabalhando com tipos de referência anuláveis](#)

Tutorial: gerar e consumir fluxos assíncronos usando C# 8,0 e .NET Core 3,0

21/01/2022 • 10 minutes to read

O C# 8,0 introduz **fluxos assíncronos**, que modelam uma fonte de transmissão de dados. Os fluxos de dados geralmente recuperam ou geram elementos de forma assíncrona. Os fluxos assíncronos dependem de novas interfaces introduzidas no .NET Standard 2,1. Essas interfaces têm suporte no .NET Core 3,0 e posterior. 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 a 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 precisará configurar seu computador para executar o .NET Core, incluindo o compilador C# 8,0. O compilador C# 8 está disponível a partir do [Visual Studio 2019 versão 16,3](#) ou do [SDK do .net Core 3,0](#).

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.

WARNING

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 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 do repositório [dotnet/docs](#) na pasta [Csharp/Whats-New/tutoriais](#) .

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:

```

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 fonte 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`:

```

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 GitHub GraphQL](#) para obter detalhes sobre a API do GitHub GraphQL.) O `RunPagedQueryAsync` método enumera os problemas do mais recente para o mais antigo. 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 alocar 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 tornam o algoritmo mais difícil de entender em sua primeira leitura. Mais tipos e APIs estão envolvidos. Você deve rastrear as comunicações por meio do `CancellationTokenSource` e seu associado `CancellationToken` para entender onde o cancelamento é solicitado e onde ele é 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:

```
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:

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Substitua essas três linhas pelo seguinte código:

```
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 `finished` deve ser semelhante ao seguinte código:

```

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:

```

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`:

```
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 [IAsyncEnumerable<T>](#) deriva de [IAsyncDisposable](#). Isso significa que o loop anterior descartará o fluxo de forma assíncrona quando o loop for concluído. Você pode imaginar que o loop é semelhante ao seguinte código:

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
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 [TaskAsyncEnumerableExtensions.ConfigureAwait](#) método de extensão. Para obter mais informações sobre contextos de sincronização e como capturar o contexto atual, consulte o artigo sobre como [consumir o padrão assíncrono baseado em tarefa](#).

Os fluxos assíncronos dão suporte ao cancelamento usando o mesmo protocolo que outros `async` métodos. Você modificaria a assinatura do método de iterador assíncrono da seguinte maneira para dar suporte ao cancelamento:

```

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 [EnumeratorCancellationAttribute](#) atributo faz com que o compilador gere código para o [IAsyncEnumerable<T>](#) que torna o token passado para [GetAsyncEnumerator](#) visível para o corpo do iterador assíncrono como esse argumento. Dentro [runQueryAsync](#) do, você pode examinar o estado do token e cancelar mais trabalho, se solicitado.

Você usa outro método de extensão, [WithCancellation](#), para passar o token de cancelamento para o fluxo assíncrono. Você modificaria o loop enumerando os problemas da seguinte maneira:

```

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 [Csharp/Whats-New/tutoriais](#).

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 é incluído diretamente no `await foreach` loop. 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 aplicativo 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.

Tutorial: Escrever um manipulador de interpolação de cadeia de caracteres personalizado

21/01/2022 • 13 minutes to read

Neste tutorial, você aprenderá a:

- 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 SDK do .NET 6](#).

Este tutorial pressupõe que você esteja familiarizado com o C# e o .NET, incluindo Visual Studio ou a CLI do .NET.

Nova delinea

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. Ele 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 parar o processamento depois que esse tamanho 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 todo o código do cliente a estar em conformidade.

Neste tutorial, você criará um manipulador de interpolação de cadeia de caracteres para um dos principais cenários de desempenho: bibliotecas de log. Dependendo do nível de log configurado, o trabalho para construir uma mensagem de log não é necessário. Se o registro em log estiver desligado, o trabalho para construir uma cadeia de caracteres 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, 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 executar apenas o trabalho necessário, se necessário.

Implementação inicial

Vamos começar de uma classe básica que dá suporte a `Logger` diferentes níveis:

```

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);
    }
}

```

Isso `Logger` dá suporte a seis níveis diferentes. Quando uma mensagem não passa no filtro de nível de log, não há saída. A API pública para o logger 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

Esta etapa é criar um manipulador *de cadeia de caracteres interpolado* que recria 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 `int` parâmetros, `literalLength` e `formatCount` . (Mais parâmetros são permitidos).
- Um método `AppendLiteral` público 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 `LogInterpolatedStringHandler` tipo que atende a estes requisitos:

```
[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 para experimentar o novo manipulador de cadeia de `Logger` caracteres interpolado:

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Você não precisa remover o método original, o compilador preferirá um método com um parâmetro de manipulador interpolado em vez de um método com um parâmetro quando o argumento for uma expressão de cadeia de caracteres `LogMessage` `string` interpolada.

Você pode verificar se o novo manipulador é invocado usando o seguinte código como o programa principal:

```
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:

```

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.

```

Ao rastrear 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 espaço reservados.
- O compilador adiciona chamadas a `AppendLiteral` para cada seção da cadeia de `AppendFormatted` caracteres literal e para cada espaço reservado.
- O compilador invoca o `LogMessage` método usando o como o argumento `CoreInterpolatedStringHandler`.

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 funcionalidades 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 aprimorará seu manipulador para que ele funcione menos quando a cadeia de caracteres construída não for gravado 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 as alterações no Manipulador. Primeiro, adicione um campo para acompanhar se o manipulador estiver habilitado. Adicione dois parâmetros ao construtor: um para especificar o nível de log dessa mensagem e o outro uma referência ao objeto de log:

```

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 só adeça literais ou objetos formatados quando a cadeia de caracteres final for usada:

```

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 para que o compilador passe os `LogMessage` parâmetros adicionais para o construtor do manipulador. Isso é tratado usando o [System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute](#) no argumento do manipulador:

```

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 esse mapa para os `LogMessage` parâmetros que seguem os `literalLength` `formattedCount` parâmetros e necessários. A cadeia de caracteres vazia (""), especifica o receptor. O compilador substitui o valor do objeto representado por pelo próximo `Logger` argumento para o construtor do `this` manipulador. O compilador substitui o valor de `level` para o argumento a seguir. Você pode fornecer qualquer número de argumentos para qualquer manipulador que você escreve. 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:

```

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 `AppendLiteral` métodos e estão sendo `AppendFormat` 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 cria. Ainda há algumas melhorias a fazer.

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 espaço reservados. Ao fazer essa alteração, também vamos alterar o tipo de retorno dos outros

métodos e , de para (se qualquer um desses métodos tiver tipos de retorno diferentes, você obterá um erro de `AppendFormatted` `AppendLiteral` `void` `bool` compilação). Essa alteração habilita *o curto-circuito*. Os métodos retornam para indicar que o processamento da expressão de cadeia de caracteres `false` interpolada deve ser interrompido. Retornar `true` indica que ele deve continuar. Neste exemplo, você o está usando para interromper o processamento quando a cadeia de caracteres resultante não for necessária. O curto-círculo dá suporte a ações mais finas. Você pode parar de processar a expressão depois que ela atingir um determinado comprimento, para dar suporte a buffers de comprimento fixo. Ou alguma condição pode indicar que os elementos restantes não são necessários.

```
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:

```
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 tempo curto" para a hora atual. O exemplo anterior mostrou uma das sobrecargas para o `AppendFormatted` método 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 que cria em cadeia de caracteres. Você pode escrever sobrecargas de `AppendFormatted` que assumem esses tipos em vez de um argumento genérico. O compilador escolherá a melhor sobrecarga. O runtime usa essa técnica para converter em saída `System.Span<T>` 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 acompanha o .NET 6 contém nove sobrecargas de para `AppendFormatted` usos diferentes. Você pode usá-lo como uma referência ao criar um manipulador para suas finalidades.

Execute o exemplo agora, e você verá que, para a `Trace` mensagem, apenas o primeiro `AppendLiteral` é chamado:

```

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 `out bool` parâmetro final. Definir esse parâmetro como `false` indica que o manipulador não deve ser chamado para processar a expressão de cadeia de caracteres interpolada:

```

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 `enabled` campo. 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:

```

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 `Append` métodos foi invocado.

Este exemplo ilustra um ponto importante para manipuladores de cadeia de caracteres interpolados,

especialmente quando as bibliotecas de log são usadas. Qualquer efeito colateral nos espaços reservados pode não ocorrer. Adicione o seguinte código ao programa principal e veja esse comportamento em ação:

```
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 `index` variável é incrementada cinco vezes cada iteração do loop. Como os espaços reservados são avaliados apenas para `Critical`, `Error` e `Warning` os níveis, não para `Information` e `Trace`, o valor final de `index` não corresponde à expectativa:

```
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
```

Os 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 tempo de execução do .NET já usou esse recurso para melhorar o desempenho em várias áreas. Você pode fazer uso do mesmo recurso em suas próprias bibliotecas. Para explorar ainda mais, examine o [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#). Ele fornece uma implementação mais completa do que você criou aqui. Você verá muitas outras sobrecargas possíveis para os `Append` métodos.

Usar interpolação de cadeia de caracteres para construir cadeia de caracteres formatadas

21/01/2022 • 8 minutes to read

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 do .NET [Olá, Mundo em 10](#) minutos tem instruções para configurar seu ambiente de desenvolvimento local no Windows, Linux ou macOS. Você também pode 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:

```
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:

```
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 do C# exibirá 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 cliente para obter a representação de cadeia de caracteres de uma Vegetable instância. No exemplo, Vegetable.ToString() o método retorna o valor da propriedade que é Name inicializada no Vegetable construtor:

```
public Vegetable(string name) => Name = name;
```

Em seguida, criamos uma instância Vegetable da classe chamada usando o item new operador e fornecendo um nome para o construtor Vegetable :

```
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:

```
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:

```
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:

```

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:

```

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`.

```

Console.WriteLine($"{DateTime.Now, -20:d} {DateTime.Now, -10:HH} [{1063.342, 15:N2}] feet");

```

A saída é semelhante ao seguinte:

```

[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#

21/01/2022 • 6 minutes to read

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](#).

NOTE

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:

```
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:

```
{<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:

```
{<interpolationExpression>:<formatString>}
```

O seguinte exemplo mostra como especificar cadeias de caracteres de formato padrão e personalizadas para expressões que produzem resultados numéricos ou de data e hora:

```
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:

```
{<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:

```
{<interpolationExpression>,<alignment>:<formatString>}
```

O seguinte exemplo mostra como especificar o alinhamento e usa caracteres de barra vertical ("|") para delimitar campos de texto:

```
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 textual interpolada começa com o `$` caractere seguido pelo `@` caractere. A partir do C# 8,0, você pode usar `$` os `@` tokens e em qualquer ordem: `$@"..."` e `@$"..."` são cadeias de caracteres textuais 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:

```
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.");
// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\\Users\\{userName}\\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// 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:

```
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:

```
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:

```
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

21/01/2022 • 10 minutes to read

Este tutorial ensina uma série de recursos no .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 ecoa o conteúdo desse arquivo para o console. A saída para o console é conduzida a fim de corresponder à leitura em voz alta. Você pode acelerar ou reduzir o ritmo pressionando as chaves '`<`' (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 compilá-las uma a uma.

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 a fazer modificações, vamos executar o aplicativo simples de Olá, Mundo. Depois de criar o aplicativo, digite `dotnet run` no prompt de comando. Esse comando executa o NuGet processo de restauração do pacote, cria o executável do aplicativo e executa o executável.

O código do aplicativo Olá, Mundo simples está 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, consulte uma `namespace` instrução. 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`):

```
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 enumeradores 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 enumeradores são métodos que contêm uma ou mais `yield return` instruções. 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 de sintaxe C# que podem ser novos para você. A `using` instrução neste 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 digitada implicitamente*. 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 `Main` método:

```
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 impressa 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 um pouco 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 para retornar palavras únicas em vez de linhas inteiras. Isso é feito com essas modificações. Substitua a instrução `yield return line;` pelo seguinte código:

```
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:

```
Console.WriteLine(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. Isso é fácil de corrigir. Você só controlará o tamanho de cada linha e gerará uma nova linha sempre que o comprimento da linha atingir um determinado limite. Declare uma variável local após a declaração de `words` no método `ReadFrom` que contém o comprimento da linha:

```
var lineLength = 0;
```

Em seguida, adicione o seguinte código após a instrução `yield return word + " "` (antes da chave de fechamento):

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Execute o exemplo e você poderá ler em voz alta em seu ritmo pré-configurado.

Tarefas assíncronas

Nesta etapa final, você adicionará o código para gravar a saída de forma assíncrona em uma tarefa, enquanto também executa outra tarefa para ler a entrada do usuário se quiser acelerar ou diminuir a exibição do texto, ou parar completamente a exibição do texto. Isso tem algumas etapas e, no final, você terá todas as atualizações necessárias. A primeira etapa é criar um método de `Task` retorno assíncrono que represente o código que você criou até agora para ler e exibir o arquivo.

Adicione este método à sua `Program` classe (é tirado do corpo do seu `Main` método):

```
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.

Chame esse novo método em seu método `Main`:

```
ShowTeleprompter().Wait();
```

Aqui, em `Main`, o código aguarda de forma síncrona. Use o operador `await` em vez de esperar de forma síncrona sempre que possível. Mas, no método de um aplicativo de `Main` console, você não pode usar o `await` operador. Isso resultaria no encerramento do aplicativo antes da conclusão de todas as tarefas.

NOTE

Se você usar o C# 7.1 ou posterior, poderá criar aplicativos de console com o `async Main` método.

Em seguida, você precisa escrever o segundo método assíncrono para ler no Console e observar as chaves '<' (menor que), '>' (maior que) e 'X' ou 'x'. Este é o método que você adiciona para essa tarefa:

```

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 que lê uma chave do Console e modifica uma variável local que representa o atraso quando o usuário pressiona as [Action](#) teclas '<' (menor que) ou '>' (maior que). O método delegado termina quando o usuário pressiona as teclas 'X' ou 'x', que permitem que o usuário pare 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 possa 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:

```

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 essa classe no `TeleprompterConsole` namespace, conforme mostrado. Você também precisará adicionar uma instrução na parte superior do arquivo para que você possa referenciar os métodos e sem os nomes de classe ou `using static` `Min` `Max` namespace delimitadores. Uma `using static` instrução importa os métodos de uma classe. Isso é diferente da instrução `using` sem , que importa todas as classes de um `static` namespace.

```
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:

```
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:

```
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`:

```
RunTeleprompter().Wait();
```

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 viu os conceitos básicos de E/S de Arquivo e Console, o bloqueio e o uso sem bloqueio da programação assíncrona baseada em tarefas, um tour pela linguagem C# e como os programas C# são organizados e a CLI do .NET.

Para obter mais informações sobre E/S de arquivo, consulte [E/S de arquivo e fluxo](#). Para obter mais informações sobre o modelo de programação assíncrona usado neste tutorial, consulte [Programação assíncrona baseada em tarefas](#) e [programação assíncrona](#).

Tutorial: Fazer solicitações HTTP em um aplicativo de console .NET usando C#

21/01/2022 • 8 minutes to read

Este tutorial cria um aplicativo que emite solicitações HTTP para um serviço REST 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 desserialização*.

O tutorial mostra como:

- Enviar solicitações HTTP.
- Deserializar respostas JSON.
- Configure a destrialização com atributos.

Se você preferir acompanhar o exemplo [final deste](#) tutorial, poderá baixá-lo. Para obter instruções de download, consulte [Exemplos e tutoriais](#).

Pré-requisitos

- [SDK do .NET 5.0 ou posterior](#)
- Um editor de código como [Visual Studio Code](#), que é um editor de plataforma cruzada de código aberto. Você pode executar o aplicativo de exemplo 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:

```
dotnet new console --name WebAPIClient
```

Esse comando cria os arquivos iniciais para um aplicativo "Olá, Mundo" básico. O nome do projeto é "WebAPIClient".

3. Navegue até o diretório "WebAPIClient" e execute o aplicativo.

```
cd WebAPIClient
```

```
dotnet run
```

`dotnet run` é executado `dotnet restore` automaticamente para restaurar as dependências de que o aplicativo precisa. Ele também é executado `dotnet build` se necessário.

Fazer solicitações HTTP

Esse aplicativo chama a [API GitHub para](#) obter informações sobre os projetos no [.NET Foundation Umbrella](#). O ponto de extremidade é <https://api.github.com/orgs/dotnet/repos>. Para recuperar informações, ele faz uma solicitação HTTP GET. Os navegadores também fazem solicitações HTTP GET, para que você possa colar essa URL

na barra de endereços do navegador para ver quais informações você receberá e processará.

Use a `HttpClient` classe 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 `Program.cs` arquivo no diretório do projeto e adicione o seguinte método assíncrono à `Program` classe :

```
private static async Task ProcessRepositories()
{
}
```

2. Adicione uma `using` diretiva na parte superior do arquivo `Program.cs` para que o compilador C# reconheça o `Task` tipo:

```
using System.Threading.Tasks;
```

Se você executar neste ponto, a compilação terá êxito, mas avisará que esse método não contém operadores e, portanto, `dotnet build` será executado de forma `await` síncrona. Você adicionará `await` operadores mais tarde à medida que preencher o método .

3. Substitua o método `Main` pelo seguinte código:

```
static async Task Main(string[] args)
{
    await ProcessRepositories();
}
```

Esse código:

- Altera a assinatura de `Main` adicionando o `async` modificador e alterando o tipo de retorno para `Task` .
- Substitui a `Console.WriteLine` instrução por uma chamada para `ProcessRepositories` que usa a palavra-chave `await` .

4. Na classe `Program` , crie uma instância estática do para lidar com `HttpClient` solicitações e respostas.

```
namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            //...
        }
    }
}
```

5. No método , chame o ponto de GitHub que retorna uma lista de todos os `ProcessRepositories` repositórios na organização do .NET Foundation:

```

private static async Task ProcessRepositories()
{
    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 stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.WriteLine(msg);
}

```

Esse código:

- Configura cabeçalhos HTTP para todas as solicitações:
 - Um `Accept` header para aceitar respostas JSON
 - Um `User-Agent` header. Esses headers são verificados pelo código GitHub servidor e são necessários para recuperar informações de GitHub.
- Chama `HttpClient.GetStringAsync(String)` para fazer uma solicitação da Web e recuperar a resposta. Esse método inicia uma tarefa que faz a solicitação da Web. Quando a solicitação retorna, a tarefa lê o fluxo de resposta e extrai o conteúdo do fluxo. O corpo da resposta é retornado como `String` um , que está disponível quando a tarefa é concluída.
- Aguarda a tarefa para a cadeia de caracteres de resposta e imprime a resposta no console.

6. Adicione duas `using` diretivas na parte superior do arquivo:

```

using System.Net.Http;
using System.Net.Http.Headers;

```

7. Crie o aplicativo e execute-o.

```

dotnet run

```

Não há nenhum aviso de build porque o `ProcessRepositories` agora contém um operador `await` .

A saída é uma exibição longa do texto JSON.

Desserializar o resultado JSON

As etapas a seguir convertem a resposta JSON em objetos C#. Você usa a `System.Text.Json.JsonSerializer` classe para deserializar JSON em objetos .

1. Crie um arquivo chamado `repo.cs` e adicione o seguinte código:

```

using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}

```

O código anterior define uma classe para representar o objeto JSON retornado da API GitHub. Você usará

essa classe para exibir uma lista de nomes de repositório.

O JSON para um objeto de repositório contém dezenas de propriedades, mas somente a `name` propriedade será deserializada. O serializador ignora automaticamente as propriedades JSON para as quais não há nenhuma combinação na classe de destino. Esse recurso facilita a criação de tipos que funcionam com apenas um subconjunto de campos em um pacote JSON grande.

A convenção C# é [capitalizar](#) a primeira letra de nomes de propriedade, mas a propriedade aqui começa com uma letra minúscula porque corresponde exatamente ao que `name` está no JSON. Posteriormente, você verá como usar nomes de propriedade C# que não corresponderem aos nomes de propriedade JSON.

2. Use o serializador para converter JSON em objetos C#. Substitua a chamada para `GetStringAsync(String)` no método com as seguintes `ProcessRepositories` linhas:

```
var streamTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
```

O código atualizado substitui `GetStringAsync(String)` por `GetStreamAsync(String)`. Esse método serializador usa um fluxo em vez de uma cadeia de caracteres como sua origem.

O primeiro argumento para `JsonSerializer.DeserializeAsync< TValue >(Stream, JsonSerializerOptions, CancellationToken)` é uma `await` expressão. `await` As expressões podem aparecer em praticamente qualquer lugar em seu código, embora até agora você as tenha visto apenas 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 é genérico, o que significa que você fornece argumentos de tipo para que tipo de objetos devem `DeserializeAsync` ser criados com base no texto JSON. Neste exemplo, você está desserializando para um `List<Repository>`, que é outro objeto genérico, um `System.Collections.Generic.List<T>`. A `List<T>` classe armazena uma coleção de objetos. O argumento `type` declara o tipo de objetos armazenados no `List<T>`. O argumento `type` é sua `Repository` classe, porque 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:

```
var msg = await stringTask;
Console.WriteLine(msg);
```

pelo código a seguir:

```
foreach (var repo in repositories)
    Console.WriteLine(repo.name);
```

4. Adicione as `using` seguintes diretivas na parte superior do arquivo:

```
using System.Collections.Generic;
using System.Text.Json;
```

5. Execute o aplicativo.

```
dotnet run
```

A saída é uma lista dos nomes dos repositórios que fazem parte do .NET Foundation.

Configurar a desterialização

- Em `repo.cs`, altere a propriedade para e adicione um atributo para especificar como essa `name` propriedade aparece no `Name` `[JsonPropertyName]` JSON.

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

- Adicione o `System.Text.Json.Serialization` namespace às `using` diretivas:

```
using System.Text.Json.Serialization;
```

- Em `Program.cs`, atualize o código para usar a nova capitalização da `Name` propriedade:

```
Console.WriteLine(repo.Name);
```

- Execute o aplicativo.

A saída é a mesma.

Refatorar o código

O método `ProcessRepositories` pode fazer o trabalho assíncrono e retornar uma coleção de repositórios. Altere esse método para `List<Repository>` retornar e move o código que grava as informações no método `Main`.

- Altere a assinatura de `ProcessRepositories` para retornar uma tarefa cujo resultado é uma lista de objetos `Repository`:

```
private static async Task<List<Repository>> ProcessRepositories()
```

- Retorne os repositórios depois de processar a resposta JSON:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

O compilador gera o `Task<T>` objeto para o valor de retorno porque você marcou esse método como `async`.

- Modifique `Main` o método para capturar os resultados e gravar cada nome de repositório no console. O método `Main` agora tem esta aparência:

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

- Execute o aplicativo.

A saída é a mesma.

Desterializar mais propriedades

As etapas a seguir adicionam código para processar mais das propriedades no pacote JSON recebido. Provavelmente, você não deseja processar todas as propriedades, mas adicionar mais algumas demonstra outros recursos do C#.

1. Adicione as seguintes propriedades à `Repository` definição de classe:

```
[JsonPropertyName("description")]
public string Description { get; set; }

[JsonPropertyName("html_url")]
public Uri GitHubHomeUrl { get; set; }

[JsonPropertyName("homepage")]
public Uri Homepage { get; set; }

[JsonPropertyName("watchers")]
public int Watchers { get; set; }
```

Os `Uri` tipos e têm funcionalidades built-in para converter de e para representação de cadeia `int` de caracteres. Nenhum código extra é necessário para desserilizar 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 lançará uma exceção.

2. Atualize `Main` o método para exibir os valores da propriedade:

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    Console.WriteLine();
}
```

3. Execute o aplicativo.

A lista agora inclui as propriedades adicionais.

Adicionar uma propriedade de data

A data da última operação de push é formatada dessa maneira na resposta JSON:

```
2016-02-08T21:27:00Z
```

Esse formato é para Tempo Universal Coordenado (UTC), portanto, o resultado da desserilização é um valor `DateTime` cuja `Kind` propriedade é `Utc`.

Para obter uma data e hora representadas em seu fuso horário, você precisa escrever um método de conversão personalizado.

1. Em `repo.cs`, adicione uma propriedade para a representação UTC da data e hora e uma propriedade que retorna a data convertida para a `public` hora `LastPush` `readonly` local:

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

public DateTime LastPush => LastPushUtc.ToLocalTime();
```

A `LastPush` propriedade é definida usando um membro apto para *expressão* para `get` o acessador. Não há nenhum `set` acessador. Omitir `set` o acessador 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:

```
Console.WriteLine(repo.LastPush);
```

3. Execute o aplicativo.

A saída inclui a data e a hora do último push para cada repositório.

Próximas etapas

Neste tutorial, você criou um aplicativo que faz solicitações da Web e analisará os resultados. Sua versão do aplicativo agora deve corresponder ao [exemplo concluído](#).

Saiba mais sobre como configurar a serialização JSON em [Como serializar e desterializar \(marshal e unmarshal\) JSON no .NET](#).

Trabalhar com Language-Integrated consulta (LINQ)

21/01/2022 • 16 minutes to read

Introdução

Este tutorial ensina os recursos no .NET Core e da linguagem C#. Você aprenderá a:

- Gere sequências com LINQ.
- Escreva métodos que podem ser facilmente usados em consultas LINQ.
- Distinguir entre avaliação ávida e lento.

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 criará constrói um baralho de cartas e, em seguida, executa uma sequência de embaralhamentos, escrevendo a sequência toda vez. 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 pode encontrar as instruções de instalação na página [Download do .NET Core](#). Você pode executar esse aplicativo em Windows, Ubuntu Linux ou OS X ou em um contêiner do Docker. Será necessário instalar o editor de código de sua preferência. As descrições abaixo usam [Visual Studio Code](#) que é um editor de plataforma cruzada de código aberto. 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`:

```
// 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 as valores e naipes como `IEnumerable<T>`s de cadeias de caracteres:

```
// 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:

```
// 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:

```
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:\>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`:

```
// 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:

```
// 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:

```
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 enumera ambas as sequências adquiridas por meio de e de uma vez, os elementos e a criação de uma sequência: seu baralho de cartas agora [Take Skip](#) embaralhado. [interLeaving](#) Escrever um método LINQ que funciona com duas sequências exige que você compreenda como [IEnumarable<T>](#) funciona.

A interface [IEnumarable<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:

```
public static IEnumarable<T> InterleaveSequenceWith<T>
    (this IEnumarable<T> first, IEnumarable<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:

```
// 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:

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!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:

```
// 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:

```
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^a 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.

```
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`:

```
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:

```

// 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 as 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:

```

public static void Main(string[] args)
{
    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 [ToArray](#) 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](#)

Usar atributos em C#

21/01/2022 • 7 minutes to read

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 uma variedade de destinos.

Considere o atributo `[Obsolete]`. 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 verá como adicionar atributos a seu código, como criar e usar seus próprios atributos e como usar alguns atributos que são criados no .NET Core.

Pré-requisitos

Você precisará configurar seu computador para executar o .NET Core. Encontre as instruções de instalação na página [Downloads do .NET Core](#). Você pode executar esse aplicativo no Windows, Ubuntu Linux, macOS ou em um contêiner do Docker. Será necessário instalar o editor de código de sua preferência. As descrições abaixo usam [Visual Studio Code](#) que é um editor de plataforma cruzada de código aberto. No entanto, você pode usar quaisquer ferramentas que esteja familiarizado.

Criar o aplicativo

Agora que você instalou todas as ferramentas, crie um novo aplicativo do .NET Core. Para usar o gerador de linha de comando, execute o seguinte comando no shell de sua preferência:

```
dotnet new console
```

Esse comando criará arquivos de projeto do .NET Core bare-bare-bare. Você precisará executar `dotnet restore` para restaurar as dependências necessárias para compilar esse projeto.

Você não precisa executar `dotnet restore` porque ele é executado implicitamente por todos os comandos que exigem a ocorrência de uma restauração, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` e `dotnet pack`. Para desabilitar a restauração implícita, use a `--no-restore` opção.

O `dotnet restore` comando ainda é útil em determinados cenários em que a restauração explícita faz sentido, como [compilações de integração contínua em Azure DevOps Services](#) ou em sistemas de compilação que precisam controlar explicitamente quando a restauração ocorre.

Para obter informações sobre como gerenciar feeds do NuGet, consulte a [dotnet restore documentação](#).

Para executar o programa, use `dotnet run`. Você deve ver a saída do "Olá, Mundo" no console.

Como 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`. Ele é usado para sinalizar que o código está obsoleto e não deve mais ser usado. Você pode colocar este atributo em uma classe, por exemplo, usando colchetes.

```
[Obsolete]
public class MyClass
{
}
```

Observe que, embora a classe seja chamada de `ObsoleteAttribute`, só é necessário usar `[Obsolete]` no código. Isso é uma convenção que a linguagem C# segue. 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. Faça isso passando um parâmetro de cadeia de caracteres para o atributo obsoleto.

```
[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.

Como criar seu próprio atributo

Criar um atributo é tão simples quanto herdar de uma classe base `Attribute`.

```
public class MySpecialAttribute : Attribute
{
}
```

Com os itens acima, agora posso usar `[MySpecial]` (ou `[MySpecialAttribute]`) como um atributo em qualquer lugar na base do código.

```
[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 (mais sobre isso posteriormente no tutorial).

Há uma “pegadinha” aqui. Conforme mencionado acima, somente determinados tipos podem ser passados como argumentos ao usar atributos. No entanto, ao criar um tipo de atributo, o compilador C# não impedirá você de criar esses parâmetros. No exemplo abaixo, criei um atributo com um construtor que compila bem.

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str) {
    }
}
```

No entanto, não será possível usar esse construtor com a sintaxe de atributo.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{}
```

O descrito acima causará 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 em um número de "destinos". 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# permitirá 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!

```
[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 obterá um erro do compilador como

```
Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on
'class, struct' declarations
```

```
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 farão nada.

Para localizar e agir sobre os atributos, geralmente é necessário [Reflexão](#). Não abordarei Reflexão detalhadamente neste tutorial, mas a ideia básica é a Reflexão permite que você escreva um código em C# que examine outro código.

Por exemplo, você pode usar a Reflexão para obter informações sobre uma classe (adicione

```
using System.Reflection;
```

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

Isso imprimirá 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`, etc.), você poderá usar o método `GetCustomAttributes`. Isso retornará 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).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Isso será impresso 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 na BCL (biblioteca de classes base)

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`, todas as chamadas a esse método (mas não o próprio método) serão removidas pelo compilador C#. Normalmente, isso é usado para depuração (diagnóstico).
- `[CallerMemberName]`. Esse atributo pode ser usado em parâmetros e reside no namespace `System.Runtime.CompilerServices`. Este é um atributo que é usado para injetar o nome do método que está chamando outro método. Isso normalmente é usado como uma forma de eliminar 'cadeias de caracteres mágicas' ao implementar `INotifyPropertyChanged` em diversas estruturas de interface do usuário. Por exemplo:

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        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. Isso pode ajudar a evitar erros relacionados à digitação e também possibilita a refatoração/renomeação mais suave.

Resumo

Atributos trazem poder declarativo para C#, mas eles são uma forma de metadados de código e não agem sozinhos.

Tipos de referência anuláveis

21/01/2022 • 18 minutes to read

Antes do C# 8.0, todos os tipos de referência eram anulados. *Tipos de referência que podem ser anulados* refere-se a um grupo de recursos introduzidos no C# 8.0 que você pode usar para minimizar a probabilidade de que seu código cause o runtime a lançar [System.NullReferenceException](#). *Os tipos de referência que podem ser anulados* incluem três recursos que ajudam a evitar essas exceções, incluindo a capacidade de marcar explicitamente um tipo de referência como *anulado*:

- Análise de fluxo estático aprimorada que determina se uma variável pode estar `null` antes de desreferenciá-la.
- Atributos que anotam APIs para que a análise de fluxo determine *o estado nulo*.
- Anotações de variável que os desenvolvedores usam para declarar explicitamente o estado *nulo pretendido* para uma variável.

As anotações de variáveis e análise de estado nulo são desabilitadas por padrão para projetos existentes, o que significa que todos os tipos de referência continuam sendo — anulados. 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* que permite valor nulo, consulte [Contextos que permitem valor nulo](#).

O restante deste artigo descreve como essas três áreas de recurso funcionam para produzir avisos quando seu código pode **estar desreferenciando** um `null` valor. Desreferenciar uma variável significa acessar um de seus membros usando o operador `.` (ponto), conforme mostrado no exemplo a seguir:

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

Quando você desreferencia uma variável cujo valor é `null`, o runtime lança um [System.NullReferenceException](#).

Análise de estado nulo

Análise de estado nulo _ rastreia _ null-state de referências. Essa análise estática emite avisos quando seu código pode desreferenciar `null`. Você pode resolver esses avisos para minimizar as reações quando o runtime lança um [System.NullReferenceException](#). O compilador usa a análise estática para determinar o *estado nulo* de uma variável. Uma variável não é *nula ou talvez nula*. O compilador determina que uma variável não é *nula* de duas maneiras:

1. A variável foi atribuída a um valor que é conhecido como *não nulo*.
2. A variável foi verificada `null` e não foi modificada desde essa verificação.

Qualquer variável que o compilador não tenha determinado como *não nulo* é considerada *talvez nula*. A análise fornece avisos em situações em que você pode accidentalmente desreferenciar um `null` valor. O compilador produz avisos com base no *estado nulo*.

- Quando uma variável *não é nula*, essa variável pode ser desreferenciada com segurança.
- Quando uma variável é *talvez nula*, essa variável deve ser verificada para garantir que não seja antes `null` de desreferenciá-la.

Considere o exemplo a seguir:

```

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 *talvez nulo* quando a primeira mensagem é impressa. Não há nenhum aviso para a segunda mensagem. A linha final de código produz um aviso porque `originalMessage` pode ser nulo. O exemplo a seguir mostra um uso mais prático para percorrer uma árvore de nós para a raiz, processando cada nó durante a travessia:

```

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 desreferenciado quando é *talvez nulo*. A `current` variável é verificada `null` antes `current.Parent` de ser acessada e antes de passar `current` para a `ProcessNode` ação. Os exemplos anteriores mostram como o compilador determina o estado nulo para variáveis locais quando inicializado, atribuído ou comparado a `null`.

NOTE

Foram adicionadas várias melhorias à atribuição definida e à análise de estado nulo no C# 10. Ao atualizar para o C# 10, você pode encontrar menos avisos anulados que são falsos positivos. Você pode saber mais sobre as melhorias na especificação de [recursos para aprimoramentos de atribuição definidos](#).

Atributos em assinaturas de API

A análise de estado nulo precisa de dicas de desenvolvedores para entender a semântica das APIs. Algumas APIs fornecem verificações nulas e devem alterar o estado nulo de uma variável de *talvez nulo* para *não nulo*. Outras APIs retornam expressões que não *são nulas* ou *talvez nulas*, dependendo do estado *nulo* dos argumentos de entrada. Por exemplo, considere o seguinte código que exibe uma mensagem:

```

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 fornece `IsNullOrWhiteSpace` uma verificação nula. Você aplica atributos para informar

ao compilador que `message` não é *nulo* se e somente se `IsNullOrEmptySpace` retornar `false`. No exemplo anterior, a assinatura inclui o `[NotNullWhen(false)]` para indicar o estado nulo de `message`:

```
public static bool IsNullOrEmptySpace([NotNullWhen(false)] string message);
```

Os atributos fornecem informações detalhadas sobre o estado nulo de argumentos, valores de retorno e membros da instância de objeto usada para invocar um membro. Os detalhes sobre cada atributo podem ser encontrados no artigo de referência de linguagem sobre atributos [de referência que podem ser anulados](#). Todas as APIs de runtime do .NET foram anotadas no .NET 5. Você melhora a análise estática anotando suas APIs para fornecer informações semânticas sobre o estado nulo de argumentos e valores de retorno.

Anotações de variável que podem ser anuladas

A *análise de estado nulo* fornece análise robusta para a maioria das variáveis. O compilador precisa de mais informações suas para variáveis de membro. O compilador não pode fazer suposições sobre a ordem em que 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 pode ser definido como `null`, o compilador deve assumir que seu estado nulo é `null talvez nulo` no início de cada método.

Você usa anotações que podem declarar se uma variável é um tipo de referência que pode ser anulada ou um tipo de referência não anulada. Essas anotações fazem instruções importantes sobre o estado *nulo* para variáveis:

- **Uma referência não deve ser nula.** O estado padrão de uma variável de referência não anulada não é *nulo*. O compilador impõe regras que garantem que seja seguro desreferenciar essas variáveis sem primeiro verificar se ela não é nula:
 - A variável deve ser inicializada para um valor não nulo.
 - A variável nunca pode receber o valor `null`. O compilador emite um aviso quando o código atribui uma expressão *talvez nula* a uma variável que não deve ser nula.
- **Uma referência pode ser nula.** O estado padrão de uma variável de referência que pode ser anulada é *talvez nulo*. O compilador impõe regras para garantir que você tenha verificado corretamente uma `null` referência:
 - A variável só poderá ser desreferenciada quando o compilador puder garantir que o valor não seja `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 *talvez nula* a uma variável que pode ser nula.

Qualquer variável de referência que não deveria ser tem um estado `null nulo de não nulo`. Qualquer variável de referência que possa `null` ser inicialmente tem o estado *nulo 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`:

```
string? name;
```

Qualquer variável em que o não é anexado 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 habilitar esse recurso. No entanto, todas as variáveis locais de tipo implícito (declaradas usando `var`) são tipos de referência

anuladas. Como as seções anteriores mostraram, a análise estática determina o estado nulo das variáveis locais para determinar se elas são *talvez nulas*.

Às vezes, você deve substituir um aviso quando sabe que uma variável não é nula, mas o compilador determina que seu estado *nulo* é *talvez nulo*. Use o operador [null-operator](#) após um nome de variável para forçar o estado ! *nulo a não ser nulo*. Por exemplo, se você sabe que a variável não é, mas o compilador emite um aviso, você pode escrever o código a seguir para substituir `name` `null` a análise do compilador:

```
name!.Length;
```

Tipos de referência que anuláveis e tipos de valor anulado fornecem um conceito semântico semelhante: uma variável pode representar um valor ou objeto ou essa variável pode ser `null`. No entanto, tipos de referência que anuláveis e tipos de valor anuáveis são implementados de forma diferente: os tipos de valor anulado são implementados usando e os tipos de referência que anuáveis são implementados por atributos lidos [System.Nullable<T>](#) pelo compilador. Por exemplo, `string?` e `string` são representados pelo mesmo tipo: [System.String](#). No entanto, `int?` `int` e são representados por e, [System.Nullable<System.Int32>](#) [System.Int32](#) respectivamente.

IMPORTANT

A habilitação de anotações que permitem valor nulo pode alterar Entity Framework Core determina se um membro de dados é necessário. Você pode saber mais detalhes no artigo sobre [conceitos básicos Entity Framework Core: trabalhando com tipos de referência que podem ser anulados](#).

Genéricos

Genéricos exigem regras detalhadas para lidar `T?` com qualquer parâmetro de tipo `T`. As regras são necessariamente detalhadas devido ao histórico e à implementação diferente para um tipo de valor anulado e um tipo de referência que pode ser anulado. [Os tipos de valor que podem ser anulados](#) são implementados usando o [System.Nullable<T>](#) struct. [Os tipos de referência que podem ser anulados](#) são implementados como anotações de tipo que fornecem regras semânticas para o compilador.

No C# 8.0, o uso sem restrições para ser `T?` um ou um não foi `T struct class` compilado. Isso permitiu que o compilador interpretava `T?` claramente. Essa restrição foi removida no C# 9.0, definindo as seguintes regras para um parâmetro de tipo ir pouco `T` restrito:

- Se o argumento de tipo para um tipo de referência, referencia o tipo de referência que pode ser `T T?` nulo correspondente. Por exemplo, se `T` for um , então será um `string T? string?`.
- Se o argumento de tipo para `T` for um tipo de valor, referencia o mesmo tipo de `T?` valor, `T`. Por exemplo, se `T` for um , o também será um `int T? int`.
- Se o argumento de tipo para um tipo de referência que pode ser `T` anulado, `T?` referencia esse mesmo tipo de referência que pode ser anulado. Por exemplo, se `T` for um , também será um `string? T? string?`
- Se o argumento de tipo para um tipo de valor que pode ser `T` anulado, `T?` referencia esse mesmo tipo de valor que pode ser nulo. Por exemplo, se `T` for um , também será um `int? T? int?`.

Para valores de retorno, `T?` é equivalente a ; para valores de `[MaybeNull]T` argumento, é equivalente a `T? [AllowNull]T` . Para obter mais informações, consulte o artigo sobre [Atributos para análise de estado nulo](#) na referência de linguagem.

Você pode especificar um comportamento diferente usando [restrições](#):

- A restrição significa que deve ser um tipo de referência `class T` não anulada (por `string` exemplo,). O

compilador produzirá um aviso se você usar um tipo de referência que pode ser anulado, como `string?` para `T`.

- A restrição significa que deve ser um tipo de referência, não anuável () ou um tipo de referência que pode ser `class?` `T` anulado `string` (por `string?` exemplo). Quando o parâmetro type é um tipo de referência que pode ser anulado, como , uma expressão de faz referência ao mesmo tipo de referência que pode ser `string?` `T?` anulado, como `string?` .
- A restrição significa que deve ser um tipo de referência não `notnull` `T` anulada ou um tipo de valor não anulado. Se você usar um tipo de referência que pode ser anulado ou um tipo de valor anulado para o parâmetro de tipo, o compilador produzirá um aviso. Além disso, quando é um tipo de valor, o valor de retorno é esse tipo de valor, não o tipo de valor `T` anulado correspondente.

Essas restrições ajudam a fornecer mais informações ao compilador sobre como `T` será usado. Isso ajuda quando os desenvolvedores escolhem o tipo para e fornece uma melhor análise de estado nulo quando uma `T` instância do tipo genérico é usada.

Contextos que permitem valor nulo

Os novos recursos que protegem contra a geração de um `System.NullReferenceException` podem ser interrupções quando ligado em uma base de código existente:

- Todas as variáveis de referência explicitamente digitadas são interpretadas como tipos de referência não anuladas.
- O significado da `class` restrição em genéricos foi alterado para significar um tipo de referência que não pode ser anulado.
- Novos avisos são gerados devido a essas novas regras.

Você deve optar explicitamente por usar esses recursos em seus projetos existentes. Isso fornece um caminho de migração e preserva a compatibilidade com compatibilidade com vert. 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 anulada** determina o comportamento do compilador. Há quatro valores para o contexto de anotação que pode ser anulado:

- *disabled*: o compilador se comporta da mesma forma que o C# 7.3 e anterior:
 - Avisos que podem ser anulados estão desabilitados.
 - Todas as variáveis de tipo de referência são tipos de referência que podem ser anuladas.
 - Não é possível declarar uma variável como um tipo de referência que pode ser anulada usando `?` o sufixo no tipo .
 - Você pode usar o operador de anulação `!` nula, , mas ele não tem nenhum efeito.
- *habilitado*: o compilador habilita toda a análise de referência nula e todos os recursos de linguagem.
 - Todos os novos avisos que permitem valor nulo estão habilitados.
 - Você pode usar o `?` sufixo para declarar um tipo de referência que pode ser anulado.
 - Todas as outras variáveis de tipo de referência são tipos de referência não anuladas.
 - O operador null null null suprime avisos para uma possível atribuição a `null` .
- *avisos*: o compilador executa toda a análise nula e emite avisos quando o código pode desreferenciar `null` .
 - Todos os novos avisos que permitem valor nulo estão habilitados.
 - O uso do sufixo para declarar um tipo de referência que pode ser `?` anulado produz um aviso.
 - Todas as variáveis de tipo de referência têm permissão para serem nulas. No entanto, os membros têm o *estado* nulo de *não nulo* na chave de abertura de todos os métodos, a menos que declarados com o `?` sufixo .
 - Você pode usar o operador de anulação nula, `!` .
- *anotações*: o compilador não executa a análise nula nem emite avisos quando o código pode desreferenciar

`null`.

- Todos os novos avisos que podem ser anulados estão desabilitados.
- Você pode usar o `?` sufixo para declarar um tipo de referência que pode ser anulado.
- Todas as outras variáveis de tipo de referência são tipos de referência não anuladas.
- Você pode usar o operador de anulação `!` nula, , mas ele não tem nenhum efeito.

O contexto de anotação anuável e o contexto de aviso anulado podem ser definidos para um projeto usando o elemento `<Nullable>` em seu arquivo `.csproj`. Esse elemento configura como o compilador interpreta a nulidade de tipos e quais avisos são emitidos. A tabela a seguir mostra os valores permitidos e resume os contextos especificados.

CONTEXTO	AVISOS DE DESREFERÊNCIA	AVISOS DE ATRIBUIÇÃO	TIPOS DE REFERÊNCIA	<code>? </code> SUFIXO	<code>! </code> OPERADOR
<code>disable</code>	Desabilitado	Desabilitado	Todos são anuáveis	Não pode ser usado	Não tem nenhum efeito
<code>enable</code>	habilitado	habilitado	Não anulável, a menos que declarado com <code>? </code>	Declara o tipo que pode ser anulado	Suprime avisos para uma possível <code>null</code> atribuição
<code>warnings</code>	habilitado	Não aplicável	Todos são anulados, mas os membros são considerados <i>não nulos na chave de abertura de métodos</i>	Produz um aviso	Suprime avisos para uma possível <code>null</code> atribuição
<code>annotations</code>	Desabilitado	Desabilitado	Não anulável, a menos que declarado com <code>? </code>	Declara o tipo que pode ser anulado	Não tem nenhum efeito

As variáveis de tipo de referência no código compilado antes do C# 8 ou em um contexto desabilitado são *anuladas*. Você pode atribuir uma variável literal ou talvez nula a uma variável que é `null` *anulada*. No entanto, o estado padrão de uma *variável anulada* não é *nulo*.

Você pode escolher qual configuração é melhor para seu projeto:

- Escolha *desabilitar* para projetos herdados que você não deseja atualizar com base em diagnóstico ou novos recursos.
- Escolha *avisos* para determinar onde seu código pode `System.NullReferenceException` lançar s. Você pode resolver esses avisos antes de modificar o código para habilitar tipos de referência que não permitem valor nulo.
- Escolha *anotações para expressar sua* intenção de design antes de habilite avisos.
- Escolha *habilitar* para novos projetos e projetos ativos em que você deseja proteger contra exceções de referência nula.

Exemplo:

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

Você também pode usar diretivas para definir esses mesmos contextos em qualquer lugar no código-fonte. Eles

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 para **habilitar**.
- `#nullable disable` : define o contexto de anotação que pode ser anulado e o contexto de aviso anulado para **desabilitar**.
- `#nullable restore` : restaura o contexto de anotação que pode ser anulado e o contexto de aviso que pode ser anulado para as configurações do projeto.
- `#nullable disable warnings` : de definir o contexto de aviso que pode ser anulado para **desabilitar**.
- `#nullable enable warnings` : de definir o contexto de aviso que permite valor nulo para **habilitar**.
- `#nullable restore warnings` : restaura o contexto de aviso que pode ser anulado para as configurações do projeto.
- `#nullable disable annotations` : de definir o contexto de anotação que pode ser anulado para **desabilitar**.
- `#nullable enable annotations` : de definir o contexto de anotação que permite valor nulo para **habilitar**.
- `#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	USO
padrão do projeto	padrão do projeto	Padrão
enable	disable	Corrigir avisos de análise
enable	padrão do projeto	Corrigir avisos de 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 herdados ao projeto migrado
padrão do projeto	disable	Raramente
disable	padrão do projeto	Raramente

Essas nove combinações fornecem controle apurado sobre o diagnóstico 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.

IMPORTANT

O contexto anulado global não se aplica a arquivos de código gerados. Em qualquer estratégia, o contexto que pode ser *anulado* é *desabilitado* para qualquer 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 em um comentário na parte superior do `<auto-generated/>` arquivo. Ele pode estar em qualquer linha nesse comentário, mas o bloco de comentário deve ser o primeiro elemento no arquivo.
3. Inicie o nome do arquivo com `TemporaryGeneratedFile_`
4. End the file name with `.designer.cs`, `.generated.cs`, `.g.cs` ou `.g.i.cs`.

Os geradores podem optar por usar a `#nullable` diretiva de pré-processador.

Por padrão, os contextos de aviso e anotação anuáveis são **desabilitados**. Isso significa que o código existente é compilado sem alterações e sem gerar novos avisos. Começando com o .NET 6, novos projetos incluem `<Nullable>enable</Nullable>` o elemento em todos os modelos de projeto.

Essas opções fornecem duas estratégias distintas para [atualizar uma base de código](#) existente para usar tipos de referência que podem ser anulados.

Armadilhas conhecidas

Matrizes e structs que contêm tipos de referência são armadilhas conhecidas em referências que podem ser anuladas e a análise estática que determina a segurança nula. Em ambas as situações, uma referência não anulada pode ser inicializada para `null`, sem gerar avisos.

Estruturas

Um struct que contém tipos de referência que não permitem valor nulo permite atribuir `default` a ele sem avisos. Considere o exemplo a seguir:

```
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 enquanto os tipos de referência não `PrintStudent(default)` anuáveis `FirstName` e `LastName` são nulos.

Outro caso mais comum é quando você lida com structs genéricos. Considere o exemplo a seguir:

```

#ifndef 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 será em tempo de operação e será atribuída a uma cadeia de caracteres não `Bar` `null` anulada sem avisos.

Matrizes

As matrizes também são uma armadilha conhecida em tipos de referência que podem ser anulados. Considere o exemplo a seguir que não produz nenhum aviso:

```

using System;

#ifndef 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 anuladas, enquanto seus elementos são todos inicializados como `null`. Em seguida, a `s` variável recebe um `null` valor (o primeiro elemento da matriz). Por fim, a `s` variável é desreferenciada, causando uma exceção de runtime.

Confira também

- [Proposta de tipos de referência que podem ser anuladas](#)
- [Especificação de tipos de referência que podem ser anulados de rascunho](#)
- [Anotações de parâmetro de tipo sem restrição](#)
- [Introdução ao tutorial de referências que permitem valor nulo](#)
- [Nullable \(opção do compilador C#\)](#)

Atualizar uma codebase com tipos de referência anuláveis para melhorar avisos de diagnóstico nulos

21/01/2022 • 7 minutes to read

Os [tipos de referência anuláveis](#) permitem que você declare se as variáveis de um tipo de referência devem ou não ter um `null` valor atribuído. A análise estática do compilador e os avisos quando seu código pode ser desreferenciado `null` são o benefício mais importante desse recurso. Uma vez habilitado, o compilador gera avisos que ajudam a evitar lançar um [System.NullReferenceException](#) quando seu código é executado.

Se a base de código for relativamente pequena, você poderá ativar o [recurso em seu projeto](#), tratar avisos e aproveitar os benefícios do diagnóstico aprimorado. Bases de código maiores podem exigir uma abordagem mais estruturada para tratar avisos ao longo do tempo, habilitando o recurso para alguns à medida que você tratar 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, os valores de *estado nulo* de *talvez-nulo* e *não-nulo* e as anotações anuláveis. Quando estiver familiarizado com esses conceitos e termos, você estará pronto para migrar seu código.

Planeje sua migração

Independentemente de como você atualiza sua base de código, o objetivo é que os avisos anuláveis e anotações anuláveis sejam habilitadas em seu projeto. Depois de atingir essa meta, você terá a `<nullable>Enable</nullable>` configuração em seu projeto. Você não precisará de nenhum dos pragmas para ajustar as configurações em outro lugar.

A primeira opção é definir o padrão para o projeto. Suas opções são:

1. ***O valor nulo desabilita como o padrão `_:_disable`*** é o padrão se você não adicionar um `Nullable` elemento ao arquivo de projeto. Use esse padrão quando você não estiver adicionandoativamente novos arquivos à base de código. A atividade principal é atualizar a biblioteca para usar tipos de referência anuláveis. Usar esse padrão significa adicionar um pragma anulável a cada arquivo à medida que você atualiza seu código.
2. **Habilitação anulável como padrão**: defina esse padrão quando estiver desenvolvendoativamente novos recursos. Você quer todo o novo código para beneficiar tipos de referência anuláveis e análise estática anulável. Usar esse padrão significa que você deve adicionar um `#pragma nullable disable` à parte superior de cada arquivo. Você removerá esse pragma à medida que começar a abordar os avisos nesse arquivo.
3. **Avisos anuláveis como o padrão `_:_`**: escolha esse padrão para uma migração de duas fases. Na primeira fase, resolva os avisos. Na segunda fase, ative as anotações para declarar o `_null` estado esperado de uma variável *. Usar esse padrão significa que você deve adicionar um `#pragma nullable disable` à parte superior de cada arquivo.
4. **Anotações que permitem valor nulo** como padrão. Anote o código antes de abordar os avisos.

Habilitar `Nullable` como o padrão cria mais trabalho antecipado para adicionar o pragma a cada arquivo. A vantagem é que todos os novos arquivos de código adicionados ao projeto serão habilitados para permitir valor nulo. Qualquer trabalho novo terá reconhecimento de `Nullable`; somente o código existente deve ser atualizado. Desabilitar `Nullable` como padrão funcionará melhor se a biblioteca estiver estável e o foco principal do desenvolvimento é adotar tipos de referência anuláveis. Você ativa os tipos de referência anuláveis ao anotar APIs. Quando tiver terminado, habilite os tipos de referência anuláveis para o projeto inteiro. Ao criar um novo arquivo, você deve adicionar os pragmas e torná-lo indesejado de forma anulável. Se algum desenvolvedor da

sua equipe esquecer, esse novo código estará no registro posterior do trabalho para tornar todos os incompatíveis com o código anulável.

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

IMPORTANT

O contexto anulável global não se aplica a arquivos de código gerados. Em qualquer estratégia, o contexto anulável é *desabilitado* para qualquer arquivo de origem marcado como gerado. Isso significa que qualquer API em arquivos gerados não é anotada. Há quatro maneiras de um arquivo ser marcado como gerado:

1. Em `editorconfig`, especifique `generated_code = true` em uma seção que se aplica a esse arquivo.
2. Put `<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 comentário deve ser o primeiro elemento no 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 optar por usar a `#nullable` diretiva de pré-processador.

Entender contextos e avisos

Habilitar avisos e anotações controla como o compilador exibe os tipos de referência e a nulidade. Cada tipo tem um dos três nullabilities:

- *alheios*: todos os tipos de referência são anuláveis *alheios* quando o contexto de anotação é desabilitado.
- Não *nulo*: um tipo de referência não anotado, `c`, não é *nulo* quando o contexto de anotação está habilitado.
- *Nullable*: um tipo de referência anotada, `c?`, é *anulável*, mas um aviso pode ser emitido quando o contexto de anotação é desabilitado. Variáveis declaradas com `var` são *anuláveis* quando o contexto de anotação está habilitado.

O compilador gera avisos com base nessa nulidade:

- tipos não *nulos* causam avisos se um `null` valor potencial é atribuído a eles.
- tipos *anuláveis* causam avisos se eles forem desreferenciados quando *-talvez-NULL*.
- os tipos *alheios* causam avisos se eles forem desreferenciados quando *-NULL* e o contexto de aviso estiver habilitado.

Cada variável tem um estado anulável padrão que depende da sua nulidade:

- Variáveis anuláveis têm um *estado nulo* padrão de *talvez-NULL*.
- Variáveis não anuláveis têm um *estado nulo* padrão de *não NULL*.
- Variáveis alheios anuláveis têm um *estado nulo* padrão de *não NULL*.

Antes de habilitar tipos de referência anuláveis, todas as declarações em sua codebase são *alheios anuláveis*. Isso é importante porque significa que todos os tipos de referência têm um *estado nulo* padrão de *não NULL*.

Avisos de endereço

Se o seu projeto usa Entity Framework Core, você deve ler suas diretrizes sobre como [trabalhar com tipos de referência anuláveis](#).

Ao iniciar a migração, você deve começar habilitando apenas avisos. Todas as declarações permanecem *alheios anuláveis*, mas você verá avisos quando você desreferenciar um valor após seu *estado nulo* mudar para *talvez-NULL*. Conforme você resolve esses avisos, você verificará em relação a nulo em mais locais e sua codebase se

tornará mais resiliente. Para aprender técnicas específicas para diferentes situações, consulte o artigo sobre [técnicas para resolver avisos anuláveis](#).

Você pode endereçar avisos e habilitar anotações em cada arquivo ou classe antes de continuar com outro código. No entanto, geralmente é mais eficiente abordar os avisos gerados enquanto o contexto é *avisos* antes de habilitar as anotações de tipo. Dessa forma, todos os tipos são *alheios* até que você tenha abordado 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 *alheios* para *nonnull*. Todas as variáveis declaradas com `var` são *anuláveis*. Essa alteração geralmente apresenta novos avisos. A primeira etapa para abordar os avisos do compilador é usar `?` anotações nos tipos de parâmetro e de retorno para indicar quando argumentos ou valores de retorno podem ser `null`. Conforme você faz essa tarefa, seu objetivo não é apenas corrigir avisos. A meta mais importante é fazer com que o compilador entenda sua intenção por possíveis valores nulos.

Atributos estendem anotações de tipo

Vários atributos foram adicionados para expressar informações adicionais sobre o estado nulo de variáveis. As regras para suas APIs são provavelmente mais complicadas do que *não-nulas* ou *talvez-nulas* para todos os parâmetros e valores de retorno. Muitas de 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 de sua API são encontrados no artigo sobre [atributos que afetam a análise anulável](#).

Próximas etapas

Depois de solucionar todos os avisos depois de habilitar as anotações, você pode definir o contexto padrão para que seu projeto seja *habilitado*. Se você adicionou quaisquer pragmas em seu código para a anotação anulável ou o contexto de aviso, poderá removê-los. Ao longo do tempo, você poderá ver novos avisos. Você pode escrever código que apresente avisos. Uma dependência de biblioteca pode ser atualizada para tipos de referência anuláveis. Essas atualizações alterarão os tipos nessa biblioteca de *alheios anulável* para não *nulo* ou *anulável*.

Aprenda técnicas para resolver avisos que podem ser anulados

21/01/2022 • 8 minutes to read

A finalidade dos tipos de referência que podem ser anulados é minimizar a chance de seu aplicativo lançar um [System.NullReferenceException](#) quando executado. Para atingir essa meta, o compilador usa análise estática e emite avisos quando seu código tem construções que podem levar a exceções de referência nula. Você fornece ao compilador informações para sua análise estática aplicando anotações e atributos de tipo. Essas anotações e atributos descrevem a nulidade de argumentos, parâmetros e membros de seus tipos. Neste artigo, você aprenderá técnicas diferentes para resolver os avisos que podem ser anulados que o compilador gera de sua análise estática. As técnicas descritas aqui são para código C# geral. Saiba como trabalhar com tipos de referência que podem ser anulados e Entity Framework núcleo em Trabalhando com tipos de referência que podem ser [anulados](#).

Você abordará quase todos os avisos usando uma das quatro técnicas:

- Adicionar verificações nulas necessárias.
- Adicionando `? !` anotações que podem ser anuladas ou anuladas.
- Adicionando atributos que descrevem semântica nula.
- Inicializando corretamente as variáveis.

Possível desreferência de null

Um conjunto de avisos alerta que você está desreferenciando uma variável cujo estado *nulo* é *talvez nulo*. Um exemplo pode ser:

```
string message = null;
Console.WriteLine(message.Length);
```

Para remover esses avisos, você precisa adicionar código para alterar o estado nulo dessa variável para *não nulo* antes de desreferenciá-lo.

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 reescrito como:

```
string message = null;
if (message is not null)
{
    Console.WriteLine(message.Length);
}
```

Quando o código gerar um aviso de que ele pode estar desreferenciando uma referência *talvez nula*, verifique se você fez uma verificação nula. Se você não fez isso, adicione um. O aviso do compilador ajudou você a resolver um possível bug.

Outras instâncias quando você recebe esses avisos podem ser falsos positivos. Você pode ter um método de utilitário privado que testa se há nulo. O compilador não sabe que o método fornece uma verificação nula. Considere o exemplo a seguir que usa um método de utilitário privado, `IsNotNull` :

```
public void WriteMessage(string? message)
{
    if (IsNotNull(message))
        Console.WriteLine(message.Length);
}
```

O compilador avisa que você pode estar desreferenciando nulo ao gravar a propriedade porque sua análise estática determina `message.Length` que `message` pode ser `null`. Você pode saber que `IsNotNull` fornece uma verificação nula e, quando ela retorna `true`, o estado *nulo* de `message` deve ser *não nulo*. Você deve dizer ao compilador esses fatos. Uma maneira é usar o operador de anulação nula, `!`. Você pode alterar a `WriteLine` instrução para corresponder ao seguinte código:

```
Console.WriteLine(message!.Length);
```

O operador `null null null` torna a expressão *não nula*, mesmo se ela *fosse talvez-nula* sem `!` o aplicado. Neste exemplo, uma solução melhor é adicionar um atributo à assinatura de `IsNotNull`:

```
private static bool IsNotNull([NotNullWhen(true)] object? obj) => obj != null;
```

O [System.Diagnostics.CodeAnalysis.NotNullWhenAttribute](#) informa ao compilador que o argumento usado para o `obj` parâmetro não é *nulo* quando o método retorna `true`. Quando o método retorna `false`, o argumento tem o mesmo estado `false` *nulo* que tinha antes de o método ser chamado.

TIP

Há um conjunto rico de atributos que você pode usar para descrever como seus métodos e propriedades afetam o *estado nulo*. Você pode aprender sobre eles no artigo de referência de linguagem sobre atributos de análise [estática anuladas](#).

Corrigir um aviso para desreferenciar uma *variável talvez nula* envolve uma das três técnicas:

- Adicione uma verificação nula ausente.
- Adicione atributos de análise nulo em APIs para afetar a análise estática de *estado nulo* do compilador. Esses atributos informam o compilador quando um valor ou argumento de retorno deve ser *talvez nulo* ou *não nulo* depois de chamar o método.
- Aplique o operador de anulação `!` nula à expressão para forçar o estado a *não nulo*.

Possível valor nulo atribuído a uma referência não anulada

O compilador emite esses avisos quando você tenta atribuir uma expressão que pode ser nula a uma variável que não pode ser anulada. Por exemplo:

```
string? TryGetMessage(int id) => "";

string msg = TryGetMessage(42); // Possible null assignment.
```

Você pode tomar uma das três ações para resolver esses avisos. Uma delas é adicionar a `?` anotação para tornar a variável um tipo de referência que pode ser anulada. Essa alteração pode causar outros avisos. Alterar uma variável de uma referência não anulada para uma referência que pode ser anulada altera seu estado nulo padrão de *not-null* para *maybe-null*. A análise estática do compilador pode encontrar instâncias em que você desreferencia uma variável que seja *talvez nula*.

As outras ações instruem o compilador de que o lado direito da atribuição não é *nulo*. A expressão no lado

direito pode ser marcada como nula antes da atribuição, conforme mostrado no exemplo a seguir:

```
string notNullMsg = TryGetMessage(42) ?? "Unknown message id: 42";
```

Os exemplos anteriores demonstram a atribuição ao valor de retorno de um método. Você pode anotar o método (ou propriedade) para indicar quando um método retorna um valor não nulo. O [System.Diagnostics.CodeAnalysis.NotNullIfNotNullAttribute](#) geralmente especifica que um valor de retorno não é *nulo* quando um argumento de entrada *não é nulo*. Outra alternativa é adicionar o operador de anulação nula ao `!` lado direito:

```
string msg = TryGetMessage(42)!;
```

Corrigir um aviso para atribuir uma expressão *talvez nula* a uma variável *não nula* envolve uma das quatro técnicas:

- Altere o lado esquerdo da atribuição para um tipo que pode ser anulado. Essa ação pode introduzir novos avisos quando você desreferenciar essa variável.
- Forneça uma verificação nula antes da atribuição.
- Anote a API que produz o lado direito da atribuição.
- Adicione o operador null ao lado direito da atribuição.

Referência não anulada não inicializada

Outros avisos são gerados quando uma variável de referência não anulada não é inicializada quando declarada ou em um construtor. Considere a seguinte classe como um exemplo:

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

Nem `FirstName` nem `LastName` são inicializados garantidos. Se esse código for novo, considere alterar a interface pública. O exemplo acima pode ser atualizado da seguinte forma:

```
public class Person
{
    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Se você precisar criar um objeto antes de definir o nome, poderá inicializar as propriedades usando `Person` um valor não nulo padrão:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;
    public string LastName { get; set; } = string.Empty;
}
```

Outra alternativa pode ser alterar esses membros para tipos de referência que podem ser anulados. A `Person` classe pode ser definida da seguinte forma se deve ser permitido para o `null` nome:

```
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. Os [System.Diagnostics.CodeAnalysis.MemberNotNullAttribute](#) atributos [System.Diagnostics.CodeAnalysis.MemberNotNullWhenAttribute](#) e informam ao compilador que um membro *não é nulo* depois que o método foi chamado.

Por fim, você pode usar o operador de anulação nula para indicar que um membro é inicializado em outro código. Para outro exemplo, considere as seguintes classes que representam um Entity Framework Core modelo:

```
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 está definida como *um valor não nulo*. Na verdade, a base `DbContext` executa a inicialização do conjunto. A análise estática do compilador não escolhe isso. Para obter mais informações sobre como trabalhar com tipos de referência que podem ser anulados e Entity Framework Core, consulte o artigo sobre Como trabalhar com tipos de referência que podem ser anulados [no EF Core](#).

Corrigir um aviso para não inicializar um membro não anular envolve uma das quatro técnicas:

- Altere os construtores ou inicializadores de campo para garantir que todos os membros não anulados sejam inicializados.
- Altere um ou mais membros para que sejam tipos que podem ser anulados.
- Anote os métodos auxiliares para indicar quais membros são atribuídos.
- Adicione um inicializador a `null!` para indicar que o membro é inicializado em outro código.

Incompatibilidade na declaração de nulidade

Outros avisos indicam incompatibilidades de nulidade entre assinaturas para métodos, delegados ou parâmetros de tipo. Por exemplo:

```
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 `virtual` método em uma classe base e um com `override` nulidade diferente. A classe base retorna uma cadeia de caracteres não anulada, mas a classe derivada retorna uma cadeia de caracteres que pode ser anulada. Se os `string` e `string?` são invertidos, ele seria 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 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 a implementação desse método. Ou um tipo delegado e a expressão para esse delegado podem ser diferentes. Um parâmetro de tipo e o argumento de tipo podem ser diferentes na nulidade.

Para corrigir esses avisos, atualize a declaração apropriada.

O código não corresponderá à declaração de atributo

As seções anteriores discutiram como você pode usar Atributos para análise estática anulada para informar o compilador sobre a semântica nula do seu código. O compilador avisa se o código não cumpre as promessas desse atributo. Considere o seguinte método:

```
public bool TryGetMessage(int id, [NotNullWhen(true)] out string? message)
{
    message = null;
    return true;
}
```

O compilador produz um aviso porque `message` o parâmetro é atribuído `null` e o método retorna `true`. O `NotNullWhen` atributo indica que isso não deve acontecer.

Para resolver esses avisos, atualize seu código para que ele corresponda às expectativas dos atributos que você aplicou. Você pode alterar os atributos ou o algoritmo.

Métodos em (C#)

21/01/2022 • 22 minutes to read

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.

NOTE

Este tópico aborda os métodos nomeados. Para obter informações sobre funções anônimas, consulte [expressões lambda](#).

Assinaturas de método

Os métodos são declarados em um `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.

IMPORTANT

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:

```
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();
}
```

Observe que 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 que o parâmetro denominado 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.

```

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`.

```

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.

```

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 os argumentos nomeados quando os argumentos nomeados estão nas posições corretas. O exemplo a seguir invoca o método `TestMotorcycle.Drive` do exemplo anterior usando um argumento posicional e um argumento nomeado.

```
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`.

```

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.)

```

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.

```
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.

```

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 `ref` ou `out` palavra-chave ou. Você também pode passar um valor por referência para evitar copiar, mas ainda evitar modificações usando a `in` palavra-chave.

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.

```

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.

```

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 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 `Main` método 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.

```

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`. A partir do C# 7.1, você pode usar o `default` literal, pois o compilador pode inferir o tipo da declaração do parâmetro.
- Uma expressão da forma `new ValType()`, em que `ValType` é um tipo de valor. Isso invoca o construtor sem parâmetros implícito do tipo de valor, que não é de fato um membro do tipo.

NOTE

No C# 10 e posterior, quando uma expressão do formulário `new ValType()` invoca o construtor sem parâmetros explicitamente definido 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 `default(ValType)` expressão ou o `default` literal para fornecer o valor de parâmetro padrão. Para obter mais informações sobre construtores sem parâmetros, consulte a seção [construtores sem parâmetros e inicializadores de campo](#) 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.

```
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.

O 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 `Required`, mas usa um argumento nomeado para fornecer um valor ao parâmetro `description` ao omitir o `optionalInt` argumento.

```
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 chamada 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 tem 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:

```
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:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
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. A partir do C# 7.0, 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`.

```
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:

```
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:

```
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 `GetPersonInfo` pode ser modificada da seguinte maneira:

```
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.

```

using System;

public class ArrayValueExample
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8 };
        DoubleValues(values);
        foreach (var value in values)
            Console.WriteLine("{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 possui 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 que o 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.

NOTE

Um método assíncrono retorna ao chamador quando encontra o primeiro objeto esperado que ainda não está concluído ou chega ao final do método `Async`, o que ocorre primeiro.

Um método assíncrono normalmente tem um tipo de retorno `Task<TResult>` de `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. Começando com o C# 7.0, 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 de `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.

```
using System;
using System.Threading.Tasks;

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 métodos assíncronos, consulte [programação assíncrona com tipos de retorno Async e Await e 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 `=>`:

```
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 não usar 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>`, `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

21/01/2022 • 9 minutes to read

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:

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

Uma definição de propriedade contém declarações para um acessador `get` e `set` que recupera e atribui o valor dessa propriedade:

```
public class Person
{
    public string FirstName { get; set; }

    // remaining implementation removed from listing
}
```

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:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

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:

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Quando uma implementação de propriedade é uma única expressão, você pode usar *membros aptos para expressão* para o getter ou setter:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

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.

Cenários

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.

Validação

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:

```

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;
    // remaining implementation removed from listing
}

```

O exemplo anterior pode ser simplificado usando uma expressão `throw` como parte da validação do setter de propriedade:

```

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;
    // remaining implementation removed from listing
}

```

O exemplo acima aplica a regra de que o nome não pode ser em branco ou espaço em branco. Se um desenvolvedor escreve

```
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.

Somente leitura

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`:

```

public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}

```

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 pode 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.

Você também pode restringir modificações a uma propriedade para que ela possa ser definida somente em um construtor ou um inicializador de propriedade. Você pode modificar a classe `Person` da seguinte maneira:

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

Esse recurso é mais comumente usado para inicializar coleções que são expostas como propriedades somente leitura:

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
```

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:

```
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:

```
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:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == 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:

```

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 == 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

Do C# 7.3 em diante, atributos de campo podem ser anexados ao campo de suporte gerado pelo compilador em propriedades autoimplementadas. Por exemplo, considere uma revisão da classe `Person` que adiciona uma propriedade `Id` de inteiro exclusivo. Você escreve a `Id` propriedade usando uma propriedade auto-implementada, mas seu design não chama para persistir a `Id` propriedade. 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:

```

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.

```
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;
    // remaining implementation removed from listing
}
```

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, confira [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 você digitou errado o nome da propriedade.

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

21/01/2022 • 9 minutes to read

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:

```
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:

```
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.

```
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 nextPage = new Page(startingIndex, 1000);
    addPageToCache(nextPage);
    return nextPage;
}

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 essa linguagem de design para modelar qualquer tipo de coleção onde 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.

```
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 está usando um método público para definir 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 pelo computador são populares para o conjunto Mandelbrot definir cores para o número de iterações necessárias para determinar se um ponto está fora do conjunto.)

```
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:

```

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. Essas duas `using` diretivas:

```

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 de 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 estendem o conceito de [Propriedades](#), em que um membro é tratado como um item de dados de fora da classe, mas como um método no interior. Os indexadores permitem que os argumentos localizem um único item em uma propriedade que representa um conjunto de itens.

Iterators

21/01/2022 • 6 minutes to read

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 *iterador* para os elementos dessa classe. Um *iterador* é um objeto que atravessa um contêiner, especialmente listas. 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 forma 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:

```
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 para 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 forma assíncrona, você pode usar a instrução para consumir a sequência de forma `await foreach` assíncrona:

```
await foreach (var item in asyncSequence)
{
    Console.WriteLine(item.ToString());
}
```

Quando uma sequência é [System.Collections.Generic.IEnumerable<T>](#) um, você usa `foreach`. Quando uma sequência é [System.Collections.Generic.IAsyncEnumerable<T>](#) um, você usa `await foreach`. No último caso, a sequência é gerada de forma 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:

```
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:

```
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:

```
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 uma contraparte assíncrona. Em cada caso, você substituiria o tipo de retorno de `IEnumerable<T>` por um `IAsyncEnumerable<T>`. Por exemplo, o exemplo anterior teria a seguinte versão assíncrona:

```

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:

```

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 método:

```

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 e uma `return` instrução no mesmo método. O código a seguir não será compilado:

```

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:

```

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 uma situação em que talvez você queira retornar uma coleção vazia ou os primeiros cinco números ímpares, com base em um argumento booleana. Você poderia escrever isso como esses dois métodos:

```

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.

Aprofunde-se 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. Ele também minimiza os erros que os desenvolvedores fazem ao não gerenciar corretamente os recursos.

O compilador converte o loop `foreach` mostrado no primeiro exemplo em algo semelhante a esse constructo:

```
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 `GetEnumerator()` por implementa a interface `IDisposable`. A expansão completa gera um código mais semelhante a esse:

```
{
    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:

```
{
    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 geral síncrono, a `finally` cláusula se expande para:

```
finally
{
    (enumerator as IDisposable)?.Dispose();
}
```

O caso assíncrono geral expande para:

```
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

No entanto, se o tipo de for um tipo lacrado e não houver nenhuma conversão implícita do tipo de para ou , a cláusula se `enumerator` `enumerator` `IDisposable` `IAsyncDisposable` `finally` expandirá para um bloco vazio:

```
finally
{
}
```

Se houver uma conversão implícita do tipo de para e for um tipo de valor não `enumerator` `IDisposable` `enumerator` anulado, a `finally` cláusula se expandirá para:

```
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

21/01/2022 • 2 minutes to read

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 adiante nesta série, o código C# que você escreve para algoritmos como esse é tipo seguro e usa as regras de linguagem e o compilador para garantir que os tipos correspondam a argumentos e tipos de retorno.

[Ponteiros de função](#) foram adicionados ao C# 9 para cenários semelhantes, onde 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 delegate. Usando ponteiros de função, você pode especificar diferentes convenções.

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. Os delegados permitem que os desenvolvedores aprendam um conceito e usem esse mesmo conceito em vários problemas de software diferentes.

Em segundo lugar, a equipe queria dar suporte a chamadas de método single ou multicast. (Delegados multicast são delegados que encadeiam várias chamadas de método. Você verá exemplos [posteriormente 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 um padrão de evento que é um padrão específico em que delegados ou qualquer algoritmo de ligação tardia é muito útil. A equipe queria garantir que o código para delegados pudesse fornecer a base para o padrão de eventos .NET.

O resultado de todo esse trabalho era o suporte do delegado e do evento no C# e .NET. Os artigos restantes nesta seção abordarão os recursos de linguagem, o suporte à biblioteca e os idiomas comuns usados quando você trabalhar com delegados.

Você aprenderá sobre a palavra-chave `delegate` e qual código ela gera. Você aprenderá sobre os recursos na classe `System.Delegate` e como esses recursos são usados. Você aprenderá como criar delegados de segurança de tipos e como criar métodos que podem ser invocados por meio de delegados. Você também aprenderá a trabalhar com eventos e delegados usando expressões Lambda. Você verá onde delegados se tornam um dos blocos de construção para LINQ. Você aprenderá como os delegados são a base para o padrão de eventos .NET.

e como eles são diferentes.

Vamos começar.

[Próximo](#)

System.Delegate e a palavra-chave `delegate`

21/01/2022 • 6 minutes to read

[Anterior](#)

Este artigo aborda as classes no .NET que dão suporte a delegados e como elas são mapeadas para a `delegate` palavra-chave.

Definir tipos delegados

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:

```
// 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.

NOTE

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.

```
// 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:

```
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:

```
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()`:

```
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:

```
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:

```
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 `System.Delegate` classe e sua única subclasse direta, `System.MulticastDelegate` fornecem o suporte à estrutura para criar delegados, registrar métodos como destinos delegados e invocar todos os métodos que são registrados como um destino 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 oferecem suporte a delegados, vamos examinar como os delegados com rigidez de tipos são usados, criados e invocados.

[Próximo](#)

Delegados Fortemente Tipados

21/01/2022 • 2 minutes to read

[Anterior](#)

No artigo anterior, você viu como criar tipos de delegado específicos usando a palavra-chave `delegate`.

A classe `Delegate` abstrata fornece a infraestrutura para acoplamento flexível e invocação. 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:

```
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:

```
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.

Também há um especializado `Predicate<T>` tipo para um delegado que retorna um teste em um único valor:

```
public delegate bool Predicate<in T>(T obj);
```

Você pode observar que para qualquer tipo `Predicate`, há um tipo `Func` estruturalmente equivalente, por exemplo:

```
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

21/01/2022 • 8 minutes to read

[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:

```
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:

```
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.

Crie seus próprios componentes 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 de 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.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        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:

```
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:

```
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 delegados definidos na estrutura principal torna mais fácil para os usuários trabalhar 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.

Saída de formato

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:

```

public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}

```

```

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}";
        WriteMessage(outputMsg);
    }
}

```

Em seguida, vamos usar aquele argumento `Severity` para filtrar as mensagens que são enviadas para o log de saída.

```

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}";
        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.

Este é o agente baseado em arquivo:

```

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:

```
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:

```
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:

```
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 infraestrutura para dar suporte ao sistema de arquivos corretamente. O delegado 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 implementá-lo `IDisposable` para fechar o arquivo quando estiver concluído. Cada método tem suas vantagens e desvantagens. Ambos criam um pouco mais de acoplamento entre as classes.

Nenhum do código na `Logger` classe precisaria ser atualizado para dar suporte a qualquer cenário.

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()`:

```
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. Usando delegados no design, esses diferentes componentes são livremente acoplados. Isso traz vários benefícios. É fácil criar novos mecanismos de saída e anexá-los ao sistema. Esses outros mecanismos precisam de apenas um método: o método que grava a mensagem de log. É um design 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

21/01/2022 • 3 minutes to read

[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 suporte a eventos

O design de linguagem para eventos tem como alvo estas metas:

- Habilita um acoplamento muito mínimo entre uma origem do evento e um sink de evento. 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 fontes de 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`:

```
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 relata 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:

```
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 `+=`:

```
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.

Cancela a assinatura usando o operador `-=`:

```
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

21/01/2022 • 8 minutes to read

[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:

```
void OnEventRaised(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:

```
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.

```
public class FileSearcher
{
    public event EventHandler<FileEventArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileEventArgs(file));
        }
    }
}
```

Definindo e acionando eventos semelhantes a campo

A maneira mais simples de adicionar um evento à sua classe é declarar esse evento como um campo público, como no exemplo anterior:

```
public event EventHandler<FileEventArgs> 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:

```
EventHandler<FileEventArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

e remover manipulador:

```
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 ser capazes de parar o processamento, se esse arquivo for aquele que era 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 EventArguments 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 só 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 `FileEventArgs`:

```
public class FileEventArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileEventArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Este novo campo é inicializado automaticamente para `false`, o valor padrão para um campo booleano, 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:

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileEventArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

Uma vantagem desse padrão é que ele não é uma alteração significativa. Nenhum dos assinantes solicitou um cancelamento antes e ainda não o 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:

```
EventHandler<FileEventArgs> 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.

```
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.

```
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:

```

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)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(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))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}

```

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.

```

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.

Em seguida, você verá algumas alterações nesses padrões na versão mais recente do .NET.

[Próximo](#)

O padrão de eventos atualizado do .NET Core Event

21/01/2022 • 4 minutes to read

[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:

```
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:

```
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 continua a execução porque ele é `async`.) O comportamento real do tempo de execução 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

21/01/2022 • 3 minutes to read

[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 de delegados ou eventos geralmente é difícil, porque os dois recursos de linguagem são semelhantes. De fato, os eventos são criados usando o suporte de linguagem para delegados.

Ambos oferecem um cenário de associação tardia: eles habilitam cenários em que um componente se comunica chamando um método que só é conhecido em tempo de executar. 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 código precisar chamar o código fornecido pelo assinante, você deverá usar um design com base 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 da que contém um evento só podem adicionar e remover ouvintes de eventos; somente a classe que contém o evento pode invocar o evento. Os eventos normalmente são membros de classe pública.

Por comparação, delegados geralmente são passados como parâmetros e armazenados como membros de classe privada, se eles são armazenados.

Ouvintes de evento frequentemente têm vida útil mais longa

Os ouvintes de eventos têm tempo de vida mais longo 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)

21/01/2022 • 3 minutes to read

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`.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.WriteLine(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 usam muitas construções 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 [relações 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 às 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

21/01/2022 • 13 minutes to read

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[]`):

```
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`.

```
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` :

```
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.

```
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.

```

static void Main()
{
    // 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);
    }
}
// Outputs: 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:

```

//Query syntax
IQueryable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IQueryable<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:

```

int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IQueryable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

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:

```

// Use of var is optional here and in all queries.
// queryCities is an IQueryable<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.

```
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:

```
IEnumerable<City> cityQuery =  
    from country in countries  
    from city in country.Cities  
    where city.Population > 10000  
    select city;
```

Para obter mais informações, consulte [da cláusula](#).

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 e cuja chave é um tipo com valor sendo a primeira letra dos nomes `Country` `char` dos países.

```
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.

```
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.

```
// 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 `em`

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.

```
// 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.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

Para obter mais informações, consulte [a 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`.

```
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`. Produtos cujos `Category` não corresponderem a nenhuma cadeia de caracteres `categories` em são filtrados. A `select` instrução projeta um novo tipo cujas propriedades são retiradas de `cat` e `prod`.

```
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`.

```
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.

```
var queryGroupMax =  
    from student in students  
    group student by student.GradeLevel into studentGroup  
    select new  
    {  
        Level = studentGroup.Key,  
        HighestScore =  
            (from student2 in studentGroup  
             select student2.Scores.Average())  
            .Max()  
    };
```

Para obter mais informações, [consulte Executar uma subconsistência em uma operação de agrupação](#).

Confira também

- [Guia de programação em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Palavras-chave de consulta \(LINQ\)](#)
- [Visão geral dos operadores de consulta padrão](#)

LINQ em C#

21/01/2022 • 2 minutes to read

Esta seção contém links para tópicos que fornecem informações mais detalhadas sobre a LINQ.

Nesta seção

[Introdução às consultas LINQ](#)

Descreve as três partes da operação de consulta LINQ básica que são comuns a todas as linguagens e fontes de dados.

[Tipos de LINQ e genérico](#)

Fornece uma breve introdução sobre como os tipos genéricos são usados na LINQ.

[Transformações de dados com LINQ](#)

Descreve as várias maneiras que você pode transformar os dados recuperados em consultas.

[Relações de tipo em operações de consulta LINQ](#)

Descreve como tipos são preservados e/ou transformados nas três partes de uma operação de consulta LINQ.

[Sintaxe de consulta e sintaxe de método no LINQ](#)

Compara a sintaxe de método e a sintaxe de consulta como dois modos para expressar uma consulta LINQ.

[Recursos do C# que dão suporte ao LINQ](#)

Descreve os constructos da linguagem em C# que dão suporte ao LINQ.

Seções relacionadas

[Expressões de consulta LINQ](#)

Inclui uma visão geral de consultas na LINQ e fornece links para recursos adicionais.

[Visão geral dos operadores de consulta padrão](#)

Apresenta os métodos padrão usados na LINQ.

Escrever consultas LINQ em C#

21/01/2022 • 4 minutes to read

Este artigo mostra as três maneiras de escrever uma consulta LINQ em C#:

1. Usar a sintaxe de consulta.
2. Usar a sintaxe do método.
3. Usar uma combinação da sintaxe de consulta e da sintaxe de método.

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.

NOTE

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.

```
// Query #1.  
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
  
// The query variable can also be implicitly typed by using var  
IQueryable<int> filteringQuery =  
    from num in numbers  
    where num < 3 || num > 7  
    select num;  
  
// Query #2.  
IQueryable<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:

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 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 parâmetros `Action` ou `Func`, eles serão fornecidos na forma de uma [expressão lambda](#), conforme mostrado no exemplo a seguir:

```
// 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:

```
// 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.

```
// 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:

```
var numCount = (from num in numbers...
```

Ela pode ser escrita na sintaxe de método da seguinte maneira:

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Ela pode ser escrita usando a tipagem explícita da seguinte maneira:

```
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

21/01/2022 • 2 minutes to read

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.

Este aplicativo serve como a estrutura para muitos outros exemplos nesta seção que usam as mesmas fontes de dados `students`.

Exemplo

A consulta a seguir retorna os alunos que receberam uma pontuação de 90 ou mais em sua primeira prova.

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79 }},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94 }},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87 }},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82 }},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70 }},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78 }},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91 }},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39 }},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60 }},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92 }},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 94, 92, 91, 91 }},
    };
}
```

```

        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
                     Year = GradeLevel.FourthYear,
                     ExamScores = new List<int> { 96, 85, 91, 60}}
    };
#endregion

// Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public static 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}");
    }
}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

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.

Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Interpolação de cadeia de caracteres](#)

Como retornar uma consulta de um método (guia de programação C#)

21/01/2022 • 2 minutes to read

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.

```
class MQ
{
    // QueryMethod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref 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 (string 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 (string s in app.QueryMethod1(ref nums))
{
    Console.WriteLine(s);
}

IEnumerable<string> myQuery2;
// QueryMethod2 returns a query as the value of its out parameter.
app.QueryMethod2(ref nums, out myQuery2);

// Execute the returned query.
Console.WriteLine("\nResults of executing myQuery2:");
foreach (string s in myQuery2)
{
    Console.WriteLine(s);
}

// You can modify a query by using query composition. A saved query
// is nested inside a new query definition that revises the results
// of the first query.
myQuery1 = from item in myQuery1
            orderby item descending
            select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (string s in myQuery1)
{
    Console.WriteLine(s);
}

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}

```

Confira também

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

Armazenar os resultados de uma consulta na memória

21/01/2022 • 2 minutes to read

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:

- [ToList](#)
- [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

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

    static void Main()
    {

        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();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

Confira também

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

Agrupar resultados de consultas

21/01/2022 • 8 minutes to read

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](#).

Exemplo de classe auxiliar e fonte de dados

Todos os exemplos neste tópico usam as seguintes fontes de dados e classes auxiliares.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 87, 80, 84}}
    }
}
```

```

        Year = GradeLevel.FourthYear,
        ExamScores = new List<int>{ 75, 84, 91, 39}),
    new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
        Year = GradeLevel.SecondYear,
        ExamScores = new List<int>{ 97, 92, 81, 60}),
    new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
        Year = GradeLevel.ThirdYear,
        ExamScores = new List<int>{ 68, 79, 88, 92}),
    new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
        Year = GradeLevel.FirstYear,
        ExamScores = new List<int>{ 94, 92, 91, 91}),
    new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
        Year = GradeLevel.FourthYear,
        ExamScores = new List<int>{ 96, 85, 91, 60})
};

#endregion

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public 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}");
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

Exemplo de propriedade agrupar por 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 subcadeia para a chave. A operação de agrupamento usa o comparador de igualdade padrão para o tipo.

Cole o seguinte método na classe `StudentClass`. Altere a instrução de chamada no método `Main` para `sc.GroupBySingleProperty()`.

```

public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by a single property in an object:
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 agrupar 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.

Cole o seguinte método na classe `StudentClass`. Altere a instrução de chamada no método `Main` para `sc.GroupBySubstring()`.

```

public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by something other than a property of the object:
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
*/

```

Agrupar por exemplo de 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.

```

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

```

Cole o seguinte método na classe `StudentClass`. Altere a instrução de chamada no método `Main` para `sc.GroupByRange()`.

```

public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        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 queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"{item.LastName}, {item.FirstName}");
        }
    }
}

/* Output:
Group by numeric range and project into a new anonymous type:
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
*/

```

Agrupar por exemplo de comparação

O exemplo a seguir mostra como agrupar elementos de origem usando uma expressão de comparação booliana. Neste exemplo, a expressão booliana 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.

Cole o seguinte método na classe `StudentClass`. Altere a instrução de chamada no método `Main` para `sc.GroupByBoolean()`.

```

public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                                group new { student.FirstName, student.LastName } 
                                by student.ExamScores.Average() > 75 into studentGroup
                                select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}
/* Output:
Group by a Boolean into two groups with string keys
"True" and "False" and project into a new anonymous type:
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
*/

```

Agrupar 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.

Cole o seguinte método na classe `StudentClass`. Altere a instrução de chamada no método `Main` para `sc.GroupByCompositeKey()`.

```

public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
Group and order by a compound key:
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

21/01/2022 • 2 minutes to read

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

NOTE

Este exemplo contém referências a objetos que são definidos no código de exemplo em [Consultar uma coleção de objetos](#).

```

public void QueryNestedGroups()
{
    var queryNestedGroups =
        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 queryNestedGroups)
    {
        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($"{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

21/01/2022 • 2 minutes to read

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

NOTE

Este exemplo contém referências a objetos que são definidos no código de exemplo em [Consultar uma coleção de objetos](#).

```
public void QueryMax()
{
    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.

```
public void QueryMaxUsingMethodSyntax()
{
    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

21/01/2022 • 7 minutes to read

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
A	think
Um	that
B	Linq
C	está
Um	really
B	cool
B	!

Os seguintes grupos serão criados nesta ordem:

1. We, think, that
2. Linq
3. está
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:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```

namespace ChunkIT
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IGrouping<TKey, TSource> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return 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;
    }

    // The value that is used to determine matching elements
    private readonly TKey key;

    // 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 object m_Lock;

    // REQUIRES: enumerator != null && predicate != null
    public Chunk(TKey key, Ienumerator<TSource> enumerator, Func<TSource, bool> predicate)
    {
        this.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 => key;

    // 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 IEnumrator<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();
    }
}

// A simple named type is used for easier viewing in the debugger. Anonymous types
// work just as well with the ChunkBy operator.
public class KeyValuePair
{
    public string Key { get; set; }
    public string Value { get; set; }
}

class Program
{
    // The source sequence.
    public static IEnumerable<KeyValuePair> list;

    // Query variable declared as class member to be available
    // on different threads.
    static IGrouping<string, KeyValuePair>> query;

    static void Main(string[] args)
    {
        // Initialize the source sequence with an array initializer.
        list = new[]
        {
            new KeyValuePair{ Key = "A", Value = "We" },
            new KeyValuePair{ Key = "A", Value = "think" },
            new KeyValuePair{ Key = "A", Value = "that" },
            new KeyValuePair{ Key = "B", Value = "Linq" },
            new KeyValuePair{ Key = "C", Value = "is" },
            new KeyValuePair{ Key = "A", Value = "really" },
            new KeyValuePair{ Key = "B", Value = "cool" },
            new KeyValuePair{ Key = "B", Value = "!" }
        };
        // Create the query by using our user-defined query operator.
        query = list.ChunkBy(p => p.Key);

        // 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($"Group key = {item.Key}");
            foreach (var inner in item)
            {
                Console.WriteLine($"{inner.Value}");
            }
        }

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
}

```

Para usar o método de extensão em seu projeto, copie a classe estática `MyExtensions` para um arquivo de código-fonte novo ou existente e se for necessário, adicione uma diretiva `using` para o namespace em que ele está localizado.

Confira também

- LINQ (Consulta Integrada à Linguagem)

Especificar dinamicamente filtros de predicado em tempo de execução

21/01/2022 • 2 minutes to read

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. O exemplo é construído de duas maneiras. Primeiro, o projeto é executado filtrando valores que são fornecidos no programa. Em seguida, o projeto é executado novamente usando a entrada fornecida em tempo de execução.

Para filtrar usando o método Contains

1. Abra um novo aplicativo de console e nomeie-o como `PredicateFilters`.
2. Copie a classe `StudentClass` de [Consultar uma coleção de objetos](#) e cole-a no namespace `PredicateFilters` sob a classe `Program`. `StudentClass` fornece uma lista de objetos `Student`.
3. Comente o método `Main` em `StudentClass`.
4. Substitua a classe `Program` pelo código a seguir:

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Adicione a seguinte linha ao método `Main` na classe `DynamicPredicates`, sob a declaração de `ids`.

```
QueryById(ids);
```

6. Execute o projeto.
7. A saída a seguir é exibida em uma janela do console:

Garcia: 114

O'Donnell: 112

Omelchenko: 111

8. A próxima etapa é executar o projeto novamente, desta vez usando a entrada inserida em tempo de execução em vez da matriz `ids`. Altere `QueryByID(ids)` para `QueryByID(args)` no método `Main`.
9. Execute o projeto com os argumentos de linha de comando `122 117 120 115`. Quando o projeto é executado, esses valores se tornam elementos de `args`, o parâmetro do método `Main`.
10. A saída a seguir é exibida em uma janela do console:

Adams: 120

Feng: 117

Garcia: 115

Tucker: 122

Para filtrar usando uma instrução switch

1. Você pode usar uma instrução `switch` para selecionar entre consultas alternativas predeterminadas. No exemplo a seguir, `studentQuery` usa uma cláusula `where` diferente dependendo de qual nível de ensino ou ano, é especificado em tempo de execução.
2. Copie o método a seguir e cole-o na classe `DynamicPredicates`.

```

// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;
        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

```

3. No método `Main`, substitua a chamada para `QueryByID` pela chamada a seguir, que envia o primeiro elemento da matriz `args` como seu argumento: `QueryByYear(args[0])`.
4. Execute o projeto com um argumento de linha de comando de um valor inteiro entre 1 e 4.

Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [cláusula WHERE](#)

Executar junções internas

21/01/2022 • 10 minutes to read

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.

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.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // 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 };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"{ownerAndPet.PetName} is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following 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

```

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.

```

class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
        new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
        new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
        new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> 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;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price

```

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.

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // 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 };

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name, with \"{obj.DogName}\".");
    }
}

// This code produces the following 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".

```

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.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists

```

```

// Create two lists.
List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

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 };

Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

var query2 = from person in people
            join pet in pets on person equals pet.Owner
            select new { OwnerName = person.FirstName, PetName = pet.Name };

Console.WriteLine("\nThe equivalent operation using Join():");
foreach (var v in query2)
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following 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

```

Confira também

- [Join](#)
- [GroupJoin](#)
- [Executar junções agrupadas](#)
- [Executar junções externas esquerdas](#)
- [Tipos anônimos](#)

Executar junções agrupadas

21/01/2022 • 5 minutes to read

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).

NOTE

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.

WARNING

`Enumerable.GroupJoin` não tem equivalente direto em termos tradicionais de banco de dados relacional. No entanto, esse método implementa um superconjunto de junções internas e junções externas esquerdas. Essas duas operações podem ser escritas 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.

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`.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // 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($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following 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.

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("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);
}

// This code produces the following 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

21/01/2022 • 2 minutes to read

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.

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. (Consulte [Executar junções internas](#) para ver uma explicação desse processo.) Neste exemplo, a lista de objetos `Person` é ingressada internamente na lista de objetos `Pet` com base em um objeto que corresponde a `Person` `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.

NOTE

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`.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { 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} : {v.PetName}");
    }
}

// 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

21/01/2022 • 2 minutes to read

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.

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.

```
class HowToOrderJoins
{
    #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()
    {
        HowToOrderJoins app = new HowToOrderJoins();
        app.OrderJoin1();

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```

        Console.WriteLine(),
    }

    void OrderJoin1()
    {
        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

21/01/2022 • 2 minutes to read

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:

```
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:

```
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

21/01/2022 • 5 minutes to read

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.

NOTE

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](#).

Exemplo 1

O primeiro método no exemplo a seguir 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.

O segundo método 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`.

```
class CustomJoins
{
    #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},
};

// Specify the second data source.
List<Product> products = new List<Product>()
{
    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},
    new Product{Name="Ice Cream", CategoryID=007},
    new Product{Name="Mackerel", CategoryID=012},
};

#endregion

static void Main()
{
    CustomJoins app = new CustomJoins();
    app.CrossJoin();
    app.NonEquijoin();

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

void CrossJoin()
{
    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}");
    }
}

void NonEquijoin()
{
    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, CategoryID = p.CategoryID };

    Console.WriteLine("Non-equijoin query:");
    foreach (var v in nonEquijoinQuery)
    {
        Console.WriteLine($"{v.CategoryID},-{5}{v.Product}");
    }
}

/* Output:
Cross Join Query:
1   Tea
1   Mustard
1   Pickles
1   Carrots
1   Bok Choy

```

```

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
Non-equijoin query:
1   Tea
2   Mustard
2   Pickles
3   Carrots
3   Bok Choy
Press any key to exit.
*/

```

Exemplo 2

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.

```

class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../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],
                ID = Convert.ToInt32(x[2]),

```

```

        ExamScores = (from scoreAsText in s.Skip(1)
                      select Convert.ToInt32(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()}.");
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* 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

21/01/2022 • 2 minutes to read

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 é `null`, e a consulta não tratar `null` valores, um `NullReferenceException` será gerado quando você executar a consulta.

Você pode escrever o código defensivamente para evitar uma exceção de referência nula conforme mostrado no exemplo a seguir:

```
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` neste exemplo funciona porque `Products.CategoryID` é do tipo `int?`, que é a abreviação de `Nullable<int>`.

Em uma cláusula de junção, se apenas uma das chaves de comparação for um tipo de valor anulável, você poderá converter a outra em 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?`:

```
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 `equals` palavra-chave de consulta é usada. O C# 9 adiciona a [correspondência de padrões](#), 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 consulta é uma biblioteca que traduz expressões de consulta C# em um formato de dados nativo, como Entity Framework Core. Os provedores de consulta implementam a `System.Linq.IQueryProvider` interface para criar fontes de dados que implementam a `System.Linq.IQueryable<T>` interface.

Confira também

- [Nullable<T>](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Tipos de valor anuláveis](#)

Tratar exceções em expressões de consulta

21/01/2022 • 2 minutes to read

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 [Práticas recomendadas 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.

```

class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        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");
            goto Exit;
        }

        // 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());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}

```

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 às consultas LINQ](#).

```

class QueryThatThrows
{
    static void Main()
    {
        // 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;

        // Runtime exceptions are thrown when 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);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* 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 \(Consulta Integrada à Linguagem\)](#)

Escrever um código C# seguro e eficiente

21/01/2022 • 18 minutes to read

O C# fornece recursos que permitem escrever código seguro verificável com melhor desempenho. Se você aplicar essas técnicas com cuidado, menos cenários exigirão código não seguro. Esses recursos tornam fácil usar referências a tipos de valor como argumentos e retornos de método. Quando realizadas com segurança, essas técnicas minimizam a cópia de tipos de valor. Usando tipos de valor, é possível minimizar a quantidade de alocações e passagens de coleta de lixo.

Grande parte do código de exemplo neste artigo usa recursos adicionados no C# 7.2. Para usar esses recursos, certifique-se de que seu projeto não está configurado para usar uma versão anterior. Para obter mais informações, consulte [Configurar a versão do idioma](#).

Uma vantagem de usar tipos de valor é que eles geralmente evitam alocações de heap. A desvantagem é que eles são copiados por valor. Essa troca dificulta a otimização de algoritmos que operam em grandes quantidades de dados. Os recursos de linguagem destacados neste artigo fornecem mecanismos que permitem código seguro e eficiente usando referências a tipos de valor. Use esses recursos criteriosamente para minimizar tanto as alocações quanto as operações de cópia.

Algumas das diretrizes neste artigo referem-se a práticas de codificação que são sempre aconselháveis, não apenas para o benefício de desempenho. Use a `readonly` palavra-chave quando expressar com precisão a intenção de design:

- Declare structs imutáveis `readonly` como .
- Declare `readonly` membros para structs mutáveis.

O artigo também explica algumas otimizações de baixo nível que são aconselháveis quando você executar um profiler e identificar gargalos:

- Use o `in` modificador de parâmetro.
- Use `ref readonly return` instruções.
- Use `ref struct` tipos.
- Use `nint` os tipos `nuint` e .

Essas técnicas equilibram duas metas concorrentes:

- Minimize as alocações no heap.

Variáveis que são [tipos de referência](#) têm uma referência a um local na memória e são alocadas no heap gerenciado. Somente a referência é copiada quando um tipo de referência é passado como um argumento para um método ou retornado de um método. Cada novo objeto requer uma nova alocação e, posteriormente, deve ser recuperado. A coleta de lixo leva tempo.

- Minimize a cópia de valores.

As variáveis que são [tipos de valor](#) contêm diretamente seu valor e o valor normalmente é copiado quando passado para um método ou retornado de um método. Esse comportamento inclui copiar o valor de ao chamar iteradores e métodos de instância `this` assíncrona de structs. A operação de cópia leva tempo, dependendo do tamanho do tipo.

Este artigo usa o seguinte conceito de exemplo da estrutura de ponto 3D para explicar suas recomendações:

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

diferentes exemplos usam diferentes implementações deste conceito.

Declarar structs imutáveis como `readonly`

Declare um `readonly struct` para indicar que um tipo é imutável. O `readonly` modificador informa ao compilador que sua intenção é criar um tipo imutável. O compilador impõe essa decisão de design com as seguintes regras:

- Todos os membros do campo devem ser somente leitura.
- Todas as propriedades devem ser somente leitura, incluindo propriedades autoimplementadas.

Essas duas regras são suficientes para garantir que nenhum membro de um `readonly struct` modifica o estado desse struct. O `struct` é imutável. A estrutura `Point3D` pode ser definida como um struct imutável, conforme mostrado no exemplo a seguir:

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Siga esta recomendação sempre que sua intenção de design for criar um tipo de valor imutável. Quaisquer melhorias no desempenho são um benefício adicional. As `readonly struct` palavras-chave expressam claramente sua intenção de design.

Declarar `readonly` membros para structs mutáveis

No C# 8.0 e posterior, quando um tipo de struct é `readonly` mutável, declare membros que não modificam o estado como membros .

Considere um aplicativo diferente que precisa de uma estrutura de ponto 3D, mas deve dar suporte à mutabilidade. A versão a seguir da estrutura de ponto 3D adiciona o modificador somente aos membros que `readonly` não modificam a estrutura. Siga este exemplo quando seu design deve dar suporte a modificações no struct por alguns membros, mas você ainda deseja os benefícios da imposição `readonly` em alguns membros:

```

public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

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

    public readonly override string ToString() => $"{X}, {Y}, {Z}";
}

```

O exemplo anterior mostra muitos dos locais em que você pode aplicar o `readonly` modificador: métodos, propriedades e acessadores de propriedade. Se você usar propriedades auto-implementadas, o compilador adicionará o `readonly` modificador ao `get` acessador para propriedades de leitura/gravação. O compilador adiciona o modificador às declarações de propriedade `readonly` auto-implementadas para propriedades com apenas `get` um acessador.

Adicionar o modificador a membros que não modificam o estado `readonly` fornece dois benefícios relacionados. Primeiro, o compilador impõe sua intenção. Esse membro não pode modificar o estado do `struct`. Em segundo lugar, o compilador não criará cópias [de defesa de](#) `in` parâmetros ao acessar um `readonly` membro. O compilador pode fazer essa otimização com segurança porque garante que o `struct` não seja modificado por um `readonly` membro.

Instruções `ref readonly return` de uso

Use um `ref readonly` retorno quando ambas as condições a seguir são verdadeiras:

- O valor de retorno é `struct` maior que `IntPtr.Size`.
- O tempo de vida de armazenamento é maior que o método que retorna o valor.

É possível retornar valores por referência quando o valor que está sendo retornado não é local para o método de retorno. Retornar por referência significa que somente a referência é copiada, não a estrutura. No exemplo a seguir, a propriedade `Origin` não pode usar um retorno `ref`, porque o valor que está sendo retornado é uma variável local:

```
public Point3D Origin => new Point3D(0,0,0);
```

No entanto, a seguinte definição de propriedade pode ser retornada por referência, porque o valor retornado é um membro estático:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

Você não deseja chamadores que modificam a origem, então deve retornar o valor por `ref readonly`:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

Retornar `ref readonly` permite que você salvar a cópia de estruturas maiores e preserve a imutabilidade de seus membros de dados internos.

No site de chamada, os chamadores fazem a opção de usar a propriedade `origin` como um `ref readonly` ou como um valor:

```
var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;
```

A primeira atribuição no código anterior faz uma cópia da constante `origin` e atribui essa cópia. A segunda atribui uma referência. Observe que o modificador `readonly` deve ser parte da declaração da variável. A referência à qual ele se relaciona não pode ser modificada. As tentativas de modificá-la resultam em um erro em tempo de compilação.

O modificador `readonly` é necessário na declaração de `originReference`.

O compilador impõe que o autor da chamada não pode modificar a referência. As tentativas de atribuir o valor diretamente geram um erro em tempo de compilação. Em outros casos, o compilador aloca uma cópia [de defesa](#), a menos que possa usar com segurança a referência somente leitura. As regras de análise estática determinam se o struct pode ser modificado. O compilador não cria uma cópia de defesa quando o struct é um ou o membro é `readonly struct` `readonly` membro do struct. Cópias de defesa não são necessárias para passar o struct como um `in` argumento.

Usar o `in` modificador de parâmetro

As seções a seguir explicam o que o modificador faz, como usá-lo e quando `in` usá-lo para otimização de desempenho:

- As `out` `ref` palavras-chave, `in` e

- Usar `in` parâmetros para structs grandes
- Uso opcional de `in` no site de chamada
- Evitar cópias de defesa

As `out` `ref` palavras-chave, `in` e

A `in` palavra-chave complementa as `ref` `out` palavras-chave e para passar argumentos por referência. A `in` palavra-chave especifica que o argumento é passado por referência, mas o método chamado não modifica o valor. O modificador pode ser aplicado a qualquer membro que aceita `in` parâmetros, como métodos, delegados, lambdas, funções locais, indexadores e operadores.

Com a adição da `in` palavra-chave, o C# fornece um vocabulário completo para expressar sua intenção de design. Os tipos de valor são copiados no momento em que são passados para um método chamado quando você não especifica nenhum dos modificadores a seguir na assinatura do método. Cada um desses modificadores especifica que uma variável é passada por referência, evitando a cópia. Cada modificador expressa uma intenção diferente:

- `out`: esse método define o valor do argumento usado como este parâmetro.
- `ref`: Esse método pode modificar o valor do argumento usado como esse parâmetro.
- `in`: Esse método não modifica o valor do argumento usado como esse parâmetro.

Adicione o modificador `in` para passar um argumento por referência e declare que sua intenção de design é passar argumentos por referência para evitar cópias desnecessárias. Você não pretende modificar o objeto usado como esse argumento.

O modificador `in` complementa `out` e `ref` de outras formas também. Não é possível criar sobrecargas de um método que diferem somente na presença de `in`, `out` ou `ref`. Essas novas regras apresentam o mesmo comportamento que sempre foi definido para os parâmetros `out` e `ref`. Como os modificadores `out` e `ref`, os tipos de valor não estão demarcados, porque o modificador `in` é aplicado. Outro recurso dos `in` parâmetros é que você pode usar valores literais ou constantes para o argumento para um `in` parâmetro.

O `in` modificador também pode ser usado com tipos de referência ou valores numéricos. No entanto, os benefícios nesses casos são mínimos, se houver.

Há várias maneiras pelas quais um compilador impõe a natureza somente leitura de um argumento `in`. Em primeiro lugar, o método chamado não pode ser atribuído diretamente a um parâmetro `in`. Não é possível atribuí-lo diretamente a nenhum campo de um parâmetro `in` quando esse valor é um tipo `struct`. Além disso, não é possível passar um parâmetro `in` para nenhum método usando o modificador `ref` ou `out`. Essas regras se aplicam a qualquer campo de um parâmetro `in`, considerando que o campo seja um tipo `struct` e o parâmetro também seja um tipo `struct`. Na verdade, essas regras são aplicadas a várias camadas de acesso de membro, considerando que os tipos, em todos os níveis de acesso de membro, sejam `structs`. O compilador impõe que os `struct` tipos passados como `in` argumentos e seus `struct` Membros sejam variáveis somente leitura quando usados como argumentos para outros métodos.

Usar `in` parâmetros para estruturas grandes

Você pode aplicar o `in` modificador a qualquer `readonly struct` parâmetro, mas essa prática provavelmente melhorará o desempenho somente para tipos de valor que são consideravelmente maiores do que `IntPtr.Size`. Para tipos simples (como `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` e `bool` `enum` tipos), quaisquer ganhos potenciais de desempenho são mínimos. Alguns tipos simples, como `decimal` com 16 bytes de tamanho, são maiores que as referências de 4 ou 8 bytes, mas não o suficiente para fazer uma diferença significativa no desempenho na maioria dos cenários. E o desempenho pode degradar usando passagem por referência para tipos menores do que `IntPtr.Size`.

O código a seguir mostra um exemplo de um método que calcula a distância entre dois pontos no espaço 3D.

```

private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}

```

Os argumentos são duas estruturas que contêm três duplas. Uma dupla tem 8 bytes. Então, cada argumento tem 24 bytes. Ao especificar o modificador `in`, você passa uma referência de 4 ou 8 bytes para esses argumentos, dependendo da arquitetura do computador. A diferença no tamanho é pequena, mas pode ser somada quando o aplicativo chama esse método em um loop rígido usando muitos valores diferentes.

No entanto, o impacto de qualquer otimização de nível baixo, como o uso do `in` modificador, deve ser medido para validar um benefício de desempenho. Por exemplo, você pode imaginar que `in` o uso de em um parâmetro `GUID` seria benéfico. O `Guid` tipo tem 16 bytes de tamanho, duas vezes o tamanho de uma referência de 8 bytes. Mas essa diferença pequena provavelmente não resultará em um benefício de desempenho mensurável, a menos que esteja em um método que esteja em um prazo de quente crítico para seu aplicativo.

Uso opcional de `in` no site de chamada

Ao contrário de um `ref` `out` parâmetro ou, você não precisa aplicar o `in` modificador no site de chamada. O código a seguir mostra dois exemplos de como chamar o `CalculateDistance` método. O primeiro usa duas variáveis locais transmitidas por referência. O segundo inclui uma variável temporária criada como parte da chamada de método.

```

var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());

```

Omitir o `in` modificador no site de chamada informa ao compilador que ele tem permissão para fazer uma cópia do argumento por qualquer um dos seguintes motivos:

- Existe uma conversão implícita, mas não uma conversão de identidade do tipo de argumento no tipo de parâmetro.
- O argumento é uma expressão, mas não tem uma variável de armazenamento conhecida.
- Há uma sobrecarga que é distinguível pela presença ou ausência de `in`. Nesse caso, a sobrecarga pelo valor é uma correspondência melhor.

Essas regras são úteis conforme você atualiza o código existente para usar argumentos de referência somente leitura. Dentro do método chamado, você pode chamar qualquer método de instância que usa parâmetros por valor. Nessas instâncias, uma cópia do parâmetro `in` é criada.

Uma vez que o compilador pode criar uma variável temporária para qualquer parâmetro `in`, você também pode especificar valores padrão para qualquer parâmetro `in`. O código a seguir especifica a origem (ponto 0, 0, 0) como o valor padrão para o segundo ponto:

```

private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}

```

Para forçar o compilador a passar argumentos somente leitura por referência, especifique o modificador `in` nos argumentos no site de chamada, conforme mostrado no código a seguir:

```

distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);

```

Esse comportamento facilita a adoção de parâmetros `in` ao longo do tempo nas grandes bases de código em que os ganhos de desempenho são possíveis. Primeiro você adiciona o modificador `in` às assinaturas de método. Em seguida, você pode adicionar o `in` modificador em sites de chamada e criar `readonly struct` tipos para habilitar o compilador para evitar a criação de cópias defensivas de `in` parâmetros em mais locais.

Evitar cópias defensivas

Passar um `struct` como o argumento para um `in` parâmetro somente se ele for declarado com o `readonly` modificador ou o método acessar somente `readonly` os membros da estrutura. Caso contrário, o compilador deve criar *cópias defensivas* em muitas situações para garantir que os argumentos não sejam modificados. Considere o exemplo a seguir que calcula a distância de um ponto 3D da origem:

```

private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}

```

A `Point3D` estrutura *não* é uma `struct` somente leitura. Há seis chamadas de acesso de propriedade diferentes no corpo deste método. No primeiro exame, você pode imaginar que esses acessos são seguros. No fim das contas, um acessador `get` não deve modificar o estado do objeto. Mas não há nenhuma regra de linguagem que impõe isso. É apenas uma convenção comum. Qualquer tipo pode implementar um acessador `get` que modificou o estado interno.

Sem alguma garantia de linguagem, o compilador deve criar uma cópia temporária do argumento antes de chamar qualquer membro não marcado com o `readonly` modificador. O armazenamento temporário é criado na pilha, os valores do argumento são copiados para o armazenamento temporário e o valor é copiado para a pilha para cada acesso de membro como o argumento `this`. Em muitas situações, essas cópias prejudicam o desempenho que passa por valor é mais rápido do que a referência de passagem por somente leitura quando o tipo de argumento não é um `readonly struct` e o método chama Membros que não estão marcados `readonly`. Se você marcar todos os métodos que não modificam o estado de `struct` como `readonly`, o compilador poderá determinar com segurança que o estado de `struct` não será modificado e uma cópia defensiva não será necessária.

Se o cálculo de distância usar a estrutura imutável, os `ReadOnlyPoint3D` objetos temporários não serão necessários:

```

private static double CalculateDistance3(in ReadonlyPoint3D point1, in ReadonlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}

```

O compilador gera um código mais eficiente quando você chama os membros de um `readonly struct`. A referência `this`, em vez de uma cópia do destinatário, sempre é um parâmetro `in` passado por referência para o método do membro. Essa otimização economiza cópias quando você usa um `readonly struct` como um argumento `in`.

Não transmita um tipo de valor anulável como um `in` argumento. O `Nullable<T>` tipo não é declarado como uma estrutura somente leitura. Isso significa que o compilador deve gerar cópias defensivas para qualquer argumento de tipo de valor anulável passado para um método usando o modificador `in` na declaração de parâmetro.

Você pode ver um programa de exemplo que demonstra as diferenças de desempenho usando o [BenchmarkDotNet](#) em nosso [repositório de exemplos](#) no github. Ele compara a passagem de um struct mutável por valor e por referência com a passagem de um struct imutável por valor e por referência. O uso do struct imutável e da passagem por referência é mais rápido.

Tipos de uso `ref struct`

Use um `ref struct` ou um `readonly ref struct`, como `Span<T>` ou `ReadOnlySpan<T>`, para trabalhar com blocos de memória como uma sequência de bytes. A memória usada pelo span é restrita a um único quadro de pilha. Essa restrição permite que o compilador faça várias otimizações. A principal motivação para esse recurso foi `Span<T>` e as estruturas relacionadas. Você obterá melhorias de desempenho com esses aprimoramentos usando APIs do .NET novas e atualizadas que fazem uso do tipo `Span<T>`.

A declaração de um struct como `readonly ref` combina os benefícios e as restrições das declarações `ref struct` e `readonly struct`. A memória usada pelo alcance somente leitura está restrita a um único registro de ativação, e a memória usada pelo alcance de somente leitura não pode ser modificada.

Você pode ter requisitos semelhantes para trabalhar com a memória criada usando o `stackalloc` ou o usando a memória de APIs de interoperabilidade. Você pode definir seus próprios tipos `ref struct` para essas necessidades.

Uso `nint` e `nuint` tipos

Os [tipos inteiros de tamanho nativo](#) são inteiros de 32 bits em um processo de 32 bits ou inteiros de 64 bits em um processo de 64 bits. Use-os para cenários de interoperabilidade, bibliotecas de nível baixo e para otimizar o desempenho em cenários em que o inteiro de matemática é usado extensivamente.

Conclusões

Usar tipos de valor minimiza o número de operações de alocação:

- Armazenamento para tipos de valor é de pilha alocada para variáveis locais e argumentos de método.
- o armazenamento para tipos de valor que são membros de outros objetos é alocado como parte desse objeto, não como uma alocação separada.
- o armazenamento para valores retornados de tipo de valor é alocado em pilhas.

Compare isso com tipos de referência nestas mesmas situações:

- Armazenamento para tipos de referência é heap alocado para variáveis locais e argumentos de método. A referência é armazenada na pilha.
- O armazenamento para tipos de referência que são membros de outros objetos são alocados separadamente no heap. O objeto recipiente armazena a referência.
- O armazenamento para valores retornados de tipo de referência é alocado em heap. A referência a esse armazenamento é armazenada na pilha.

Minimizar alocações implica compensações. Você copia mais memória quando o tamanho do `struct` é maior que o tamanho de uma referência. Normalmente, uma referência é 64 ou 32 bits e depende da CPU do computador de destino.

Essas compensações geralmente têm o mínimo de impacto no desempenho. No entanto, para estruturas grandes ou coleções maiores, o impacto no desempenho aumenta. O impacto pode ser grande em loops estreitos e em afunilamentos para programas.

Esses aprimoramentos na linguagem C# são criados para algoritmos de desempenho críticos, nos quais as alocações de memória são um importante fator para alcançar o desempenho necessário. Você pode achar que geralmente não usa esses recursos no código que grava. No entanto, esses aprimoramentos foram adotados por meio do .NET. À medida que mais APIs usarem esses recursos, você verá o desempenho dos seus aplicativos melhorar.

Confira também

- [Modificador de parâmetro in \(referência do C#\)](#)
- [ref keyword](#)
- [Ref returns e ref locals](#)

Árvores de expressão

21/01/2022 • 2 minutes to read

Se tiver usado o LINQ, você tem experiência com uma rica biblioteca em que os tipos `Func` fazem parte do conjunto de API. (Se você não estiver familiarizado com o LINQ, provavelmente deseja ler [o tutorial do LINQ](#) e o artigo sobre [expressões lambda](#) antes desta.) As *árvores de expressão* fornecem uma interação mais rica com os argumentos que são funções.

Você escreve argumentos de função, normalmente usando expressões lambda, quando cria consultas LINQ. Em uma consulta LINQ típica, esses argumentos de função são transformados em um delegado que o compilador cria.

Quando quiser ter uma interação mais avançada, você precisa usar *Árvores de expressão*. Árvores de expressão representam o código como uma estrutura que você pode examinar, modificar ou executar. Essas ferramentas oferecem a capacidade de manipular o código em tempo de execução. Você pode escrever código que examina algoritmos em execução ou injeta novos recursos. Em cenários mais avançados, você pode modificar algoritmos em execução e até mesmo converter expressões C# para outro formato para execução em outro ambiente.

Provavelmente, você já escreveu código usando Árvores de expressão. APIs do LINQ do Entity Framework aceitam Árvores de expressão como os argumentos para o padrão de expressão de consulta do LINQ. Isso permite que o [Entity Framework](#) converta a consulta que você escreveu em C# em SQL, que é executado no mecanismo do banco de dados. Outro exemplo é [Moq](#), que é uma estrutura de simulação popular para .NET.

As seções restantes deste tutorial explorarão o que são as árvores de expressão, examinarão as classes de estrutura que dão suporte a árvores de expressão e mostrarão como trabalhar com árvores de expressão. Você aprenderá a ler árvores de expressão, criar árvores de expressão, criar árvores de expressão modificadas e executar o código representado pelas árvores de expressão. Após a leitura, você estará pronto para usar essas estruturas para criar algoritmos adaptáveis avançados.

1. Árvores de Expressão Explicadas

Compreender a estrutura e os conceitos por trás das *Árvores de Expressão*.

2. Tipos de Framework com suporte a árvores de expressão

Saiba mais sobre as estruturas e classes que definem e manipulam as árvores de expressão.

3. Executando Expressões

Saiba como converter uma árvore de expressão representada como uma expressão lambda em um delegado e como executar o delegado resultante.

4. Interpretando Expressões

Saiba como percorrer e examinar *árvores de expressão* para entender que código a árvore de expressão representa.

5. Compilando Expressões

Saiba como construir os nós de uma árvore de expressão e compilar árvores de expressão.

6. Traduzindo Expressões

Saiba como compilar uma cópia modificada de uma árvore de expressão ou converter uma árvore de expressão em um formato diferente.

7. Resumindo

Examine informações sobre as árvores de expressão.

Árvores de Expressão Explicadas

21/01/2022 • 4 minutes to read

Anterior – Visão geral

Uma Árvore de expressão é uma estrutura de dados que define o código. Elas se baseiam nas mesmas estruturas que um compilador usa para analisar o código e gerar a saída compilada. Ao ler este tutorial, você notará certa semelhança entre árvores de expressão e os tipos usados nas APIs Roslyn para criar [Analyzers e CodeFixes](#). (Analizadores e CodeFixes são NuGet pacotes que executam análise estática no código e podem sugerir 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 diferente das APIs Roslyn.

Vejamos um exemplo simples. Aqui está uma linha de código:

```
var sum = 1 + 2;
```

Se você analisar isso como uma árvore de expressão, 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`. Essa é 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`)

Isso pode parecer complicado, mas é muito eficiente. Seguindo o mesmo processo, você pode decompor expressões muito mais complicadas. Considere esta expressão:

```
var finalAnswer = this.SecretSauceFunction(  
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),  
    decisionServer.considerFinalOptions("hello")) +  
    MoreSecretSauce('A', DateTime.Now, true);
```

A expressão acima também é uma declaração de variável com uma atribuição. Neste exemplo, o lado direito da atribuição é uma árvore muito mais complicada. Eu não vou decompor essa expressão, mas considere quais seriam os diferentes nós. Há chamadas de método usando o objeto atual como um receptor, uma que tem um receptor `this` explícito e outra que não. Há chamadas de método usando outros objetos receptores, há argumentos constantes de tipos diferentes. E, finalmente, há um operador de adição binária. 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 dessa complexidade, a expressão acima cria uma estrutura de árvore que pode ser percorrida tão facilmente quanto o primeiro exemplo. Você pode continuar percorrendo os nós filho para encontrar os nós folha na expressão. Os nós pai terão referências aos filhos e 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 poderá entender 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 do C# pode analisar os programas em C# mais complexos e criar a saída apropriada desse código-fonte complicado.

Uma vez que estiver familiarizado com a estrutura das árvores de expressão, você descobrirá que o conhecimento adquirido permitirá que você trabalhe com muitos outros cenários ainda mais avançados. Há um poder incrível nas árvores de expressão.

Além de mover algoritmos para serem executados em outros ambientes, as árvores de expressão podem ser usadas para tornar mais fácil escrever algoritmos que inspecionam o código antes de executá-lo. Você pode escrever um método cujos argumentos são expressões e, em seguida, examinar essas expressões antes de executar o código. A árvore de expressão é uma representação completa do código: você pode ver os valores de qualquer subexpressão. Você pode ver os nomes de métodos e propriedades. Você pode ver o valor de qualquer expressão de constante. Você também pode converter uma árvore de expressão em um delegado executável e executar 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, algumas expressões de C# não podem ser criadas 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 os cria usando loops `for`, `foreach`, `while` ou `do`. Como você verá [mais adiante nesta série](#), as APIs para árvores de expressão oferecem suporte a uma única expressão de loop, com expressões `break` e `continue` que controlam 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.

[Próximo – Tipos de estruturas que dão suporte às árvores de expressão](#)

Tipos de Framework com suporte a árvores de expressão

21/01/2022 • 3 minutes to read

[Anterior – Árvores de expressão explicadas](#)

Há uma grande lista de classes do .NET Core Framework que funcionam com árvores de expressão. Veja a lista completa em [System.Linq.Expressions](#). Em vez de percorrer a lista completa, vamos entender como as classes de estrutura 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).

Tudo começa com `System.Linq.Expression`

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. Uma vez que você souber qual o tipo do nó, poderá 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. Eu segui a prática de verificar o tipo de nó, converter em uma expressão de acesso variável e verificar as propriedades do tipo de expressão específico:

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.First();

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Criando 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 de seus nós folha. Por exemplo, esse código cria uma expressão de Adicionar:

```
// 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#.

Navegando nas 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, aqui estão as técnicas que eu uso para trabalhar com Árvores de expressão:

1. Observar os membros da enumeração `ExpressionType` para determinar possíveis nós que devem ser examinados. Isso é realmente 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 ao examinar cada uma dessas três áreas. Invariavelmente, você encontrará o que precisa ao começar com uma dessas três etapas.

[Próximo – Executar árvores de expressão](#)

Executar árvores de expressão

21/01/2022 • 6 minutes to read

[Anterior – Tipos de estruturas que dão suporte às árvores de expressão](#)

Uma *árvore de expressão* é uma estrutura de dados que representa algum código. Ela não é código compilado nem executável. Se você quiser executar o código do .NET que é representado por uma árvore de expressão, é necessário convertê-lo em instruções IL executáveis.

Expressões lambda para funções

Você pode converter qualquer `LambdaExpression` ou qualquer tipo derivado de `LambdaExpression` em IL executável. Outros tipos de expressão 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 uma `ConstantExpression`. Isso significa que algo é útil?) Qualquer árvore de expressão que seja um `LambdaExpression`, ou um tipo derivado de `LambdaExpression` pode ser convertida em IL. O tipo de expressão `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.

Na maioria dos casos, isso cria um mapeamento simples entre uma expressão e o delegado correspondente. Por exemplo, uma árvore de expressão que é 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 `LambdaExpression` contém membros `Compile` e `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 `MethodInfo` com a IL que representa a saída compilada da árvore de expressão. Observe que `CompileToMethod` só está disponível na estrutura de área de trabalho completa, não no .NET Core.

Opcionalmente, você também pode fornecer um `DebugInfoGenerator` que receberá as informações de depuração de símbolo para o objeto delegado gerado. Isso permite que você converta a árvore de expressão em um objeto delegado e tenha as informações de depuração completas sobre o delegado gerado.

Uma expressão seria convertida em um delegado usando o seguinte código:

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

Observe que o tipo delegado se baseia 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 para o tipo delegado correto para fazer com que as ferramentas de tempo de compilação verifiquem a lista de argumentos ou o tipo de retorno.

Execução e tempos de vida

O código é executado ao invocar o delegado que foi criado quando você chamou `LambdaExpression.Compile()`.

Você pode ver isso acima do local em que `add.Compile()` retorna um delegado. Invocar o delegado, chamando `func()`, 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).

Eu alerto contra a tentativa de criar mecanismos de cache mais sofisticados para aumentar o desempenho ao evitar chamadas de compilação desnecessárias. A comparação de duas árvores de expressão arbitrárias para determinar se elas representam o mesmo algoritmo também será algo demorado para executar. Provavelmente você descobrirá que o tempo de computação economizado para evitar quaisquer chamadas extras à `LambdaExpression.Compile()` será mais demorado que o tempo da execução do código que determina se 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.

Em geral, o compilador garantirá que isso seja verdadeiro. No entanto, se sua expressão acessa uma variável que implementa `IDisposable`, é possível que seu código descarte o objeto enquanto ele ainda é mantido pela árvore de expressão.

Por exemplo, esse código funciona bem, porque `int` não implementa `IDisposable`:

```
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 essa classe (bastante artificial) que implementa `IDisposable`:

```

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;
    }
}

```

Ao usá-la em uma expressão, como mostrado abaixo, você obterá uma `ObjectDisposedException` ao executar o código referenciado pela propriedade `Resource.Argument`:

```

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, quando você executar o delegado retornado desse método, você terá um `ObjectDisposedException` lançado no ponto de execução.

Parece realmente estranho ter um erro de runtime que representa um constructo de tempo de compilação, mas esse é o mundo em que entramos quando trabalhamos com árvores de expressão.

Há muitas permutações desse problema, portanto é difícil oferecer diretrizes gerais para evitá-lo. Tenha cuidado ao acessar variáveis locais quando estiver definindo expressões e tenha cuidado ao acessar o estado no objeto atual (representado por `this`) quando estiver criando uma árvore de expressão que pode ser retornada por 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 chamado. Você vai se deparar com uma `ReferencedAssemblyNotFoundException` nos casos em que ele não estiver presente.

Resumo

As árvores de expressão que representam expressões lambda podem ser compiladas para criar um delegado que pode ser executado. Isso fornece 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 bastante previsíveis e serão capturados em seus primeiros testes de qualquer código usando as árvores de expressão.

[Próximo – Interpretando expressões](#)

Interpretando Expressões

21/01/2022 • 14 minutes to read

Anterior – Executando expressões

Agora, vamos escrever código para examinar a estrutura de um *árvore de expressão*. Cada nó em uma árvore de expressão será um objeto de uma classe derivada de `Expression`.

Esse design torna visitar todos os nós em uma árvore de expressão 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.

Examinando 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:

```
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}");
```

Isso imprimirá o seguinte:

```
This is 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. Este é o código:

Examinando uma expressão de adição simples

Vamos começar com o exemplo de adição da introdução desta seção.

```
Expression<Func<int>> sum = () => 1 + 2;
```

Não estou usando `var` para declarar essa árvore de expressão, pois isso não é possível porque o lado direito da atribuição é de um tipo implícito.

O nó raiz é um `LambdaExpression`. Para obter o código interessante no lado direito do operador, você precisa encontrar um dos `=>` filhos do `LambdaExpression`. Faremos 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, precisaremos visitar recursivamente alguns nós. Esta é uma primeira implementação simples:

```

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($"{Environment.NewLine}\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($"{Environment.NewLine}\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"{Environment.NewLine}\tParameter Type: {right.Type.ToString()}, Name: {right.Name}");

```

Este exemplo imprime a seguinte 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

```

Você vai notar que há muita repetição no exemplo de código acima. 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:

```

// 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 => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:

```

```

        return new ParameterVisitor((ParameterExpression)node);
    case ExpressionType.Add:
        return new BinaryVisitor((BinaryExpression)node);
    default:
        Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
        return default(Visitor);
    }
}

// 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($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.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($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.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.ToString()}, 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 pode visitar qualquer `LambdaExpression` arbitrário. Há muitos orifícios, ou seja, que o código que criei procura apenas uma amostra muito pequena 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 que precisa adicionar um novo tipo de expressão.)

Quando executa esse visitante na expressão de adição mostrada acima, você obtém a saída a seguir:

```

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.

Examinando uma expressão de adição com vários níveis

Vamos testar um exemplo mais complicado, mas ainda limitar os tipos de nó somente à adição:

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Antes de executar isso no algoritmo de visitante, tente pensar no que poderia ser 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:

```
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 *deixada associativa*.

Para executar esse exemplo e ver a árvore de expressão completa, eu 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:

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Crie um visitante para essa soma e execute o visitante; você verá esta 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 também pode executar qualquer um dos outros exemplos pelo código visitante e ver que árvore ele representa. Veja um exemplo da expressão `sum3` acima (com um parâmetro adicional para impedir que o compilador calcule a constante):

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Esta é a saída do visitante:

```
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 deste 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 ele funcione para isto:

```
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 escrevi este 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 e outras estruturas de controle comuns em C#. Estou limitado ao uso de expressões. Em segundo lugar, não posso chamar recursivamente a mesma expressão. Eu poderia se ela já fosse um delegado, mas não posso chamá-la em sua forma de árvore de expressão. Na seção sobre como [criar árvores de expressão](#), você aprenderá técnicas para superar essas limitações.

Nesta expressão, você encontrará todos esses tipos de nós:

1. Igual (expressão binária)

2. Multiplicar (expressão binária)
3. Condicional (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 terá cisto todos os nós potenciais. Então, você tem tudo de que você precisa. O resultado seria algo semelhante a:

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

O `ConditionalVisitor` e o `MethodCallVisitor` processam esses dois nós:

```

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($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{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:

```
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
```

Estendendo a biblioteca de exemplo

Os exemplos nesta seção mostram as principais técnicas para visitar e examinar nós em uma árvore de expressão. Eu encobri várias ações de que talvez você precise para nos concentrarmos 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 poderá alimentá-lo com muitas expressões que o fariam falhar. Uma implementação completa é incluída no .NET Standard sob 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. Eu o compus para deixar as estruturas usadas claras e para realçar as técnicas usadas para visitar os nós e analisar o que há lá. Uma implementação de produção dedicaria mais atenção ao desempenho do que eu dediquei.

Mesmo com essas limitações, você deve estar bem no processo de escrever algoritmos que leem e entendem árvores de expressão.

[Próximo – Compilando expressões](#)

Criando árvores de expressão

21/01/2022 • 5 minutes to read

Anterior – Interpretando expressões

Todas as árvores de expressão que você viu até agora foram criadas pelo compilador C#. Tudo que eu tive que fazer foi criar uma expressão lambda que foi atribuída a uma variável tipada como um `Expression<Func<T>>` ou algum tipo semelhante. Essa não é a única maneira de criar uma árvore de expressão. Para muitos cenários, você pode achar que precisa criar uma expressão na memória em tempo de execução.

Criar árvores de expressão é complicado pelo fato de que essas árvores de expressão são imutáveis. Ser imutável significa que você precisa criar a árvore de folhas até a raiz. As APIs que você usará para criar as árvores de expressão refletem esse fato: os métodos que você usará para criar um nó usam todos os seus filhos como argumentos. Vejamos alguns exemplos para mostrar as técnicas a você.

Criando nós

Vamos começar de forma relativamente simples mais uma vez. Vamos usar a expressão de adição com que tenho trabalhado durante essas seções:

```
Expression<Func<int>> sum = () => 1 + 2;
```

Para construir essa árvore de expressão, você precisará criar os nós de folha. Os nós de folha são constantes, de modo que você pode usar o método `Expression.Constant` para criar os nós:

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Em seguida, você criará a expressão de adição:

```
var addition = Expression.Add(one, two);
```

Depois que tiver a expressão de adição, você pode criar a expressão lambda:

```
var lambda = Expression.Lambda(addition);
```

Essa é uma expressão lambda muito simples, pois não contém argumentos. Posteriormente nesta seção, você verá como mapear argumentos para parâmetros e criar expressões mais complicadas.

Para expressões que são simples como essa, você pode combinar todas as chamadas em uma única instrução:

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Criando uma árvore

Estes são 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:

```
Expression<Func<double, double, double>> distanceCalc =  
    (x, y) => Math.Sqrt(x * x + y * y);
```

Você começará criando expressões de parâmetro para `x` e `y`:

```
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:

```
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`.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });  
var distance = Expression.Call(sqrtMethod, sum);
```

Depois, por fim, você coloca a chamada de método em uma expressão lambda e define os argumentos para a expressão lambda:

```
var distanceLambda = Expression.Lambda(  
    distance,  
    xParameter,  
    yParameter);
```

Neste exemplo mais complicado, você verá mais algumas técnicas de que frequentemente precisará 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 `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.

Criando código profundamente

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:

```

Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};

```

Acima, observe que eu não criei a árvore de expressão, apenas 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. Ele é complicado pelo fato de que não há uma API para criar um loop `while`. Em vez disso, você precisa criar um loop que contém um teste condicional e um destino para o rótulo para interromper o loop.

```

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 gostamos de evitar em nossas tarefas de codificação cotidianas.

Para esta seção, também atualizei o código de visitante para visitar cada nó nessa árvore de expressão e gravar informações sobre os nós que são 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. Para obter instruções de download, consulte [Exemplos e tutoriais](#).

Examinando as APIs

As APIs de árvore de expressão são algumas das mais difíceis de navegar no .NET Core, mas não tem problema. Sua finalidade é uma tarefa bastante complexa: escrever código que gera código em tempo de execução. Eles são necessariamente complicadas para fornecer um equilíbrio entre dar suporte a todas as estruturas de

controle disponíveis na linguagem C# e manter a área de superfície das APIs tão pequena quanto for razoável. Esse equilíbrio significa que muitas estruturas de controle são representadas não por seus constructos em C#, mas por constructos que representam a lógica subjacente que o compilador gera desses constructos de nível superior.

Além disso, no momento, há expressões de C# que não podem ser criadas diretamente usando os métodos de classe `Expression`. Em geral, esses serão os operadores e expressões mais novos adicionadas no C# 5 e no C# 6. (Por exemplo, expressões `async` não podem ser criadas e o novo operador `?.` não pode ser criado diretamente.)

[Próximo – Traduzindo expressões](#)

Traduzir árvores de expressões

21/01/2022 • 6 minutes to read

Anterior – Compilando expressões

Nesta seção final, você aprenderá a visitar cada nó em uma árvore de expressão, enquanto estiver criando uma cópia modificada dessa árvore de expressão. Essas são as técnicas que você usará em dois cenários importantes. O primeiro é entender os algoritmos expressados por uma árvore de expressão para que ela possa ser movida para outro ambiente. O segundo é quando você deseja alterar o algoritmo que foi criado. Isso poderia ser feito para adicionar registro em log, interceptar chamadas de método e monitorá-las ou outras finalidades.

Mover é visitar

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 colocou na árvore.

Vamos ver isso em ação, visitando uma árvore de expressão e criando uma nova árvore com alguns nós de substituição. Neste exemplo, vamos substituir qualquer constante com uma constante que seja dez vezes maior. Caso contrário, vamos deixar a árvore de expressão intacta. Em vez de ler o valor da constante e substituí-la por uma nova constante, faremos essa substituição através da troca do nó constante por um novo nó que executa a multiplicação.

Aqui, quando você encontrar um nó constante, você criará um novo nó de multiplicação cujos filhos serão a constante original e a constante `10`:

```
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;
}
```

Ao substituir o nó original pelo substituto, uma nova árvore será formada, contendo as nossas modificações. Podemos verificar isso compilando e executando a árvore substituída.

```

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.

Este exemplo mostra a importância das árvores de expressão serem imutáveis. Observe que a nova árvore criada acima contém uma mistura de nós recém-criados e nós da árvore existente. E isso é seguro, porque os nós da árvore existente não podem ser modificados. Isso pode resultar 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 isso criando um segundo visitante que percorre a árvore de nós de adição e calcula o resultado. Você pode fazer isso com algumas modificações no visitante que utilizamos até agora. Nessa nova versão, o visitante retornará a soma parcial da operação de adição até este ponto. Para uma expressão constante, esse é simplesmente o valor da expressão constante. Para uma expressão de adição, o resultado será a soma dos operandos esquerdos e direitos, uma vez que essas árvores forem percorridas.

```

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 we 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 bastante acessíveis. Esse código visita filhos em uma pesquisa de profundidade inicial. Ao encontrar um nó constante, o visitante retorna o valor da constante. Depois que o visitante visitar ambos os filhos, esses filhos terão calculado 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 será 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:

```

private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"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 mesma expressão produz o seguinte resultado:

```

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 acima. 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`:

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Aqui está a saída ao examinar essa expressão:

```
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 completamente diferente. Os nós são percorridos em uma ordem diferente, porque a árvore foi construída com operações diferentes que ocorrem primeiro.

Para saber 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 uma discussão completa a respeito de todo o trabalho necessário para compilar uma biblioteca de finalidade geral que move árvores de expressão para outra linguagem, leia [esta série](#) escrita por 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 poder das árvores de expressão. Você pode examinar um conjunto de códigos, fazer as alterações que desejar nesse código e executar a versão modificada. Como as árvores de expressão são imutáveis, você pode criar novas árvores usando os componentes de árvores existentes. Isso minimiza a quantidade de memória necessária para criar árvores de expressão modificadas.

[Próximo – Resumindo](#)

Resumo de árvores de expressão

21/01/2022 • 2 minutes to read

[Anterior – Movendo expressões](#)

Nesta série, você viu como é possível usar as *árvores de expressão* para criar programas dinâmicos que interpretam o código como dados e criam uma nova funcionalidade com base nesse código.

Você pode examinar as árvores de expressão para entender a intenção de um algoritmo. E não apenas examinar esse código. Você pode criar novas árvores de expressão que representam versões modificadas do código original.

E também pode usar as árvores de expressão para examinar um algoritmo e mover esse algoritmo para outra linguagem ou ambiente.

Limitações

Há alguns elementos mais recentes da linguagem C# que não se convertem bem em árvores de expressão. As árvores de expressão não podem conter expressões `await` ou expressões lambda `async`. Muitos dos recursos adicionados na versão 6 do C# não aparecem exatamente como escritos nas árvores de expressão. Em vez disso, os recursos mais recentes serão expostos em árvores de expressões na sintaxe anterior equivalente. Isso pode não ser realmente uma limitação como você imagina. Na verdade, isso significa que é provável que o seu código, que interpreta árvores de expressão, vai continuar funcionando da mesma forma que quando os novos recursos de linguagem forem introduzidos.

Mesmo com essas limitações, as árvores de expressão permitem criar algoritmos dinâmicos que se apoiam na interpretação e modificação do código que é representado como uma estrutura de dados. Elas são uma ferramenta poderosa e é um dos recursos do ecossistema do .NET que habilita as avançadas bibliotecas, como o Entity Framework, a realizarem o que fazem.

Interoperabilidade (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

A interoperabilidade permite que você mantenha e aproveite os investimentos existentes em código não gerenciado. O código que é executado sob o controle do CLR (Common Language Runtime) é chamado de *código gerenciado*, e o código que é executado fora do CLR é chamado de *código não gerenciado*. COM, COM+, componentes do C++, componentes do ActiveX e a API do Microsoft Windows são exemplos de código não gerenciado.

O .NET habilita a interoperabilidade com código não tratado por meio de serviços de invocação de plataforma, [System.Runtime.InteropServices](#) namespace, interoperabilidade C++ e interoperabilidade COM (interoperabilidade COM).

Nesta seção

[Visão geral sobre interoperabilidade](#)

Descreve métodos para fins de interoperabilidade entre código gerenciado em C# e código não gerenciado.

[Como acessar objetos de interoperabilidade do Office usando recursos do C#](#)

Descreve os recursos que são introduzidos no Visual C# para facilitar a programação do Office.

[Como usar propriedades indexadas na programação para interoperabilidade COM](#)

Descreve como usar propriedades indexadas para acesso propriedades COM que têm parâmetros.

[Como usar invocação de plataforma para executar um arquivo WAV](#)

Descreve como usar os serviços de invocação de plataforma para reproduzir um arquivo de som .wav no sistema operacional Windows.

[Passo a passo: programação do Office](#)

Mostra como criar uma planilha do Excel e um documento do Word com um link para a planilha.

[Exemplo de classe COM](#)

Demonstra como expor uma classe C# como um objeto COM.

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

- [Marshal.ReleaseComObject](#)
- [Guia de Programação em C#](#)
- [Interoperação com código não gerenciado](#)
- [Passo a passo: programação do Office](#)

Controle de versão em C#

21/01/2022 • 5 minutes to read

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 etc. ao aplicar informações de versão à sua biblioteca .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: ao atualizar um comportamento de método exige que você altere sua assinatura também, você deve criar uma sobrecarga para que a chamada de código para 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.

NOTE

Tornar os argumentos repulsivos opcionais deve ter muito pouco efeito, especialmente se ele 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 criar suas bibliotecas de forma que as informações que provavelmente sejam alterados regularmente sejam armazenadas no arquivo. Dessa forma, quando essas informações são atualizadas, o arquivo de configuração de versões mais antigas precisa ser substituído pelo novo sem a necessidade de recomilação da `app.config` 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 `arquivoapp.config` para atualizar a versão de uma biblioteca que seu aplicativo usa. Ao adicionar 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 você atualizaria o `arquivoapp.config` aplicativo para usar a versão de patch do em vez da versão com a qual ele `1.0.1` foi `ReferencedLibrary 1.0.0` compilado originalmente.

```
<dependentAssembly>
  <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
  <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

NOTE

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:

```
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

```
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.

```
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

```
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 das técnicas discutidas e sua compreensão das situações em que usá-las serão muito longos para a adoção da transição entre as versões de uma biblioteca.

Instruções (C#)

21/01/2022 • 3 minutes to read

Na seção como do guia do C#, você pode 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 várias dicas e truques que são práticas de desenvolvedor C# comuns:

- [Iniciar 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

Você cria classes, registros e estruturas para implementar seu programa. Essas técnicas costumam ser usadas ao escrever classes, registros ou estruturas.

- [Declare propriedades implementadas automaticamente.](#)
- [Declare e use propriedades de leitura/gravação.](#)
- [Defina constantes.](#)
- [Substitua o método `ToString` para fornecer saída de cadeia de caracteres.](#)
- [Definir propriedades abstratas.](#)
- [Use os recursos de documentação para documentar seu código.](#)
- [Implemente membros de interface explicitamente para manter a interface pública concisa.](#)
- [Implemente explicitamente membros de duas interfaces.](#)

Trabalhando com coleções

Esses artigos ajudam você a trabalhar com coleções de dados.

- [Iniciar 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.

- [Comparar 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 `int`.
- 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.
- Conformidade com as diretrizes do .net quando seu código publica 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.
- Classificar elementos em várias 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#

21/01/2022 • 2 minutes to read

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 é geralmente a maneira mais fácil de separar uma cadeia de caracteres em limites de palavras. Ele também é usado para dividir cadeias de caracteres em outras cadeias ou caractere específico.

NOTE

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.

```
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 um separador.

```
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 torna mais fácil para formatos como arquivos de valores separados por vírgulas (CSV) que representam dados tabulares. 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 usa espaços, vírgulas, pontos, dois-pontos e tabulações como separando caracteres, que são passados para [Split](#) em uma matriz. O loop, na parte inferior do código, exibe cada uma das palavras na matriz retornada.

```

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:

```

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).

```

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 do C#)

21/01/2022 • 4 minutes to read

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.

NOTE

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 capacidade de leitura no código-fonte. O código concatena as cadeias de caracteres menores para criar o literal de cadeia de caracteres longo. As partes são concatenadas em uma única cadeia de caracteres no tempo de compilação. Não há nenhum custo de desempenho em tempo de execução, independentemente do número de cadeias de caracteres envolvidas.

```
// 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 `+` `+=` `String.Format` `String.Concat` `String.Join` caracteres, você pode usar os operadores ou , interpolação de cadeia de caracteres ou os métodos , `StringBuilder.Append` ou . 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:

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// 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:

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}." ;
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

NOTE

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 os espaço reservados também são cadeias de caracteres constantes.

String.Format

Outro método para concatenar cadeias de caracteres é [String.Format](#). Esse método funciona bem quando você está criando uma cadeia de caracteres de um pequeno número de cadeias de caracteres de componente.

StringBuilder

Em outros casos, você pode estar combinando cadeias de caracteres em um loop em que não sabe quantas cadeias de caracteres de origem você está combinando e 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.

```
// 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 StringBuilder](#)

classe.

[String.Concat](#) OU [String.Join](#)

Outra opção para unir cadeias de caracteres de uma coleção é usar o método [String.Concat](#). Use [StringJoin](#) ou método se as cadeias de caracteres de origem devem ser separadas por um separador. O código a seguir combina uma matriz de palavras usando os dois métodos:

```
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:

```
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 ela cria uma cadeia de caracteres intermediária para cada iteração. Se a otimização do desempenho for crítica, considere [StringBuilder](#) a classe ou o método [String.Concat](#) ou [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 cadeias de caracteres

21/01/2022 • 4 minutes to read

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.

NOTE

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 [String.Contains](#) [String.StartsWith](#) métodos, e [String.EndsWith](#) pesquisam uma cadeia de caracteres para um texto específico. O exemplo a seguir mostra cada um desses métodos e uma variação que usa uma pesquisa que não diferencia maiúsculas de minúsculas:

```
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 [StringComparison.CurrentCultureIgnoreCase](#) valor de enumeração 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.

```

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 a seguir usa o único `\`, que deve ter escape como `\\` em uma cadeia de caracteres C#).

PADRÃO	SIGNIFICADO
<code>the</code>	corresponder ao texto "the"
<code>(eir)?</code>	corresponder a 0 ou 1 ocorrência de "eir"
<code>\s</code>	corresponder a um caractere de espaço em branco

```

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();
    }
}

```

TIP

Os métodos `string` são geralmente melhores opções quando você está procurando por uma cadeia de caracteres exata. As 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

```
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 usar cadeias de caracteres no .NET](#)

Como modificar o conteúdo da cadeia de caracteres em C#

21/01/2022 • 6 minutes to read

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 novo modificado `string` ao executar cada exemplo.

NOTE

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.

```
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 '_':

```
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.

```
// 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:

```
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. Para obter mais informações, consulte [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](#).

```

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 localizar 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.

```

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);

```

Criar programaticamente o conteúdo da cadeia de caracteres

Como as cadeias de caracteres são imutáveis, todos os exemplos anteriores criam cadeias temporárias ou matrizes de caractere. Em cenários de alto desempenho, pode ser desejável evitar essas alocações de heap. O .NET Core fornece um [String.Create](#) método que permite que você preencha programaticamente o conteúdo de caractere de uma cadeia de caracteres por meio de um retorno de chamada enquanto evita as alocações de

cadeia de caracteres temporárias intermediárias.

```
// 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 **não é recomendável** modificar o conteúdo da cadeia de caracteres depois que uma cadeia de caracteres é criada. Isso interromperá as coisas de maneiras imprevisíveis. Por exemplo, se alguém estagiárior uma cadeia de caracteres que tenha o mesmo conteúdo que o seu, ele obterá 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#

21/01/2022 • 12 minutes to read

Você compara cadeias de caracteres para responder uma dessas 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.

NOTE

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 `!=` e, respectivamente

Execute uma comparação ordinal que diferencia maiúsculas de minúsculas e, no caso de `String.Equals` um `StringComparison` argumento, pode ser fornecido para alterar suas regras de classificação. O exemplo a seguir demonstra que:

```
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 as 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.

O teste para igualdade com `String.Equals` e os `==` `!=` operadores and é diferente da comparação de cadeias de caracteres usando os `String.CompareTo` `Compare(String, String)` métodos e. Embora os testes de igualdade executem uma comparação ordinal que diferencia maiúsculas de minúsculas, os métodos de comparação executam uma comparação de diferenciação de maiúsculas e minúsculas, com a cultura atual. Como os métodos de comparação padrão geralmente fazem diferentes tipos de comparações, recomendamos que você sempre esclareça a intenção do seu código, chamando uma sobrecarga que especifica explicitamente o tipo de comparação a ser feita.

Comparações ordinais que não diferenciam maiúsculas de minúsculas

O método `String.Equals(String, StringComparison)` permite que você especifique um valor `StringComparison` de `StringComparison.OrdinalIgnoreCase` para uma comparação ordinal 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:

```
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, 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

As cadeias de caracteres também podem ser ordenadas usando regras linguísticas para a cultura atual. Às vezes, isso é conhecido como "ordem de classificação de palavra". Quando você executa uma comparação linguística, alguns caracteres Unicode não alfanuméricos podem ter pesos especiais atribuídos. Por exemplo, o hífen "-" pode ter um pequeno peso atribuído para que "co-op" e "Coop" apareçam ao lado um do outro na ordem de classificação. Além disso, alguns caracteres Unicode podem ser equivalentes a uma sequência de instâncias `Char`. O exemplo a seguir usa a frase "They dance in the street." em alemão, com o "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".

```

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");
}

```

Este exemplo demonstra a natureza dependente de sistema operacional das comparações linguísticas. O host da janela interativa é um host Linux. As comparações de linguísticas e ordinais produzem os mesmos resultados. se você executar esse mesmo exemplo em um host Windows, verá a seguinte saída:

```

<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison

```

No Windows, 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.

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":

```

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.Ordinal);
    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. Além disso, experimente este código de exemplo localmente em um computador Windows, e você obterá os seguintes resultados:

```

<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison

```

As comparações linguísticas dependem da cultura atual e são dependentes do SO. Leve isso em conta ao trabalhar com comparações de cadeias de caracteres.

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:

```
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\rSorted 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 foi encontrada, o valor retornado indicará onde seria encontrado.

```

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 de minúsculas. Você usaria os métodos [String.Compare](#) estáticos para especificar regras de comparação diferentes.

```
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\rSorted 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:

```

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](#).

Igualdade de referência e centralização de cadeia de caracteres

Nenhum dos exemplos usou [ReferenceEquals](#). Esse método determina se duas cadeias de caracteres são o mesmo objeto, o que pode levar a resultados inconsistentes em comparações de cadeias de caracteres. O exemplo a seguir demonstra o recurso de *centralização da cadeia de caracteres* do C#. Quando um programa declara duas ou mais variáveis de cadeia de caracteres idênticas, o compilador armazena todas no mesmo local. Chamando o método [ReferenceEquals](#), você pode ver que as duas cadeias de caracteres na verdade se referem ao mesmo objeto na memória. Use o método [String.Copy](#) para evitar a centralização. Depois que a cópia for feita, as duas cadeias de caracteres terão locais de armazenamento diferentes, mesmo que tenham o mesmo valor. Execute o exemplo a seguir para mostrar que as cadeias de caracteres `a` e `b` são *centralizadas*, ou seja, que elas compartilham o mesmo armazenamento. As cadeias de caracteres `a` e `c` não são.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

Quando você testa cadeias de caracteres quanto a igualdade, é necessário usar os métodos que especificam explicitamente o tipo de comparação que você pretende executar. O código fica muito mais legível e fácil de manter. Use as sobrecargas dos métodos das classes [System.String](#) e [System.Array](#) que aceitam um parâmetro de enumeração [StringComparison](#). Você especifica o tipo de comparação a ser executado. Evite usar os operadores `==` e `!=` ao testar a igualdade. Os métodos de instância [String.CompareTo](#) sempre executam uma comparação ordinal que diferencia maiúsculas de minúsculas. Basicamente, eles são adequados para colocar cadeias de caracteres em ordem alfabética.

Você pode internalizar uma cadeia de caracteres ou recuperar uma referência a uma cadeia de caracteres interna existente chamando o método [String.Intern](#). Para determinar se uma cadeia de caracteres está internalizada, chame o método [String.IsInterned](#).

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

21/01/2022 • 2 minutes to read

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.

```
// 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

21/01/2022 • 6 minutes to read

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. Essa é a missão principal das APIs do Roslyn: abrir as caixas opacas e permitir que as ferramentas e os usuários finais compartilhem na riqueza de compiladores de informações 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 muitas oportunidades de inovação em áreas como metaprogramação, geração de código e transformação, uso interativo das linguagens c# e Visual Basic e incorporação de c# e Visual Basic em linguagens específicas de domínio.

o SDK do .NET Compiler Platform permite que você crie *analisadores* e *correções de código* que encontrem e corrijam erros de codificação. _ 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 lidar com erros de codificação encontrados por analisadores ou diagnóstico de 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 geralmente levam a bugs, código não sustentável ou violação de diretriz padrão.

além de analisadores e correções de código, o SDK do .NET Compiler Platform também permite que você crie *refatorações de código*. ele também fornece um conjunto único de APIs que permitem que você examine e entenda 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.

TIP

Antes de criar seu próprio analisador, confira os internos. Para obter mais informações, consulte [regras de estilo de código](#).

Fornecer diretrizes com pacotes de biblioteca

Há uma infinidade de bibliotecas disponíveis para desenvolvedores do .NET em 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, por meio de experiência, padrões que funcionam bem e padrões que são mais bem 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 guias de início rápido, exemplos e 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. Selecione **Modificar**
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. Verifique a caixa para o **editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Selecione **Modificar**
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**.

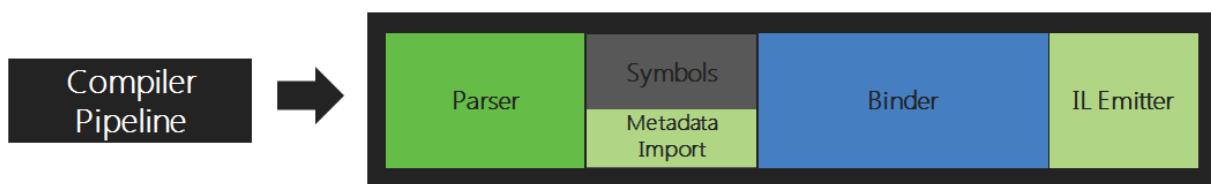
Entender o modelo do SDK do .NET Compiler Platform

21/01/2022 • 3 minutes to read

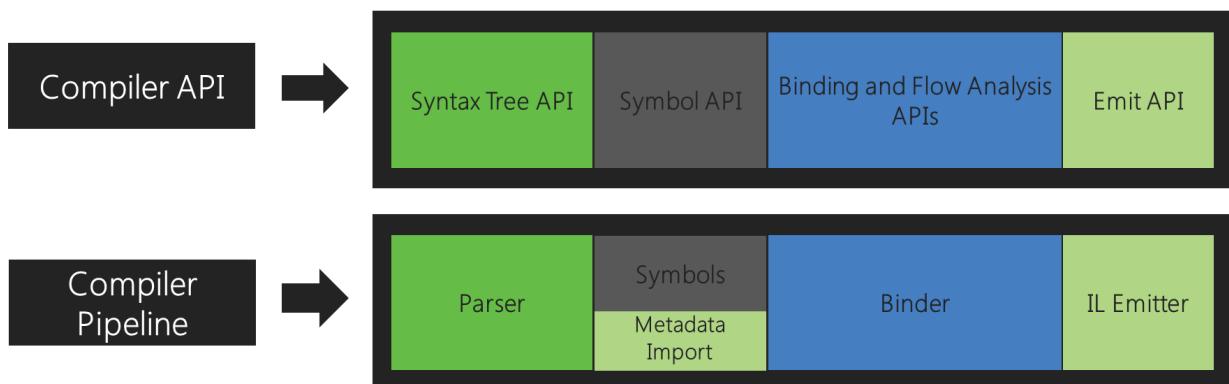
Os compiladores processam o código escrito seguindo regras estruturadas que geralmente diferem da forma como os humanos leem e entendem um código. Uma compreensão básica do modelo usado pelos compiladores é essencial para compreender as APIs usadas ao criar ferramentas baseadas no Roslyn.

Áreas funcionais do pipeline do compilador

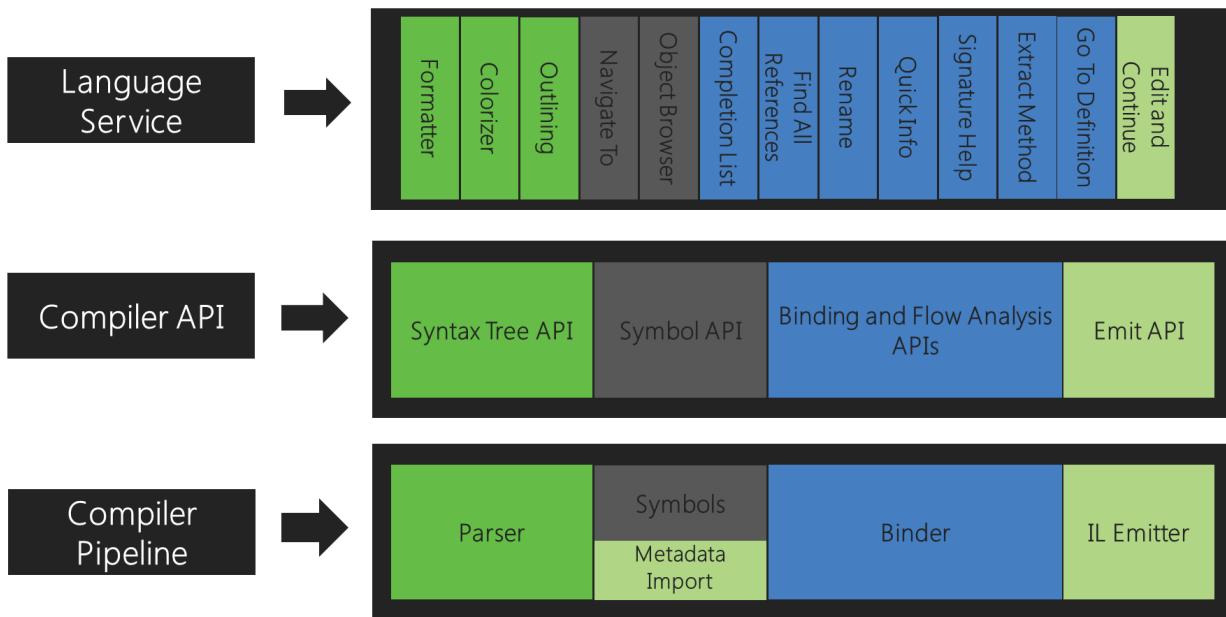
O SDK do .NET Compiler Platform expõe a análise de código dos compiladores C# e Visual Basic para você como um consumidor, fornecendo uma camada de API que espelha um pipeline de compilador tradicional.



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 símbolo hierárquico, a fase de associação expõe o resultado da análise semântica do compilador e a fase de emitir é uma API que produz códigos de byte 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 delineamento de código usam as árvores de sintaxe, o Pesquisador de Objetos e os recursos de navegação usam a tabela de símbolos, as refatorações e Ir para Definição usam o modelo semântico e Editar e Continuar usa todos eles, incluindo a API de Emitir.

Camadas de API

O SDK do compilador .NET consiste em várias camadas de APIs: APIs do compilador, APIs de diagnóstico, APIs de script e APIs de workspaces.

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. As duas APIs são semelhantes em forma, mas personalizadas para alta fidelidade a cada idioma individual. Essa camada não tem dependências em componentes do Visual Studio.

APIs de diagnóstico

Como parte de sua análise, o compilador pode produzir um conjunto de diagnósticos que abrangem tudo, desde erros de sintaxe, semântica e atribuição definida a vários avisos e diagnósticos informacionais. 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. Ele permite que o diagnóstico definido pelo usuário, como aqueles produzidos por ferramentas como StyleCop, sejam produzidos junto com o diagnóstico definido pelo compilador. A produção de diagnóstico dessa maneira tem o benefício de integrar naturalmente com ferramentas como MSBuild e Visual Studio, que dependem do diagnóstico para experiências como interromper um build com base na política e mostrar alternâncias ao vivo no editor e sugerir 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. Ele ajuda você a organizar todas as informações sobre os projetos em 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 de 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

21/01/2022 • 8 minutes to read

A árvore de sintaxe é uma estrutura de dados imutável fundamental exposta pelas APIs do compilador. Essas árvores representam a estrutura lexical e sintática do código-fonte. Elas servem duas finalidades importantes:

- Para permitir ferramentas – como um IDE, suplementos, ferramentas de análise de código e refatoração – para ver e processar a estrutura sintática do código-fonte no projeto de um usuário.
- Para habilitar ferramentas – como refatorações e um IDE, para criar, modificar e reorganizar o código-fonte de maneira natural sem precisar usar edições de texto direto. 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.

Árvores de sintaxe têm três atributos de chave:

- Eles contêm todas as informações de origem com total fidelidade. Fidelidade total significa que a árvore de sintaxe contém todas as informações encontradas no texto de origem, todas as construções gramaticais, todos os tokens léxicos e todo o restante entre eles, incluindo o espaço em branco, os comentários e as diretivas de pré-processador. Por exemplo, cada literal mencionado na fonte é representado exatamente como foi digitado. As árvores de sintaxe também capturam erros no código-fonte quando o programa está incompleto ou malformado, representando tokens ignorados ou ausentes.
- Eles podem produzir o texto exato do qual foram analisados. De qualquer nó de sintaxe, é possível obter a representação de texto da subárvore com raiz nesse nó. Essa capacidade significa que as árvores de sintaxe podem ser usadas como uma maneira de construir e editar o texto de origem. Criando uma árvore que você tem, por implicação, criou o texto equivalente e fazendo uma nova árvore fora das alterações em uma árvore existente, você editou o texto com eficiência.
- Eles são imutáveis e thread-safe. Depois que uma árvore é obtida, é um instantâneo do estado atual do código e nunca é alterado. 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 descendentes, como [DescendantNodes](#) , [DescendantTokens](#) ou [DescendantTrivia](#) -que representam uma lista de todos os nós, tokens ou Trívia que existem na subárvore com raiz por esse 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 `\u03c0`.

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.

Você pode acessar o Trívia inspecionando as [SyntaxToken.LeadingTrivia](#) coleções ou um token [SyntaxToken.TrailingTrivia](#) . 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 [Span](#) propriedade é o intervalo de texto desde o início do primeiro token na subárvore do nó até o final do último token. Esse intervalo não inclui nenhum desafio à esquerda ou à direita.

A [FullSpan](#) propriedade é o intervalo de texto que inclui o span normal do nó, além do intervalo de qualquer Trívia à esquerda ou à direita.

Por exemplo:

```
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 um idioma. Cada idioma, C# ou Visual Basic, tem uma única `SyntaxKind` enumeração ([Microsoft.CodeAnalysis.CSharp.SyntaxKind](#) e [Microsoft.CodeAnalysis.VisualBasic.SyntaxKind](#), respectivamente) que lista todos os nós, tokens e elementos trivias possíveis 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.

TIP

É recomendável verificar os tipos usando os [IsKind](#) métodos de extensão (para C#) ou [IsKind](#) (para vb).

Erros

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 código que não está de acordo com a sintaxe definida do idioma, ele usa uma das duas técnicas para criar uma árvore de sintaxe:

- Se o analisador espera um tipo específico 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 em que possa continuar a análise. Nesse caso, os tokens ignorados são anexados como um nó de desafio com o tipo `SkippedTokensTrivia`.

Trabalhar com semântica

21/01/2022 • 3 minutes to read

As [árvore 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 é 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 estar 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 [IMethodSymbol.ReturnType](#) propriedade informa 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

21/01/2022 • 2 minutes to read

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

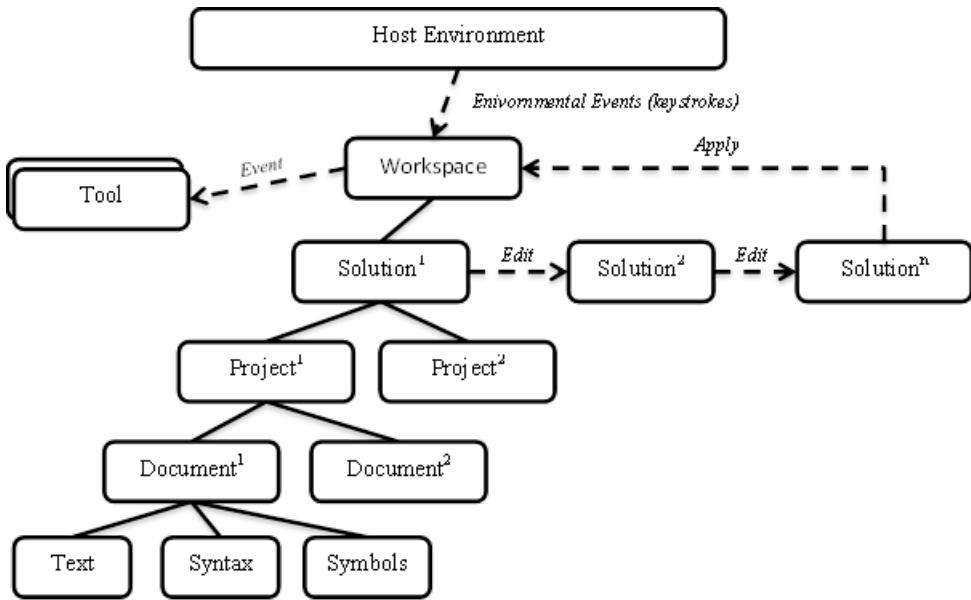
Embora um workspace possa ser alterado sempre que uma tecla é pressionada, você pode trabalhar com o modelo da solução de forma isolada.

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 muitas oportunidades de inovação em áreas como metaprogramação, geração de código e transformação, 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

21/01/2022 • 9 minutes to read

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. Selecione **Modificar**
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. Verifique a caixa para o **editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o [Instalador do Visual Studio](#)
2. Selecione **Modificar**
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**.

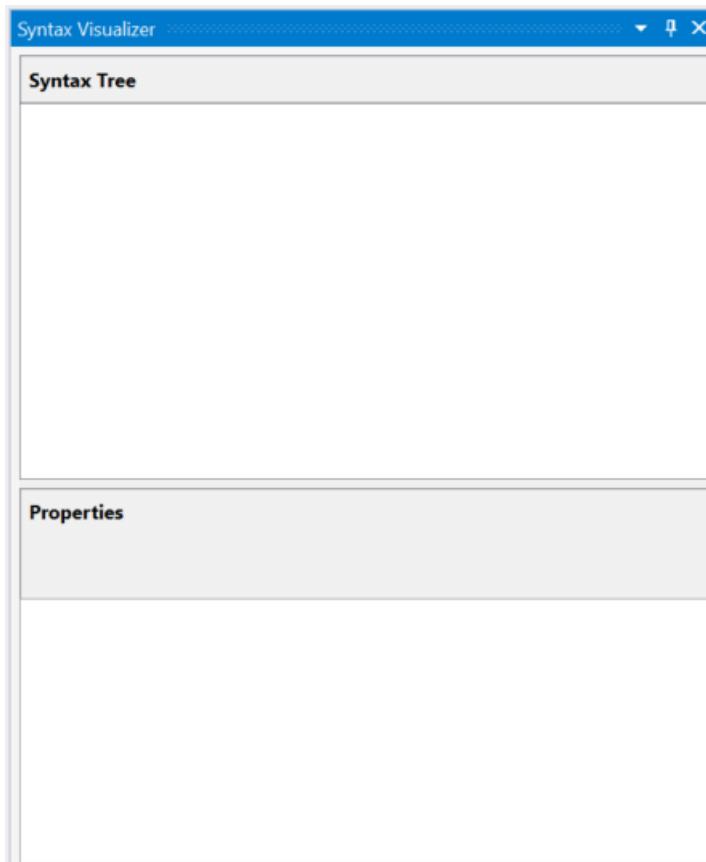
Familiarize-se com os conceitos usados no SDK do .NET Compiler Platform lendo o artigo de [visão geral](#). Ele fornece uma introdução a árvores de sintaxe, nós, tokens e trívia.

Visualizador de sintaxe

O **Syntax Visualizer** permite a inspeção da árvore de sintaxe para o arquivo de código C# ou Visual Basic na

janela atual do editor ativo dentro do Visual Studio IDE. O visualizador pode ser lançado clicando em **Exibir Outros Windows > > Syntax Visualizer**. Você também pode usar a barra de ferramentas de **Início Rápido** no canto superior direito. Digite "syntax" e o comando para abrir o **Visualizador de sintaxe** deverá aparecer.

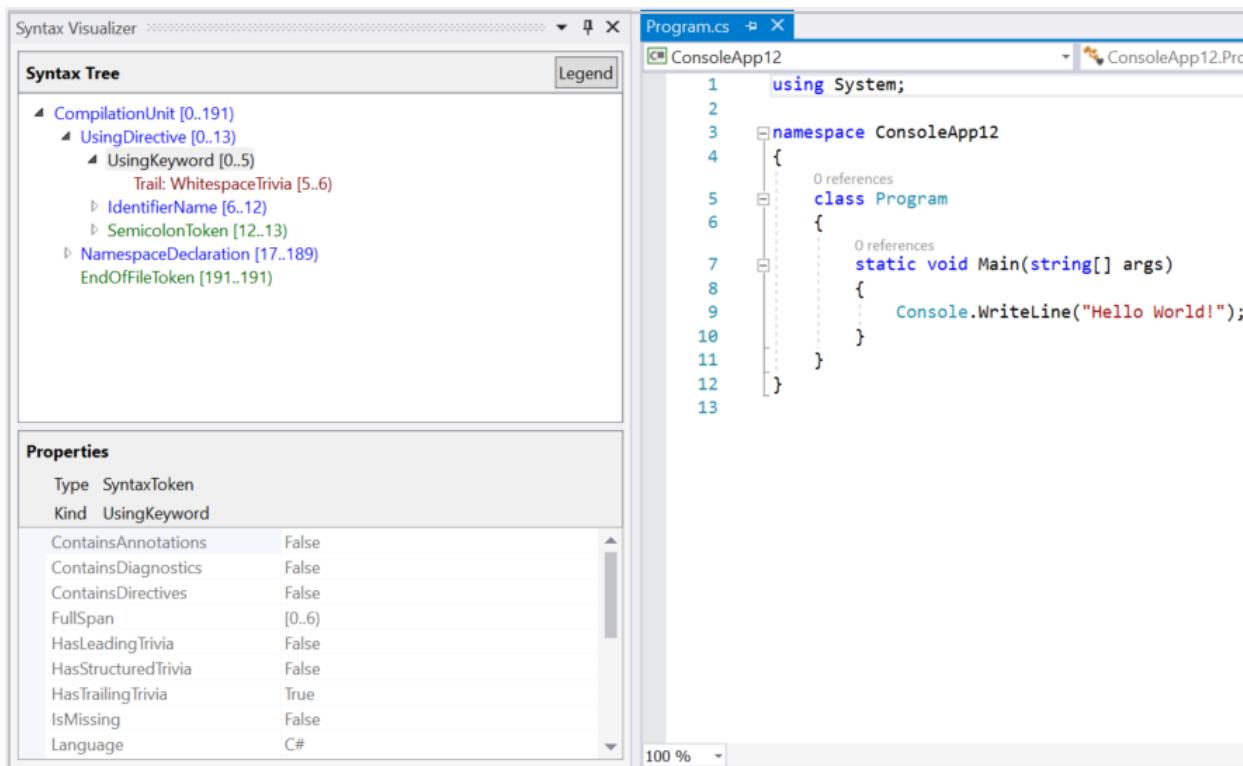
Este comando abre o Visualizador de sintaxe como uma janela de ferramentas flutuante. Se você não tiver uma janela de editor de código aberta, a exibição ficará em branco, conforme mostrado na figura a seguir.



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 Project**. 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 C# /Visual Basic existente nesta instância Visual Studio, e o visualizador exibe a árvore de sintaxe desse arquivo. 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#](#)
- [Visual Basic](#)



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 Visual Basic, se você selecionar a linha que contém "Module Module1" no editor de código, o visualizador navegará 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.

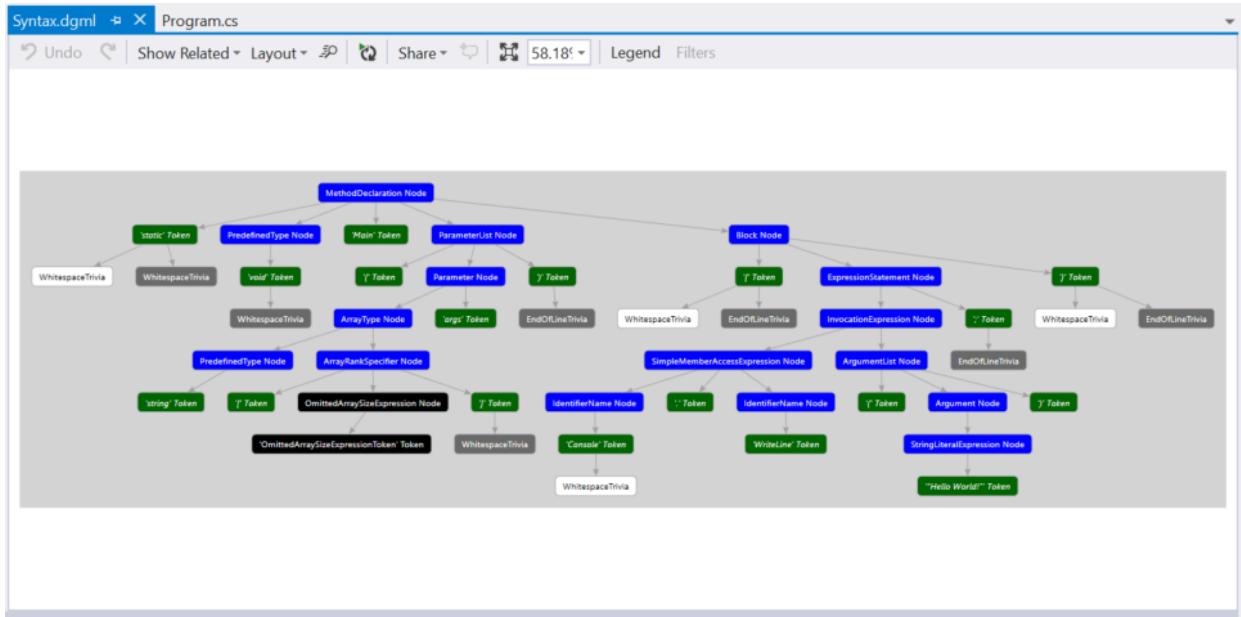
Pare de digitar quando já tiver digitado `Console.`. A árvore tem alguns itens coloridos em rosa. Neste ponto, há erros (também conhecidos como "Diagnóstico") no código digitado. Esses erros são anexados a nós, tokens e trívia na árvore de sintaxe. O visualizador mostra quais itens têm erros anexados 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.

- C#
- Visual Basic

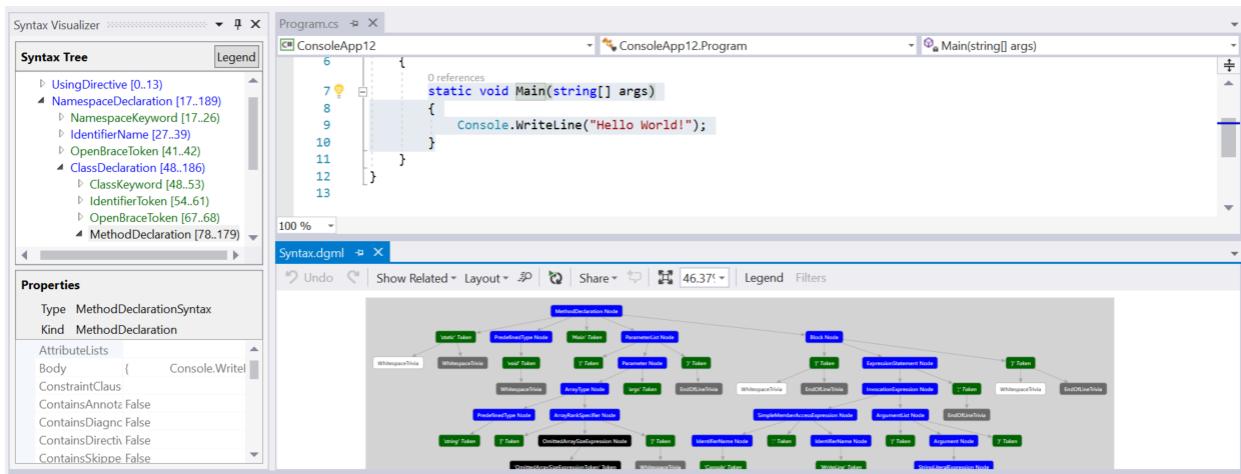
O visualizador exibe uma representação gráfica da subárvore com raiz no item selecionado. Repita essas etapas para o nó **MethodDeclaration** correspondente ao método `Main()` no exemplo de C#. O visualizador exibe um gráfico de sintaxe que tem a seguinte aparência:



O visualizador de grafo de sintaxe tem uma opção para exibir uma legenda para seu esquema de coloração. 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:

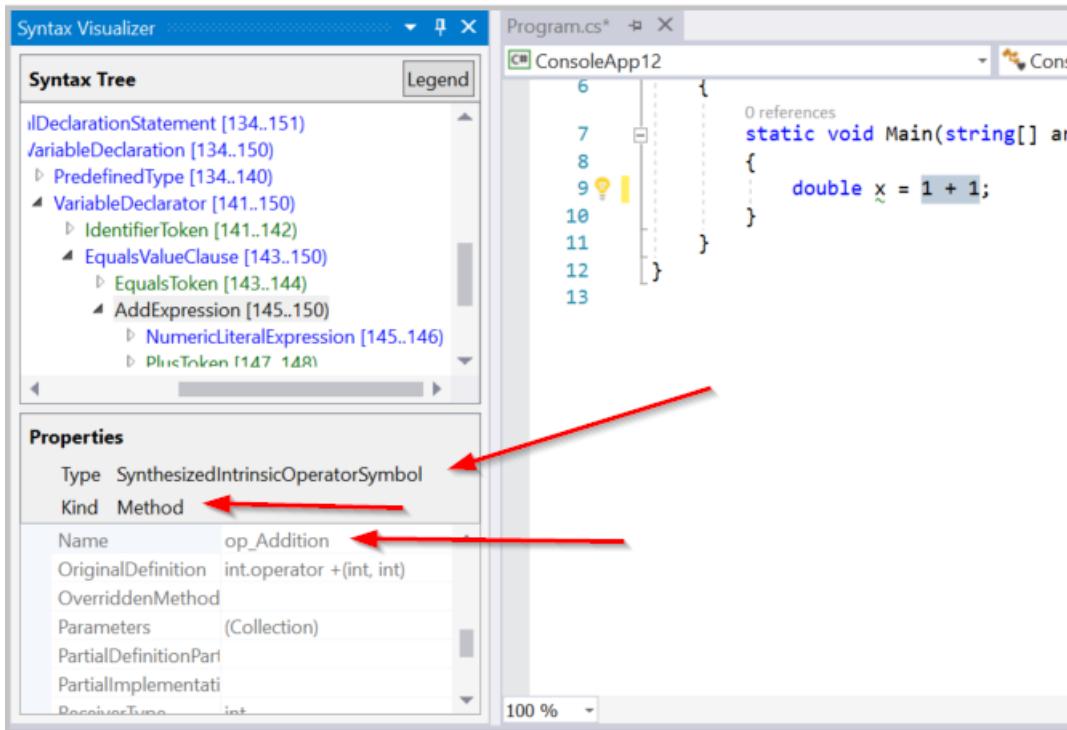


Outra opção é colocar a janela de gráfico de sintaxe em um segundo monitor, em uma configuração de dois monitores.

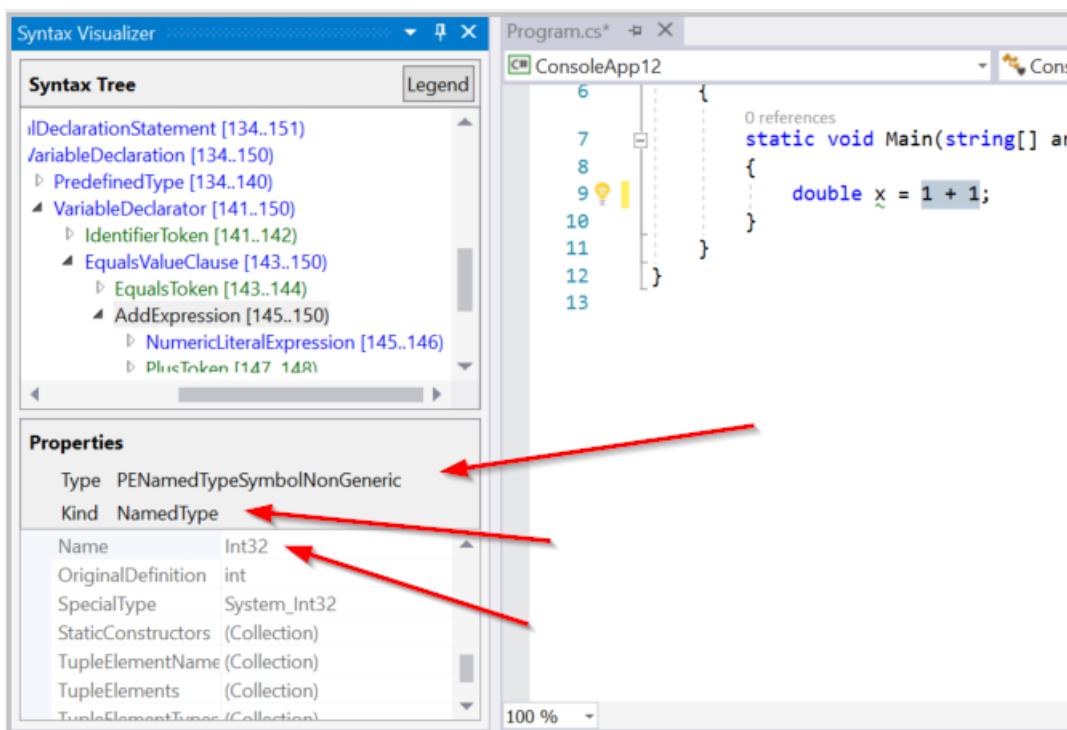
Inspecionando semântica

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 Nô, incluindo propriedades que podem não estar presentes em todos os nós.

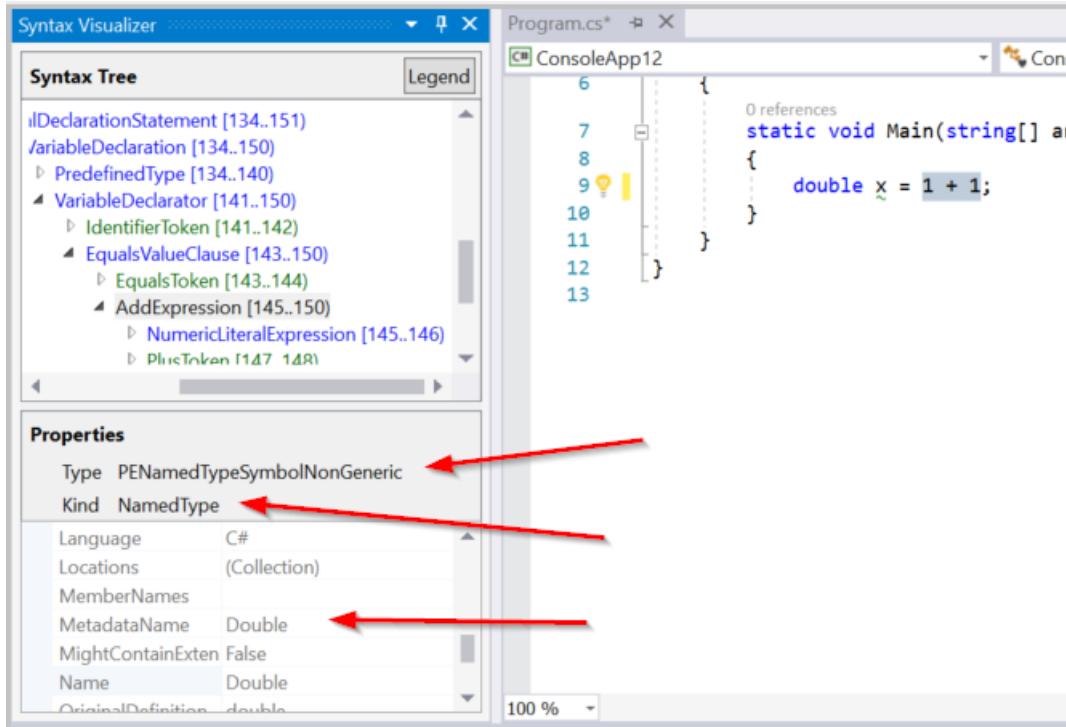
A grade de propriedade do visualizador é atualizada conforme mostrado na figura a seguir: o símbolo da expressão é um **SynthesizedIntrinsicOperatorSymbol** com **Kind = Method**.



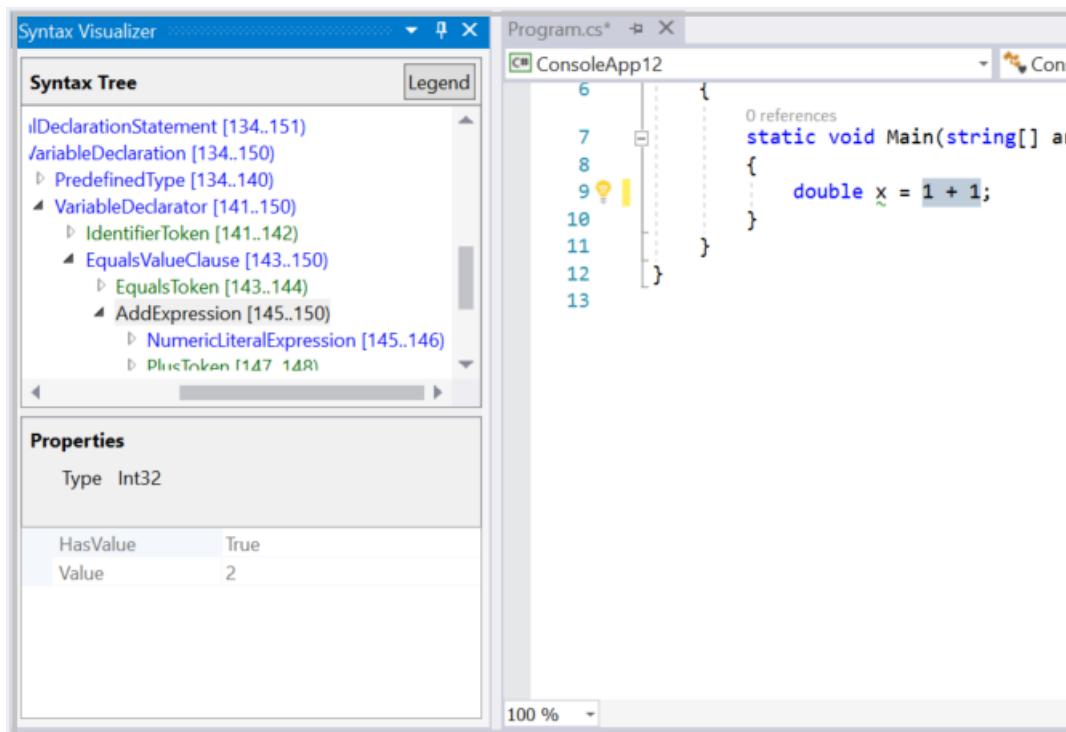
Experimente **Exibir TypeSymbol (se houver)** para o mesmo nó **AddExpression**. A grade de propriedade no visualizador é atualizada, conforme mostrado na figura a seguir, indicando que o tipo da expressão selecionada é `Int32`.



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 em Visual Basic. Digite `Dim x As Double = 1 + 1` um arquivo Visual Basic dados. 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 em Visual Basic. Atualize seu arquivo Visual Basic principal com o seguinte código:

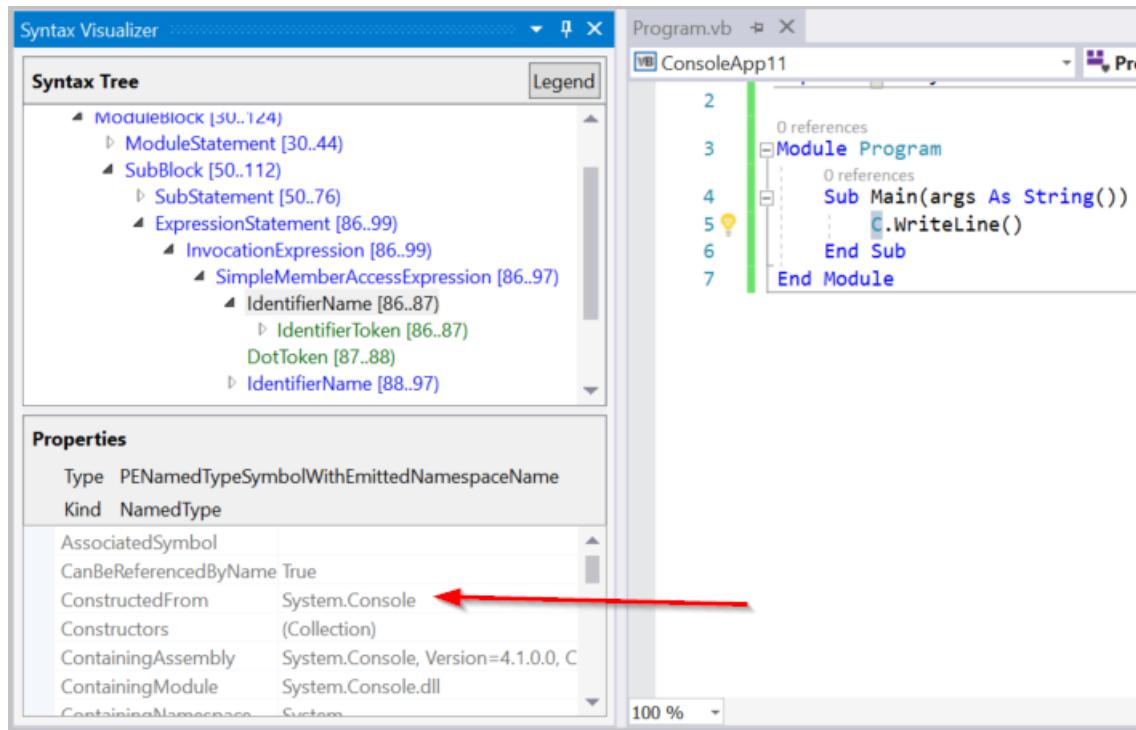
```

Imports C = System.Console

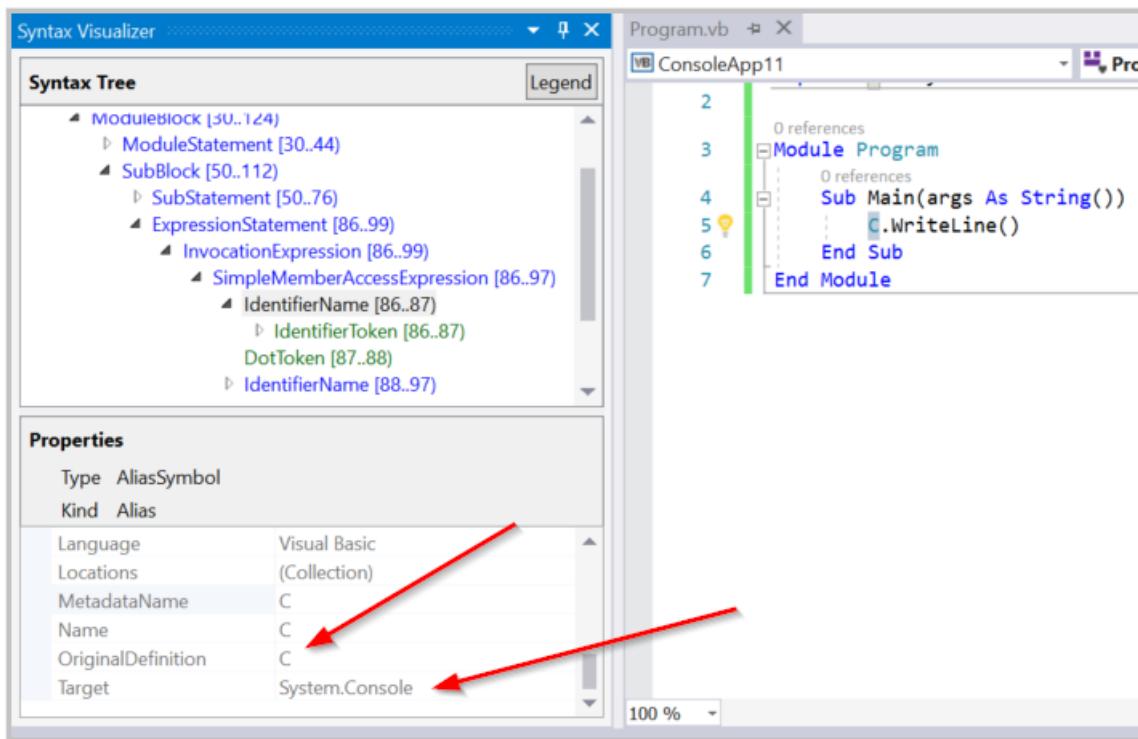
Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module

```

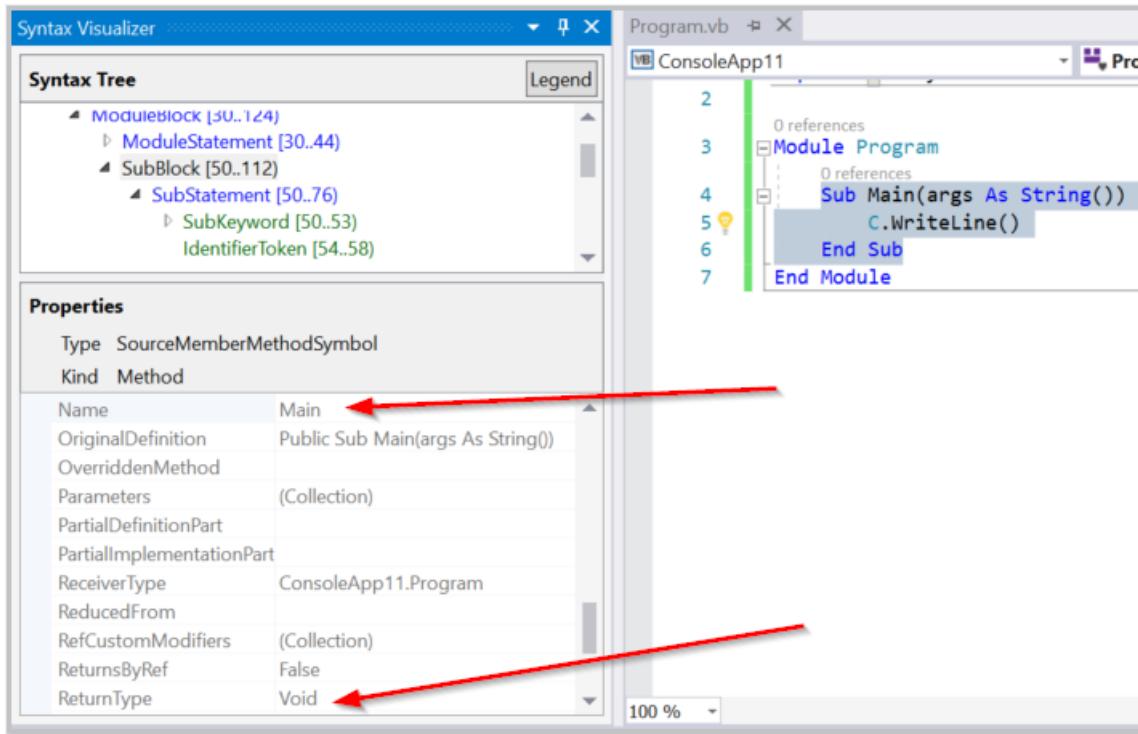
Esse código introduz um alias chamado `c` que mapeia para o tipo `System.Console` na parte superior do arquivo e usa esse alias em `Main()`. Selecione o uso desse alias, o `c` em `c.WriteLine()`, dentro do método `Main()`. O visualizador seleciona o nó **IdentifierName** correspondente no visualizador. 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 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 trívias. 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 sua exibição conforme você navega pelo código, editando e alterando a origem. Ele poderá se tornar uma distração quando você não estiver usando.

Geradores de origem

21/01/2022 • 8 minutes to read

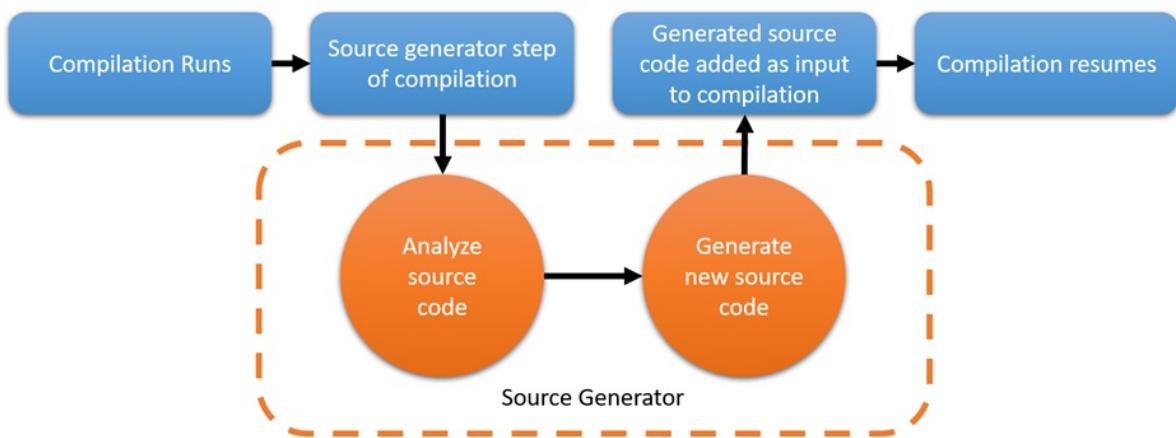
este artigo fornece uma visão geral dos geradores de origem que são fornecidos como parte do SDK do .NET Compiler Platform ("Roslyn"). Os geradores de origem são um recurso de compilador C# que permite aos desenvolvedores de C# inspecionar o código do usuário à medida que ele está sendo compilado e gerar novos arquivos de origem em C# dinamicamente que são adicionados à compilação do usuário. Dessa forma, você pode ter o código que é executado durante a compilação e inspeciona seu programa para produzir arquivos de origem adicionais que são compilados junto com o restante do seu código.

Um gerador de origem é um novo tipo de componente que os desenvolvedores de C# podem escrever, permitindo que você faça duas coisas principais:

1. Recuperar um *objeto de compilação* que representa todo o código de usuário que está sendo compilado. Esse objeto pode ser inspecionado e você pode escrever um código que funcione com a sintaxe e os modelos semânticos para o código que está sendo compilado, assim como ocorre com os analisadores 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 código-fonte adicional como entrada para uma compilação enquanto o código está sendo compilado.

Quando combinadas, essas duas coisas são o que torna os geradores de origem tão úteis. Você pode inspecionar o código do usuário com todos os metadados avançados que o compilador cria durante a compilação e emitir o código C# de volta para a 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 qualquer analisador. Ele é utilizável em ambientes em que .NET Standard componentes podem ser carregados e executados.

IMPORTANT

No momento, apenas .NET Standard assemblies 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 pelas tecnologias hoje:

- Reflexão de tempo de execução.
- MSBuild tarefas de malabarismos.
- Combinando de linguagem intermediária (IL) (não discutido neste artigo).

Os geradores de origem podem ser uma melhoria em relação a cada abordagem.

Reflexão de tempo de execução

A reflexão de tempo de execução é uma tecnologia poderosa que foi adicionada ao .NET há muito tempo. Há inúmeros cenários para usá-lo. Um cenário muito comum é executar alguma análise do código do usuário quando um aplicativo é iniciado e usar esses dados para gerar coisas.

Por exemplo, ASP.NET Core usa reflexão quando o serviço web é executado pela primeira vez para descobrir construções que você definiu para que ele possa "conectar" itens como controladores e páginas razor. Embora isso permita que você escreva código direto com abstrações poderosas, ele vem com uma penalidade de desempenho no tempo de execução: 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 tempo de execução que descobre informações sobre o seu código termine de ser executado. Embora essa penalidade de desempenho não seja enorme, é um custo fixo que você não pode melhorar por conta própria em seu próprio aplicativo.

Com um gerador de origem, a fase de descoberta de controlador da inicialização poderia acontecer em tempo de compilação analisando o código-fonte e emitindo o código necessário para "conectar" seu aplicativo. Isso pode resultar em alguns tempos de inicialização mais rápidos, pois uma ação que ocorre em tempo de execução hoje pode ser enviada para o tempo de compilação.

MSBuild tarefas de malabarismos

Os geradores de origem podem melhorar o desempenho de maneiras que não estão limitadas à reflexão em tempo de execução para descobrir tipos também. Alguns cenários envolvem a chamada de MSBuild tarefa C# (chamada CSC) várias vezes para que eles possam inspecionar 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 eliminar a necessidade de fazer malabarismos MSBuild tarefas como essa, pois os geradores de origem não oferecem apenas alguns benefícios de desempenho, mas também permitem que as ferramentas operem no nível certo de abstração.

Outra capacidade que os geradores de origem podem oferecer é dispensando o uso de algumas APIs de "tipo de cadeia de caracteres", como o funcionamento do roteamento de ASP.NET Core entre controladores e páginas razor. 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 a quantidade de vezes que um literal de cadeia de caracteres digitado incorretamente leva a uma solicitação que não está atingindo o controlador correto.

Introdução aos geradores de origem

Neste guia, você explorará a criação de um gerador de origem usando a [ISourceGenerator API](#).

1. Crie um aplicativo de console .NET. Este exemplo usa o .NET 6.

2. Substitua a `Program` classe pelo seguinte:

```
namespace ConsoleApp;

partial class Program
{
    static void Main(string[] args)
    {
        HelloFrom("Generated Code");
    }

    static partial void HelloFrom(string name);
}
```

NOTE

Você pode executar este exemplo como está, mas nada acontecerá ainda.

3. Em seguida, criaremos um projeto de gerador de origem que implementará o `partial void HelloFrom` método equivalente.
4. Crie um projeto de biblioteca do .NET standard que tenha como alvo o `netstandard2.0` moniker da estrutura de destino (TFM):

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

<PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="Microsoft.CodeAnalysis.CSharp" Version="4.0.1" PrivateAssets="all" />
    <PackageReference Include="Microsoft.CodeAnalysis.Analyzers" Version="3.3.3" PrivateAssets="all" />
</ItemGroup>

</Project>
```

TIP

O projeto gerador de origem precisa direcionar o `netstandard2.0` TFM, caso contrário, ele não funcionará.

5. Crie um novo arquivo C# chamado `HelloSourceGenerator.cs` que especifica seu próprio gerador de origem, assim:

```

using Microsoft.CodeAnalysis;

namespace SourceGenerator
{
    [Generator]
    public class HelloSourceGenerator : ISourceGenerator
    {
        public void Execute(GeneratorExecutionContext context)
        {
            // Code generation goes here
        }

        public void Initialize(GeneratorInitializationContext context)
        {
            // No initialization required for this one
        }
    }
}

```

Um gerador de origem precisa implementar a [Microsoft.CodeAnalysis.ISourceGenerator](#) interface e ter o [Microsoft.CodeAnalysis.GeneratorAttribute](#). Nem todos os geradores de origem exigem inicialização, e esse é o caso com este exemplo implementação — em que [ISourceGenerator.Initialize](#) está vazio.

6. Substitua o conteúdo do [ISourceGenerator.Execute](#) método pela seguinte implementação:

```

using Microsoft.CodeAnalysis;

namespace SourceGenerator
{
    [Generator]
    public class HelloSourceGenerator : ISourceGenerator
    {
        public void Execute(GeneratorExecutionContext context)
        {
            // Find the main method
            var mainMethod = context.Compilation.GetEntryPoint(context.CancellationToken);

            // Build up the source code
            string source = $@"// Auto-generated code
using System;

namespace {mainMethod.ContainingNamespace.ToString()}
{{"
                public static partial class {mainMethod.ContainingType.Name}
                {{
                    static partial void HelloFrom(string name) =>
                        Console.WriteLine($"Generator says: Hi from '{name}'");
                }}
            }}"
            ;
            var typeName = mainMethod.ContainingType.Name;

            // Add the source code to the compilation
            context.AddSource($"{typeName}.g.cs", source);
        }

        public void Initialize(GeneratorInitializationContext context)
        {
            // No initialization required for this one
        }
    }
}

```

A partir do `context` objeto, podemos acessar o ponto de entrada ou o método de compilações `Main`. A

`mainMethod` instância é um `IMethodSymbol`, e representa um método ou símbolo semelhante ao método (incluindo o construtor, destruidor, operador ou assessor de propriedade/evento). A partir desse objeto, podemos ter um motivo sobre o namespace contido (se houver um) e o tipo. O `source` neste exemplo é uma cadeia de caracteres interpolada que modelará o código-fonte a ser gerado, onde os inteiros interpolados são preenchidos com as informações de namespace e tipo que o contém. O `source` é adicionado ao `context` com um nome de dica.

TIP

O `hintName` parâmetro do `GeneratorExecutionContext.AddSource` método 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 origem.

7. Agora temos um gerador funcional, mas é necessário conectá-lo ao nosso aplicativo de console. Edite o projeto de aplicativo de console original e adicione o seguinte, substituindo o caminho do projeto pelo projeto de .NET Standard que você criou acima:

```
<!-- Add this as a new ItemGroup, replacing paths and names appropriately -->
<ItemGroup>
    <ProjectReference Include="..\PathTo\SourceGenerator.csproj"
        OutputItemType="Analyzer"
        ReferenceOutputAssembly="false" />
</ItemGroup>
```

Essa não é uma referência de projeto tradicional e precisa ser editada manualmente para incluir os `OutputItemType` `ReferenceOutputAssembly` atributos e para obter informações adicionais sobre `OutputItemType` `OS` `ReferenceOutputAssembly` atributos e de `ProjectReference`, consulte [Common MSBuild project items: ProjectReference](#).

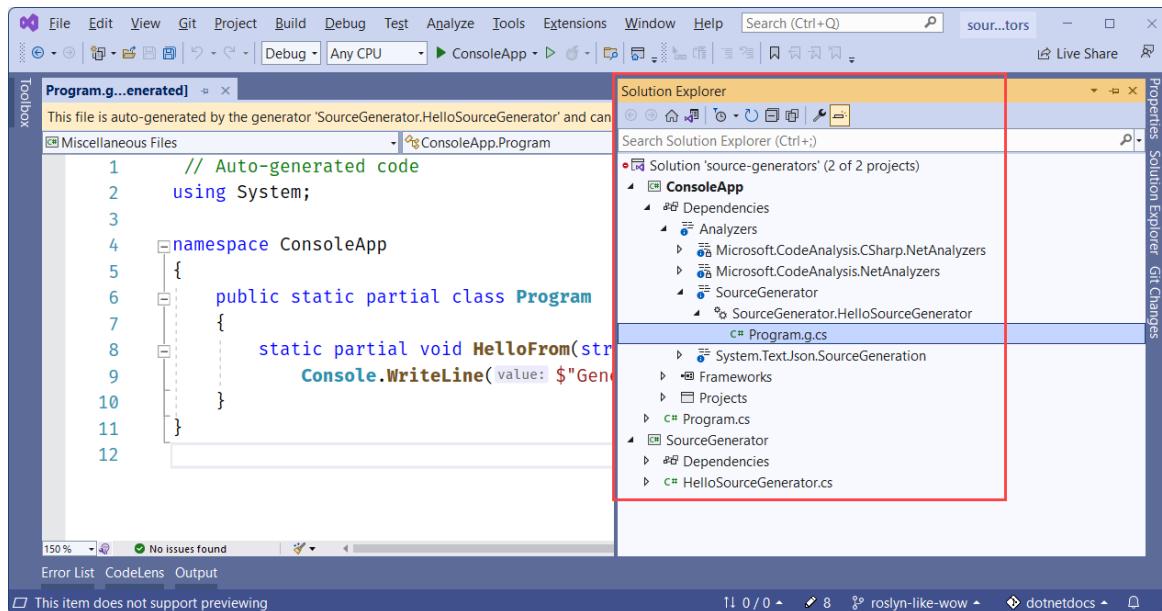
8. Agora, ao executar o aplicativo de console, você verá que o código gerado é executado e impresso na tela. O aplicativo de console em si não implementa o `HelloFrom` método, em vez disso, ele é gerado pela origem durante a compilação do projeto do gerador de origem. Veja a seguir um exemplo de saída do aplicativo:

```
Generator says: Hi from 'Generated Code'
```

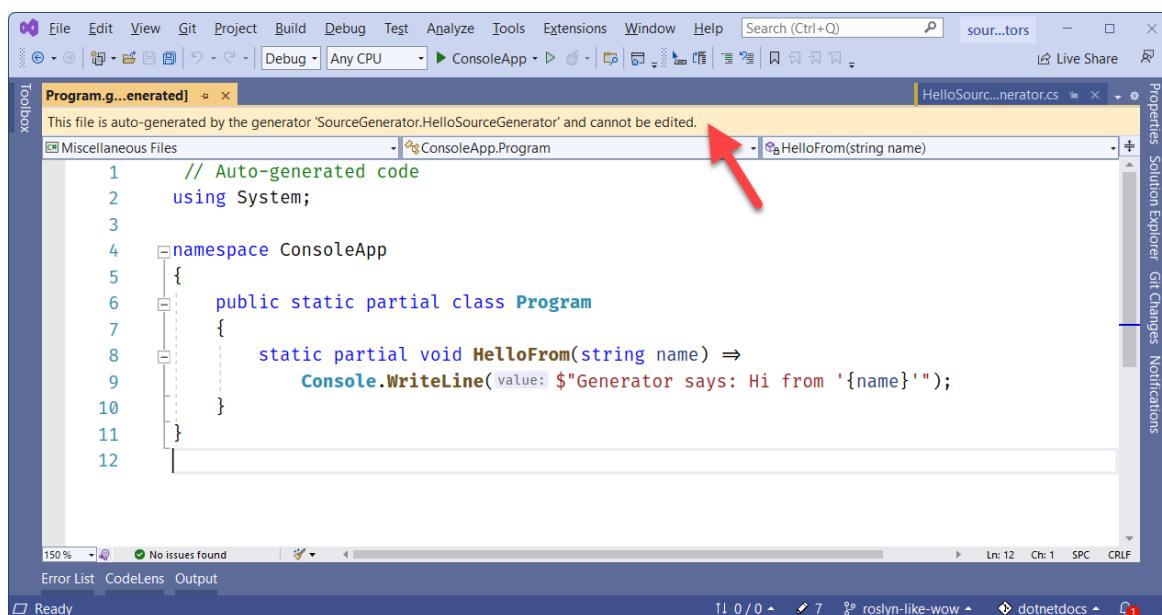
NOTE

talvez seja necessário reiniciar Visual Studio para ver o IntelliSense e se livrar de erros, pois a experiência de ferramentas está sendo aprimorada ativamente.

9. se você estiver usando Visual Studio, poderá ver os arquivos gerados de origem. Na janela **Gerenciador de soluções** expanda os analisadores de dependências > > **SourceGenerator** > **SourceGenerator**. **HelloSourceGenerator** e clique duas vezes no arquivo *Program.g.cs*.



quando você abre esse arquivo gerado, Visual Studio indica que o arquivo é gerado automaticamente e que não pode ser editado.



Próximas etapas

O guia do [gerador de fontes de origem](#) apresenta alguns desses exemplos com algumas abordagens recomendadas para solucioná-los. além disso, temos um [conjunto de amostras disponíveis em GitHub](#) que você pode experimentar por conta própria.

Você pode saber mais sobre os geradores de origem nestes tópicos:

- [Documento de design de geradores de origem](#)
- [Guia do gerador de origem](#)

Introdução à análise de sintaxe

21/01/2022 • 14 minutes to read

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:

```
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 de árvore continua até os níveis mais baixos: a cadeia de caracteres "Olá, Mundo!" é um **token literal da cadeia de caracteres** que é um 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. Selecione **Modificar**
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. Verifique a caixa para o editor **DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Selecione **Modificar**
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.

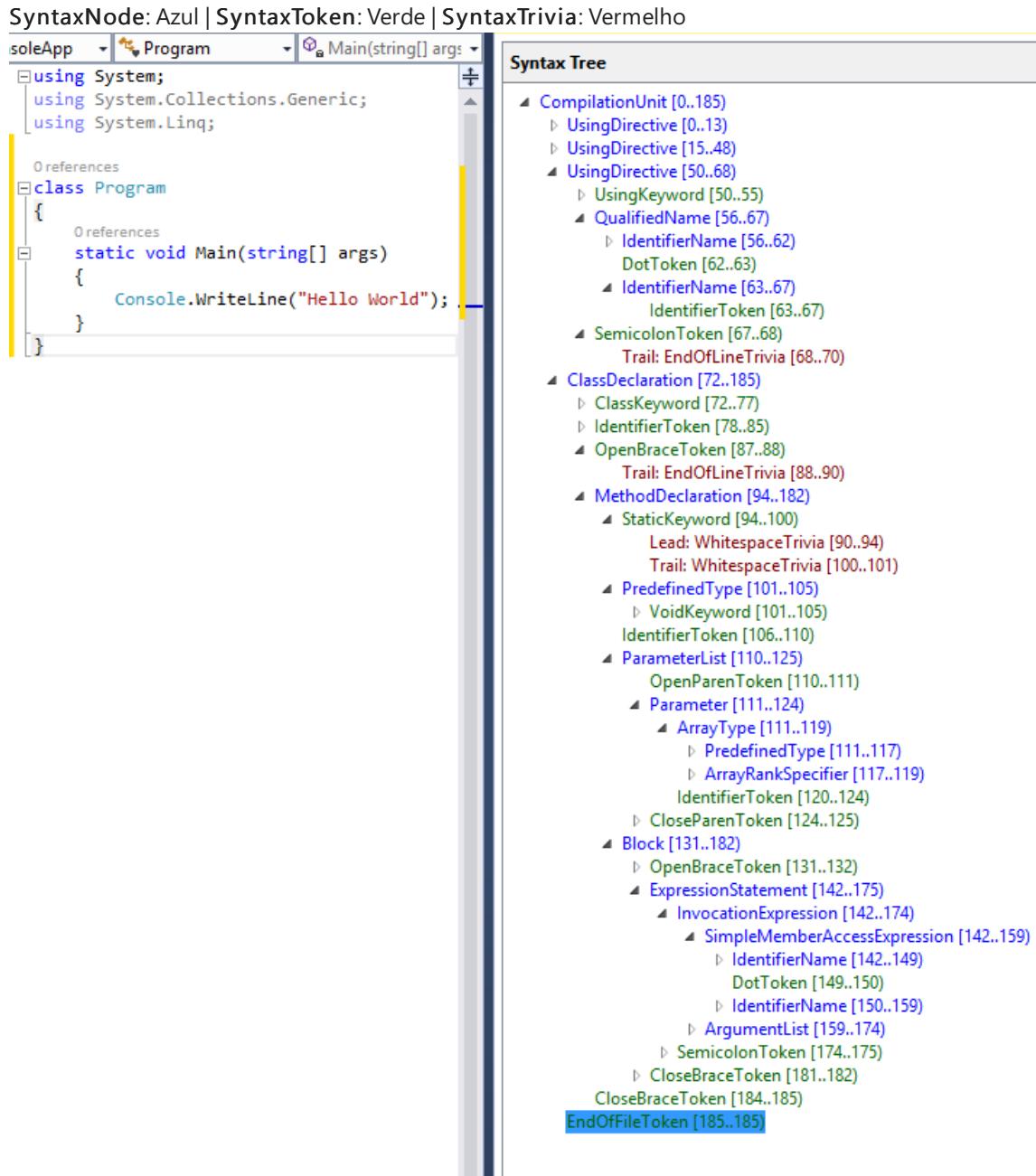
Os quatro principais blocos de construção de árvores de sintaxe são:

- 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 Parse da [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree](#) classe (ou [Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree](#)) para analisar o texto em C# (ou 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** em Visual Studio, escolha **exibir > outros Windows > Syntax Visualizer**. Por exemplo, o arquivo de origem C# anterior examinado usando o **Visualizador de Sintaxe** se parecerá com a figura a seguir:



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 amostra no GitHub](#).

NOTE

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#:

- em Visual Studio, escolha **arquivo > novo > Project** para exibir a nova caixa de diálogo de Project.
- em extensibilidade do Visual C# > , escolha **ferramenta de Code Analysis autônoma**.
- Nomeie o projeto "**SyntaxTreeManualTraversal**" e clique em OK.

Você analisará o programa "Olá, Mundo!" básico mostrado anteriormente. Adicione o texto ao programa Olá, Mundo como uma constante em sua classe `Program`:

```
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`:

```
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:

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"    \t{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 :
```

```
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`:

```
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](#).

```
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:

```
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.ToString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Execute o programa para ver todas as informações que você descobriu sobre este programa:

```
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`:

```
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 amostra no 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:

```
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:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Em seguida, crie uma nova classe. em Visual Studio, escolha **Project > 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](#).

```
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:

```
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)`:

```
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 coletor:

```
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:

```
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

21/01/2022 • 8 minutes to read

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. Selecione **Modificar**
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. Verifique a caixa para o **editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Selecione **Modificar**
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 arquivo de código ou snippet de código Visual Basic ou C# possa ser analisado sintaticamente isoladamente, não é significativo fazer perguntas como "qual é o tipo dessa variável" em um vácuo. 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 amostra no GitHub](#).

NOTE

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 Project** para exibir a caixa de diálogo Project Novo.
- Em **Extensibilidade do Visual C#**, escolha Ferramenta de Code Analysis > **Autônomo**.
- Nomeie o projeto "**SemanticQuickStart**" e clique em OK.

Você analisará o programa "Olá, Mundo!" básico mostrado anteriormente. Adicione o texto ao programa Olá, Mundo como uma constante em sua classe `Program`:

```
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`:

```
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:

```
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 neste 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:

```
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`:

```
// 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`:

```
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}
```

Execute o programa e você deverá ver a seguinte 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 . . .
```

NOTE

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 um `Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax`, a cadeia de caracteres "Olá, Mundo!" exibida pelo console.

Você localiza 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`:

```
// 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:

```
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`:

```
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`:

```
var methods = allMembers.OfType<IMethodSymbol>();
```

Em seguida, adicione outro filtro para retornar somente os métodos que são públicos e retornam um `string`:

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Selecione apenas a propriedade de nome e somente os nomes distintos, removendo quaisquer sobrecargas:

```
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:

```
foreach (string name in (from method in stringTypeSymbol
    .GetMembers().OfType<IMethodSymbol>()
    where method.ReturnType.Equals(stringTypeSymbol) &&
    method.DeclaredAccessibility == Accessibility.Public
    select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Compile e execute o programa. Você deve ver o seguinte resultado:

```
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

21/01/2022 • 12 minutes to read

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. Selecione **Modificar**
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. Verifique a caixa para o **editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Selecione **Modificar**
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 trivia, 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 os nós existentes pela nova árvore que você criou.
```

Inicie o Visual Studio e crie um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#. No Visual Studio, escolha **Arquivo > > Novo Project** para exibir a caixa de diálogo **Novo**. Em **Extensibilidade do Visual C#**, escolha uma Ferramenta > de **Code Analysis Autônomo**. 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 using à parte superior do **Program.cs**.

```
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**:

```
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`:

```
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`:

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Execute o programa novamente para ver que você criou a árvore para adicionar o código.

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`:

```
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!");
        }
    }
}";
```

NOTE

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:

```
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`:

```
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:

```
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 Code Analysis **autônomo** em C#. No Visual Studio, clique com o botão direito do mouse no nó da solução `SyntaxTransformationQuickStart`. Escolha **Adicionar > Novo Project** para exibir a caixa de diálogo **Novo Project**. Em **Extensibilidade do Visual C#**, escolha Ferramenta de Code Analysis > **Autônomo**. 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 **Project > Adicionar Classe...**. Na caixa de diálogo **Adicionar Novo Item**, digite como o nome do `TypeInferenceRewriter.cs` arquivo.

Adicione o seguinte usando diretivas ao arquivo `TypeInferenceRewriter.cs`:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Em seguida, faça a classe `TypeInferenceRewriter` se estender à classe `CSharpSyntaxRewriter`:

```
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:

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

Substitua o método `VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax)`:

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{
}
```

NOTE

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 um `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:

```
Type variable = expression;
```

Se você quiser explorar por conta própria, considere estender a amostra finalizada para esses tipos de declarações de variáveis:

```
// 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:

```

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:

```

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:

```
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:

```

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;
}

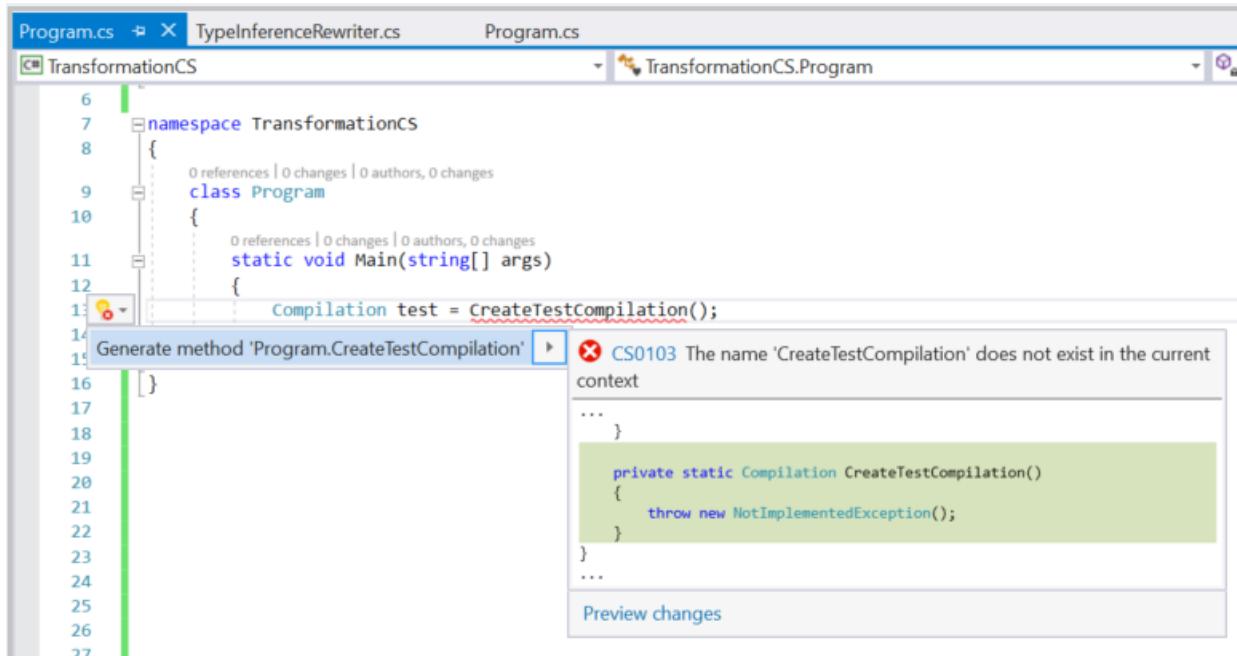
```

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:

```
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, inicialize um novo `TypeInferenceRewriter` com o `SemanticModel` para essa árvore:

```
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:

```
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! Ainda há uma etapa: criar um [Compilation](#) teste. 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:

```
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 refactoring!

Tutorial: escrever seu primeiro analisador e correção de código

21/01/2022 • 25 minutes to read

O SDK do .NET Compiler Platform fornece as ferramentas necessárias para criar diagnósticos personalizados (analisadores), correções de código, refactoring de código e supressores de diagnóstico destinados a C# ou Visual Basic código. Um **analisador** contém código que reconhece violações de sua regra. 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

Você precisará 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. Selecione **Modificar**
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. Verifique a caixa para o editor **DGML**.

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**

2. Selecione **Modificar**
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 a caixa de diálogo Novo Projeto.
- Em **Visual C# > Extensibilidade**, escolha **Analisador com correção de código (.NET Standard)**.
- Nomeie seu projeto como "**MakeConst**" e clique em OK.

NOTE

Você pode receber um erro de compilação (*MSB4062: A tarefa "CompareBuildTaskVersion" não pôde ser carregada*). Para corrigir isso, atualize os NuGet na solução com NuGet Gerenciador de Pacotes ou use na **Update-Package** janela Gerenciador de Pacotes Console.

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 um 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 inicia uma segunda instância do Visual Studio que carregou o novo analisador. Pressione F5 para iniciar o projeto VSIX.

NOTE

Os analisadores devem ser .NET Standard 2.0 porque podem ser executados no ambiente do .NET Core (builds de linha de comando) e .NET Framework ambiente (Visual Studio).

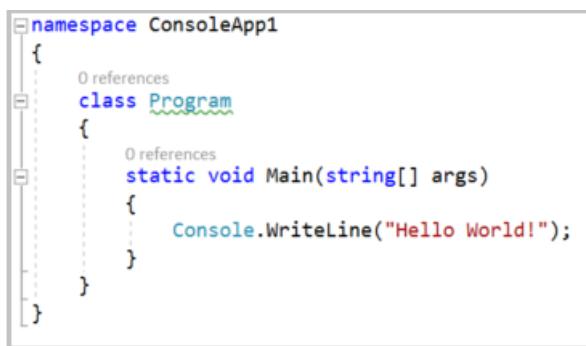
TIP

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 do Roslyn, você precisa excluí-lo manualmente de `%LocalAppData% \ Microsoft \ VisualStudio`. O nome da pasta do Hive do Roslyn terminará `Roslyn` em, por exemplo, `16.0_9ae182f9Roslyn`. Observe que talvez seja necessário limpar a solução e recomendar-a depois de excluir o hive.

Na segunda instância Visual Studio que você acabou de iniciar, crie um novo projeto de Aplicativo de Console 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:



```
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

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.

TIP

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 escreve um analisador que relata ao usuário quaisquer declarações de variável local que possam ser convertidas em constantes locais. Por exemplo, considere o seguinte código:

```
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`:

```
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 `DiagnosticAnalyzer` classe.

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 figura a seguir:

Name	Value	Comment
<code>AnalyzerDescription</code>	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
<code>AnalyzerMessageFormat</code>	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
<code>AnalyzerTitle</code>	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.Analyzer.Initialize`, localize a linha que registra a ação em símbolos:

```
context.RegisterSymbolAction>AnalyzeSymbol, SymbolKind.NamedType);
```

Substitua-a com a seguinte linha:

```
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:

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Altere `Category` o para " " em `Usage MakeConstAnalyzer.cs`, conforme mostrado no código a seguir:

```
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:

```
int x = 0;
Console.WriteLine(x);
```

A primeira etapa é encontrar declarações locais. Adicione o seguinte código a `AnalyzeNode` em `MakeConstAnalyzer.cs`:

```
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:

```
// 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 um único 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](#):

```
// 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](#):

```
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:

```
int x = 0;  
Console.WriteLine(x);
```

A lâmpada deve aparecer e o analisador deve relatar um diagnóstico. No entanto, dependendo da sua versão Visual Studio, você verá:

- A lâmpada, que ainda usa ou usa a correção de código gerada pelo modelo, dirá que ela pode ficar maiúscula.
- Uma mensagem de faixa na parte superior do editor dizendo que 'MakeConstCodeFixProvider' encontrou um erro e foi desabilitado.". Isso porque o provedor de correção de código ainda não foi alterado e ainda espera encontrar [TypeDeclarationSyntax](#) elementos em vez de [LocalDeclarationStatementSyntax](#) elementos.

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](#):

```
- 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:

```
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:

```
// 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:

```
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`:

```
// 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:

```
// 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:

```
// 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:

```
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`:

```
// 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 = 2;
    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 teste de unidade.

TIP

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 `DiagnosticDescriptor` um fornecido pelo `DiagnosticAnalyzer.SupportedDiagnostics`.
- `{|ExpectedDiagnosticId:text|}` : indica que um diagnóstico com Id `ExpectedDiagnosticId` é relatado para `text`.

Substitua os testes de modelo na `MakeConstUnitTest` classe pelo seguinte método de teste:

```
[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 seja aprovado. No Visual Studio, abra o Gerenciador de Testes selecionando Teste > Windows > Gerenciador de Testes. Em seguida, 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:

```
[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 tratou:

- Declarações que já são `const`, porque elas já são `const`:

```
[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:

```
[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:

```
[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:

```
[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:

```

// 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:

```

// 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. Portanto, a implementação atual converterá uma declaração incorreta como `int i = "abc"` em uma constante local. Adicione um método de teste para este caso:

```
[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 constante permitido para um tipo de referência é `null`, exceto no caso do `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:

```
[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:

```
[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:

```

[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 *MakeConstAnalyzer.cs* e localize o loop foreach em que cada um dos inicializadores da declaração local é verificado para garantir que eles sejam atribuídos com valores constantes. Imediatamente *antes* do primeiro loop foreach, chame `context.SemanticModel.GetTypeInfo()` para recuperar informações detalhadas sobre o tipo declarado da declaração local:

```

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:

```
// 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.

```
// 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 deve escrever um pouco mais de código no provedor de correção de código para substituir a `var` palavra-chave pelo nome de tipo correto. Retorne ao `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`:

```

// 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 `using` diretiva para usar o `Simplifier` tipo:

```
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 visualização 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 `var` palavra-chave agora é manipulada corretamente.

Por fim, adicione o seguinte código:

```
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#

21/01/2022 • 2 minutes to read

Esta seção fornece informações detalhadas sobre os principais recursos e recursos da linguagem C# 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 for um iniciante completo com programação ou com C#, talvez queira visitar a Introdução aos [Tutorial](#)s do C# ou ao Tutorial do [.NET In-Browser](#), em que nenhum conhecimento de programação anterior é necessário.

Para obter informações sobre palavras-chave, operadores e diretivas de pré-processador específicas, consulte [Referência de C#](#). Para obter mais informações sobre as especificações da linguagem C#, consulte [Especificações da linguagem C#](#).

Seções de programa

[Dentro de um programa em C#](#)

[Main\(\) e Command-Line argumentos](#)

Seções da linguagem

[Instruções, expressões e operadores](#)

[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 de 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#)

21/01/2022 • 2 minutes to read

Esta seção explica conceitos de programação na linguagem C#.

Nesta seção

TÍTULO	DESCRIÇÃO
Assemblies no .NET	Descreve como criar e usar um assemblies.
Programação assíncrona com Async e Await (C#)	Descreve como criar soluções assíncronas usando as palavras-chave <code>async</code> e <code>await</code> no C#. Inclui um passo a passo.
Atributos (C#)	Discute como fornecer informações adicionais sobre como programar elementos como tipos, campos, métodos e propriedades por meio de atributos.
Coleções (C#)	Descreve alguns dos tipos de coleções fornecidos pelo .NET. Demonstra como usar coleções simples e coleções de pares chave/valor.
Covariância e contravariância (C#)	Mostra como habilitar a conversão implícita de parâmetros de tipo genérico em interfaces e delegados.
Árvores de expressão (C#)	Explica como você pode usar árvores de expressão para habilitar a modificação dinâmica de código executável.
Iteradores (C#)	Descreve os iteradores, que são usados para percorrer coleções e retornar elementos um por vez.
LINQ (Consulta Integrada à Linguagem) (C#)	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.
Reflexão (C#)	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.
Serialização (C#)	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 com `async` e `await`

21/01/2022 • 16 minutes to read

O TAP (modelo de programação [assíncrona](#)) da tarefa fornece uma abstração sobre código assíncrono. Você escreve o código como uma sequência de instruções, como usual. Você pode ler o código como se cada instrução fosse concluída antes do início da próxima. O compilador executa muitas transformações porque algumas dessas instruções podem iniciar o trabalho e retornar um que [Task](#) representa o trabalho em andamento.

Essa é a meta dessa sintaxe: habilitar um código que leia como uma sequência de instruções, mas que execute em uma ordem muito mais complicada com base na alocação de recurso externo e em quando as tarefas são concluídas. Isso é semelhante à maneira como as pessoas dão instruções para processos que incluem tarefas assíncronas. Ao longo deste artigo, você usará um exemplo de instruções para fazer um café da manhã para ver como as palavras-chave facilitam o raciocínio sobre o código, que inclui uma série de `async` `await` instruções assíncronas. Você deve escrever as instruções de maneira parecida com a lista a seguir para explicar como fazer um café da manhã:

1. Encher uma xícara de café.
2. Aquecer uma frigideira e, em seguida, fritar dois ovos.
3. Frita três fatias de bacon.
4. Torrar dois pedaços de pão.
5. Adicionar manteiga e a geleia na torrada.
6. Encher um copo com suco de laranja.

Se tivesse experiência em culinária, você executaria essas instruções **assincronamente**. Você iniciaria aquecendo a frigideira para os ovos e, em seguida, começaria a preparar o bacon. Você colocaria o pão na torradeira e começaria a preparar os ovos. Em cada etapa do processo, iniciaria uma tarefa e voltaria sua atenção para as tarefas que estivessem prontas para a sua atenção.

Preparar o café da manhã é um bom exemplo de trabalho assíncrono que não é paralelo. Uma pessoa (ou um thread) pode lidar com todas essas tarefas. Continuando com a analogia do café da manhã, uma pessoa pode fazer café da manhã assincronamente iniciando a tarefa seguinte antes de concluir a primeira. O preparo progride independentemente de haver alguém observando. Assim que inicia o aquecimento da frigideira para os ovos, você pode começar a fritar o bacon. Quando começar a preparar o bacon, você pode colocar o pão na torradeira.

Para um algoritmo paralelo, você precisaria de vários cozinheiros (ou threads). Um prepararia os ovos, outro o bacon e assim por diante. Cada um se concentraria apenas naquela tarefa específica. Cada cozinheiro (ou thread) ficaria bloqueado de forma síncrona, esperando que o bacon estivesse pronto para ser virado ou que a torrada pulasse.

Agora, considere essas mesmas instruções escritas como instruções em C#:

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    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();
}

```

```
    return new Egg(),
}

private static Coffee PourCoffee()
{
    Console.WriteLine("Pouring coffee");
    return new Coffee();
}

}
```



O café da manhã preparado de forma síncrona levou aproximadamente 30 minutos porque o total é a soma de cada tarefa individual.

NOTE

As `Coffee`, `classes`, `,,`, `e`, `estão`, `Egg`, `Bacon`, `Toast`, `Juice` vazias. Elas são simplesmente classes de marcador para fins de demonstração, não contêm nenhuma propriedade e não servem para nenhuma outra finalidade.

Os computadores não interpretam essas instruções da mesma forma que as pessoas. O computador ficará bloqueado em cada instrução até que o trabalho seja concluído, antes de passar para a próxima instrução. Isso cria um café da manhã insatisfatório. As tarefas posteriores não seriam iniciadas até que as tarefas anteriores fossem concluídas. Levaria muito mais tempo para criar o café da manhã e alguns itens ficariam frios antes de serem servidos.

Se você quiser que o computador execute as instruções acima de forma assíncrona, deverá escrever o código assíncrono.

Essas questões são importantes para os programas que você escreve atualmente. Ao escrever programas de cliente, você quer que a interface do usuário responda de acordo com as solicitações do usuário. Seu aplicativo não deve fazer um telefone parecer travado enquanto ele está baixando dados da Web. Ao escrever programas de servidor, você não quer threads bloqueados. Esses threads poderiam servir a outras solicitações. O uso de código síncrono quando existem alternativas assíncronas afeta sua capacidade de aumentar de forma menos custosa. Você paga pelos threads bloqueados.

Aplicativos modernos bem-sucedidos exigem código assíncrono. Sem suporte de linguagem, escrever código assíncrono exigia retornos de chamada, eventos de conclusão ou outros meios que obscureciam a intenção original do código. A vantagem do código síncrono é que suas ações passo a passo facilitam a verificação e a compreensão. Modelos assíncronos tradicionais forçavam você a se concentrar na natureza assíncrona do código e não nas ações fundamentais do código.

Não bloquear, mas aguardar

O código anterior demonstra uma prática inadequada: construção de código síncrono para realizar operações assíncronas. Como escrito, esse código bloqueia o thread que o está executando, impedindo-o de realizar qualquer outra tarefa. Ele não será interrompido enquanto qualquer uma das tarefas estiver em andamento. Seria como se você fixasse o olhar na torradeira depois de colocar o pão. Você ignoraria qualquer pessoa que estivesse conversando com você até que a torrada pulasse.

Vamos começar atualizando esse código para que o thread não seja bloqueado enquanto houver tarefas em execução. A palavra-chave `await` oferece uma maneira sem bloqueio de iniciar uma tarefa e, em seguida, continuar a execução quando essa tarefa for concluída. Uma versão assíncrona simples do código de fazer café da manhã ficaria como o snippet a seguir:

```
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

O tempo total decorrido é aproximadamente o mesmo que a versão síncrona inicial. O código ainda precisa aproveitar alguns dos principais recursos da programação assíncrona.

TIP

Os corpos de método `FryEggsAsync`, `FryBaconAsync` e `ToastBreadAsync` foram todos atualizados para retornar, e `Task<Egg>`, `Task<Bacon>` e `Task<Toast>` respectivamente. Os métodos são renomeados de sua versão original para incluir o sufixo "Async". Suas implementações são mostradas como parte da [versão final](#) mais adiante neste artigo.

Esse código não bloqueia enquanto os ovos ou o bacon são preparados. Entretanto, esse código não iniciará outras tarefas. Você ainda colocaria o pão na torradeira e ficaria olhando até ele pular. Mas, pelo menos, você responderia a qualquer pessoa que quisesse sua atenção. Em um restaurante em que vários pedidos são feitos, o cozinheiro pode iniciar o preparo de outro café da manhã enquanto prepara o primeiro.

Agora, o thread trabalhando no café da manhã não fica bloqueado aguardando qualquer tarefa iniciada que ainda não tenha terminado. Para alguns aplicativos, essa alteração já basta. Um aplicativo de GUI ainda responde ao usuário com apenas essa alteração. No entanto, neste cenário, você quer mais. Você não deseja que cada uma das tarefas componentes seja executada em sequência. É melhor iniciar cada uma das tarefas componentes antes de aguardar a conclusão da tarefa anterior.

Iniciar tarefas simultaneamente

Em muitos cenários, convém iniciar várias tarefas independentes imediatamente. Em seguida, conforme cada tarefa é concluída, você pode continuar outro trabalho que esteja pronto. Na analogia do café da manhã, é assim que você prepara o café da manhã muito mais rapidamente. Você também prepara tudo quase ao mesmo tempo. Você terá um café da manhã quente.

O [System.Threading.Tasks.Task](#) e os tipos relacionados são classes que você pode usar para pensar nas tarefas que estão em andamento. Elas permitem que você escreva código que se assemelhe mais à maneira como você realmente prepara o café da manhã. Você começaria a preparar os ovos, o bacon e a torrada ao mesmo tempo. Como cada um exige ação, você voltaria sua atenção para essa tarefa, cuidaria da próxima ação e aguardaria algo mais que exigisse sua atenção.

Você inicia uma tarefa e espera o objeto [Task](#) que representa o trabalho. Você vai `await` cada tarefa antes de trabalhar com o respectivo resultado.

Vamos fazer essas alterações no código do café da manhã. A primeira etapa é armazenar as tarefas para as operações quando elas forem iniciadas, em vez de aguardá-las:

```
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!");
```

Em seguida, você pode mover as instruções `await` do bacon e dos ovos até o final do método, antes de servir o café da manhã:

```

Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

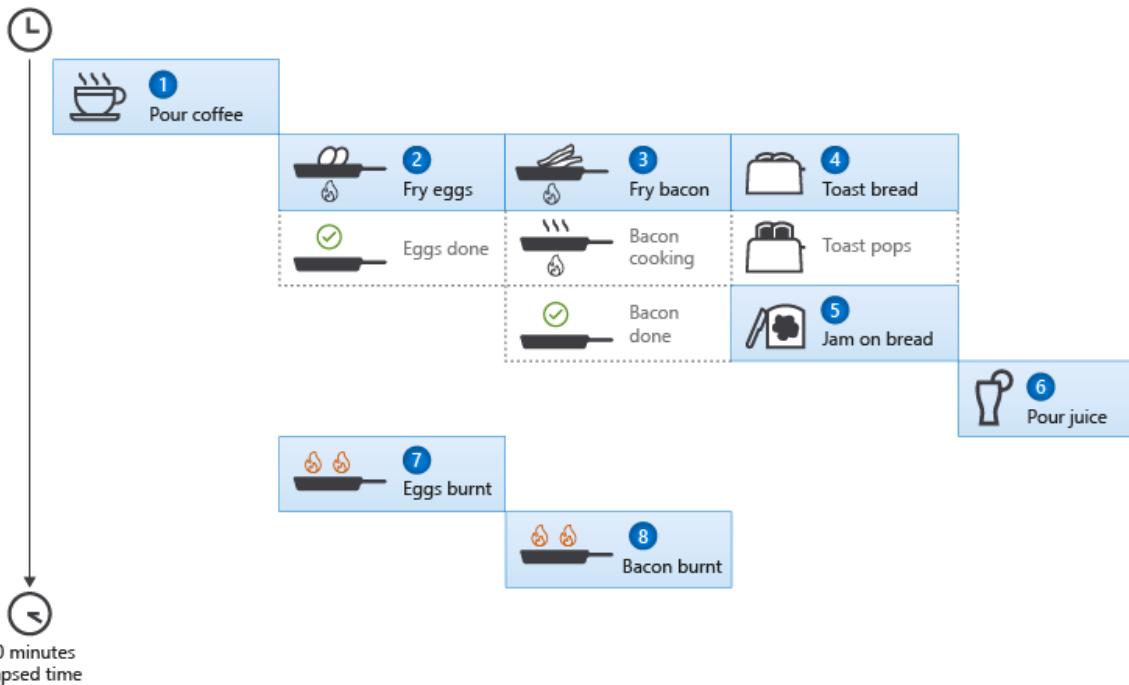
Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

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!");

```



O café da manhã preparado de forma assíncrona levou aproximadamente 20 minutos, desta vez a economia é porque algumas tarefas foram executadas simultaneamente.

O código anterior funciona melhor. Você inicia todas as tarefas assíncronas ao mesmo tempo. Você aguarda cada tarefa somente quando precisar dos resultados. O código anterior pode ser semelhante a um código em um aplicativo Web que faz solicitações de diferentes microsserviços e combina os resultados em uma única página. Você fará todas as solicitações imediatamente e, em seguida, `await` em todas essas tarefas e comporá a página da Web.

Composição com tarefas

Você prepara tudo para o café da manhã ao mesmo tempo, exceto a torrada. Preparar a torrada é a composição de uma operação assíncrona (torrar o pão) com operações síncronas (adicionar a manteiga e a geleia). A atualização deste código ilustra um conceito importante:

IMPORTANT

A composição de uma operação assíncrona seguida por trabalho síncrono é uma operação assíncrona. Explicando de outra forma, se qualquer parte de uma operação for assíncrona, toda a operação será assíncrona.

O código anterior mostrou que você pode usar objetos `Task` ou `Task<TResult>` para manter tarefas em execução. Você `await` em cada tarefa antes de usar seu resultado. A próxima etapa é criar métodos que declarem a combinação de outro trabalho. Antes de servir o café da manhã, você quer aguardar a tarefa que representa torrar o pão antes de adicionar manteiga e geleia. Você pode declarar esse trabalho com o código a seguir:

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

O método anterior tem o modificador `async` na sua assinatura. Isso sinaliza ao compilador que esse método contém uma instrução `await`; ele contém operações assíncronas. Este método representa a tarefa que torra o pão e, em seguida, adiciona manteiga e geleia. Esse método retorna um `Task<TResult>` que representa a composição dessas três operações. O principal bloco de código agora se torna:

```
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!");
}
```

A alteração anterior ilustrou uma técnica importante para trabalhar com código assíncrono. Você pode compor tarefas, separando as operações em um novo método que retorna uma tarefa. Você pode escolher quando aguardar essa tarefa. Você pode iniciar outras tarefas simultaneamente.

Exceções assíncronas

Até este ponto, você implicitamente assumiu que todas essas tarefas foram concluídas com êxito. Os métodos assíncronos lançam exceções, assim como suas contrapartes síncronas. O suporte assíncrono para exceções e tratamento de erros busca as mesmas metas que o suporte assíncrono em geral: você deve escrever um código que leia como uma série de instruções síncronas. As tarefas geram exceções quando não podem ser concluídas

com êxito. O código do cliente pode capturar essas exceções quando uma tarefa iniciada é awaited. Por exemplo, vamos pressupor que o torradeira capture incêndio enquanto faz o sistema de notificação. Você pode simular isso modificando o `ToastBreadAsync` método para corresponder ao seguinte código:

```
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

Você receberá um aviso quando compilar o código anterior em relação ao código inacessível. Isso é intencional, porque, uma vez que o torradeira capture incêndios, as operações não continuarão normalmente.

Execute o aplicativo depois de fazer essas alterações e você produzirá uma saída semelhante ao seguinte texto:

```
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)
```

Observe que há algumas tarefas sendo concluídas entre quando o torradeira capture incêndio e a exceção é observada. Quando uma tarefa executada de forma assíncrona gera uma exceção, essa tarefa *falha*. O objeto Task contém a exceção gerada na `Task.Exception` propriedade. As tarefas com falha geram uma exceção quando são aguardadas.

Há dois mecanismos importantes a serem compreendidos: como uma exceção é armazenada em uma tarefa com falha e como uma exceção é desempacotada e relançada quando o código aguarda uma tarefa com falha.

Quando o código executado de forma assíncrona gera uma exceção, essa exceção é armazenada no `Task`. A `Task.Exception` propriedade é um `System.AggregateException` porque mais de uma exceção pode ser lançada durante o trabalho assíncrono. Qualquer exceção gerada é adicionada à `AggregateException.InnerExceptions` coleção. Se essa `Exception` propriedade for nula, um novo `AggregateException` será criado e a exceção gerada será o primeiro item da coleção.

O cenário mais comum para uma tarefa com falha é que a `Exception` propriedade contém exatamente uma exceção. Ao codificar `awaits` uma tarefa com falha, a primeira exceção na `AggregateException.InnerExceptions` coleção é relançada. É por isso que a saída deste exemplo mostra um `InvalidOperationException` em vez de um `AggregateException`. A extração da primeira exceção interna torna o trabalho com métodos assíncronos o mais semelhante possível ao trabalho com suas contrapartes síncronas. Você pode examinar a `Exception` propriedade em seu código quando seu cenário pode gerar várias exceções.

Antes de prosseguir, comente essas duas linhas em seu `ToastBreadAsync` método. Você não quer iniciar outro incêndio:

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Aguardar tarefas com eficiência

A série de instruções `await` no final do código anterior pode ser melhorada usando métodos da classe `Task`. Uma dessas APIs é a `WhenAll`, que retorna um `Task` que é concluído ao final de todas as tarefas na lista de argumentos, conforme mostrado no código a seguir:

```
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!");
```

Outra opção é usar `WhenAny`, que retorna uma `Task<Task>` que é concluída quando qualquer um dos argumentos é concluído. Você pode aguardar a tarefa retornada, sabendo que ela já foi concluída. O código a seguir mostra como você poderia usar `WhenAny` para aguardar a primeira tarefa concluir e, em seguida, processar seu resultado. Depois de processar o resultado da tarefa concluída, você remove essa tarefa concluída da lista de tarefas passada para `WhenAny`.

```
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");
    }
    breakfastTasks.Remove(finishedTask);
}
```

Depois de todas essas alterações, a versão final do código tem esta aparência:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    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");
                }
                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");

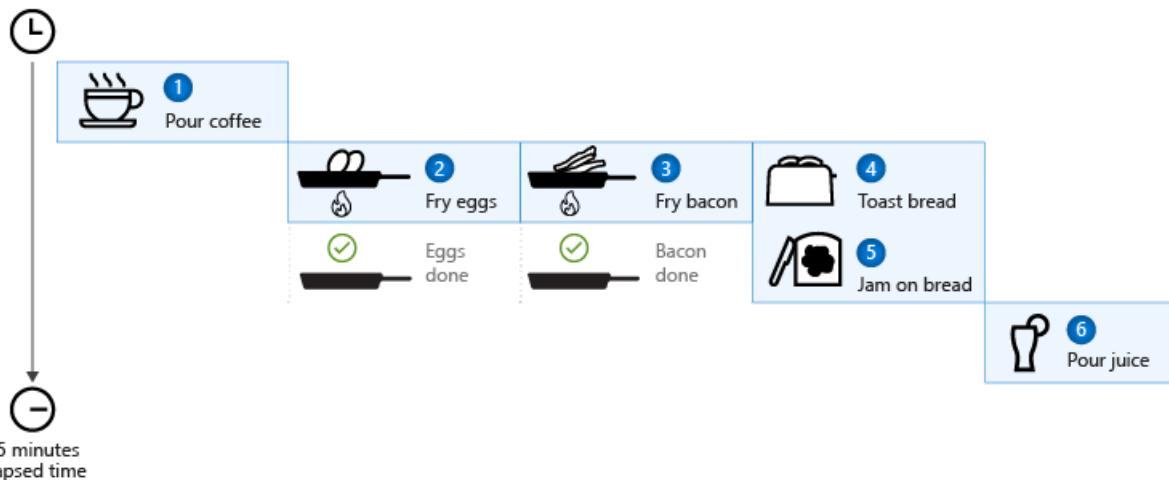
    return new Bacon();
}

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();
}
}

```



A versão final do café da manhã preparado de forma assíncrona levou aproximadamente 15 minutos, pois algumas tarefas foram executadas simultaneamente, e o código monitorou várias tarefas de uma vez e levou apenas uma ação quando fosse necessário.

Esse código final é assíncrono. Ele reflete mais precisamente como uma pessoa poderia preparar um café da manhã. Compare o código anterior com o primeiro exemplo de código neste artigo. As ações principais permanecem claras ao ler o código. Você pode ler esse código da mesma forma como faria ao ler essas instruções para fazer um café da manhã no início deste artigo. Os recursos de linguagem para `async` e `await` fornecem a tradução que todas as pessoas fazem para seguir essas instruções escritas: iniciar tarefas assim que possível e não ficar bloqueado ao aguardar a conclusão de tarefas.

Próximas etapas

[Explore cenários do mundo real para programas assíncronos](#)

Programação assíncrona

21/01/2022 • 11 minutes to read

Se você tiver necessidades vinculadas a E/S (como solicitar dados de uma rede, acessar um banco de dados ou ler e escrever em um sistema de arquivos), será melhor utilizar 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 no nível da linguagem, que permite escrever facilmente código assíncrono sem precisar lidar com retornos de chamada ou estar em conformidade com uma biblioteca que dá 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 o código com limite de E/S, você aguarda uma operação que retorna um `Task` ou dentro de um `Task<T> async` método.
- Para o código vinculado à CPU, você aguarda uma operação que é iniciada em um thread em segundo plano com o `Task.Run` método .

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 [haja maneiras de](#) abordar o código assíncrono diferente de e , este artigo se concentra nos `async` `await` constructos de nível de linguagem.

Exemplo de E/S vinculada: Baixar dados de um serviço Web

Talvez seja necessário baixar alguns dados de um serviço Web quando um botão for pressionado, mas não quiser bloquear o thread da interface do usuário. Isso pode ser feito desta forma:

```
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 dados de forma assíncrona) sem ficar confuso em interagir com `Task` objetos.

Exemplo de limite de 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 aguarda seu resultado usando `await`. Isso permite que a interface do usuário se sinta suave à medida que o trabalho está sendo feito.

```
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, não requer o gerenciamento manual de um thread em segundo plano e faz isso de maneira sem bloqueio.

O que acontece nos bastidores

Há muitas partes móveis no que diz respeito a operações assíncronas. Se você estiver curioso sobre o que está acontecendo sob os cobertos de `e`, consulte o [Task](#) [Task<T>](#) artigo [Async em detalhes](#) para obter mais informações.

No lado do C#, o compilador transforma seu código em um computador de estado que acompanha coisas como gerar execução quando um é atingido e retomar a execução quando um trabalho em segundo plano `await` é concluído.

Para teoricamente, essa é uma implementação do [Modelo de Promessa de assincronia](#).

Principais partes a 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 E/S

Os dois primeiros exemplos deste guia mostraram como você pode usar e para o trabalho vinculado à `async` `await` E/S e à CPU. É fundamental que você possa identificar quando um trabalho que você precisa fazer é vinculado à E/S ou à CPU, pois ele pode afetar muito o desempenho do código e potencialmente levar ao uso incorretamente 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 um cálculo caro?

Se você respondeu "sim", seu trabalho é vinculado à CPU.

Se o trabalho que você tem for vinculado à E/S, use `async` e `await` sem `Task.Run`. Você não deve usar a biblioteca de paralelismo de tarefas. O motivo para isso é descrito em [Assíncrono em Profundidade](#).

Se o trabalho que você tem for vinculado à CPU e você se importar com a capacidade de resposta, use e , mas gerará o trabalho `async` em outro thread `await com Task.Run` . Se o trabalho for apropriado para simultaneidade e paralelismo, considere também usar a [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

Esse snippet baixa o HTML da home page em e conta o número de vezes que a cadeia de <https://dotnetfoundation.org> ocorre no HTML. Ele usa ASP.NET para definir um método de controlador de API Web, que executa essa tarefa e retorna o número.

NOTE

Se você pretende fazer análise de HTML no código de produção, não use expressões regulares. Use uma biblioteca de análise.

```
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:

```

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;
}

```

Aguarde a conclusão de várias tarefas

Você pode encontrar em uma situação em que precisa recuperar várias partes de dados simultaneamente. A API contém dois métodos, e , que permitem escrever código assíncrono que executa uma espera sem bloqueio em `Task` vários trabalhos em segundo `Task.WhenAll` `Task.WhenAny` plano.

Este exemplo mostra como você pode obter os dados `User` para um conjunto de `userId`s.

```

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 outra maneira de escrever isso de forma mais sucinta, usando LINQ:

```

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));
    return await Task.WhenAll(getUserTasks);
}

```

Embora seja menos código, tenha 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()`.

Informações e conselhos importantes

Com a programação assíncrona, há alguns detalhes a ter em mente que podem impedir um comportamento inesperado.

- `async` os métodos precisam ter um `await` palavra-chave em seu corpo ou eles nunca produzirão!

É importante ter isso em mente. Se não for usado no corpo de um método, o compilador C# gerará um aviso, mas o código será compilado e executado como se fosse `await` `async` um método normal. Isso é incrivelmente ineficiente, pois o computador de estado gerado pelo compilador C# para o método assíncrono não está realizando nada.

- Adicione "Assíncrono" como o sufixo de cada nome de método assíncrono que você escreve.

Essa é a convenção usada no .NET para diferenciar mais facilmente métodos síncronos e assíncronos. Determinados métodos que não são explicitamente chamados pelo seu código (como manipuladores de eventos ou métodos de controlador da Web) não se aplicam necessariamente. Como eles não são explicitamente chamados pelo seu código, ser explícito sobre o nome entre eles não é tão importante.

- O `async void` 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:

- Exceções lançadas em `async void` um método não podem ser capturadas fora desse método.
- `async void` os métodos são difíceis de testar.
- `async void` Os métodos poderão causar efeitos colaterais ruins se o chamador não estiver esperando que eles sejam assíncronos.
- **Vá com cuidado ao usar lambdas assíncronas em expressões LINQ**

As expressões lambda no LINQ usam a execução adiada, o que significa que o código pode acabar sendo executado em um momento em que você não está esperando isso. 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. Async e LINQ são poderosos, mas devem ser usados juntos da 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 um pode resultar em deadlocks e threads de contexto bloqueados e pode exigir tratamento `Task` de erros mais complexo. A tabela a seguir fornece diretrizes sobre como lidar com a espera de tarefas sem bloqueio:

USE ISTO...	EM VEZ DISTO...	AO 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 Task.WhenAny</code>	<code>Task.WaitAny</code>	Aguardar a conclusão de qualquer tarefa
<code>await 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. Nos casos em que um método declarado com o modificador 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 `async` do 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 `Task.ConfigureAwait(Boolean)` método?". O método permite que uma `Task` instância configure seu awaiter. Essa é uma consideração importante e defini-la incorretamente pode potencialmente ter implicações de desempenho e até mesmo deadlocks. Para obter mais informações `ConfigureAwait` sobre , consulte as Perguntas frequentes sobre `ConfigureAwait`.

- **Escrever código com menos monitoração de estado**

Não dependa 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 mantinível.

Outros recursos

- A [Programação assíncrona em detalhes](#) fornece mais informações sobre o funcionamento de Tarefas.
- [O modelo de programação assíncrona Task \(C#\)](#).

Modelo de programação assíncrona de tarefa

21/01/2022 • 14 minutes to read

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# 5](#) apresentou uma programação assíncrona de abordagem simplificada que aproveita o suporte assíncrono no .NET Framework 4.5 e superior, no .NET Core e no Windows Runtime. 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.

O Async 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	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Trabalhando com arquivos	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile
Trabalhando com imagens		MediaCapture BitmapEncoder BitmapDecoder
Programação WCF	Operações síncronas e assíncronas	

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.

Os métodos assíncronos são fáceis de escrever

As palavras-chave `async` e `await` em C# são a parte central da programação assíncrona. usando essas duas palavras-chave, você pode usar recursos no .NET Framework, no .NET Core ou no Windows Runtime para criar um método assíncrono quase tão fácil 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. Quase tudo no código deve parecer familiar para você.

você pode encontrar um exemplo completo de Windows Presentation Foundation (WPF) disponível para download da [programação assíncrona com `async` e `await` em C#](#).

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.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.

```
string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

As características a seguir resumem o que torna o exemplo anterior 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` (começando com o C# 7.0).

Para obter mais informações, consulte a seção [tipos de retorno e parâmetros](#).

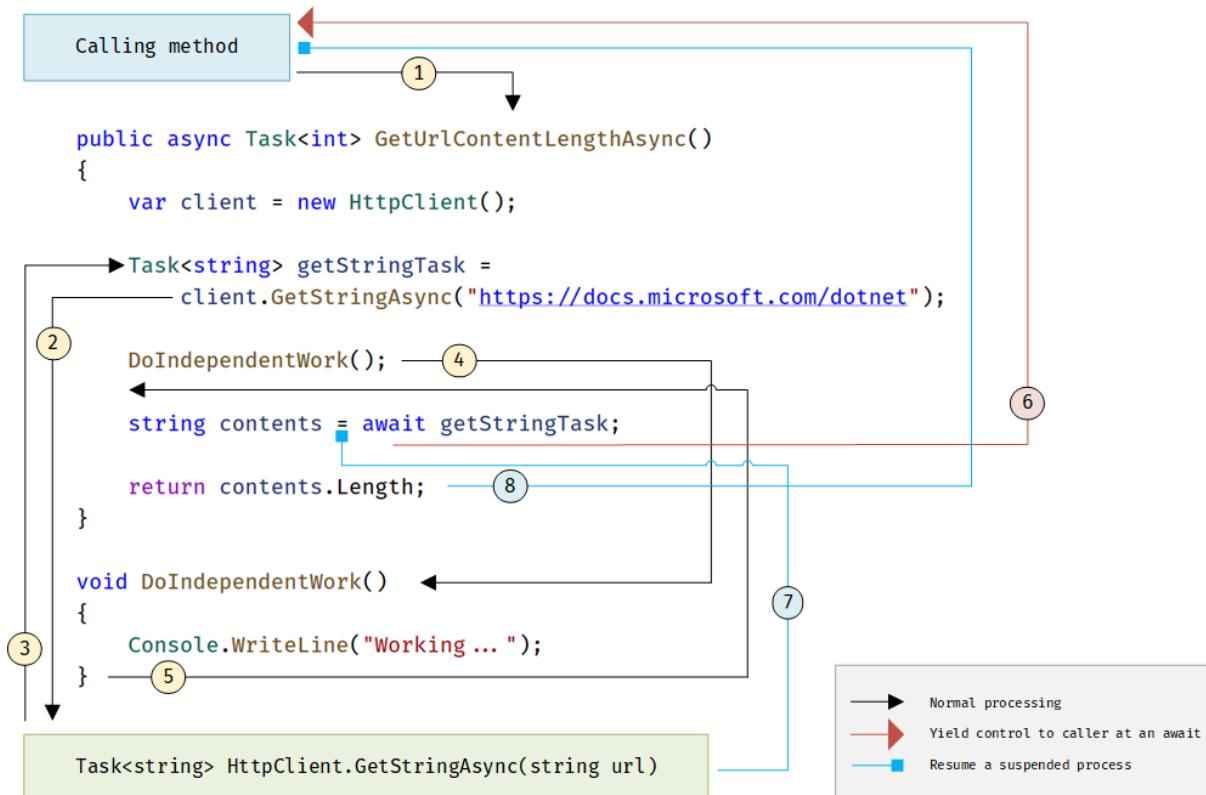
- O método geralmente inclui, pelo menos, uma expressão `await`, que marca um ponto em que o método não poderá continuar enquanto a operação assíncrona aguardada não for concluída. Enquanto isso, o método é suspenso e o controle retorna para o chamador do método. A próxima seção deste tópico ilustra 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 em versões anteriores do .NET Framework, consulte [TPL e programação assíncrona de .NET Framework tradicional](#).

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 no diagrama correspondem às etapas a seguir, iniciadas quando um método de chamada chama o

método Async.

1. Um método de chamada chama e aguarda o `GetUrlContentLengthAsync` método Async.
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.

NOTE

Se `GetStringAsync` (e, portanto, `getStringTask`) for concluído antes que `GetUrlContentLengthAsync` o aguarde, o controle permanecerá em `GetUrlContentLengthAsync`. A despesa de suspender e de retornar para `GetUrlContentLengthAsync` seria desperdiçada se o processo assíncrono `getStringTask` já tiver sido concluído e `GetUrlContentLengthAsync` não tiver que esperar pelo 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 de API

Você pode estar curioso para saber onde encontrar métodos como `GetStringAsync` que oferecem suporte à programação assíncrona. .NET Framework 4,5 ou superior e o .NET Core contêm muitos membros que funcionam com o `async` e o `await`. Você pode reconhecê-los pelo sufixo "Async" que é anexado ao nome do membro e por seu tipo de retorno de `Task` ou `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 `await` expressão em um método assíncrono não bloqueia o thread atual enquanto a tarefa esperada 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 `Task.Run` método, a programação assíncrona é melhor do que `BackgroundWorker` para operações associadas à CPU porque a programação assíncrona separa os detalhes de coordenação da execução do código do trabalho que `Task.Run` transfere para o 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 `await` expressão não constitui uma saída do método e os `finally` blocos não são executados.

- O método assíncrono marcado pode ser aguardado por métodos que o chamam.

Um método assíncrono normalmente contém uma ou mais ocorrências de um `await` operador, mas a ausência de `await` expressões não causa um erro de compilador. Se um método assíncrono não usar um `await` operador para marcar um ponto de suspensão, o método será executado como um método síncrono, apesar do `async` modificador. 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 de retorno e parâmetros

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 especifica `Task<TResult>` como o tipo de retorno se o método contiver uma `return` instrução que especifica um operando do tipo `TResult`.

Você usará `Task` como o tipo de retorno se o método não tiver nenhuma instrução `return` ou se tiver uma instrução `return` que não retorna um operando.

Começando com C# 7.0, também será possível 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.Extension`.

O exemplo a seguir mostra como você declara e chama um método que retorna um `Task<TResult>` ou um `Task`:

```
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 um `void` tipo de retorno não pode ser aguardado, e o chamador de um método de retorno nulo não pode capturar nenhuma exceção que o método gera.

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, consulte [tipos de retorno assíncrono \(C#\)](#). Para obter mais informações sobre como capturar exceções nos métodos assíncronos, consulte [try-catch](#).

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 retornam tipos comumente awaitable (por exemplo, `Task`, `Task<T>`, `ValueTask`, `ValueTask<T>`) devem ter nomes que terminem 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)

TÍTULO	DESCRIÇÃO
Como fazer várias solicitações da Web em paralelo usando Async e Await (C#)	Demonstra como iniciar várias tarefas ao mesmo tempo.
Tipos de retorno assíncrono (C#)	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.	Mostra como adicionar a seguinte funcionalidade à sua solução assíncrona: <ul style="list-style-type: none">- Cancelar uma lista de tarefas (C#)- Cancelar tarefas após um período de tempo (C#)- Processar tarefa assíncrona como concluída (C#)
Usando Async para acesso a arquivos (C#)	Lista e demonstra as vantagens de usar <code>async</code> e <code>await</code> para acessar arquivos.
Padrão assíncrono baseado em tarefa (toque)	Descreve um padrão assíncrono, o padrão é baseado nos <code>Task</code> tipos e <code>Task<TResult></code> .

TÍTULO	DESCRIÇÃO
Vídeos sobre assincronia no Channel 9	Fornece links para uma variedade de vídeos sobre programação assíncrona.

Confira também

- [Async](#)
- [await](#)
- [Programação assíncrona](#)
- [Visão geral da assincronia](#)

Tipos de retorno assíncronos (C#)

21/01/2022 • 9 minutes to read

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.
- Começando com o C# 7.0, qualquer tipo que tenha um método acessível `GetAwaiter`. O objeto retornado pelo método `GetAwaiter` deve implementar a interface `System.Runtime.CompilerServices.ICriticalNotifyCompletion`.
- Começando com o C# 8.0, `IAsyncEnumerable<T>`, para um método assíncrono que retorna um fluxo assíncrono.

Para obter mais informações sobre métodos assíncronos, consulte [Programação assíncrona com async e await \(C#\)](#).

Também existem vários outros tipos específicos para Windows cargas de trabalho:

- `DispatcherOperation`, para operações assíncronas limitadas 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 o progresso e retornam um valor.

Tipo de retorno de 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 `WaitAndApologizeAsync` método não contém uma `return` instrução, portanto, o método retorna um `Task` objeto. Retornar um `Task` permite que seja `WaitAndApologizeAsync` aguardado. O `Task` tipo não inclui uma propriedade porque não tem nenhum valor de `Result` retorno.

```

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 operand direito de um `await` é um `Task`, a expressão produz um resultado de `Task<TResult>` `await T`. Quando o operand direito de um `await` é um `Task`, o e seu `Task await` operand são uma instrução.

Você pode separar a chamada para `WaitAndApologizeAsync` do aplicativo de um operador `await`, como mostra o 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.

```

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 `<TResult>` retorno de tarefa

O `Task<TResult>` tipo de retorno é usado para um método assíncrono que contém uma instrução de retorno na qual o operand é `TResult`.

No exemplo a seguir, o `GetLeisureHoursAsync` método contém uma `return` instrução que retorna um inteiro. A declaração do método deve especificar um tipo de retorno de `Task<int>`. O `FromResult` método assíncrono é um espaço reservado para uma operação que retorna um `DayOfWeek`.

```

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 recupera o resultado de um separando a chamada para do aplicativo `await` de , como mostra o código a `Task<T>` `GetLeisureHoursAsync` `await` 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` .

IMPORTANT

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 para bloquear o thread principal para que o método pudesse imprimir o no `Result` `Console` antes do aplicativo `Main` `message` terminar.

```

var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);

```

Tipo de retorno void

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 até a conclusão sem aguardar a conclusão do método assíncrono chamado. O chamador deve ser independente de quaisquer valores ou exceções que o método assíncrono gera.

O chamador de um método assíncrono que retorna nulo não pode capturar exceções lançadas do método. Essas exceções semhandled provavelmente causarão falha no aplicativo. Se um método que retorna um `Task<TResult>` ou lança uma exceção, a exceção é armazenada na tarefa retornada. A exceção é relrown quando a tarefa é aguardada. Certifique-se de que qualquer método assíncrono que possa produzir uma exceção tenha um tipo de retorno de ou e que as chamadas para `Task<TResult>` o método sejam aguardadas.

Para obter mais informações sobre como capturar exceções em métodos assíncronos, consulte a seção Exceções em métodos [assíncronos](#) do artigo `try-catch`.

O exemplo a seguir mostra o comportamento de um manipulador de eventos assíncrono. No código de exemplo, um manipulador de eventos assíncrono deve permitir que o thread principal saiba quando ele for finalizado. Em seguida, o thread principal pode aguardar um manipulador de eventos assíncronos ser concluído antes de sair do programa.

```
using System;
using System.Threading.Tasks;

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>

A partir do C# 7.0, um método assíncrono pode retornar qualquer tipo que tenha um método acessível que retorna uma instância de um `GetAwaiter` *tipo awaite*. Além disso, o tipo retornado do `GetAwaiter` método deve ter o atributo `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`. Você pode saber mais no artigo sobre Atributos lidos pelo compilador ou a especificação de recurso para Tarefa, como tipos de retorno.

Esse recurso é o complemento para [expressões a esperadas](#), que descreve os requisitos para o operando `await`. Os tipos de retorno assíncronos generalizados permitem que o compilador `async` gere métodos que retornam tipos diferentes. Os tipos de retorno assíncronos generalizados habilitaram melhorias de desempenho nas bibliotecas do .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. Para usar o tipo `System.Threading.Tasks.ValueTask<TResult>`, você deve adicionar o pacote NuGet `System.Threading.Tasks.Extensions` ao seu projeto. O exemplo a seguir usa a estrutura `ValueTask<TResult>` para recuperar o valor de dois lançamentos de dados.

```

using System;
using System.Threading.Tasks;

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 e é direcionado para uso em ambientes especializados. Considere usar os tipos `Task`, `Task<T>` ou `ValueTask<T>` disso, que abrangem a maioria dos cenários de código assíncrono.

No C# 10 e posterior, você pode aplicar o atributo `[AsyncMethodBuilder]` a um método assíncrono (em vez da declaração de tipo de retorno assíncrono) para substituir o construtor desse `AsyncMethodBuilder` tipo. Normalmente, você aplicaria esse atributo para usar um construtor diferente fornecido no runtime do .NET.

Fluxos assíncronos com `IAsyncEnumerable<T>`

A partir do C# 8.0, um método assíncrono pode retornar um *fluxo assíncrono*, representado por `IAsyncEnumerable<T>`. Um fluxo assíncrono fornece uma maneira de enumerar itens lidos de um fluxo quando os elementos são gerados em partes com chamadas assíncronas repetidas. O exemplo a seguir mostra um método assíncrono que gera um fluxo assíncrono:

```
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 forma assíncrona. Depois que cada linha é lida, o código enumera cada palavra na cadeia de caracteres. Os chamadores enumeram cada palavra usando a `await foreach` instrução. O método aguarda quando precisa ler de forma assíncrona 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 (C#)

21/01/2022 • 4 minutes to read

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 [CancellationTokenSource](#) instância a cada tarefa. Se você selecionar a tecla `Enter`, cancelará todas as tarefas que ainda não foram concluídas.

Este tutorial abrange:

- Criando um aplicativo de console .NET
- Escrevendo um aplicativo assíncrono que dá suporte ao cancelamento
- Demonstrando o cancelamento de sinalização

Pré-requisitos

Este tutorial exige o seguinte:

- [SDK do .NET 5 ou posterior](#)
- IDE (ambiente de desenvolvimento integrado)
 - é recomendável [Visual Studio](#), [Visual Studio Code](#) ou [Visual Studio para Mac](#)

Criar aplicativo de exemplo

Crie um novo aplicativo de console .NET Core. Você pode criar um usando o `dotnet new console` comando ou de [Visual Studio](#). Abra o arquivo `Program.cs` em seu editor de código favorito.

Substituir usando instruções

Substitua as instruções `using` existentes por estas declarações:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Adicionar campos

Na `Program` definição de classe, adicione estes três campos:

```

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://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.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` mantém todas as URLs que o aplicativo planeja processar.

Atualizar ponto de entrada do aplicativo

O ponto de entrada principal no aplicativo de console é o `Main` método. Substitua o método existente pelo seguinte:

```

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();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    Console.WriteLine("Application ending.");
}

```

O `Main` método atualizado 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 `Task` instância chamada `cancelTask`, que lerá os traços de tecla do console. Se a tecla Enter for pressionada, será feita uma chamada para `CancellationTokenSource.Cancel()`. Isso sinalizará o cancelamento. Em seguida, a `sumPageSizesTask` variável é atribuída a partir do `SumPageSizesAsync` método. Em seguida, as duas tarefas são passadas para `Task.WhenAny(Task[])` o, o que continuará quando qualquer uma das duas tarefas for concluída.

Criar o método de tamanhos de página de soma assíncrona

Abaixo do `Main` método, adicione o `SumPageSizesAsync` método:

```
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($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}
```

O método começa instanciando e iniciando um `Stopwatch`. Em seguida, ele executa um loop em cada URL nas `s_urlList` chamadas e `ProcessUrlAsync`. Com cada iteração, o `s_cts.Token` é passado para o `ProcessUrlAsync` método e o código retorna um `Task<TResult>`, em que `TResult` é um inteiro:

```
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 `ProcessUrlAsync` método abaixo do `SumPageSizesAsync` método:

```
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 `client` instância fornecida para obter a resposta como um `byte[]`. A `CancellationToken` instância é passada para os `HttpClient.GetAsync(String, CancellationToken)` `HttpContent.ReadAsByteArrayAsync()` métodos e. O `token` é usado para se registrar para o cancelamento solicitado. O comprimento é retornado depois que a URL e o comprimento são gravados no console.

Exemplo de saída de aplicativo

```
Application started.  
Press the ENTER key to cancel...  
  
https://docs.microsoft.com 37,357  
https://docs.microsoft.com/aspnet/core 85,589  
https://docs.microsoft.com/azure 398,939  
https://docs.microsoft.com/azure/devops 73,663  
https://docs.microsoft.com/dotnet 67,452  
https://docs.microsoft.com/dynamics365 48,582  
https://docs.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.

```
using System;  
using System.Collections.Generic;  
using System.Diagnostics;  
using System.Net.Http;  
using System.Threading;  
using System.Threading.Tasks;  
  
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://docs.microsoft.com",  
        "https://docs.microsoft.com/aspnet/core",  
        "https://docs.microsoft.com/azure",  
        "https://docs.microsoft.com/azure/devops",  
        "https://docs.microsoft.com/dotnet",  
        "https://docs.microsoft.com/dynamics365",  
        "https://docs.microsoft.com/education",  
        "https://docs.microsoft.com/enterprise-mobility-security",  
        "https://docs.microsoft.com/gaming",  
        "https://docs.microsoft.com/graph",  
        "https://docs.microsoft.com/microsoft-365",  
        "https://docs.microsoft.com/office",  
        "https://docs.microsoft.com/powershell",  
        "https://docs.microsoft.com/sql",  
        "https://docs.microsoft.com/surface",  
        "https://docs.microsoft.com/system-center",  
        "https://docs.microsoft.com/visualstudio",  
        "https://docs.microsoft.com/windows",  
        "https://docs.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();

await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

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 (C#)

21/01/2022 • 2 minutes to read

Você pode cancelar uma operação assíncrona após um período de tempo usando o método se não quiser aguardar a operação `CancellationTokenSource.CancelAfter` ser finalada. Esse método agenda o cancelamento de todas as tarefas associadas que não são concluídas dentro do período de tempo designado pela `CancelAfter` expressão.

Este exemplo adiciona ao código desenvolvido em Cancelar uma lista de tarefas (C#) para baixar uma lista de sites e exibir o comprimento do conteúdo 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#)
- [SDK do .NET 5 ou posterior](#)
- IDE (ambiente de desenvolvimento integrado)
 - Recomendamos [Visual Studio](#), [Visual Studio Code](#) ou [Visual Studio para Mac](#)

Atualizar ponto de entrada do aplicativo

Substitua o método `Main` existente pelo seguinte:

```
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 de instrução no console. Dentro do `try catch`, uma chamada para agenda `CancellationTokenSource.CancelAfter(Int32)` um cancelamento. Isso sinaliza o cancelamento após um período de tempo.

Em seguida, `SumPageSizesAsync` o método é aguardado. Se o processamento de todas as URLs ocorrer mais rapidamente do que o cancelamento agendado, o aplicativo terminará. No entanto, se o cancelamento agendado for disparado antes que todas as URLs sejam processadas, `OperationCanceledException` um será lançado.

Exemplo de saída do aplicativo

```
Application started.

https://docs.microsoft.com           37,357
https://docs.microsoft.com/aspnet/core   85,589
https://docs.microsoft.com/azure        398,939
https://docs.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.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

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://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.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($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"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#)

21/01/2022 • 9 minutes to read

Usando [Task.WhenAny](#) o, você pode iniciar várias tarefas ao mesmo tempo e processá-las uma a uma conforme elas são concluídas, em vez de processá-las na ordem em que elas são 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 versão prévia do 17.0.0](#) com a carga de [trabalho de desenvolvimento do .net desktop](#) instalada. O SDK do .NET 6,0 é instalado automaticamente quando você seleciona essa carga de trabalho.
- o [SDK do .net 6,0](#) com um editor de código de sua escolha, como [Visual Studio Code](#).

Criar aplicativo de exemplo

Crie um novo aplicativo de console .NET Core destinado ao .NET 6,0. Você pode criar um usando o comando [dotnet New console](#) ou de Visual Studio.

Abra o arquivo *Program.cs* em seu editor de código e substitua o código existente por este código:

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Adicionar campos

Na `Program` definição de classe, adicione os dois campos a seguir:

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

O `HttpClient` expõe a capacidade de enviar solicitações HTTP e receber respostas http. O `s_urlList` mantém todas as URLs que o aplicativo planeja processar.

Atualizar ponto de entrada do aplicativo

O ponto de entrada principal no aplicativo de console é o `Main` método. Substitua o método existente pelo seguinte:

```
static Task Main() => SumPageSizesAsync();
```

O `Main` método atualizado 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 `Main` método, adicione o `SumPageSizesAsync` método:

```

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 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:

```

IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);

```

Devido à [execução retardada](#) com o LINQ, você chama `Enumerable.ToList` para iniciar cada tarefa.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

O `while` loop executa as seguintes etapas para cada tarefa na coleção:

1. Aguarda uma chamada para `WhenAny` para identificar a primeira tarefa na coleção que concluiu seu download.

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. Remove a tarefa da coleção.

```
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 falhado, `await` o lançará a primeira exceção filha armazenada no `AggregateException`, ao contrário da leitura da `Task<TResult>.Result` propriedade, que geraria o `AggregateException`.

```
total += await finishedTask;
```

Adicionar método de processo

Adicione o seguinte `ProcessUrlAsync` método abaixo do `SumPageSizesAsync` método:

```
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 `client` instância 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.

Caution

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.

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
        "https://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };

    static Task Main() => SumPageSizesAsync();
}
```

```

        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");
    }

    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://docs.microsoft.com/windows          25,513
// https://docs.microsoft.com/gaming          30,705
// https://docs.microsoft.com/dotnet          69,626
// https://docs.microsoft.com/dynamics365      50,756
// https://docs.microsoft.com/surface          35,519
// https://docs.microsoft.com                39,531
// https://docs.microsoft.com/azure/devops     75,837
// https://docs.microsoft.com/xamarin          60,284
// https://docs.microsoft.com/system-center     43,444
// https://docs.microsoft.com/enterprise-mobility-security 28,946
// https://docs.microsoft.com/microsoft-365      43,278
// https://docs.microsoft.com/visualstudio      31,414
// https://docs.microsoft.com/office           42,292
// https://docs.microsoft.com/azure             401,113
// https://docs.microsoft.com/graph            46,831
// https://docs.microsoft.com/education         25,098
// https://docs.microsoft.com/powershell        58,173
// https://docs.microsoft.com/aspnet/core       87,763
// https://docs.microsoft.com/sql               53,362

// Total bytes returned: 1,249,485
// Elapsed time: 00:00:02.7068725

```

Confira também

- [WhenAny](#)
- [Programação assíncrona com `async` e `await` \(C#\)](#)

Usando `Task.WhenAny` o, você pode iniciar várias tarefas ao mesmo tempo e processá-las uma a uma conforme elas são concluídas, em vez de processá-las na ordem em que elas são 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](#) com a carga de **trabalho de desenvolvimento do .net desktop** instalada. O SDK do .NET é instalado automaticamente quando você seleciona essa carga de trabalho.
- o [SDK do .net 5,0](#) com um editor de código de sua escolha, como [Visual Studio Code](#).

Criar aplicativo de exemplo

Crie um novo aplicativo de console .NET Core que tenha como alvo o .NET 5,0 ou o .NET Core 3,1. Você pode criar um usando o comando [dotnet New console](#) ou de Visual Studio. Abra o arquivo *Program.cs* em seu editor de código favorito.

Substituir usando instruções

Substitua as instruções `using` existentes por estas declarações:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

Adicionar campos

Na `Program` definição de classe, adicione os dois campos a seguir:

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

O `HttpClient` expõe a capacidade de enviar solicitações HTTP e receber respostas http. O `s_urlList` mantém todas as URLs que o aplicativo planeja processar.

Atualizar ponto de entrada do aplicativo

O ponto de entrada principal no aplicativo de console é o `Main` método. Substitua o método existente pelo seguinte:

```
static Task Main() => SumPageSizesAsync();
```

O `Main` método atualizado agora é considerado um [Async Main](#), que permite um ponto de entrada assíncrono no executável. É expressa uma chamada para `SumPageSizesAsync`.

Criar o método de tamanhos de página de soma assíncrona

Abaixo do `Main` método, adicione o `SumPageSizesAsync` método:

```

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 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:

```

IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);

```

Devido à [execução retardada](#) com o LINQ, você chama `Enumerable.ToList` para iniciar cada tarefa.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

O `while` loop executa as seguintes etapas para cada tarefa na coleção:

1. Aguarda uma chamada para `WhenAny` para identificar a primeira tarefa na coleção que concluiu seu download.

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. Remove a tarefa da coleção.

```
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 falhado, `await` o lançará a primeira exceção filha armazenada no `AggregateException`, ao contrário da leitura da `Task<TResult>.Result` propriedade, que geraria o `AggregateException`.

```
total += await finishedTask;
```

Adicionar método de processo

Adicione o seguinte `ProcessUrlAsync` método abaixo do `SumPageSizesAsync` método:

```
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 `client` instância 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.

Caution

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.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace ProcessTasksAsTheyFinish
{
    class Program
    {
        static readonly HttpClient s_client = new HttpClient
        {
            MaxResponseContentBufferSize = 1_000_000
        };

        static readonly IEnumerable<string> s_urlList = new string[]
        {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
        };
    }
}
```

```

        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };

    static Task Main() => SumPageSizesAsync();

    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");
    }

    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://docs.microsoft.com/windows                25,513
// https://docs.microsoft.com/gaming               30,705
// https://docs.microsoft.com/dotnet                69,626
// https://docs.microsoft.com/dynamics365           50,756
// https://docs.microsoft.com/surface              35,519
// https://docs.microsoft.com                      39,531
// https://docs.microsoft.com/azure/devops          75,837
// https://docs.microsoft.com/xamarin              60,284
// https://docs.microsoft.com/system-center         43,444
// https://docs.microsoft.com/enterprise-mobility-security 28,946
// https://docs.microsoft.com/microsoft-365          43,278
// https://docs.microsoft.com/visualstudio          31,414
// https://docs.microsoft.com/office                 42,292
// https://docs.microsoft.com/azure                  401,113
// https://docs.microsoft.com/graph                 46,831
// https://docs.microsoft.com/education             25,098
// https://docs.microsoft.com/powershell            58,173
// https://docs.microsoft.com/aspnet/core            87,763
// https://docs.microsoft.com/sql                   53,362

// Total bytes returned: 1,249,485
// Elapsed time: 00:00:02.7068725

```

Confira também

- [WhenAny](#)

- Programação assíncrona com `async` e `await` (C#)

Acesso a arquivo assíncrono (C#)

21/01/2022 • 5 minutes to read

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 geralmente 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 finito sobre as operações de E/S de arquivo, use a classe `FileStream`, que tem uma opção que faz com que a E/S assíncrona ocorra no nível `FileStream` 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.Aynchronous` na chamada do construtor.

Você não poderá usar essa opção com a `StreamReader` ou `StreamWriter` se abri-las diretamente especificando um caminho de arquivo. No entanto, você poderá usar essa opção se fornecer um `Stream` que a classe `FileStream` abriu. As chamadas assíncronas são mais rápidas em aplicativos de interface do usuário mesmo que um thread do pool de threads seja bloqueado, porque o thread da interface do usuário não é bloqueado durante a espera.

Escrever texto

Os exemplos a seguir escrevem 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 assíncrono está na definição de métodos que usam a instrução `await`.

Exemplo simples

```

public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}

```

Exemplo de controle finito

```

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:

```

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

```

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 [ReadAsync](#) método retorna um [Task <Int32>](#), portanto, a

avaliação do await produz um valor após a conclusão da `Int32 numRead` operação. Para obter mais informações, consulte [Tipos de retorno assíncronos \(C#\)](#).

```
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 escrevendo 10 arquivos de texto.

Exemplo simples

```
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 totalmente do processamento paralelo e não do processamento assíncrono. As vantagens da assincronia são que ela não encadeia vários threads e que ela não acarreça o thread da interface do usuário.

```

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 \(C#\)](#)

Atributos (C#)

21/01/2022 • 5 minutes to read

Os atributos fornecem um método eficiente de associação de metadados, ou informações declarativas, ao código (assemblies, tipos, métodos, propriedades e etc.). Após um atributo ser associado a uma entidade de programa, o atributo poderá ser consultado no tempo de execução usando uma técnica chamada *reflexão*. Para obter mais informações, consulte [Reflexão \(C#\)](#).

Os atributos têm as seguintes propriedades:

- Os atributos adicionam metadados ao seu programa. Os *metadados* são informações sobre os tipos definidos em um programa. Todos os assemblies .NET contêm um conjunto de metadados especificado que descreve os tipos e os membros de tipo definidos no assembly. Você pode adicionar atributos personalizados para especificar qualquer informação adicional necessária. Para obter mais informações, consulte [Criando atributos personalizados \(C#\)](#).
- Você pode aplicar um ou mais atributos a assemblies completos, módulos ou elementos de programas menores, como classes e propriedades.
- Os atributos podem aceitar argumentos da mesma forma que métodos e propriedades.
- Seu programa pode examinar seus próprios metadados ou os metadados em outros programas usando reflexão. Para obter mais informações, consulte [Acessando atributos usando reflexão \(C#\)](#).

Usando atributos

Os atributos podem ser colocados em quase qualquer declaração, embora um atributo específico possa restringir os tipos de declarações em que ele é válido. No C#, você especifica um atributo colocando o nome do atributo entre colchetes ([]) acima da declaração da entidade à qual ele se aplica.

Neste exemplo, o atributo `SerializableAttribute` é usado para aplicar uma característica específica a uma classe:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

Um método com o atributo `DllImportAttribute` é declarado como este exemplo:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

Mais de um atributo pode ser colocado em uma declaração como o seguinte exemplo mostra:

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Alguns atributos podem ser especificados mais de uma vez para uma determinada entidade. Um exemplo de

um atributo multiuso é [ConditionalAttribute](#):

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

NOTE

Por convenção, todos os nomes de atributo terminam com a palavra "Atributo" para distingui-los de outros itens nas bibliotecas do .NET. No entanto, você não precisa especificar o sufixo de atributo ao usar atributos no código. Por exemplo, `[DllImport]` é equivalente a `[DllImportAttribute]`, mas `DllImportAttribute` é o nome real do atributo na biblioteca de classes do .NET.

Parâmetros de atributo

Muitos atributos têm parâmetros, que podem ser nomeados, sem nome ou posicionais. Quaisquer parâmetros de posição devem ser especificados em uma determinada ordem e não podem ser omitidos. Parâmetros nomeados são opcionais e podem ser especificados em qualquer ordem. Os parâmetros posicionais são especificados primeiro. Por exemplo, esses três atributos são equivalentes:

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

O primeiro parâmetro, o nome da DLL, é positional e sempre vir em primeiro lugar; os outros são nomeados. Nesse caso, ambos os parâmetros nomeados são padronizados como false e, portanto, podem ser omitidos. Parâmetros de posição correspondem aos parâmetros do construtor de atributo. Parâmetros nomeados ou opcionais correspondem a propriedades ou a campos do atributo. Consulte a documentação do atributo individual para obter informações sobre valores de parâmetro padrão.

Para obter mais informações sobre os tipos de parâmetro permitidos, consulte a seção [atributos](#) da [especificação da linguagem C#](#)

Destinos do atributo

O *destino* de um atributo é a entidade à qual o atributo se aplica. Por exemplo, um atributo pode ser aplicado a uma classe, um método específico ou um assembly inteiro. Por padrão, um atributo se aplica ao elemento que o segue. Mas você pode identificar explicitamente, por exemplo, se um atributo é aplicado a um método, ou a seu parâmetro ou a seu valor retornado.

Para identificar explicitamente um atributo de destino, use a seguinte sintaxe:

```
[target : attribute-list]
```

A lista de possíveis valores `target` é mostrada na tabela a seguir.

VALOR DE DESTINO	APLICA-SE A
<code>assembly</code>	Assembly inteiro
<code>module</code>	Módulo do assembly atual

VALOR DE DESTINO	APLICA-SE A
field	Campo em uma classe ou um struct
event	Evento
method	Método ou acessadores de propriedade <code>get</code> e <code>set</code>
param	Parâmetros de método ou parâmetros de acessador de propriedade <code>set</code>
property	Propriedade
return	Valor retornado de um método, indexador de propriedade ou acessador de propriedade <code>get</code>
type	Struct, classe, interface, enum ou delegado

Especifique o valor de destino `field` para aplicar um atributo ao campo de suporte criado para uma [propriedade autoimplementada](#).

O exemplo a seguir mostra como aplicar atributos a módulos e assemblies. Para obter mais informações, consulte [Atributos comuns \(C#\)](#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

O exemplo a seguir mostra como aplicar atributos a métodos, parâmetros de método e valores de retorno de método em 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

Independentemente dos destinos nos quais `ValidatedContract` é definido como válido, o destino `return` deve ser especificado, mesmo se `ValidatedContract` forem definidos para serem aplicados somente a valores de retorno. Em outras palavras, o compilador não usará as informações de `AttributeUsage` para resolver os destinos de atributos ambíguos. Para obter mais informações, consulte [AttributeUsage \(C#\)](#).

Usos comuns para atributos

A lista a seguir inclui alguns dos usos comuns de atributos no código:

- Marcar métodos usando o atributo `WebMethod` nos serviços Web para indicar que o método deve ser chamado por meio do protocolo SOAP. Para obter mais informações, consulte [WebMethodAttribute](#).
- Descrever como realizar marshaling de parâmetros de método ao interoperar com código nativo. Para obter mais informações, consulte [MarshalAsAttribute](#).
- Descrever as propriedades COM para classes, métodos e interfaces.
- Chamar o código não gerenciado usando a classe [DllImportAttribute](#).
- Descrever o assembly em termos de versão, título, descrição ou marca.
- Descrever quais membros de uma classe serializar para persistência.
- Descrever como fazer mapeamento entre nós XML e membros de classe para serialização de XML.
- Descrever os requisitos de segurança para métodos.
- Especificar as características usadas para impor a segurança.
- Controlar otimizações pelo compilador JIT (Just-In-Time) para que o código permaneça fácil de depurar.
- Obter informações sobre o chamador de um método.

Seções relacionadas

Para obter mais informações, consulte:

- [Criando atributos personalizados \(C#\)](#)
- [Acessando atributos usando reflexão \(C#\)](#)
- [Como criar uma União C/C++ usando atributos \(C#\)](#)
- [Atributos comuns \(C#\)](#)
- [Informações do chamador \(C#\)](#)

Confira também

- [Guia de programação C#](#)
- [Reflexão \(C#\)](#)
- [Atributos](#)
- [Usando atributos em C #](#)

Criando atributos personalizados (C#)

21/01/2022 • 2 minutes to read

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:

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct)  
]  
public class AuthorAttribute : System.Attribute  
{  
    private string name;  
    public double version;  
  
    public AuthorAttribute(string name)  
    {  
        this.name = name;  
        version = 1.0;  
    }  
}
```

O nome da `AuthorAttribute` classe é o nome do atributo, , mais o `Author` `Attribute` sufixo . 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 positional. 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:

```
[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.

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct,  
                      AllowMultiple = true) // multiuse attribute  
]  
public class AuthorAttribute : System.Attribute
```

No exemplo de código a seguir, vários atributos do mesmo tipo são aplicados a uma classe.

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

Confira também

- [System.Reflection](#)
- [Guia de Programação em C#](#)
- [Escrevendo atributos personalizados](#)
- [Reflexão \(C#\)](#)
- [Atributos \(C#\)](#)
- [Acessando atributos usando reflexão \(C#\)](#)
- [AttributeUsage \(C#\)](#)

Acessando atributos usando reflexão (C#)

21/01/2022 • 2 minutes to read

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:

```
[Author("P. Ackerman", version = 1.1)]  
class SampleClass
```

é conceitualmente equivalente a esta:

```
Author anonymousAuthorObject = new Author("P. Ackerman");  
anonymousAuthorObject.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 como acima. 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.

Exemplo

Aqui está um exemplo completo. Um atributo personalizado é definido, aplicado a várias entidades e recuperado por meio da reflexão.

```
// Multiuse attribute.  
[System.AttributeUsage(System.AttributeTargets.Class |  
    System.AttributeTargets.Struct,  
    AllowMultiple = true) // Multiuse attribute.  
]  
public class Author : System.Attribute  
{  
    string name;  
    public double version;  
  
    public Author(string name)  
    {  
        this.name = name;  
  
        // Default value.  
        version = 1.0;  
    }  
  
    public string GetName()  
    {  
        return 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
{
    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 {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t); // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("  {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}

/* 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](#)
- [Guia de Programação em C#](#)
- [Recuperando informações armazenadas em atributos](#)

- [Reflexão \(C#\)](#)
- [Atributos \(C#\)](#)
- [Criando atributos personalizados \(C#\)](#)

Como criar uma União C/C++ usando atributos (C#)

21/01/2022 • 2 minutes to read

Usando atributos, você pode personalizar a forma como as estruturas são colocadas 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`.

Exemplos

Neste segmento de código, todos os campos de `TestUnion` são iniciados no mesmo local na memória.

```
// Add a using directive for System.Runtime.InteropServices.  
  
[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;  
}
```

A seguir, temos outro exemplo em que os campos são iniciados em locais diferentes definidos explicitamente.

```
// Add a using directive for System.Runtime.InteropServices.  
  
[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(0)]  
    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`, compartilham os mesmos locais de memória que `lg`. Esse tipo de controle sobre o layout do struct é útil ao usar a invocação de plataforma.

Confira também

- [System.Reflection](#)
- [Attribute](#)
- [Guia de programação C#](#)
- [Atributos](#)
- [Reflexão \(C#\)](#)
- [Atributos \(C#\)](#)
- [Criando atributos personalizados \(C#\)](#)
- [Acessando atributos usando reflexão \(C#\)](#)

Coleções (C#)

21/01/2022 • 13 minutes to read

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.

NOTE

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`.

```
// 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.

```
// 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.Write(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`.

```
// 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.Write(salmons[index] + " ");  
}  
// Output: chinook coho pink sockeye
```

O exemplo a seguir remove um elemento da coleção, especificando o objeto a ser removido.

```

// 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 `foreach` uma instrução, `for` uma instrução que itera em ordem decrescente é usada. Isso é feito porque o método `RemoveAt` faz com que os elementos após um elemento removido tenham um valor de índice menor.

```

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.

```

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

Muitas 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
Dictionary< TKey,TValue >	Representa uma coleção de pares chave-valor organizados com base na chave.
List< T >	Representa uma lista de objetos que podem ser acessados por índice. Fornece métodos para pesquisar, classificar e modificar listas.
Queue< T >	Representa uma coleção de objetos PEPS (primeiro a entrar, primeiro a sair).

CLASSE	DESCRIÇÃO
<code>SortedList<TKey,TValue></code>	Representa uma coleção de pares chave/valor que são classificados por chave com base na implementação de <code>IComparer<T></code> associada.
<code>Stack<T></code>	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 fornecem operações thread-safe eficientes para acessar itens de coleção de [System.Collections.Concurrent](#) 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
<code>ArrayList</code>	Representa uma matriz de objetos cujo tamanho é aumentado dinamicamente conforme necessário.
<code>Hashtable</code>	Representa uma coleção de pares chave-valor organizados com base no código hash da chave.
<code>Queue</code>	Representa uma coleção de objetos PEPS (primeiro a entrar, primeiro a sair).
<code>Stack</code>	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`.

```
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.

```

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 acesse um item na coleção [elements](#) usando o [elements\[symbol\]](#) no 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.

```

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, [consulte Ponto de Partida com LINQ em C#](#).

O exemplo a seguir executa uma consulta LINQ em uma [List](#) genérica. A consulta LINQ retorna uma coleção diferente que contém os resultados.

```

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`.

```

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.Write(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 [IComparable<T>](#) ou [IEnumerable](#).

Embora você possa definir uma coleção personalizada, geralmente é melhor usar as coleções incluídas no .NET,

que são descritas em Tipos de Coleções anteriormente neste artigo.

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.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.WriteLine(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 `yield return` instrução que está dentro de um `for` loop. No método `ListEvenNumbers`, cada iteração do corpo da instrução `foreach` cria uma chamada ao método iterador, que avança para a próxima instrução `yield return`.

```

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 em coleções
- Quando usar coleções genéricas
- Instruções de iteração

Covariância e contravariância (C#)

21/01/2022 • 3 minutes to read

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.

```
// 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.

```
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.

```

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 posteriores, o C# dá suporte à covariância e 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.

```

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
Variância em interfaces genéricas (C#)	Discute covariância e contravariância em interfaces genéricas e fornece uma lista de interfaces genéricas variadas no .NET.
Criando interfaces genéricas variáveis (C#)	Mostra como criar interfaces variantes personalizadas.
Usando variação em interfaces para Coleções Genéricas (C#)	Mostra como o suporte de covariância e contravariância nas interfaces <code>IEnumerable<T></code> e <code>IComparable<T></code> pode ajudar na reutilização do código.
Variação em delegados (C#)	Discute covariância e contravariância em delegados genéricos e não genéricos e fornece uma lista de delegados genéricos de variantes no .NET.
Usando variação em delegados (C#)	Mostra como usar o suporte de covariância e contravariância em delegados não genéricos para corresponder às assinaturas de método com tipos delegados.
Usando variação para delegados genéricos Func e Action (C#)	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#)

21/01/2022 • 2 minutes to read

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.

Começando com .NET Framework 4, 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.

```
IEnumerable<String> strings = new List<String>();  
IEnumerable<Object> objects = strings;
```

Em versões anteriores do .NET Framework, esse código causa um erro de compilação em C# e, se `Option Strict` estiver, em 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.

```

// 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.

```

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.

```

// 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#)

21/01/2022 • 5 minutes to read

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*.

NOTE

O .NET Framework 4 introduziu o suporte à variação para diversas interfaces genéricas existentes. Para obter a lista das interfaces variantes no .NET, consulte [variação 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.

IMPORTANT

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.

```
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 \(C#\)](#).

```
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.

```
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.

```
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.

```
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.

```
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.

```
// 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.

```
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.

```
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.

```
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 produz um erro nesse caso, mas não é especificado qual implementação de interface será escolhida em tempo de execução. Essa ambiguidade pode levar a bugs sutis em seu código. Considere o exemplo de código a seguir.

```

// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because I Enumerable<out T> is covariant.
class Pets : I Enumerable<Cat>, I Enumerable<Dog>
{
    I Enumerator<Cat> I Enumerable<Cat>.Get Enumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    I Enumerator I Enumerable.Get Enumerator()
    {
        // Some code.
        return null;
    }

    I Enumerator<Dog> I Enumerable<Dog>.Get Enumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        I Enumerable<Animal> pets = new Pets();
        pets.Get Enumerator();
    }
}

```

Neste exemplo, não está especificado como o método `pets.Get Enumerator()` 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#)

21/01/2022 • 2 minutes to read

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 ver uma lista de interfaces variantes no .NET, consulte [Variação 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`.

```
// 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`.

```
// 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#)

21/01/2022 • 6 minutes to read

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.

```
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.

```
// 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.

```
// 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.

```
// 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`.

```
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.

```
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.

```
public delegate void DContravariant<in A>(A a);
```

IMPORTANT

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.

```
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.

```
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.

```
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.

```
// 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#)

21/01/2022 • 2 minutes to read

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

```
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:

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- Um representante [MouseEventHandler](#) que define a assinatura do evento [Button.MouseClick](#). Sua assinatura é:

```
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

```
// 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#)

21/01/2022 • 2 minutes to read

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`.

```
// 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`.

```
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](#)

Árvores de expressão (C#)

21/01/2022 • 5 minutes to read

Árvores de expressão representam código em uma estrutura de dados de árvore, onde cada nó é, por exemplo, uma expressão, uma chamada de método ou uma operação binária como `x < y`.

Você pode compilar e executar código representado por árvores de expressão. Isso permite a modificação dinâmica de código executável, a execução de consultas LINQ em vários bancos de dados e a criação de consultas dinâmicas. Para obter mais informações sobre árvores de expressão no LINQ, consulte [Como usar árvores de expressão para criar consultas dinâmicas \(C#\)](#).

As árvores de expressão também são usadas no DLR (Dynamic Language Runtime) para fornecer interoperabilidade entre linguagens dinâmicas e .NET e para permitir que os compiladores emitam árvores de expressão em vez da MSIL (Microsoft Intermediate Language). Para obter mais informações sobre o DLR, consulte [Visão geral do Dynamic Language Runtime](#).

Você pode fazer o compilador C# ou do Visual Basic criar uma árvore de expressões para você com base em uma expressão lambda anônima ou criar árvores de expressão manualmente usando o namespace [System.Linq.Expressions](#).

Criando árvores de expressão de expressões Lambda

Quando uma expressão lambda é atribuída a uma variável do tipo [Expression<TDelegate>](#), o compilador emite código para criar uma árvore de expressão que representa a expressão lambda.

Os compiladores do C# podem gerar árvores de expressão apenas por meio de expressões lambda (ou lambdas de linha única). Ele não é possível analisar instruções lambdas (ou lambdas de várias linhas). Para obter mais informações sobre expressões lambda no C#, consulte [Expressões lambda](#).

Os exemplos de código a seguir demonstram como fazer os compiladores do C# criarem uma árvore de expressão que representa a expressão lambda `num => num < 5`.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Criando árvores de expressão usando a API

Para criar árvores de expressão usando a API, use a classe [Expression](#). Essa classe contém métodos de fábrica estáticos para criar nós de árvore de expressão de tipos específicos, por exemplo, [ParameterExpression](#), que representa uma variável ou parâmetro, ou [MethodCallExpression](#), que representa uma chamada de método. [ParameterExpression](#), [MethodCallExpression](#) e os outros tipos específicos de expressão também são definidos no namespace [System.Linq.Expressions](#). Esses tipos derivam do tipo abstrato [Expression](#).

O exemplo de código a seguir demonstra como criar uma árvore de expressão que representa a expressão lambda `num => num < 5` usando a API.

```

// Add the following using directive to your code file:
// using System.Linq.Expressions;

// 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 });

```

No .NET Framework 4 ou posterior, 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.

```

// 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.

Analisando árvores de expressão

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.

```
// 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
```

Imutabilidade das árvores de expressão

Árvores de expressão devem ser imutáveis. Isso significa que se você deseja modificar uma árvore de expressão, deverá criar uma nova árvore de expressão, copiando a existente e substituindo seus nós. Você pode usar um visitantes de árvore expressão para percorrer a árvore de expressão existente. Para obter mais informações, [consulte Como modificar árvores de expressão \(C#\)](#).

Compilando árvores de expressão

O tipo `Expression<TDelegate>` fornece o método `Compile` que compila o código representado por uma árvore de expressão para um delegado executável.

O exemplo de código a seguir demonstra como compilar uma árvore de expressão e executar o código resultante.

```
// Creating an expression tree.  
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.
```

Para obter mais informações, [consulte Como executar árvores de expressão \(C#\)](#).

Confira também

- [System.Linq.Expressions](#)
- [Como executar árvores de expressão \(C#\)](#)
- [Como modificar árvores de expressão \(C#\)](#)
- [Expressões lambda](#)
- [Visão geral do Dynamic Language Runtime](#)
- [Conceitos de programação \(C#\)](#)

Como executar árvores de expressão (C#)

21/01/2022 • 2 minutes to read

Este tópico mostra como executar uma árvore de expressão. 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.

NOTE

Se o tipo de delegado não for conhecido, ou seja, a expressão lambda for do tipo [LambdaExpression](#) e não [Expression<TDelegate>](#), você deverá chamar o método [DynamicInvoke](#) sobre o delegado em vez de invocá-la diretamente.

Se uma árvore de expressão não representa uma expressão lambda, você pode criar uma nova expressão lambda que tenha a árvore de expressão original como corpo, chamando o método [Lambda<TDelegate>\(Expression, IEnumerable<ParameterExpression>\)](#). Em seguida, você pode executar a expressão lambda como descrito anteriormente nesta seção.

Exemplo

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.

```
// 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
```

Compilando o código

- Inclua o namespace `System.Linq.Expressions`.

Confira também

- Árvores de expressão (C#)
- Como modificar árvores de expressão (C#)

Como modificar árvores de expressão (C#)

21/01/2022 • 2 minutes to read

Este tópico mostra como modificar uma árvore de expressão. As árvores de expressão são imutáveis, o que significa que elas não podem ser diretamente modificadas. Para alterar uma árvore de expressão, você deve criar uma cópia de uma árvore de expressão existente e, ao criar a cópia, faça as alterações necessárias. Você pode usar a classe [ExpressionVisitor](#) para percorrer uma árvore de expressão existente e copiar cada nó que ela visitar.

Para modificar uma árvore de expressão

1. Crie um novo projeto de Aplicativo de Console.
2. Adicione uma diretiva `using` ao arquivo para o namespace `System.Linq.Expressions`.
3. Adicione a classe `AndAlsoModifier` ao seu projeto.

```
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. Para fazer isso, a classe substitui o método [VisitBinary](#) do tipo base, pois as expressões `AND` condicionais 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 que é 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 que são semelhantes às árvores de expressão que são passadas, mas os nós têm suas subárvores substituídas pelas árvores de expressão que são produzidas recursivamente pelo visitante.

4. Adicione uma diretiva `using` ao arquivo para o namespace `System.Linq.Expressions`.
5. Adicione código para o método `Main` no arquivo Program.cs para criar uma árvore de expressão e passá-la ao método que a modificará.

```
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.

6. Compile e execute o aplicativo.

Confira também

- [Como executar árvores de expressão \(C#\)](#)
- [Árvores de expressão \(C#\)](#)

Consultando com base no estado de tempo de execução (C#)

21/01/2022 • 9 minutes to read

Considere o código que define um [IQueryable](#) ou [um <T> IQueryable](#) em relação a uma fonte de dados:

```
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, a mesma consulta exata será executada. Isso geralmente não é muito útil, pois talvez você queira que seu código execute consultas diferentes dependendo das condições em tempo de execução. Este artigo descreve como você pode executar uma consulta diferente com base no estado de tempo de execução.

<T> Árvores de expressão IQueryable/IQueryable

Fundamentalmente, um [IQueryable](#) tem dois componentes:

- [Expression](#)—uma representação independente de linguagem e fonte de fontes 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 permanecerá o mesmo; a árvore de expressão da consulta será diferente da consulta para consulta.

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 uma nova [IQueryable](#).

As seções a seguir descrevem técnicas específicas para consultar de forma diferente em resposta ao estado de tempo de execução:

- Usar o estado de tempo de execução de dentro da árvore de expressão
- Chamar métodos LINQ adicionais
- Varie a árvore de expressão passada para os métodos LINQ
- Construa uma árvore de expressão de [expressão <TDelegate>](#) usando os métodos de fábrica em[Expression](#)
- Adicionar nós de chamada de método à [IQueryable](#) árvore de expressão de uma
- Construir cadeias de caracteres e usar a [biblioteca dinâmica do LINQ](#)

Usar o estado de tempo de execução de dentro da árvore de expressão

Supondo que o provedor LINQ ofereça suporte a ele, a maneira mais simples de consultar dinamicamente é referenciar o estado de tempo de execução diretamente na consulta por meio de uma variável fechada, como `length` no exemplo de código a seguir:

```
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 somente porque o valor de `length` foi alterado.

Chamar métodos LINQ adicionais

Em geral, os [métodos internos do LINQ em `Queryable`](#) executam duas etapas:

- Encapsula a árvore de expressão atual em uma [`MethodCallExpression`](#) representando a chamada de método.
- Passe a árvore de expressão encapsulada de volta para o provedor, para retornar um valor por meio do método do provedor [`IQueryProvider.Execute`](#); ou para retornar um objeto de consulta traduzido por meio do [`IQueryProvider.CreateQuery`](#) método.

Você pode substituir a consulta original pelo resultado de um método de retorno [`IQueryable <T>`](#) para obter uma nova consulta. Isso pode ser feito condicionalmente com base no estado do tempo de execução, como no exemplo a seguir:

```
// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}
```

Varie a árvore de expressão passada para os métodos LINQ

Você pode passar expressões diferentes para os métodos LINQ, dependendo do estado de tempo de execução:

```

// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("", "") => x => true,
    (_, "") => x => x.StartsWith(startsWith),
    ("", _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);

```

Você também pode querer compor as várias subexpressões usando uma biblioteca de terceiros, como o [PredicateBuilder](#) da [LinqKit](#):

```

// 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 e consultas de expressão usando métodos de fábrica

Em todos os exemplos até esse ponto, nós conhecidas o tipo de elemento no momento da compilação —

`string` — e, 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](#) e, portanto, personalizar a expressão em tempo de execução para um tipo de elemento específico.

Construindo uma expressão <TDelegate>

Quando você constrói uma expressão para passar em um dos métodos LINQ, na verdade está construindo uma instância de `Expression <TDelegate>`, em que `TDelegate` é um tipo delegado, como `Func<string, bool>`, `Action` ou um tipo de delegado personalizado.

`Expressão <TDelegate>` de herda de `LambdaExpression`, que representa uma expressão lambda completa como a seguinte:

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

Um [LambdaExpression](#) tem dois componentes:

- Uma lista de parâmetros — `(string x)` — representada pela [Parameters](#) propriedade.
- Um corpo — `x.StartsWith("a")` — representado pela [Body](#) propriedade.

As etapas básicas na construção de uma [expressão <TDelegate>](#) são as seguintes:

- Defina [ParameterExpression](#) objetos para cada um dos parâmetros (se houver) na expressão lambda, usando o [Parameter](#) método de fábrica.

```
ParameterExpression x = Parameter(typeof(string), "x");
```

- Construa o corpo do seu [LambdaExpression](#), usando os [ParameterExpression](#) (`s`) que você definiu e os métodos de fábrica em [Expression](#). Por exemplo, uma expressão que representa `x.StartsWith("a")` poderia ser construída da seguinte maneira:

```
Expression body = Call(  
    x,  
    typeof(string).GetMethod("StartsWith", new[] { typeof(string) })!,  
    Constant("a")  
);
```

- Empacote os parâmetros e o corpo em uma [expressão <TDelegate>](#) de tipo de tempo de compilação, usando a [Lambda](#) sobrecarga de método de fábrica apropriada:

```
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body, x);
```

As seções a seguir descrevem um cenário no qual você pode querer construir uma [expressão <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:

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);  
record Car(string Model, int Year);
```

Para qualquer um desses tipos de entidade, você deseja filtrar e retornar somente as entidades que têm um determinado texto dentro de um de seus `string` campos. Para `Person` o, você desejaría Pesquisar `FirstName` as `LastName` Propriedades e:

```
string term = /* ... */;  
var personsQry = new List<Person>()  
    .AsQueryable()  
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

Mas `car`, para o, você desejaría Pesquisar apenas a `Model` Propriedade:

```

string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));

```

Embora você possa escrever uma função personalizada para `IQueryable<Person>` o e outra para `IQueryable<Car>` o, a função a seguir adiciona essa filtragem a qualquer consulta existente, independentemente do tipo de elemento específico.

Exemplo

```

// 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(                  // .Contains(...)
                    Property(          // .PropertyName
                        prm,           // x
                        prp
                    ),
                    containsMethod,
                    Constant(term)   // "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 `TextFilter` função usa e retorna um `IQueryable <T>` (e não apenas um `IQueryable`), você pode adicionar elementos de consulta de tipo de tempo de compilação adicionais após o filtro de texto.

```
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 `IQueryable` árvore de expressão do

Se você tiver um `IQueryable` em vez de `um <T> IQueryable`, não poderá chamar diretamente os métodos LINQ genéricos. Uma alternativa é criar a árvore de expressão interna como acima e usar a reflexão para invocar o método LINQ apropriado ao passar na árvore de expressão.

Você também pode duplicar a funcionalidade do método LINQ encapsulando a árvore inteira em um `MethodCallExpression` que representa uma chamada para o método LINQ:

```
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 de tempo de compilação `T`, portanto, você usará a `Lambda` sobrecarga que não requer informações de tipo de tempo de compilação e que produz um `LambdaExpression` em vez de uma `expressão <TDelegate>`.

A biblioteca do LINQ dinâmico

A construção de árvores de expressão usando métodos de fábrica é relativamente complexa; é mais fácil compor cadeias de caracteres. A `biblioteca LINQ dinâmica` expõe um conjunto de métodos de extensão no `IQueryable` correspondente aos métodos padrão do LINQ 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 a partir da cadeia de caracteres e pode retornar a tradução resultante `IQueryable`.

Por exemplo, o exemplo anterior poderia ser reescrito da seguinte maneira:

```
// 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);
}
```

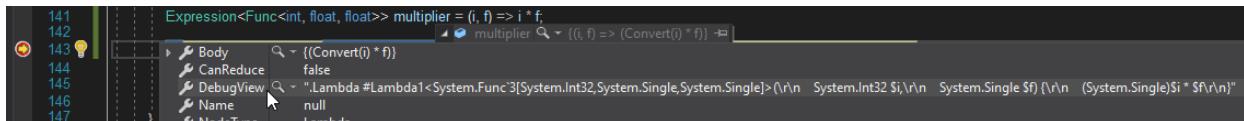
Confira também

- [Árvores de expressão \(C#\)](#)
- [Como executar árvores de expressão \(C#\)](#)
- [Especificar dinamicamente filtros de predicado em tempo de execução](#)

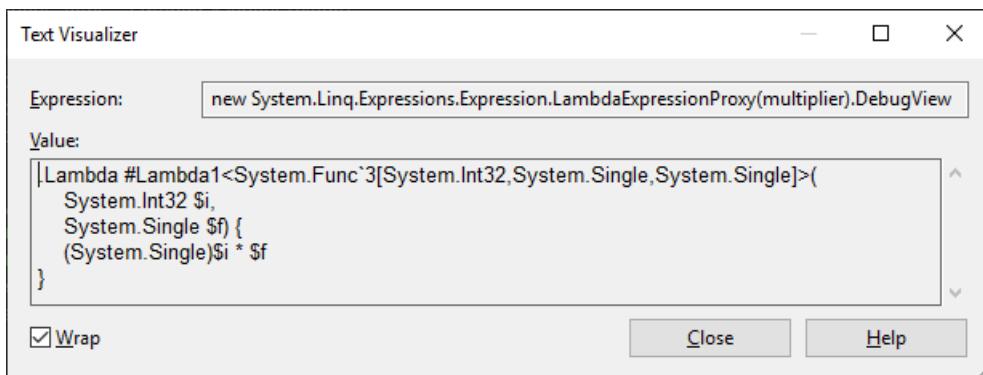
Depurando árvores de expressão no Visual Studio (C#)

21/01/2022 • 2 minutes to read

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](#). (Observe que `DebugView` está disponível apenas no modo de depuração.)

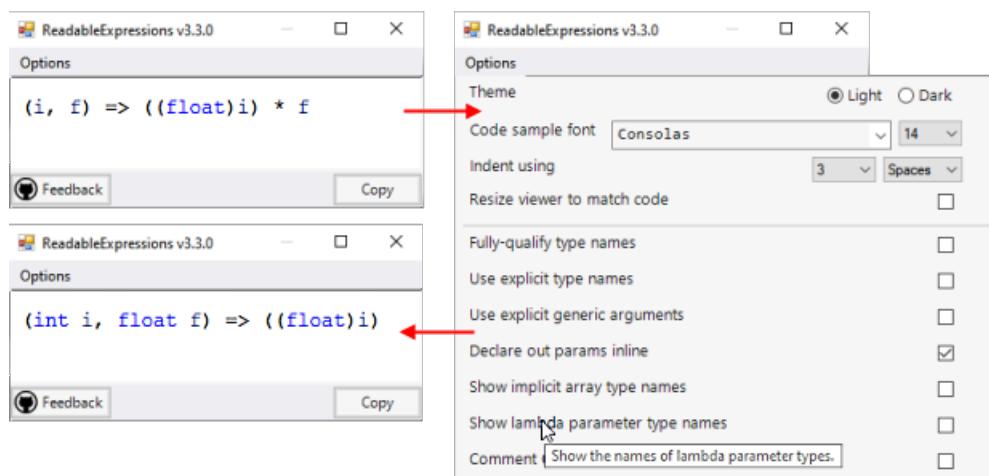


Uma vez que `DebugView` é uma cadeia de caracteres, você pode usar o [Visualizador de Texto interno](#) para exibi-lo em várias linhas, selecionando [Visualizador de Texto](#) do ícone de lupa ao lado do rótulo `DebugView`.

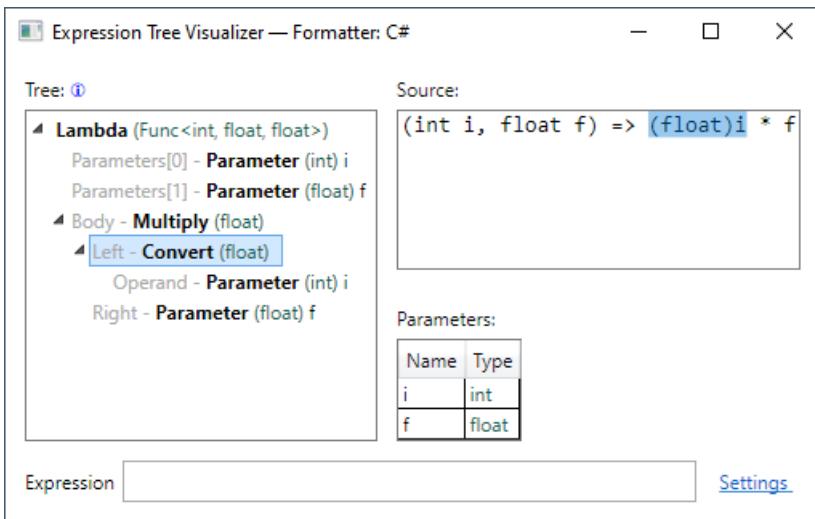


Como alternativa, você pode instalar e usar [um visualizador personalizado](#) para árvores de expressão, como:

- as [expressões legíveis](#) (licença MIT, disponível no [Visual Studio Marketplace](#)), renderizam a árvore de expressão como código passíveis C#, com várias opções de renderização:



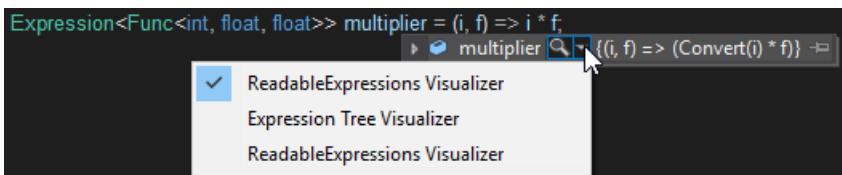
- O [Visualizador de árvore de expressões](#) (licença MIT) fornece uma exibição de árvore da árvore de expressão e seus nós individuais:



Para abrir um visualizador para uma árvore de expressão

1. Clique no 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 de visualizadores disponíveis:



2. Clique no visualizador que você deseja usar.

Confira também

- [Árvores de expressão \(C#\)](#)
- [Depurando no Visual Studio](#)
- [Criar visualizadores personalizados](#)
- [DebugView Sintaxe](#)

Sintaxe de DebugView

21/01/2022 • 2 minutes to read

A propriedade **DebugView** (disponível somente quando 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, contendo o **DebugView**.

ParameterExpression

ParameterExpression os nomes de variáveis são exibidos com o símbolo `$` no início.

Se um parâmetro não tiver um nome, será atribuído um nome gerado automaticamente, como `$var1` ou `$var2`.

Exemplos

```
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.

TYPE	PALAVRA-CHAVE	SUFIXO
<code>System.UInt32</code>	<code>uint</code>	U
<code>System.Int64</code>	<code>longo</code>	L
<code>System.UInt64</code>	<code>ULONG</code>	UL
<code>System.Double</code>	<code>double</code>	D
<code>System.Single</code>	<code>float</code>	F
<code>System.Decimal</code>	<code>decimal</code>	M

Exemplos

```

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.

Exemplos

```

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 um nome, será atribuído um nome gerado automaticamente, como

`#Lambda1` OU `#Lambda2`.

Exemplos

```

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 um nome, será atribuído um nome gerado automaticamente, como `#Label11` ou `#Label12`.

Exemplos

```
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 `#+`.

Exemplos

```
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
*/
```

Iteradores (C#)

21/01/2022 • 7 minutes to read

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.

```
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.

NOTE

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`.

```

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`.

```

static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.WriteLine(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`.

```

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 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`.

```

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)

```

```

    for each value number in theStack.DocumentStack)
    {
        Console.WriteLine("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.WriteLine("{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 IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerable 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];
        }
    }
}

```

```

        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.

Uma conversão implícita deve existir a partir do tipo de expressão na `yield return` instrução para o argumento de tipo para o `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 tem um significado especial apenas quando ela é usada antes de uma `return` ou `break` palavra-chave ou.

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 `EnumerateFiles` método, que implementa 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](#)
- [proporcionar](#)
- [Usar foreach com matrizes](#)
- [Genéricos](#)

LINQ (Consulta Integrada à Linguagem) (C#)

21/01/2022 • 4 minutes to read

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. Você escreve consultas em coleções fortemente tipadas de objetos usando palavras-chave da linguagem e operadores familiares. A família de tecnologias LINQ fornece uma experiência de consulta consistente para objetos (LINQ to Objects), bancos de dados relacionais (LINQ to SQL) e XML (LINQ to XML).

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 usa os mesmos padrões básicos de expressão de consulta para consultar e transformar dados em bancos de dados SQL, conjuntos de dados ADO.NET, documentos XML e fluxos e coleções .NET.

É possível escrever consultas do LINQ em C# para bancos de dados do SQL Server, documentos XML, conjuntos de dados ADO.NET e qualquer coleção de objetos que dá suporte a [IEnumerable](#) ou à interface genérica [IEnumerable<T>](#). O suporte ao LINQ também é fornecido por terceiros para muitos serviços Web e outras implementações de banco de dados.

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`.

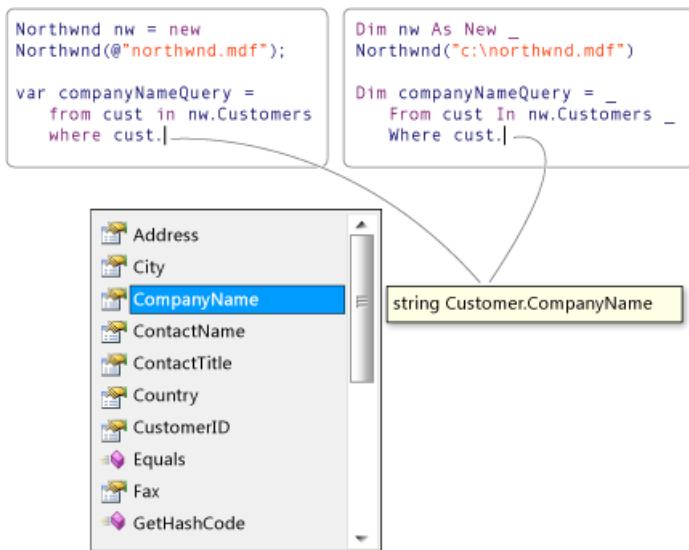
```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<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
}
```

A seguinte ilustração do Visual Studio mostra uma consulta LINQ parcialmente concluída em um banco de dados do SQL Server no C# e no Visual Basic, com verificação de tipo completa e suporte ao IntelliSense:



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.
- Expressões de consulta são fáceis de entender porque 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 [Relações 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 às 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#)

21/01/2022 • 6 minutes to read

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 oferecendo um modelo consistente para trabalhar com dados em vários tipos de fontes de dados e formatos. 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, SQL bancos de dados, ADO.NET datasets, coleções .net e qualquer outro formato para o qual um provedor de LINQ esteja disponível.

Três Partes de uma Operação de Consulta

Todas as operações de consulta LINQ consistem em três ações distintas:

1. Obter a fonte de dados.
2. Criar a consulta.
3. Executar a consulta.

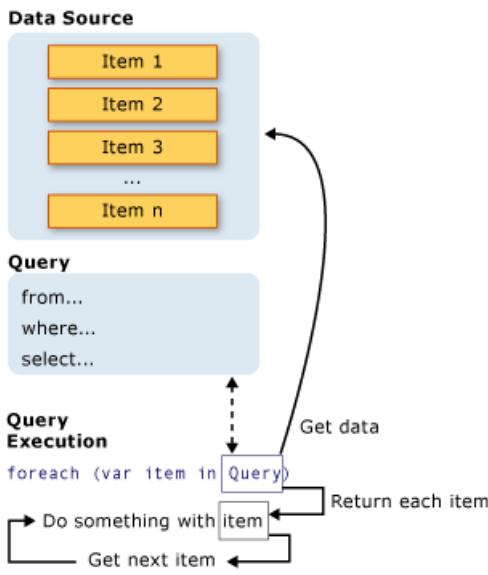
O exemplo a seguir mostra como as três partes de uma operação de consulta são expressas em código-fonte. O exemplo usa uma matriz de inteiros como uma fonte de dados para sua conveniência. No entanto, os mesmos conceitos também se aplicam a outras fontes de dados. Faremos referência a este exemplo todo o restante deste tópico.

```
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 recuperou 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. Esse fato significa que ele pode ser consultado com 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 requer nenhuma modificação ou tratamento especial para servir como uma fonte de dados LINQ. Se os dados de origem ainda não estiverem na memória como um tipo passível de consulta, o provedor LINQ deverá representá-lo como tal. por exemplo, LINQ to XML carrega um documento XML em um tipo passível de consulta `XElement`:

```
// Create a data source from an XML document.
// using System.Xml.Linq;
 XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

com LINQ to SQL o, você primeiro cria um mapeamento relacional de objeto no tempo de design manualmente ou usando as [ferramentas de 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>`.

```
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 vários provedores de LINQ. No entanto, a regra básica é muito simples: uma fonte de dados LINQ é qualquer objeto que dê suporte à `IEnumerable<T>` interface genérica ou uma interface que herde dela.

NOTE

Tipos como [ArrayList](#) o que dão suporte à interface não genérica [IEnumerable](#) também podem ser usados como uma fonte de dados LINQ. Para obter mais informações, consulte [como consultar uma 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 SQL, perceberá que a ordenação das cláusulas é revertida da ordem em SQL.) A `from` cláusula Especifica a fonte de dados, a `where` cláusula aplica o filtro e a `select` cláusula Especifica o tipo dos elementos retornados. Essas e as outras cláusulas de consulta são discutidas em detalhes na seção [linguagem de consulta integrada \(LINQ\)](#). Por enquanto, o ponto importante é que, no LINQ, a variável de consulta não executa nenhuma ação 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 \(C#\)](#).

NOTE

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:

```
// Query execution.  
foreach (int num in numQuery)  
{  
    Console.WriteLine("{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:

```
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`.

```
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 `ToDictionary`, 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 \(LINQ\)](#)

LINQ e tipos genéricos (C#)

21/01/2022 • 2 minutes to read

As consultas LINQ são baseadas em tipos genéricos, que foram introduzidos na versão 2.0 do .NET Framework. 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 digitados como `IEnumerable<T>` ou 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`.

```
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:

```
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](#).

Confira também

- [Genéricos](#)

Operações de consulta LINQ básica (C#)

21/01/2022 • 5 minutes to read

Este tópico fornece uma breve introdução às expressões de consulta LINQ e alguns dos tipos típicos de operações que você executa em uma consulta. Informações mais detalhadas estão nos tópicos a seguir:

[Expressões de Consulta LINQ](#)

[Visão geral de operadores de consulta padrão \(C#\)](#)

[Passo a passo: escrevendo consultas em C#](#)

NOTE

Se já estiver familiarizado com uma linguagem de consulta como SQL ou XQuery, você poderá ignorar a maior parte deste tópico. Leia sobre a "`from` cláusula" na próxima seção para saber mais sobre a ordem das cláusulas em expressões de consulta LINQ.

Obtendo 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 uma consulta LINQ, a `from` cláusula vem primeiro para apresentar a fonte de dados (`customers`) e a *variável de intervalo* (`cust`).

```
//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](#).

NOTE

Para fontes de dados não genéricas, como `ArrayList`, a variável de intervalo deve ser tipada explicitamente. Para obter mais informações, consulte [como consultar uma 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.

```
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:

```
where cust.City == "London" && cust.Name == "Devon"
```

Para retornar clientes de Londres ou Paris, você escreveria o seguinte código:

```
where cust.City == "London" || cust.City == "Paris"
```

Para obter mais informações, consulte a [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.

```
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.

```
// 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:

```
// 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 `join`, a cláusula sempre funciona em coleções de objetos em vez de tabelas de banco de dados diretamente.

```
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 com a `join` frequência que faz no SQL, porque as chaves estrangeiras no LINQ são representadas no modelo de 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:

```
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](#).

Confira também

- Expressões de Consulta LINQ
- Passo a passo: escrevendo consultas em C#
- Palavras-chave de consulta (LINQ)
- Tipos anônimos

Transformações de dados com LINQ (C#)

21/01/2022 • 6 minutes to read

A consulta de Language-Integrated (LINQ) não se refere apenas à recuperação de dados. Também é uma ferramenta poderosa para transformação de dados. Usando 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 seja 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:

```
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:

```

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.

```
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`:

```
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, SQL bancos de dados, ADO.NET conjuntos e fluxos XML ou documentos. O exemplo a seguir transforma objetos de uma estrutura de dados na memória em elementos XML.

```
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:

```
<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 da sequência de saída será formatada usando [interpolação de cadeia de caracteres](#).

Uma cadeia de caracteres interpolada terá uma `$` na frente das aspas de abertura da cadeia de caracteres, e as

operações poderão ser executadas dentro das chaves colocadas dentro da cadeia de caracteres interpolada.

Depois que essas operações forem executadas, os resultados serão concatenados.

NOTE

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](#).

```

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#)

21/01/2022 • 2 minutes to read

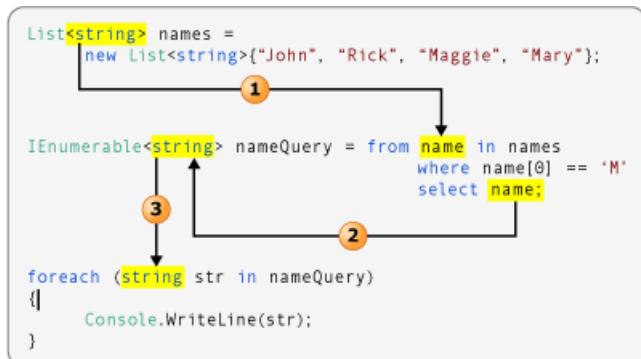
Para escrever consultas com eficiência, você precisa entender como os tipos de variáveis em uma operação de consulta completa se relacionam entre si. Se você entender essas relações, compreenderá mais facilmente os exemplos de LINQ e exemplos de código na documentação. Além disso, você compreenderá o que ocorre nos bastidores quando variáveis são tipadas de forma implícita usando `var`.

As operações de consulta LINQ são fortemente digitados 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

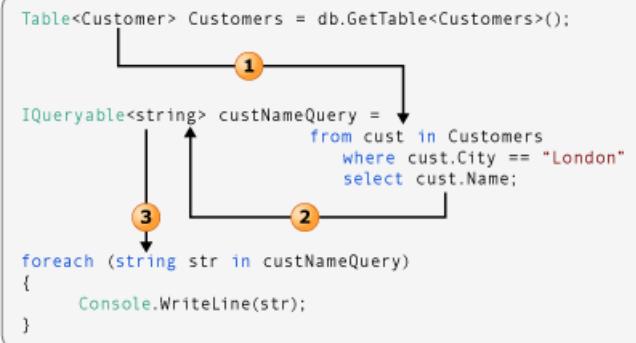
A ilustração a seguir mostra LINQ to Objects uma operação de consulta que não executa nenhuma transformação nos dados. A fonte contém uma sequência de cadeias de caracteres e a saída da consulta também é uma sequência de cadeias de caracteres.



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.

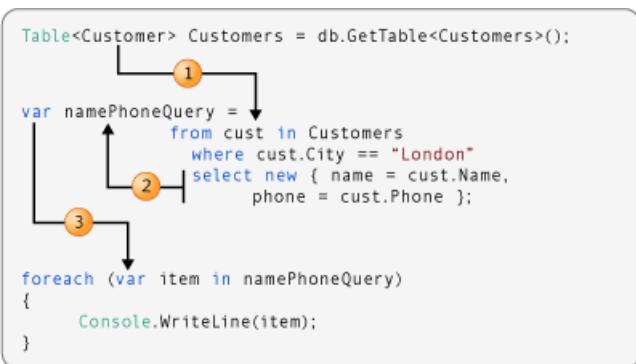
Consultas que transformam os dados de origem

A ilustração a seguir mostra uma operação de consulta de LINQ to SQL que executa transformações 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.



- O argumento de tipo da fonte de dados determina o tipo da variável de intervalo.
- 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`.
- Como `custNameQuery` é uma sequência de cadeias de caracteres, a variável de iteração do loop `foreach` também deve ser um `string`.

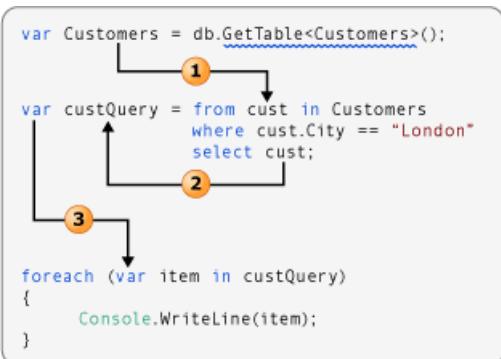
A ilustração a seguir mostra uma transformação um pouco mais complexa. A instrução `select` retorna um tipo anônimo que captura apenas dois membros do objeto `Customer` original.



- O argumento de tipo da fonte de dados sempre é o tipo da variável de intervalo na consulta.
- Como a instrução `select` produz um tipo anônimo, a variável de consulta deve ser tipada implicitamente usando `var`.
- 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](#).

Sintaxe de consulta e sintaxe de método em LINQ (C#)

21/01/2022 • 4 minutes to read

A maioria das consultas na documentação do LINQ (consulta integrada à linguagem introdutória) é escrita usando a 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 estar familiarizado com a forma de usar a sintaxe do 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*.

```

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 + " ");
        }

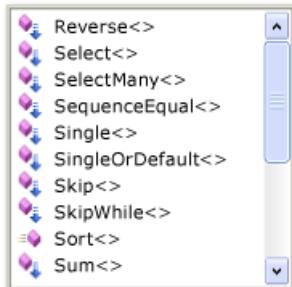
        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
Output:
6 8 10 12
6 8 10 12
*/

```

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 o LINQ, tudo o que você realmente precisa saber sobre métodos de extensão é como colocá-los em escopo em seu aplicativo usando as `using` diretivas 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 de 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 as lambdas extensivamente. No entanto, determinadas consultas só podem ser expressadas em sintaxe de método e algumas delas requerem expressões lambda. Depois de se familiarizar mais com as lambdas, você descobrirá que elas são uma ferramenta poderosa e flexível em 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.

funcionalidades do C# que dão suporte a LINQ

21/01/2022 • 3 minutes to read

A seção a seguir apresenta os novos constructos de linguagem introduzidos no C# 3.0. Embora esses novos recursos sejam todos usados em um grau com consultas LINQ, eles não são limitados a LINQ e podem ser usados em qualquer contexto em que você as achar 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 a implementação de um provedor LINQ dos métodos de extensão do operador de consulta padrão. 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.

```
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:

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

As variáveis declaradas como `var` são fortemente digitados como 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:

```
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:

```
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:

```
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:

```
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 em linha que usa o operador para separar parâmetros de entrada do corpo da função e pode ser convertida no tempo de compilação para um delegado ou uma árvore `=>` de expressão. Na programação LINQ, você encontra expressões lambda ao fazer chamadas de método direto para os 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)

21/01/2022 • 10 minutes to read

Essas instruções passo a passo demonstram os recursos de linguagem C# que são usados para gravar expressões de consulta LINQ.

Criar um Projeto C#

NOTE

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.

```

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 , consulte Variáveis locais de tipo `var` 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.

```

// 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> .
```

- Após você ter adicionado esse código, compile e execute o aplicativo para ver os resultados na janela **Console**.

```
// 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

- 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.

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Para obter mais informações, consulte [a cláusula where](#).

Modificar a Consulta

Para ordenar os resultados

- 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`:

```
orderby student.Last ascending
```

- 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.

```
orderby student.Scores[0] descending
```

- Altere a cadeia de caracteres de formato `WriteLine` para que você possa ver as pontuações:

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
```

Para obter mais informações, consulte [Cláusula orderby](#).

Para agrupar os resultados

- 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.

```
// studentQuery2 is an I Enumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

- 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:

```
// studentGroup is a I Grouping<char, Student>
foreach (var 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
```

- Execute o aplicativo e exiba os resultados na janela `Console`.

Para obter mais informações, consulte [Cláusula group](#).

Para deixar as variáveis tipadas implicitamente

- A codificação explícita `I Enumerables` 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.

```

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:

```

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.

```

// 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

- 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.

```

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

- 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`.

```

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`:

```

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#)

21/01/2022 • 3 minutes to read

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 LINQ: um que opera em objetos do tipo `IEnumerable`, outro que opera em `IQueryable` objetos do tipo `IQueryable`. 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 sintaxe de método estático ou sintaxe de método de instância.

Além disso, vários métodos de operador de consulta padrão operam em tipos diferentes daqueles baseados em `IEnumerable` ou `IQueryable`. 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 fortemente digitada de objetos. 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`, o objeto enumerável retornado captura os argumentos que foram passados `IEnumerable` 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 `IQueryable` estendem 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` 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.

```

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 usados com mais frequência têm sintaxe de palavra-chave de linguagem C# e Visual Basic dedicada que permite 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](#).

Seções relacionadas

Os links a seguir levam você a artigos que fornecem informações adicionais sobre os vários operadores de consulta padrão com base na funcionalidade.

[Classificando dados \(C#\)](#)

[Operações de conjunto \(C#\)](#)

[Filtrando dados \(C#\)](#)

[Operações de quantificador \(C#\)](#)

[Operações de projeção \(C#\)](#)

[Particionando dados \(C#\)](#)

[Operações de junção \(C#\)](#)

[Agrupando dados \(C#\)](#)

[Operações de geração \(C#\)](#)

[Operações de Igualdade \(C#\)](#)

[Operações de elemento \(C#\)](#)

[Convertendo Tipos de Dados \(C#\)](#)

[Operações de concatenação \(C#\)](#)

[Operações de agregação \(C#\)](#)

Confira também

- [Enumerable](#)
- [Queryable](#)
- [Introdução a consultas LINQ \(C#\)](#)
- [Sintaxe de expressão de consulta para operadores de consulta padrão \(C#\)](#)
- [Classificação de operadores de consulta padrão pelo modo de execução \(C#\)](#)
- [Métodos de Extensão](#)

Sintaxe de expressão de consulta para operadores de consulta padrão (C#)

21/01/2022 • 2 minutes to read

Alguns dos operadores de consulta padrão mais usados têm uma sintaxe de palavra-chave de linguagem C# dedicada que possibilita que eles sejam chamados como parte de uma *expressão de consulta*. Uma expressão de consulta é uma maneira diferente e mais legível de expressar uma consulta do que seu equivalente *baseado em método*. As cláusulas de expressão de consulta são convertidas em chamadas para os métodos de consulta em tempo de compilação.

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 C#
Cast	Use uma variável de intervalo de tipo explícito, por exemplo: <code>from int i in numbers</code> (Para obter mais informações, consulte Cláusula from .)
GroupBy	<code>group ... by</code> -OU- <code>group ... by ... into ...</code> (Para obter mais informações, consulte Group Clause .)
GroupJoin<TOuter,TInner,TKey,TResult> (IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	<code>join ... in ... on ... equals ... into ...</code> (Para obter mais informações, consulte Cláusula join .)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	<code>join ... in ... on ... equals ...</code> (Para obter mais informações, consulte Cláusula join .)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby</code> (Para obter mais informações, consulte cláusula OrderBy .)
OrderByDescending<TSource,TKey> (IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... descending</code> (Para obter mais informações, consulte cláusula OrderBy .)
Select	<code>select</code> (Para obter mais informações, consulte Cláusula select .)

MÉTODO	SINTAXE DE EXPRESSÃO DE CONSULTA C#
SelectMany	Várias cláusulas <code>from</code> . (Para obter mais informações, consulte Cláusula from.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... , ...</code> (Para obter mais informações, consulte cláusula OrderBy.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... , ... descending</code> (Para obter mais informações, consulte cláusula OrderBy.)
Where	<code>where</code> (Para obter mais informações, consulte Cláusula where.)

Confira também

- [Enumerable](#)
- [Queryable](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Classificação de operadores de consulta padrão pelo modo de execução \(C#\)](#)

Classificação de operadores de consulta padrão pelo modo de execução (C#)

21/01/2022 • 3 minutes to read

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.

Modos de execução

Imediata

A execução imediata significa que a fonte de dados é lida e a operação é realizada no ponto do código em que a consulta é declarada. Todos os operadores de consulta padrão que retornam um resultado único e não enumerável são executados imediatamente.

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.

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.

NOTE

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.

OPERADOR DE CONSULTA PADRÃO	TIPO DE RETORNO	EXECUÇÃO IMEDIATA	EXECUÇÃO ADIADA DE STREAMING	EXECUÇÃO ADIADA DE NÃO STREAMING
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		

OPERADOR DE CONSULTA PADRÃO	TIPO DE RETORNO	EXECUÇÃO IMEDIATA	EXECUÇÃO ADIADA DE STREAMING	EXECUÇÃO ADIADA DE NÃO STREAMING
LastOrDefault	TSource	X		
LongCount	Int64	X		
Max	Valor numérico único, TSource ou TResult	X		
Min	Valor numérico único, TSource ou TResult	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	TSource	X		
SingleOrDefault	TSource	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Valor numérico único	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	Matriz de TSource	X		

OPERADOR DE CONSULTA PADRÃO	TIPO DE RETORNO	EXECUÇÃO IMEDIATA	EXECUÇÃO ADIADA DE STREAMING	EXECUÇÃO ADIADA DE NÃO STREAMING
ToDictionary	Dictionary< TKey, TValue >	X		
ToList	IList< T >	X		
ToLookup	ILookup< TKey, TElement >	X		
Union	IEnumerable< T >		X	
Where	IEnumerable< T >		X	

Confira também

- [Enumerable](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Sintaxe de expressão de consulta para operadores de consulta padrão \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

Classificando dados (C#)

21/01/2022 • 2 minutes to read

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>	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Classifica valores em ordem decrescente.	<code>orderby ... descending</code>	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Executa uma classificação secundária em ordem crescente.	<code>orderby ..., ...</code>	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Executa uma classificação secundária em ordem decrescente.	<code>orderby ..., ... descending</code>	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	Inverte a ordem dos elementos em uma coleção.	Não aplicável.	Enumerable.Reverse Queryable.Reverse

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.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<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.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<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.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<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.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<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 \(LINQ\) \(C#\)](#)

Definir operações (C#)

21/01/2022 • 7 minutes to read

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 C#	MAIS INFORMAÇÕES
DISTINCT ou DistinctBy	Remove os valores duplicados de uma coleção.	Não aplicável.	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
Except ou ExceptBy	Retorna a diferença de conjunto, que significa os elementos de uma coleção que não aparecem em uma segunda coleção.	Não aplicável.	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
Intersect ou IntersectBy	Retorna a interseção de conjunto, o que significa os elementos que aparecem em cada uma das duas coleções.	Não aplicável.	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
Union ou UnionBy	Retorna a união de conjunto, o que significa os elementos únicos que aparecem em qualquer uma das duas coleções.	Não aplicável.	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

Exemplos

Alguns dos exemplos a seguir dependem de um `record` tipo que representa os planetas em nosso sistema solar.

```

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 posicional, que exige `Name` argumentos, `Type` e `OrderFromSun` para instanciá-lo. Há várias `static readonly` instâncias do planeta no `Planet` tipo. Essas são definições baseadas em conveniência para planetas bem conhecidos. O `Type` membro identifica o tipo de planeta.

```

namespace SolarSystem;

enum PlanetType
{
    Rock,
    Ice,
    Gas,
    Liquid
};

```

Distinct e DistinctBy

O exemplo a seguir descreve o comportamento do `Enumerable.Distinct` método 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`

```

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
 */

```

O `DistinctBy` é uma abordagem alternativa para `Distinct` isso `keySelector`. O `keySelector` é usado como o discriminador comparativa do tipo de origem. Considere a seguinte matriz do planeta:

```

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 em seu `PlanetType` e o primeiro planeta de cada tipo é exibido:

```

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 C# anterior:

- A `Planet` matriz é filtrada de forma distinta 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.



```

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 `ExceptBy` método é uma abordagem alternativa `Except` que leva duas sequências de tipos possivelmente heterogêneos e um `keySelector`. O `keySelector` é o mesmo tipo do tipo da segunda coleção e é usado como o discriminador comparativa do tipo de origem. Considere as seguintes matrizes do planeta:

```

Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Jupiter
};

Planet[] morePlanets =
{
    Planet.Mercury,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

```

Para localizar os planetas na primeira coleção que não estão na segunda coleção, você pode projetar os nomes do planeta como a `second` coleção e fornecer o mesmo `keySelector`:

```

// 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 C# anterior:

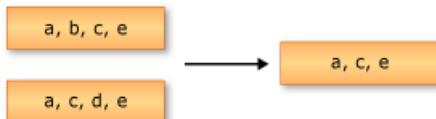
- O `keySelector` é definido como uma `static` função local que discrimina um nome de planeta.
- A primeira matriz planeta é filtrada para planetas que não são encontrados na segunda matriz planeta, com

base no nome.

- A `planet` instância resultante é gravada 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.



```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IQueryable<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 `IntersectBy` método é uma abordagem alternativa `Intersect` que leva duas sequências de tipos possivelmente heterogêneos e um `keySelector`. O `keySelector` é usado como o discriminador comparativo do tipo da segunda coleção. Considere as seguintes matrizes do planeta:

```
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; um representa os cinco primeiros planetas do sol e o segundo representa os últimos cinco planetas do sol. Como o `Planet` tipo é um tipo posicional `record`, você pode usar sua semântica de comparação de valor na forma de `keySelector`:

```

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 C# anterior:

- As duas `Planet` matrizes são interseccionadas pela 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 Union em duas sequências de cadeias de caracteres. A sequência retornada contém os elementos exclusivos das duas sequências de entrada.



```

string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IQueryable<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 `UnionBy` método é uma abordagem alternativa `Union` que leva duas sequências do mesmo tipo e um `keySelector`. O `keySelector` é usado como o discriminador comparativo do tipo de origem. Considere as seguintes matrizes do planeta:

```

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 o `keySelector` :

```

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 C# anterior:

- As duas `Planet` matrizes são reunidas usando sua `record` semântica de comparação de valor.
- As instâncias resultantes `planet` são gravadas no console.

Confira também

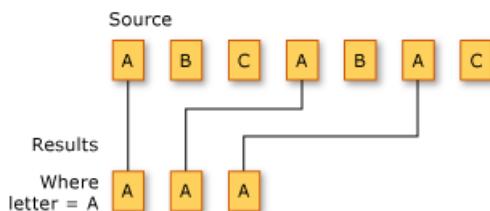
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como combinar e comparar coleções de cadeias de caracteres \(LINQ\) \(C#\)](#)
- [Como localizar a diferença de conjunto entre duas listas \(LINQ\) \(C#\)](#)

Filtrando dados (C#)

21/01/2022 • 2 minutes to read

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.	Enumerable.OfType Queryable.OfType
Where	Seleciona valores com base em uma função de predicado.	where	Enumerable.Where Queryable.Where

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.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<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

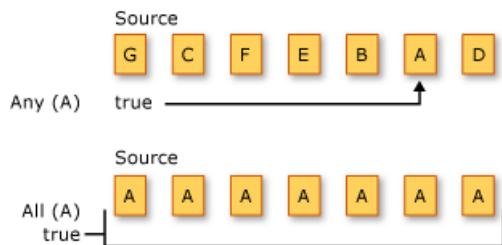
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Cláusula where](#)
- [Especificar dinamicamente filtros de predicado em tempo de executar](#)
- [Como consultar metadados de um assembly com REFLEXÃO \(LINQ\) \(C#\)](#)
- [Como consultar arquivos com um atributo ou nome especificado \(C#\)](#)
- [Como classificar ou filtrar dados de texto por qualquer palavra ou campo \(LINQ\) \(C#\)](#)

Operações de quantificador (C#)

21/01/2022 • 3 minutes to read

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 um ou mais dos elementos são o caractere 'A' e o resultado é `true`. A segunda operação pergunta se todos os elementos são o caractere 'A' e o resultado é `false`.



Os métodos de operador de consulta padrão que realizam operações de quantificador estão listados na seção a seguir.

Métodos

NOME DO MÉTODO	DESCRÍÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
Todos	Determina se todos os elementos em uma sequência satisfazem uma condição.	Não aplicável.	Enumerable.All Queryable.All
Qualquer	Determina se um ou mais dos elementos em uma sequência satisfazem uma condição.	Não aplicável.	Enumerable.Any Queryable.Any
Contém	Determina se uma sequência contém um elemento especificado.	Não aplicável.	Enumerable.Contains Queryable.Contains

Exemplos de sintaxe de expressão de consulta

Todos

O exemplo a seguir usa o `All` para verificar se todas as cadeias de caracteres são de um comprimento específico.

```

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 o `Any` para verificar se todas as cadeias de caracteres são iniciadas com 'o'.

```

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 o `Contains` para verificar se uma matriz tem um elemento específico.

```
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 dinamicamente filtros de predicado em tempo de execução](#)
- [Como consultar frases que contêm um conjunto especificado de palavras \(LINQ\) \(C#\)](#)

Operações de projeção (C#)

21/01/2022 • 6 minutes to read

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	DESCRÍÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
Selecionar	Projeta valores com base em uma função de transformação.	<code>select</code>	Enumerable.Select Queryable.Select
SelectMany	Projeta sequências de valores baseados em uma função de transformação e os mescla em uma sequência.	Use várias cláusulas <code>from</code>	Enumerable.SelectMany Queryable.SelectMany
Zip	Produz uma sequência de tuplas com elementos de 2-3 sequências especificadas.	Não aplicável.	Enumerable.Zip Queryable.Zip

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.

```
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 `from` cláusulas para projetar cada palavra de cada uma delas em uma lista de

cadeias de caracteres.

```
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 `Zip` operador de projeção. Todos os `Zip` métodos funcionam em sequências de dois ou mais tipos, que são mais heterogêneos. As duas primeiras sobrecargas retornam tuplas, com o tipo positional correspondente das sequências especificadas.

Considere as seguintes coleções:

```
// 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 juntas, use o `Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)` operador:

```
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'
```

IMPORTANT

A sequência resultante de uma operação zip nunca é mais demorada do que a seqüência mais curta. As `numbers` coleções e diferem no comprimento, e a sequência resultante omite o último elemento da `numbers` coleção, pois não tem nada a ser compactado com.

A segunda sobrecarga aceita uma `third` sequência. Vamos criar outra coleção, a saber `emoji` :

```
// A string array with 8 elements.  
IEnumerable<string> emoji = new[]  
{  
    "❑", "❑", "❑", "❑", "❑", "❑", "✓", "❑"  
};
```

Para projetar essas sequências juntas, use o `Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)` operador:

```
foreach ((int number, char letter, string em) in numbers.Zip(letters, emoji))  
{  
    Console.WriteLine(  
        $"Number: {number} is zipped with letter: '{letter}' and emoji: {em}");  
}  
// 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 `Zip` método projeta uma tupla, mas desta vez com três elementos.

A terceira sobrecarga aceita um `Func<TFirst, TSecond, TResult>` argumento que atua como um seletor de resultados. Considerando que os dois tipos das sequências estão sendo compactados, você pode projetar uma nova sequência resultante.

```
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 `Zip` sobrecarga anterior, a função especificada é aplicada aos elementos correspondentes `numbers` e `letter`, produzindo uma sequência dos `string` resultados.

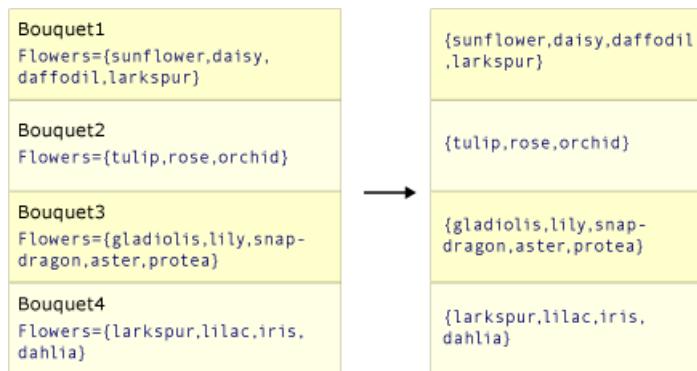
`Select` **VEZ** `SelectMany`

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

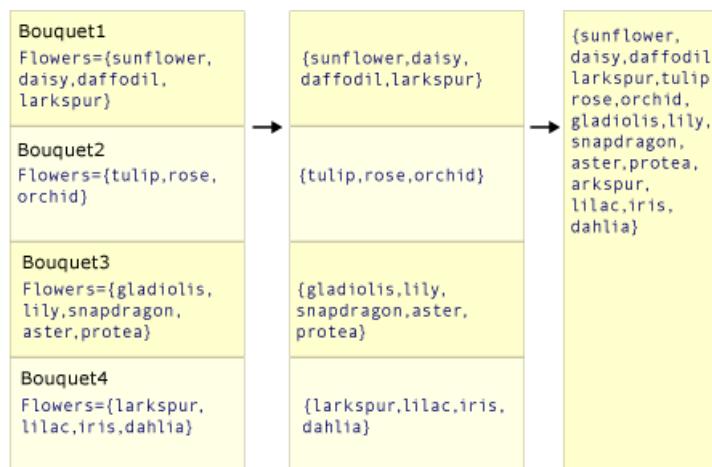
que tem o mesmo número de elementos que a coleção de origem. Por outro lado, `SelectMany` produz um único resultado geral que contém subcoleções concatenadas de cada valor de origem. A função de transformação passada como um argumento para `SelectMany` deve retornar uma sequência enumerável de valores 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 usando os dois primeiros itens de cada lista de nomes de flor 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.

```

class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new()
    {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "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" } }
    };
}

```

```

        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

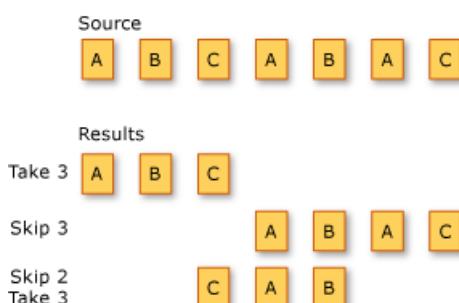
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [cláusula SELECT](#)
- [Como preencher coleções de objetos de várias fontes \(LINQ\) \(C#\)](#)
- [Como dividir um arquivo em vários arquivos usando grupos \(LINQ\) \(C#\)](#)

Particionando dados (C#)

21/01/2022 • 2 minutes to read

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	DESCRÍÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
Ignorar	Ignora elementos até uma posição especificada na sequência.	Não aplicável.	Enumerable.Skip Queryable.Skip
SkipWhile	Ignora elementos com base em uma função de predicado até que um elemento não satisfaça a condição.	Não aplicável.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Aceita elementos até uma posição especificada na sequência.	Não aplicável.	Enumerable.Take Queryable.Take
TakeWhile	Aceita elementos com base em uma função de predicado até que um elemento não satisfaça a condição.	Não aplicável.	Enumerable.TakeWhile Queryable.TakeWhile
Chunk	Divide os elementos de uma sequência em partes de um tamanho máximo especificado.	Não aplicável.	Enumerable.Chunk Queryable.Chunk

Exemplo

O `Chunk` operador é usado para dividir os elementos de uma sequência com base em um determinado `size`.

```
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 C# anterior:

- Depende de `Enumerable.Range(Int32, Int32)` para gerar uma sequência de números.
- Aplica o `Chunk` operador, dividindo a sequência em partes com um tamanho máximo de três.

Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)

Join Operações (C#)

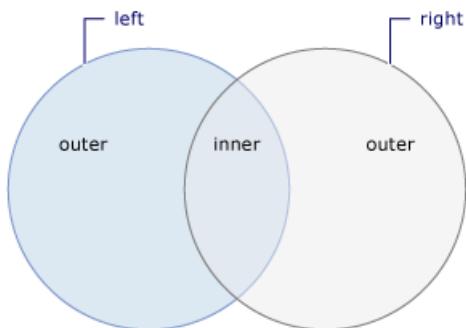
21/01/2022 • 4 minutes to read

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.

Joining é uma operação importante em consultas que direcionam 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, o [Join](#) implementa uma junção interna, um tipo de junção em que somente os objetos que têm uma correspondência no outro conjunto 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	DESCRÍÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
Join	Joins duas sequências baseadas em funções de seletor de chave e extraem pares de valores.	<pre>join ... in ... on ... equals ...</pre>	Enumerable.Join Queryable.Join

NOME DO MÉTODO	DESCRIÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
GroupJoin	Joins duas sequências baseadas em funções de seletor de chave e agrupa as correspondências resultantes para cada elemento.	<code>join ... in ... on ... equals ... into ...</code>	Enumerable.GroupJoin Queryable.GroupJoin

Exemplos de sintaxe de expressão de consulta

Join

O exemplo a seguir usa a `join ... in ... on ... equals ...` cláusula para unir duas sequências com base no valor específico:

```

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 `join ... in ... on ... equals ... into ...` cláusula para unir duas sequências com base em um valor específico e agrupa as correspondências resultantes para cada elemento:

```

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 consultas de s e produtos cruzados](#)
- [cláusula JOIN](#)

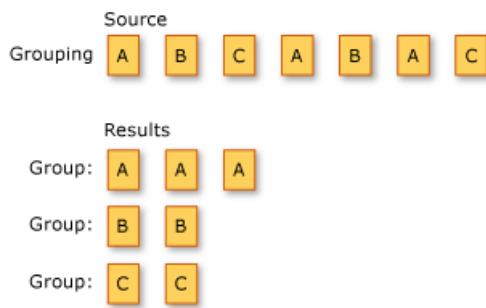
- [Join usando chaves compostas](#)
- [Como unir conteúdo de arquivos diferentes \(LINQ\) \(C#\)](#)
- [Ordenar os resultados de uma cláusula join](#)
- [Executar operações de junção personalizadas](#)
- [Executar junções agrupadas](#)
- [Executar junções internas](#)
- [Executar junções externas esquerdas](#)
- [Como preencher coleções de objetos de várias fontes \(LINQ\) \(C#\)](#)

Agrupando dados (C#)

21/01/2022 • 2 minutes to read

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	DESCRÍÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
GroupBy	Agrupa elementos que compartilham um atributo comum. Cada grupo é representado por um objeto <code>IGrouping< TKey, TElement ></code> .	<code>group ... by</code> -ou- <code>group ... by ... into ...</code>	Enumerable.GroupBy Queryable.GroupBy
ToLookup	Insere os elementos em um <code>Lookup< TKey, TElement ></code> (um dicionário one-to-many) com base em uma função de seletor de chave.	Não aplicável.	Enumerable.ToLookup

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.

```

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 muitos arquivos usando grupos \(LINQ\) \(C#\)](#)

Operações de geração (C#)

21/01/2022 • 2 minutes to read

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.	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
Vazio	Retorna uma coleção vazia.	Não aplicável.	Enumerable.Empty
Intervalo	Gera uma coleção que contém uma sequência de números.	Não aplicável.	Enumerable.Range
Repetir	Gera uma coleção que contém um valor repetido.	Não aplicável.	Enumerable.Repeat

Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)

Operações de Igualdade (C#)

21/01/2022 • 2 minutes to read

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.	Enumerable.SequenceEqual Queryable.SequenceEqual

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#)

21/01/2022 • 2 minutes to read

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 C#	MAIS INFORMAÇÕES
ElementAt	Retorna o elemento em um índice especificado em uma coleção.	Não aplicável.	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	Retorna o elemento em um índice especificado em uma coleção ou um valor padrão se o índice estiver fora do intervalo.	Não aplicável.	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
Primeiro	Retorna o primeiro elemento de uma coleção ou o primeiro elemento que satisfaz uma condição.	Não aplicável.	Enumerable.First Queryable.First
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.	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
Último	Retorna o último elemento de uma coleção ou o último elemento que satisfaz uma condição.	Não aplicável.	Enumerable.Last Queryable.Last
LastOrDefault	Retorna o último elemento de uma coleção ou o último elemento que satisfaz uma condição. Retorna um valor padrão se esse elemento não existir.	Não aplicável.	Enumerable.LastOrDefault Queryable.LastOrDefault
Single	Retorna o único elemento de uma coleção ou o único elemento que satisfaz uma condição. Gera um InvalidOperationException se não houver elemento ou mais de um elemento a ser retornado.	Não aplicável.	Enumerable.Single Queryable.Single

NOME DO MÉTODO	DESCRIÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
SingleOrDefault	Retorna o único elemento de uma coleção ou o único elemento que satisfaz uma condição. Retorna um valor padrão se não houver elemento a ser retornado. Gera um InvalidOperationException se houver mais de um elemento a ser retornado.	Não aplicável.	Enumerable.SingleOrDefault Queryable.SingleOrDefault

Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como consultar o maior arquivo ou arquivos em uma árvore de diretório \(LINQ\) \(C#\)](#)

Convertendo Tipos de Dados (C#)

21/01/2022 • 2 minutes to read

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	DESCRÍÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
AsEnumerable	Retorna a entrada digitada como IEnumerable<T> .	Não aplicável.	Enumerable.AsEnumerable
AsQueryable	Converte um IQueryable (genérico) em um IQueryable (genérico).	Não aplicável.	Queryable.AsQueryable
Conversão	Converte os elementos de uma coleção em um tipo especificado.	Use uma variável de intervalo de tipo explícito. Por exemplo: <code>from string str in words</code>	Enumerable.Cast Queryable.Cast
OfType	Filtre valores, dependendo da capacidade de serem convertidos em um tipo especificado.	Não aplicável.	Enumerable.OfType Queryable.OfType
ToArray	Converte uma coleção em uma matriz. Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToArray

NOME DO MÉTODO	DESCRIÇÃO	SINTAXE DE EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
ToDictionary	Coloca os elementos em um Dictionary< TKey, TValue > com base em uma função de seletor de chave. Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToDictionary
ToList	Converte uma coleção em um List< T > . Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToList
ToLookup	Coloca os elementos em um Lookup< TKey, TElement > (um dicionário one-to-many) com base em uma função de seletor de chave. Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToLookup

Exemplo de sintaxe de expressão de consulta

O exemplo de código a seguir usa uma variável de intervalo digitada explicitamente para converter um tipo em um subtipo antes de acessar um membro que está disponível somente no subtipo.

```

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

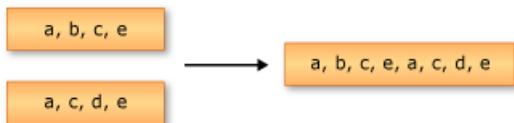
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [cláusula from](#)
- [Expressões de Consulta LINQ](#)
- [Como consultar uma ArrayList com LINQ \(C#\)](#)

Operações de concatenação (C#)

21/01/2022 • 2 minutes to read

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.	Enumerable.Concat Queryable.Concat

Confira também

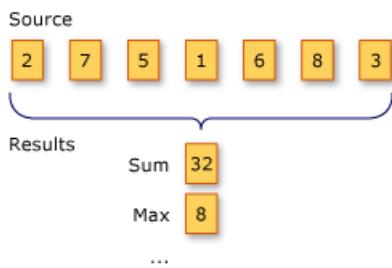
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como combinar e comparar coleções de cadeias de caracteres \(LINQ\) \(C#\)](#)

Operações de agregação (C#)

21/01/2022 • 2 minutes to read

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 DA EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
Agregado	Executa uma operação de agregação personalizada nos valores de uma coleção.	Não aplicável.	Enumerable.Aggregate Queryable.Aggregate
Média	Calcula o valor médio de uma coleção de valores.	Não aplicável.	Enumerable.Average Queryable.Average
Contagem	Conta os elementos em uma coleção e, opcionalmente, apenas os elementos que satisfazem a uma função de predicado.	Não aplicável.	Enumerable.Count Queryable.Count
LongCount	Conta os elementos em uma coleção grande e, opcionalmente, apenas os elementos que satisfazem a uma função de predicado.	Não aplicável.	Enumerable.LongCount Queryable.LongCount
Max ou MaxBy	Determina o valor máximo em uma coleção.	Não aplicável.	Enumerable.Max Enumerable.MaxBy Queryable.Max Queryable.MaxBy

NOME DO MÉTODO	DESCRIÇÃO	SINTAXE DA EXPRESSÃO DE CONSULTA C#	MAIS INFORMAÇÕES
Min ou MinBy	Retorna o valor mínimo em uma coleção.	Não aplicável.	Enumerable.Min Enumerable.MinBy Queryable.Min Queryable.MinBy
Somar	Calcula a soma dos valores em uma coleção.	Não aplicável.	Enumerable.Sum Queryable.Sum

Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como calcular 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 \(LINQ\) \(C#\)](#)

LINQ to Objects (C#)

21/01/2022 • 2 minutes to read

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 retornada por uma API do .NET.

Basicamente, o LINQ to Objects representa uma nova abordagem às coleções. Na forma antiga, você precisava escrever loops `foreach` complexos que especificavam como recuperar dados de uma coleção. 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 nos dados, mais benefícios você obterá usando LINQ em vez de técnicas de iteração tradicionais.

O objetivo desta seção é demonstrar a abordagem LINQ com alguns exemplos selecionados. Ele não se destina a ser exaustivo.

Nesta seção

[LINQ e cadeias de caracteres \(C#\)](#)

Explica como a LINQ pode ser usada para consultar e transformar cadeias de caracteres e coleções de cadeias de caracteres. Também inclui links para artigos que demonstram esses princípios.

[LINQ e reflexão \(C#\)](#)

Contém um link para um exemplo que demonstra como a LINQ usa a reflexão.

[LINQ e diretórios de arquivos \(C#\)](#)

Explica como a LINQ pode ser usada para interagir com sistemas de arquivos. Também inclui links para artigos que demonstram esses conceitos.

[Como consultar uma ArrayList com LINQ \(C#\)](#)

Demonstra como consultar um `ArrayList` no C#.

[Como adicionar métodos personalizados para consultas LINQ \(C#\)](#)

Explica como estender o conjunto de métodos que você pode usar para consultas LINQ, adicionando os métodos de extensão à interface `IEnumerable<T>`.

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

Fornece links para artigos que explicam LINQ e fornecem exemplos de código que executam consultas.

LINQ e cadeias de caracteres (C#)

21/01/2022 • 2 minutes to read

A LINQ pode ser usada para consultar e transformar as cadeias de caracteres e coleções de cadeias de caracteres. Ele pode ser especialmente útil com os dados semiestruturados em arquivos de texto. Consultas LINQ podem ser combinadas com expressões regulares e funções de cadeia de caracteres tradicionais. Por exemplo, você pode usar o método [String.Split](#) ou [Regex.Split](#) para criar uma matriz de cadeias de caracteres que você pode consultar ou modificar usando o LINQ. Você pode usar o método [Regex.IsMatch](#) na cláusula `where` de uma consulta LINQ. E você pode usar o LINQ para consultar ou modificar os resultados de [MatchCollection](#) retornados por uma expressão regular.

Você também pode usar as técnicas descritas nessa seção para transformar dados de texto semiestruturados em XML. Para obter mais informações, [consulte Como gerar XML de arquivos CSV](#).

Os exemplos nesta seção se enquadram em duas categorias:

Consultando um bloco de texto

Consultar, analisar e modificar os blocos de texto dividindo-os em uma matriz de cadeias de caracteres menores consultáveis usando o método [String.Split](#) ou o método [Regex.Split](#). Você pode dividir o texto de origem em palavras, frases, parágrafos, páginas ou quaisquer outros critérios e, em seguida, executar divisões adicionais se elas forem necessárias em sua consulta.

- [Como contar ocorrências de uma palavra em uma cadeia de caracteres \(LINQ\) \(C#\)](#)
Mostra como usar a LINQ para consultas simples em texto.
- [Como consultar frases que contêm um conjunto especificado de palavras \(LINQ\) \(C#\)](#)
Mostra como dividir os arquivos de texto em limites arbitrários e como executar consultas em cada parte.
- [Como consultar caracteres em uma cadeia de caracteres \(LINQ\) \(C#\)](#)
Demonstra que uma cadeia de caracteres é de um tipo passível de consulta.
- [Como combinar consultas LINQ com expressões regulares \(C#\)](#)
Mostra como usar expressões regulares em consultas LINQ para correspondência de padrões complexos em resultados de consulta filtrados.

Consultando dados semiestruturados em formato de texto

Muitos tipos diferentes de arquivos de texto consistem em uma série de linhas, geralmente com formatação semelhante, como arquivos delimitados por tabulação ou vírgula ou linhas de comprimento fixo. Depois de ler um arquivo de texto na memória, você pode usar a LINQ para consultar e/ou modificar as linhas. As consultas LINQ também simplificam a tarefa de combinar dados de várias fontes.

- [Como encontrar a diferença de conjunto entre duas listas \(LINQ\) \(C#\)](#)
Mostra como localizar todas as cadeias de caracteres que estão presentes em uma lista, mas não na outra.
- [Como classificar ou filtrar dados de texto por qualquer palavra ou campo \(LINQ\) \(C#\)](#)
Mostra como classificar linhas de texto com base em qualquer palavra ou campo.

- [Como reordenar os campos de um arquivo delimitado \(LINQ\) \(C#\)](#)

Mostra como reordenar campos em uma linha em um arquivo .csv.

- [Como combinar e comparar coleções de cadeias de caracteres \(LINQ\) \(C#\)](#)

Mostra como combinar listas de cadeias de caracteres de várias maneiras.

- [Como popular coleções de objetos de várias fontes \(LINQ\) \(C#\)](#)

Mostra como criar coleções de objetos usando vários arquivos de texto como fontes de dados.

- [Como unir conteúdo de arquivos diferentes \(LINQ\) \(C#\)](#)

Mostra como combinar cadeias de caracteres em duas listas em uma única cadeia de caracteres usando uma chave correspondente.

- [Como dividir um arquivo em muitos arquivos usando grupos \(LINQ\) \(C#\)](#)

Mostra como criar novos arquivos usando um único arquivo como uma fonte de dados.

- [Como calcular valores de coluna em um arquivo de texto CSV \(LINQ\) \(C#\)](#)

Mostra como executar cálculos matemáticos em dados de texto em arquivos .csv.

Confira também

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

- [Como gerar XML de arquivos CSV](#)

Como contar ocorrências de uma palavra em uma cadeia de caracteres (LINQ) (C#)

21/01/2022 • 2 minutes to read

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

```
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 ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
                        where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
                        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.

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)

Como consultar frases que contêm um conjunto especificado de palavras (LINQ) (C#)

21/01/2022 • 2 minutes to read

Este exemplo mostra como localizar frases em um arquivo de texto que contenham 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

```
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`.

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)

Como consultar caracteres em uma cadeia de caracteres (LINQ) (C#)

21/01/2022 • 2 minutes to read

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.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        I Enumerable<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 '-'
        I Enumerable<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

- [LINQ e cadeias de caracteres \(C#\)](#)
- [Como combinar consultas LINQ com expressões regulares \(C#\)](#)

Como combinar consultas LINQ com expressões regulares (C#)

21/01/2022 • 2 minutes to read

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

```
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`.

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como localizar a diferença de conjunto entre duas listas (LINQ) (C#)

21/01/2022 • 2 minutes to read

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. Copie names1.txt e names2.txt para a pasta da solução, conforme mostrado em [como combinar e comparar as coleções de cadeias de caracteres \(LINQ\) \(C#\)](#).

Exemplo

```
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 in debug mode.
        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`.

Confira também

- LINQ e cadeias de caracteres (C#)

Como classificar ou filtrar dados de texto por qualquer palavra ou campo (LINQ) (C#)

21/01/2022 • 2 minutes to read

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 em 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-o em sua pasta de solução.

Exemplo

```

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`.

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)

Como reordenar os campos de um arquivo delimitado (LINQ) (C#)

21/01/2022 • 2 minutes to read

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 representam o "sobrenome", o "nome" e a "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.

```
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

```

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`.

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)
- [Como gerar XML de arquivos CSV \(C#\)](#)

Como combinar e comparar coleções de cadeias de caracteres (LINQ) (C#)

21/01/2022 • 2 minutes to read

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:

```
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.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Exemplo

```
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.
```

```

    // 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, Nkeng
   Myrcha, Jacek
   Noriega, Fabricio
   Potra, Cristina
   Toyoshima, Tim
   28 total names in list

```

```
26 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`.

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como preencher coleções de objetos de várias fontes (LINQ) (C#)

21/01/2022 • 4 minutes to read

Este exemplo mostra como mesclar dados de diferentes fontes em uma sequência de novos tipos.

NOTE

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 `ToDictionary` na consulta de banco de dados e, em seguida, realize a junção na coleção retornada.

Para criar o arquivo de dados

Copie os arquivos de `names.csv` e `scores.csv` na 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.

```
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:

```

// 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 arquivos usando grupos (LINQ) (C#)

21/01/2022 • 2 minutes to read

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:

```
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.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Exemplo

```
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`.

Confira também

- LINQ e cadeias de caracteres (C#)
- LINQ e diretórios de arquivos (C#)

Como unir conteúdo de arquivos diferentes (LINQ) (C#)

21/01/2022 • 2 minutes to read

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.

```
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.

```
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

```
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
*/

```

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)

- LINQ e diretórios de arquivos (C#)

Como computar valores de coluna em um arquivo de texto CSV (LINQ) (C#)

21/01/2022 • 3 minutes to read

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.

```
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

```
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.
    I Enumerable<I Enumerable<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 ofToList.
    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
Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/

```

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`.

Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como consultar metadados de um assembly com reflexão (LINQ) (C#)

21/01/2022 • 2 minutes to read

As APIs de reflexão do .NET podem ser usadas para examinar os metadados em um assembly .NET e criar coleções de tipos, membros de tipo, parâmetros e assim por diante 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 localizará os nomes de todos os métodos no assembly que retornam tipos enumeráveis como matrizes.

Exemplo

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        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);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

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.

Confira também

- [LINQ to Objects \(C#\)](#)

Como consultar metadados de um assembly com reflexão (LINQ) (C#)

21/01/2022 • 2 minutes to read

As APIs de reflexão do .NET podem ser usadas para examinar os metadados em um assembly .NET e criar coleções de tipos, membros de tipo, parâmetros e assim por diante 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 localizará os nomes de todos os métodos no assembly que retornam tipos enumeráveis como matrizes.

Exemplo

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        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);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

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.

Confira também

- [LINQ to Objects \(C#\)](#)

LINQ e diretórios de arquivos (C#)

21/01/2022 • 2 minutes to read

Muitas operações do sistema de arquivos são basicamente consultas e, portanto, são bem adequadas 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 diretórios \(LINQ\) \(C#\)](#)

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#)

21/01/2022 • 2 minutes to read

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

```
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`.

Confira também

- [LINQ to Objects \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como agrupar arquivos por extensão (LINQ) (C#)

21/01/2022 • 2 minutes to read

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.

```
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 group in groupByExtList)
        {
            // Print the extension.
            Console.WriteLine(group.Key);

            // Print the files in the group.
            foreach (var file in group)
            {
                // Print the file name.
                Console.WriteLine(file.Name);
            }
        }
    }
}
```

```

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`.

Confira também

- [LINQ to Objects \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como consultar o número total de bytes em um conjunto de pastas (LINQ) (C#)

21/01/2022 • 2 minutes to read

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](#).

```

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 de 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`.

Confira também

- [LINQ to Objects \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como comparar o conteúdo de duas pastas (LINQ) (C#)

21/01/2022 • 2 minutes to read

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.

NOTE

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

```
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.

Confira também

- [LINQ to Objects \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como consultar o maior arquivo ou arquivos em uma árvore de diretório (LINQ) (C#)

21/01/2022 • 3 minutes to read

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](#).

```
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)
            select file)
            .OrderByDescending(file => file.Length)
            .First();
    }
}
```

```

(from file in fileList
    let len = GetFileLength(file)
    where len > 0
    orderby len descending
    select file)
.Fist();

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](#).

Confira também

- [LINQ to Objects \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como consultar arquivos duplicados em uma árvore de diretório (LINQ) (C#)

21/01/2022 • 3 minutes to read

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

```
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`.

Confira também

- [LINQ to Objects \(C#\)](#)
- [LINQ e diretórios de arquivos \(C#\)](#)

Como consultar o conteúdo de arquivos de texto em uma pasta (LINQ) (C#)

21/01/2022 • 2 minutes to read

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

```

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`.

Confira também

- [LINQ e diretórios de arquivos \(C#\)](#)

- LINQ to Objects (C#)

Como consultar uma ArrayList com LINQ (C#)

21/01/2022 • 2 minutes to read

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:

```
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.

```

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
*/

```

Confira também

- [LINQ to Objects \(C#\)](#)

Como adicionar métodos personalizados para consultas LINQ (C#)

21/01/2022 • 5 minutes to read

Estenda o conjunto de métodos que você usa para consultas LINQ adicionando métodos de extensão à `IEnumerable<T>` interface. Por exemplo, além da média padrão ou das operações máximas, você cria um método de agregação personalizado para calcular um único valor de uma sequência de valores. Você também cria um método que funciona como um filtro personalizado ou uma transformação de dados específica 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`.

```
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`.

```
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`.

```
// 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:

```
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`.

```
// 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, você terá que 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.

```
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.

```
// 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 chama os outros métodos da interface `IEnumerable<T>`, conforme mostrado no código a seguir:

```
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 to ADO.NET (página do portal)

21/01/2022 • 2 minutes to read

LINQ to ADO.NET permite que você consulte qualquer objeto enumerável no ADO.NET usando o modelo de programação LINQ (consulta Language-Integrated).

NOTE

A LINQ to ADO.NET de ADO.NET está localizada na seção ADO.NET do SDK do .NET Framework: [LINQ e ADO.NET](#).

Há três tecnologias linq (ADO.NET Language-Integrated) separadas: LINQ to SQL LINQ to DataSet, e LINQ to Entities. O LINQ to DataSet fornece consultas mais sofisticadas e otimizadas do que o [DataSet](#), o LINQ to SQL permite que você consulte diretamente os esquemas de banco de dados do SQL Server e o LINQ to Entities permite que você consulte um Modelo de Dados de Entidade.

LINQ to DataSet

O [DataSet](#) é um dos componentes mais amplamente usados em ADO.NET e é um elemento fundamental do modelo de programação desconectada no qual o ADO.NET se baseia. No entanto, apesar dessa importância, o [DataSet](#) limitou os recursos de consulta.

O LINQ to DataSet permite que você crie recursos mais sofisticados de consulta no [DataSet](#) usando a mesma funcionalidade de consulta que está disponível para muitas outras fontes de dados.

Para obter mais informações, consulte [LINQ to DataSet](#).

LINQ to SQL

O LINQ to SQL fornece uma infraestrutura em tempo de execução para gerenciar dados relacionais como objetos. No LINQ to SQL, o modelo de dados de um banco de dados relacional é mapeado para um modelo de objeto expresso na linguagem de programação do desenvolvedor. Quando você executa o aplicativo, o LINQ to SQL converte consultas integradas da linguagem no modelo de objeto no SQL e as envia para o banco de dados para execução. Quando o banco de dados retorna os resultados, o LINQ to SQL os converte em objetos que podem ser manipulados.

LINQ to SQL inclui suporte para procedimentos armazenados e funções definidas pelo usuário no banco de dados e para herança no modelo de objeto.

Para obter mais informações, consulte [LINQ to SQL](#).

LINQ to Entities

Por meio do Modelo de Dados de Entidade, os dados relacionais são expostos como objetos no ambiente .NET. Isso torna a camada de objeto um destino ideal para suporte a LINQ, permitindo que os desenvolvedores formulem consultas no banco de dados da linguagem usada para criar a lógica de negócios. Essa funcionalidade é conhecida como LINQ to Entities. Consulte [LINQ to Entities](#) para obter mais informações.

Confira também

- [LINQ e o ADO.NET](#)
- [LINQ \(Consulta Integrada à Linguagem\) \(C#\)](#)

Habilitando uma fonte de dados para consulta LINQ

21/01/2022 • 3 minutes to read

Há várias maneiras de estender o LINQ para permitir que qualquer fonte de dados seja consultada no padrão LINQ. A fonte de dados pode ser uma estrutura de dados, um serviço Web, um sistema de arquivos ou um banco de dados, apenas para citar algumas opções. O padrão LINQ torna mais fácil para os clientes consultar uma fonte de dados para a qual a consulta LINQ está habilitada, porque a sintaxe e o padrão da consulta não são alterados. As maneiras pelas quais o LINQ pode ser estendido para essas fontes de dados incluem o seguinte:

- implementar a `IEnumerable<T>` interface em um tipo para habilitar LINQ to Objects consulta desse tipo.
- Criando métodos de operador de consulta padrão, como `Where` e `Select` que estendem um tipo, para habilitar a consulta LINQ personalizada desse tipo.
- Criação de um provedor para sua fonte de dados que implemente a interface `IQueryable<T>`. Um provedor que implementa essa interface recebe consultas LINQ na forma de árvores de expressão, que pode ser executada de forma personalizada, por exemplo, remotamente.
- Criar um provedor para sua fonte de dados que aproveita uma tecnologia LINQ existente. Tal provedor habilitaria não apenas a consulta, mas também operações de inserção, atualização e exclusão e mapeamento para tipos definidos pelo usuário.

Este tópico aborda essas opções.

Como habilitar consultas de LINQ da sua fonte de dados

Dados na memória

Há duas maneiras de habilitar a consulta LINQ de dados na memória. Se os dados forem de um tipo que implementa `IEnumerable<T>`, você poderá consultar os dados usando LINQ to Objects. Se não fizer sentido habilitar a enumeração do seu tipo implementando a `IEnumerable<T>` interface, você poderá definir métodos do operador de consulta padrão LINQ nesse tipo ou criar métodos do operador de consulta padrão do LINQ que estendem o tipo. As implementações personalizadas dos operadores de consulta padrão devem usar a execução adiada para retornar os resultados.

Dados remotos

A melhor opção para habilitar consultas LINQ de uma fonte de dados remota é implementar a `IQueryable<T>` interface. No entanto, isso é diferente de estender um provedor como o LINQ to SQL para uma fonte de dados.

Provedores IQueryable de LINQ

Provedores LINQ que implementam `IQueryable<T>` o podem variar muito em sua complexidade. Esta seção discute os diferentes níveis de complexidade.

Um provedor menos complexo de `IQueryable` poderia fazer a interface com um único método de um serviço Web. Esse tipo de provedor é muito específico porque ele espera informações específicas nas consultas que manipula. Ele possui um sistema de tipos fechado, talvez expondo um único tipo de resultado. A maior parte da execução da consulta ocorre localmente, por exemplo, usando as implementações de `Enumerable` dos operadores de consulta padrão. Um provedor menos complexo poderia examinar somente uma expressão de

chamada do método na árvore de expressões que representa a consulta e deixar que a lógica restante da consulta fosse manipulada em outro lugar.

Um provedor de `IQueryable` de complexidade média pode destinar uma fonte de dados que tem uma linguagem de consulta parcialmente expressiva. Se o destino é um serviço Web, ele pode fazer a interface com mais de um método de serviço Web e selecionar o método a ser chamado com base na questão representada pela consulta. Um provedor de complexidade média teria um sistema de tipos maiores do que um provedor simples, mas continuaria a ser um sistema de tipo fixo. Por exemplo, o provedor pode expor os tipos que têm relação um para muitos que podem ser atravessados, mas não forneceria tecnologia de mapeamento de tipos definidos pelo usuário.

Um `IQueryable` provedor complexo, como o LINQ to SQL provedor, pode traduzir consultas completas do LINQ para uma linguagem de consulta expressiva, como SQL. Um provedor complexo é mais geral do que um provedor menos complexo porque pode manipular uma variedade mais ampla de perguntas na consulta. Ele também possui um sistema de tipos abertos e, consequentemente, deve conter uma infraestrutura extensiva para mapear tipos definidos pelo usuário. Desenvolver um provedor complexo requer uma quantidade significativa de esforço.

Confira também

- [IQueryable<T>](#)
- [IEnumerable<T>](#)
- [Enumerable](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

Suporte de ferramentas e do IDE do Visual Studio para LINQ (C#)

21/01/2022 • 2 minutes to read

O IDE (ambiente de desenvolvimento integrado) do Visual Studio fornece os seguintes recursos que dão suporte ao desenvolvimento de aplicativos LINQ:

Object Relational Designer

O Object Relational Designer é uma ferramenta de design visual que você pode usar em aplicativos [LINQ to SQL](#) para gerar classes no C# que representam os dados relacionais em um banco de dados subjacente. Para obter mais informações, consulte [Ferramentas LINQ to SQL no Visual Studio](#).

Ferramenta de linha de comando SQLMetal

SQLMetal é uma ferramenta de linha de comando que pode ser usada em processos de build para gerar classes de bancos de dados existentes para uso em aplicativos LINQ to SQL. Para obter mais informações, consulte [SqlMetal.exe \(ferramenta de geração de código\)](#).

Editor de códigos com reconhecimento de LINQ

O editor de código do C# dá amplo suporte ao LINQ com o IntelliSense e os recursos de formatação.

Suporte do Depurador do Visual Studio

O depurador do Visual Studio dá suporte à depuração de expressões de consulta. Para obter mais informações, consulte [Depurando LINQ](#).

Confira também

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

Reflexão (C#)

21/01/2022 • 2 minutes to read

A reflexão fornece objetos (do tipo `Type`) que descrevem assemblies, módulos e tipos. É possível 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. Se você estiver usando atributos em seu código, a reflexão permite acessá-los. Para obter mais informações, consulte [Atributos](#).

Aqui está um exemplo simples de reflexão usando o método – herdado por todos os tipos da classe base – para `GetType()` obter o tipo de uma `Object` variável:

NOTE

Certifique-se `using System;` de adicionar `using System.Reflection;` e na parte superior do arquivo `.cs`.

```
// Using GetType to obtain type information:  
int i = 42;  
Type type = i.GetType();  
Console.WriteLine(type);
```

A saída é: `System.Int32` .

O exemplo a seguir usa a reflexão para obter o nome completo do assembly carregado.

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

A saída é: `System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e` .

NOTE

As palavras-chave C# e não têm nenhum significado em IL (Linguagem Intermediária) e não `protected` `internal` são usadas nas APIs de reflexão. Os termos correspondentes na IL são *Família* e *Assembly*. Para identificar um método `internal` usando a reflexão, use a propriedade `IsAssembly`. Para identificar um método `protected internal`, use o `IsFamilyOrAssembly`.

Visão geral da reflexão

A reflexão é útil nas seguintes situações:

- Quando você precisa acessar atributos nos metadados do seu programa. Para obter mais informações, consulte [Recuperando informações armazenadas em atributos](#).
- Para examinar e instanciar tipos em um assembly.
- Para criar novos tipos em tempo de operação. Usar as classes em [System.Reflection.Emit](#).
- Para executar a associação tardia, acessar métodos em tipos criados em tempo de execução. Consulte o tópico [Carregando e usando tipos dinamicamente](#).

Seções relacionadas

Para mais informações:

- [Reflexão](#)
- [Exibindo informações de tipo](#)
- [Reflexão e tipos genéricos](#)
- [System.Reflection.Emit](#)
- [Recuperando informações armazenadas em atributos](#)

Confira também

- [Guia de Programação em C#](#)
- [Assemblies no .NET](#)

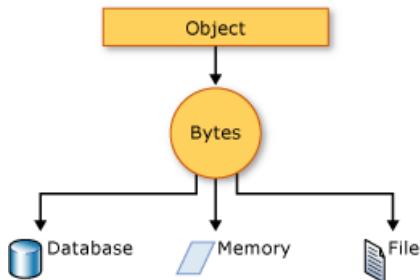
Serialização (C#)

21/01/2022 • 4 minutes to read

A serialização é o processo de converter um objeto em um fluxo de bytes para armazenar o objeto ou transmiti-lo para a memória, um banco de dados ou um arquivo. Sua finalidade principal é salvar o estado de um objeto para recriá-lo quando necessário. O processo inverso é chamado desserialização.

Como a serialização funciona

Esta ilustração mostra o processo geral de serialização:



O objeto é serializado para um fluxo que carrega os dados. O fluxo também pode ter informações sobre o tipo do objeto, como sua versão, cultura e nome do assembly. Desse fluxo, o objeto pode ser armazenado em um banco de dados, um arquivo ou memória.

Usos para serialização

A serialização permite que o desenvolvedor salve o estado de um objeto e re-crie-o conforme necessário, fornecendo armazenamento de objetos, bem como troca de dados. Por meio da serialização, um desenvolvedor pode executar ações como:

- Enviar o objeto para um aplicativo remoto usando um serviço Web
- Passando um objeto de um domínio para outro
- Passando um objeto por meio de um firewall como uma cadeia de caracteres JSON ou XML
- Manutenção de informações de segurança ou específicas do usuário entre aplicativos

Serialização JSON

O [System.Text.Json](#) namespace contém classes para serialização JavaScript Object Notation (JSON) e a serialização. O JSON é um padrão aberto que normalmente é usado para compartilhar dados pela Web.

A serialização JSON serializa as propriedades públicas de um objeto em uma cadeia de caracteres, matriz de byte ou fluxo que está em conformidade com a especificação [JSON RFC 8259](#). Para controlar a maneira [JsonSerializer](#) como serializa ou desseparaiza uma instância da classe :

- Usar um [JsonSerializerOptions](#) objeto
- Aplicar atributos do [System.Text.Json.Serialization](#) namespace a classes ou propriedades
- [Implementar conversores personalizados](#)

Serialização XML e binária

O [System.Runtime.Serialization](#) namespace contém classes para serialização e deserialização binária e XML.

A serialização binária usa a codificação binária para produzir uma serialização compacta para usos como

armazenamento ou fluxos de rede com base em soquete. Na serialização binária, todos os membros, mesmo aqueles que são somente leitura, são serializados e o desempenho é aprimorado.

WARNING

A serialização binária pode ser perigosa. Para obter mais informações, consulte [Guia de segurança do BinaryFormatter](#).

A serialização XML serializa as propriedades e os campos públicos de um objeto, ou os parâmetros e os valores de retorno de métodos, em um fluxo XML que esteja de acordo com um documento XSD (linguagem de definição de esquema XML) específico. A serialização XML resulta em classes fortemente tipadas com propriedades e campos públicos que são convertidos em XML. [System.Xml.Serialization](#) contém classes para serializar e deserializar XML. Aplique atributos a classes e a membros de classe para controlar a maneira como o [XmlSerializer](#) serializa ou desserializa uma instância da classe.

Tornando um objeto serializável

Para serialização binária ou XML, você precisa de:

- O objeto a ser serializado
- Um fluxo para conter o objeto serializado
- Uma [System.Runtime.Serialization.Formatter](#) instância

Aplique o [SerializableAttribute](#) atributo a um tipo para indicar que instâncias do tipo podem ser serializadas. Uma exceção será gerada se você tentar serializar, mas o tipo não terá o atributo [SerializableAttribute](#).

Para impedir que um campo seja serializado, aplique o [NonSerializedAttribute](#) atributo. Se um campo de um tipo serializável contiver um ponteiro, um identificador ou outra estrutura de dados que é específica de um determinado ambiente e o campo não puder ser reconstituído em um ambiente diferente, será necessário torná-lo não serializável.

Se uma classe serializada contiver referências a objetos de outras classes que estão marcadas como [SerializableAttribute](#), esses objetos também serão serializados.

Serialização básica e personalizada

A serialização binária e XML pode ser executada de duas maneiras, básica e personalizada.

A serialização básica usa o .NET para serializar automaticamente o objeto. O único requisito é que a classe tenha o [SerializableAttribute](#) atributo aplicado. O [NonSerializedAttribute](#) pode ser usado para impedir a serialização de campos específicos.

Quando você usa a serialização básica, a criação de versão de objetos pode criar problemas. Use a serialização personalizada quando problemas de criação de versão forem importantes. A serialização básica é a maneira mais fácil de executar a serialização, mas ela não fornece muito controle sobre o processo.

Na serialização personalizada, você pode especificar exatamente quais objetos vão ser serializados e como isso será feito. A classe deve ser marcada como [SerializableAttribute](#) e implementar a interface [ISerializable](#). Se você quiser que seu objeto seja desserializado de maneira personalizada também, use um construtor personalizado.

Serialização de designer

A serialização de designer é um formulário especial de serialização que envolve o tipo de persistência do objeto associado a ferramentas de desenvolvimento. A serialização de designer é o processo de conversão de um grafo do objeto em um arquivo de origem que pode, posteriormente, ser usado para recuperar o grafo do objeto. Um arquivo de origem pode conter código, marcação ou até mesmo informações de tabela do SQL.

Exemplos e tópicos relacionados

[Visão geral de System.Text.Json](#) Mostra como obter a `System.Text.Json` biblioteca.

[Como serializar e deserializar JSON no .NET](#) Mostra como ler e gravar dados de objeto de e para JSON usando a `JsonSerializer` classe .

[Passo a passo: mantendo um objeto no Visual Studio \(C#\)](#)

Demonstra como a serialização pode ser usada para manter dados de um objeto entre instâncias, permitindo que você armazene e recupere valores na próxima vez que o objeto for instanciado.

[Como ler dados de objeto de um arquivo XML \(C#\)](#)

Mostra como ler dados de objeto que foram previamente gravados em um arquivo XML usando a classe `XmlSerializer`.

[Como gravar dados de objeto em um arquivo XML \(C#\)](#)

Mostra como gravar o objeto de uma classe para um arquivo XML usando a classe `XmlSerializer`.

Como gravar dados de objeto em um arquivo XML (C#)

21/01/2022 • 2 minutes to read

Este exemplo grava o objeto de uma classe para um arquivo XML usando a classe [XmlSerializer](#).

Exemplo

```
public class XMLWrite
{
    static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
        "//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

Compilando o código

A classe precisa ter um construtor público sem parâmetros.

Programação robusta

As seguintes condições podem causar uma exceção:

- A classe que está sendo serializada não tem um construtor público sem parâmetros.
- O arquivo existe e é somente leitura ([IOException](#)).
- O caminho é muito longo ([PathTooLongException](#)).
- O disco está cheio ([IOException](#)).

Segurança do .NET

Este exemplo cria um novo arquivo, se o arquivo ainda não existe. Se um aplicativo precisar criar um arquivo, ele precisará de acesso `Create` para a pasta. Se o arquivo já existe, o aplicativo precisa apenas de acesso `Write`, um privilégio menor. Sempre que possível, é mais seguro criar o arquivo durante a implantação e somente conceder acesso `Read` a um único arquivo, em vez de acesso `Create` a uma pasta.

Confira também

- [StreamWriter](#)
- [Como ler dados de objeto de um arquivo XML \(C#\)](#)
- [Serialização \(C#\)](#)

Como ler dados de objeto de um arquivo XML (C#)

21/01/2022 • 2 minutes to read

Este exemplo lê dados de objeto que foram previamente gravados em um arquivo XML usando a classe [XmlSerializer](#).

Exemplo

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview = (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);
}
```

Compilando o código

Substitua o nome de arquivo "c:\temp\SerializationOverview.xml" pelo nome do arquivo que contém os dados serializados. Para obter mais informações sobre a serialização de dados, consulte [como gravar dados de objeto em um arquivo XML \(C#\)](#).

A classe deve ter um construtor público sem parâmetros.

Somente propriedades e campos públicos são desserializados.

Programação robusta

As seguintes condições podem causar uma exceção:

- A classe que está sendo serializada não tem um construtor público sem parâmetros.
- Os dados no arquivo não representam dados da classe a ser desserializada.
- O arquivo não existe ([IOException](#)).

Segurança do .NET

Sempre verifique as entradas e nunca desserialize dados de uma fonte não confiável. O objeto recriado é executado em um computador local com as permissões do código que o desserializou. Verifique todas as entradas antes de usar os dados no seu aplicativo.

Confira também

- [StreamWriter](#)
- [Como gravar dados de objeto em um arquivo XML \(C#\)](#)
- [Serialização \(C#\)](#)
- [Guia de programação C#](#)

Passo a passo: persistir um objeto usando o C#

21/01/2022 • 4 minutes to read

Você pode usar a serialização para manter os dados de um objeto entre instâncias, o que permite armazenar valores e recuperá-los na próxima vez que o objeto for instanciado.

Neste passo a passo, você criará um objeto `Loan` básico e persistirá seus dados em um arquivo. Em seguida, você recuperará os dados do arquivo quando recriar o objeto.

IMPORTANT

Este exemplo criará um novo arquivo se o arquivo ainda não existir. Se um aplicativo precisar criar um arquivo, esse aplicativo precisará ter a permissão `Create` para a pasta. Permissões são definidas usando listas de controle de acesso. Se o arquivo já existir, o aplicativo precisará somente da permissão `Write`, que é uma permissão menor. Sempre que possível, é mais seguro criar o arquivo durante a implantação e somente conceder permissões `Read` a um único arquivo (em vez das permissões `Create` para uma pasta). Além disso, é mais seguro gravar dados em pastas de usuário do que na pasta raiz ou na pasta Arquivos de Programas.

IMPORTANT

Este exemplo armazena dados em um arquivo de formato binário. Esses formatos não devem ser usados para dados confidenciais, como senhas ou informações de cartão de crédito.

Pré-requisitos

- Para compilar e executar, instale o [SDK do .NET](#).
- Instale seu editor de código favorito, caso ainda não tenha um.

TIP

Precisa instalar um editor de código? Experimente o [Visual Studio](#)!

- O exemplo exige C# 7.3. Confira [Selecionar a versão da linguagem C#](#)

Examine o código de exemplo online no [repositório GitHub de amostras do .NET](#).

Criando o objeto Loan

A primeira etapa é criar uma classe `Loan` e um aplicativo de console que usa a classe:

- Crie um novo aplicativo. Digite `dotnet new console -o serialization` para criar um novo aplicativo de console em um subdiretório chamado `serialization`.
- Abra o aplicativo no editor e adicione uma nova classe chamada `Loan.cs`.
- Adicione o seguinte código à classe `Loan`:

```

public class Loan : INotifyPropertyChanged
{
    public double LoanAmount { get; set; }
    public double InterestRatePercent { get; set; }

    [field:NonSerialized()]
    public DateTime TimeLastLoaded { get; set; }

    public int Term { get; set; }

    private string customer;
    public string Customer
    {
        get { return customer; }
        set
        {
            customer = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(Customer)));
        }
    }

    [field: NonSerialized()]
    public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

    public Loan(double loanAmount,
               double interestRate,
               int term,
               string customer)
    {
        this.LoanAmount = loanAmount;
        this.InterestRatePercent = interestRate;
        this.Term = term;
        this.customer = customer;
    }
}

```

Você também precisará criar um aplicativo que usa a classe `Loan`.

Serializar o objeto Loan

1. Abra o `Program.cs`. Adicione os códigos a seguir:

```
Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");
```

Adicione um manipulador de eventos ao evento `PropertyChanged` e algumas linhas para modificar o objeto `Loan` e exibir as alterações. Veja as adições no seguinte código:

```

TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);

```

Neste ponto, você pode executar o código e ver a saída atual:

```
New customer value: Henry Clay
```

```
7.5
```

```
7.1
```

A execução desse aplicativo repetidamente grava os mesmos valores. Um novo objeto Loan é criado sempre que o programa é executado. No mundo real, as taxas de juros mudam periodicamente, mas não necessariamente toda vez que o aplicativo for executado. Código de serialização significa preservar a taxa de juros mais recente entre instâncias do aplicativo. Na próxima etapa, você fará exatamente isso adicionando a serialização à classe Loan.

Usando a serialização para manter o objeto

Para manter os valores da classe Loan, primeiro você deve marcar a classe com o atributo `Serializable`.

Adicione o seguinte código acima da definição da classe Loan:

```
[Serializable()]
```

O `SerializableAttribute` informa ao compilador que tudo na classe pode ser persistido em um arquivo. Como o evento `PropertyChanged` não representa a parte do grafo do objeto que deve ser armazenada, ele não deve ser serializado. Ao fazer isso, todos os objetos anexados a esse evento serão serializados. Adicione o `NonSerializedAttribute` à declaração de campo do manipulador de eventos `PropertyChanged`.

```
[field: NonSerialized()]
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

Do C# 7.3 em diante, você pode anexar atributos ao campo de suporte de uma propriedade autoimplementada usando o valor de destino `field`. O seguinte código adiciona uma propriedade `TimeLastLoaded` e a marca como não serializável:

```
[field:NonSerialized()]
public DateTime TimeLastLoaded { get; set; }
```

A etapa seguinte é adicionar o código de serialização ao aplicativo LoanApp. Para serializar a classe e gravá-la em um arquivo, use os namespaces `System.IO` e `System.Runtime.Serialization.Formatters.Binary`. Para evitar a digitação dos nomes totalmente qualificados, você pode adicionar referências aos namespaces necessários, conforme mostrado no seguinte código:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

A próxima etapa é adicionar código para desserializar o objeto do arquivo quando o objeto for criado. Adicione uma constante à classe do nome de arquivo dos dados serializados, conforme mostrado no seguinte código:

```
const string FileName = @"../../../../SavedLoan.bin";
```

Em seguida, adicione o seguinte código após a linha que cria o objeto `TestLoan`:

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

Primeiro, é necessário verificar se o arquivo existe. Se ele existir, crie uma classe [Stream](#) para ler o arquivo binário e uma classe [BinaryFormatter](#) para converter o arquivo. Você também precisa converter do tipo de fluxo para o tipo de objeto [Loan](#).

Em seguida, é necessário adicionar um código para serializar a classe em um arquivo. Adicione o seguinte código após o código existente no método [Main](#) :

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

Neste ponto, você pode compilar e executar o aplicativo novamente. Na primeira vez em que ele é executado, observe que as taxas de juros começam em 7,5 e, em seguida, são alteradas para 7,1. Feche o aplicativo e execute-o novamente. Agora, o aplicativo imprime a mensagem indicando que ele leu o arquivo salvo e a taxa de juros é 7,1, mesmo antes do código que a altera.

Confira também

- [Serialização \(C#\)](#)
- [Guia de programação C#](#)

Instruções, expressões e operadores (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

O código C# que compõe um aplicativo consiste em instruções compostas por palavras-chave, expressões e operadores. Esta seção contém informações sobre esses elementos fundamentais de um programa C#.

Para obter mais informações, consulte:

- [Instruções](#)
- [Operadores e expressões](#)
- [Membros aptos para expressão](#)
- [Comparações de igualdade](#)

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 C#](#)
- [Conversões cast e conversões de tipo](#)

Instruções (Guia de Programação em C#)

21/01/2022 • 6 minutes to read

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:

```
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:

CATEGORIA	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">• if• switch
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">• do• for• foreach• mesmo
Instruções de atalho	Instruções de hiperlink transferem o controle para outra seção de código. Para obter mais informações, consulte estes tópicos: <ul style="list-style-type: none">• break• continua• goto• exibir• proporcionar
Instruções para tratamento de exceções	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: <ul style="list-style-type: none">• throw• try-catch• Experimente-finalmente• try – catch-finally
Marcado e desmarcado	As instruções checked e unchecked permitem que você especifique se operações numéricas podem causar um estouro quando o resultado for armazenado em uma variável que é muito pequena para conter o valor resultante. Para obter mais informações, consulte checked e unchecked .

CATEGORIA	PALAVRAS-CHAVE DO C#/OBSERVAÇÕES
A instrução <code>await</code>	<p>Se marcar um método com o modificador <code>async</code>, você poderá usar o operador <code>await</code> 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 Métodos. Para obter mais informações, consulte programação assíncrona com Async e Await.</p>
A instrução <code>yield return</code>	<p>Um iterador realiza uma iteração personalizada em uma coleção, como uma lista ou uma matriz. Um iterador usa a instrução <code>yield return</code> 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 Iteradores.</p>
A instrução <code>fixed</code>	<p>A instrução <code>fixed</code> impede que o coletor de lixo faça a realocação de uma variável móvel. Para obter mais informações, consulte fixed.</p>
A instrução <code>lock</code>	<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 lock.</p>
Instruções rotuladas	<p>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.)</p>
A instrução vazia	<p>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.</p>

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.

```
// 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.

```

// 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:

```

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 incorporada 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:

```

// 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:

```

if(pointB == true)
    //Error CS1023:
    int radius = 5;

```

Coloque a instrução inserida em um bloco para corrigir o erro:

```
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:

```
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:

```
// 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 C#](#)
- [Palavras-chave da instrução](#)
- [Operadores e expressões C#](#)

Membros aptos para expressão (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

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:

```
member => expression;
```

em que *expression* é uma expressão válida.

O suporte para definições de corpo da expressão foi introduzido para métodos e propriedades somente leitura no C# 6 e foi expandido no C# 7.0. As definições de corpo da expressão podem ser usadas com os membros de tipo listados na tabela a seguir:

MEMBRO	COM SUPORTE DESDE...
Método	C# 6
Propriedade somente leitura	C# 6
Propriedade	C# 7.0
Construtor	C# 7.0
Finalizer	C# 7.0
Indexador	C# 7.0

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`.

```

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

Começando no C# 6, você pode usar a definição de corpo da expressão para implementar uma propriedade somente leitura. Para isso, use a seguinte sintaxe:

```
.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:

```

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

Começando no C# 7.0, 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:

```
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`.

```
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.

```
using System;

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

Assim como acontece com propriedades, indexador e acessadores consistem em definições de corpo de expressão se o acessador consiste em uma única expressão que retorna um valor ou o acessador executa uma `get` `set` `get` `set` atribuição simples.

O exemplo a seguir define uma classe chamada `Sports` que inclui uma matriz `String` interna que contém os nomes de vários esportes. O indexador e `get` `set` os acessadores são implementados como definições de corpo da expressão.

```
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 \(Guia de Programação em C#\)](#).

Comparações de igualdade (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

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.

```
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. Você pode usar o `==` operador, conforme mostrado no exemplo a seguir.

```
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 correspondem e todos os valores de propriedade e campo correspondem.

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 estruturas, 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
Como testar a igualdade de referência (identidade)	Descreve como determinar se duas variáveis têm igualdade de referência.
Como definir a igualdade de valor para um tipo	Descreve como fornecer uma definição personalizada de igualdade de valor a um tipo.
Guia de programação C#	Fornece links para informações detalhadas sobre recursos importantes da linguagem C# e recursos que estão disponíveis para C# por meio do .NET.
Types	Fornece informações sobre o sistema de tipos do C# e links para mais informações.
Registros	Fornece informações sobre tipos de registro, que testam a igualdade de valor por padrão.

Confira também

- [Guia de programação C#](#)

Como definir a igualdade de valor para uma classe ou struct (Guia de Programação em C#)

21/01/2022 • 9 minutes to read

Os registros implementam automaticamente a igualdade de valor. Considere definir um em `record` vez de um quando seu tipo `class` modela dados e deve 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 a igualdade de valor quando espera adicionar objetos do tipo a uma coleção ou quando sua principal finalidade é 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 regras a seguir, suponha que `x`, `y` e `z` não sejam nulos):

1. A propriedade reflexiva: `x.Equals(x)` retorna `true`.
2. A propriedade simétrica: `x.Equals(y)` retorna o mesmo valor que `y.Equals(x)`.
3. A propriedade transitiva: se `(x.Equals(y) && y.Equals(z))` retornar `true`, `x.Equals(z)` retornará `true`.
4. Invocações sucessivas de retornam o mesmo valor, desde que os objetos referenciados por `x` e `y` não sejam modificados.
5. Qualquer valor não nulo não é igual a null. No entanto, `x.Equals(y)` lança uma exceção quando `x` é nulo. Isso interrompe as regras 1 ou 2, dependendo 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 `virtual Object.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 lançar exceções 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 se o tipo herdar diretamente de , porque a implementação de executa uma `base.Equals` verificação de igualdade de `Object Object.Equals(Object)` referência.)
 - Duas variáveis devem ser consideradas iguais somente se os tipos de tempo de run-time das variáveis que estão sendo comparadas são os mesmos. Além disso, certifique-se de que a

implementação do método para o tipo de tempo de execução seja usada se os tipos de tempo de execução e tempo de compilação de uma variável `IEquatable` `Equals` são diferentes. Uma estratégia para garantir que os tipos de tempo de execução sejam sempre comparados corretamente é implementar `IEquatable` somente em `sealed` classes. Para obter mais informações, consulte [o exemplo de classe](#) mais adiante neste artigo.

3. Opcional, mas recomendado: sobrecarregar `==` 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 a definições para "maior que" ou "menor que", implemente a interface para seu tipo e também `IComparable<T>` sobrecarregar os `<= >=` operadores e .

NOTE

A partir do C# 9.0, você pode usar registros para obter semântica de igualdade de valor sem nenhum código clichê desnecessário.

Exemplo de classe

O exemplo a seguir mostra como implementar a igualdade de valor em uma classe (tipo de referência).

```
using System;

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.

IMPORTANT

O código de exemplo anterior pode não lidar com cada cenário de herança da maneira esperada. Considere o seguinte código:

```
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 apesar da diferença nos `p2` `z` valores. A diferença é ignorada porque o compilador escolhe a `TwoDPoint` implementação de com base no tipo de tempo de `IEquatable` compilação.

A igualdade de valor integrado de `record` tipos lida com cenários como este corretamente. Se `TwoDPoint` e `ThreeDPoint` fossem `record` tipos, o resultado de seria `p1.Equals(p2) False`. Para obter mais informações, consulte [Igualdade nas record hierarquias](#)de herança de tipo .

Exemplo de struct

O exemplo a seguir mostra como implementar a igualdade de valor em um struct (tipo de valor):

```
using System;

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 `Equals` virtual em uma estrutura, 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 sobrepor explicitamente.

Confira também

- [Comparações de igualdade](#)
- [Guia de programação em C#](#)

Como testar a igualdade de referência (identidade) (guia de programação C#)

21/01/2022 • 3 minutes to read

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

```
using System;
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 sobrepor 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 tempo de execução não garante que as cadeias de caracteres criadas em tempo de execução sejam internas, nem garantem que duas cadeias de caracteres constantes iguais em assemblies diferentes sejam estagiários.

Confira também

- [Comparações de igualdade](#)

Coerções e conversões de tipo (Guia de Programação em C#)

21/01/2022 • 5 minutes to read

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:

```
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 (conversões):** conversões explícitas exigem uma [expressão de conversão](#). 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 byte 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`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Para ver uma lista completa de todas as conversões numéricas implícitas, consulte a seção [Conversões numéricas implícitas](#) do artigo Conversões numéricas integrados.

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.

```
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 o compilador de que você pretende fazer a conversão e que você está ciente de que a perda de dados pode ocorrer ou que a conversão pode falhar em tempo de operação. 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 lança um `double` para um `int`. O programa não será compilado sem a `cast`.

```
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 ver uma lista completa das conversões numéricas explícitas com suporte, consulte a seção [Conversões numéricas explícitas](#) do artigo Conversões numéricas integrados.

Para tipos de referência, uma conversão explícita será necessária se você precisar converter de um tipo base para um tipo derivado:

```

// 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.

```

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 `Test` método tem um parâmetro `Animal`, portanto, a transmissão explícita do argumento para `a` faz uma `Reptile` suposição perigosa. É mais seguro não fazer suposições, mas, em vez disso, verificar o tipo. O C# fornece o operador `is` para habilitar o teste de compatibilidade antes de realmente executar uma conversão. Para obter mais informações, consulte Como fazer a cast com segurança usando a correspondência [de padrões e os operadores as e is](#).

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#)

21/01/2022 • 6 minutes to read

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) em caixas de um tipo de valor, ele envolve o valor dentro de uma instância e o `System.Object` armazena no heap gerenciado. A conversão unboxing extraí 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`.

```
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`:

```
o = 123;
i = (int)o; // unboxing
```

Os exemplos a seguir ilustram como a conversão boxing é usada em 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.

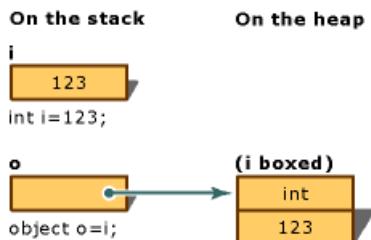
Considere a seguinte declaração de uma variável de tipo de valor:

```
int i = 123;
```

A instrução a seguir aplica implicitamente a operação de conversão boxing na variável `i`:

```
// 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:

```
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.

```
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  
*/
```

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

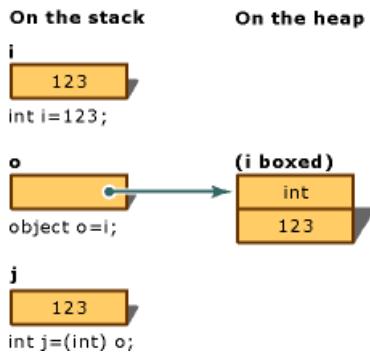
tipo de valor.

- Copiar o valor da instância para a variável de tipo de valor.

As instruções a seguir demonstram operações conversão boxing e unboxing:

```
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.

```
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:

```
int j = (short)o;
```

para:

```
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 C#)

21/01/2022 • 2 minutes to read

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 os métodos na [BitConverter](#) classe que convertem bytes (de uma matriz de bytes) em outros tipos internos.

TIPO RETORNADO	MÉTODO
<code>bool</code>	<code>ToBoolean (byte [], Int32)</code>
<code>char</code>	<code>ToChar (byte [], Int32)</code>
<code>double</code>	<code>ToDouble (byte [], Int32)</code>
<code>short</code>	<code>ToInt16 (byte [], Int32)</code>
<code>int</code>	<code>ToInt32 (byte [], Int32)</code>
<code>long</code>	<code>ToInt64 (byte [], Int32)</code>
<code>float</code>	<code>ToSingle (byte [], Int32)</code>
<code>ushort</code>	<code>ToUInt16 (byte [], Int32)</code>
<code>uint</code>	<code>ToUInt32 (byte [], Int32)</code>
<code>ulong</code>	<code>ToUInt64 (byte [], Int32)</code>

Exemplos

Este exemplo Inicializa uma matriz de bytes, reverte a matriz se a arquitetura do computador é little-endian (ou seja, o byte menos significativo é armazenado primeiro) e, em seguida, chama o método `ToInt32 (byte [], Int32)` para converter quatro bytes na matriz em um `int`. O segundo argumento para `ToInt32 (byte [], Int32)` especifica o índice de início da matriz de bytes.

NOTE

A saída pode diferir dependendo da ordenação da arquitetura do seu computador.

```
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.

NOTE

A saída pode diferir dependendo da ordenação da arquitetura do seu computador.

```
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 C#)

21/01/2022 • 5 minutes to read

Converte um `string` em um número chamando o `Parse` método ou `TryParse` encontrado em tipos numéricos (`int`, `long`, `double` e assim por diante) ou usando métodos na `System.Convert` classe.

É um pouco mais eficiente e direto chamar um `TryParse` método (por exemplo, `int.TryParse("11", out number)`) ou `Parse` método (por exemplo, `var number = int.Parse("11")`). Usar um método `Convert` é mais útil para objetos gerais que implementam `IConvertible`.

Você usa `Parse` `TryParse` métodos ou no tipo numérico que espera que a cadeia de caracteres contenha, como o `System.Int32` tipo. O método `Convert.ToInt32` usa `Parse` internamente. O `Parse` método retorna o número convertido; o `TryParse` método retorna um valor booleano que indica se a conversão foi bem-sucedida e retorna o número convertido em um `out` parâmetro. Se a cadeia de caracteres não estiver em um formato válido, o `Parse` lançará uma exceção, mas `TryParse` retornará `false`. Ao chamar um `Parse` método, você sempre deve usar a manipulação de exceção para capturar um `FormatException` quando a operação de análise falhar.

Chamar os métodos Parse ou TryParse

Os `Parse` `TryParse` métodos ignoram o espaço em branco no início e no final da cadeia de caracteres, mas todos os outros caracteres devem ser caracteres que formam o tipo numérico apropriado (,, `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, você pode usar `decimal.TryParse` para analisar "10", "10,3" ou "10", mas não pode usar esse método para analisar 10 de "10x", "1 0" (Observe o espaço incorporado), "10 3" (Observe o espaço incorporado), "10e1" (`float.TryParse` funciona aqui) e assim por diante. Uma cadeia de caracteres cujo valor é `null` ou não é possível `String.Empty` analisar 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`.

```

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 esperada para 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 strings a serem analisadas contêm alguns caracteres, o exemplo chama o [String.Concat](#) método para atribuir caracteres válidos a uma nova cadeia de caracteres. Para cadeias de caracteres maiores, pode ser usada a classe [StringBuilder](#).

```
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
decimal	ToDecimal(String)

TIPO NUMÉRICO	MÉTODO
float	ToSingle(String)
double	ToDouble(String)
short	ToInt16(String)
int	ToInt32(String)
long	ToInt64(String)
ushort	ToUInt16(String)
uint	ToUInt32(String)
ulong	ToUInt64(String)

O exemplo a seguir chama o [Convert.ToInt32\(String\)](#) método 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.

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.WriteLine("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#)

21/01/2022 • 3 minutes to read

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](#).

```
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 indivíduo [string](#) em uma matriz. Em seguida, [ToInt32\(String, Int32\)](#) ele chama 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](#).

```

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\(String, NumberStyles\)](#).

```

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](#).

```

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](#).

```
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.

```
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](#)

Usando o tipo dynamic (Guia de Programação em C#)

21/01/2022 • 5 minutes to read

O C# 4 apresenta um novo tipo, `dynamic`. O tipo é um tipo 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`. Em tempo de compilação, supõem-se que um elemento que tem o tipo `dynamic` dá suporte a qualquer operação. Portanto, você não precisa se preocupar 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 serão capturados 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 é válido porque ele contém dois argumentos. Essa chamada causa um erro do compilador. A segunda chamada para o método, `dynamic_ec.exampleMethod1(10, 4)`, não é verificada pelo compilador porque o tipo de `dynamic_ec` é `dynamic`. Portanto, nenhum erro de compilador é relatado. No entanto, o erro não escapa o aviso indefinidamente. Ele é detectado em tempo de execução e causa uma exceção de tempo de execução.

```
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();
}
```

```
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`. Em tempo de execução, as informações armazenadas são examinadas e qualquer instrução inválida causa uma exceção de 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.

```
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`:

```
var testInstance = new ExampleClass(d);
```

Exemplos de conversão são mostrados na seção a seguir, "Conversões".

Conversões

As conversões entre objetos dinâmicos e outros tipos são fáceis. Isso permite que o desenvolvedor mude entre o comportamento dinâmico e o não dinâmico.

Qualquer objeto pode ser convertido no tipo dinâmico implicitamente, conforme mostrado nos exemplos a seguir.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Por outro lado, uma conversão implícita pode ser aplicada dinamicamente a qualquer expressão do tipo `dynamic`.

```
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.

```
// 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);
```

runtimde linguagem dinâmico

o DLR (dynamic language runtime) é uma API que foi introduzida no .NET Framework 4. Ele 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

O C# 4 inclui vários recursos que aprimoram a experiência de interoperar com APIs COM, como as APIs de Automação do Office. Entre os aperfeiçoamentos estão o uso do tipo `dynamic` e de [argumentos nomeados e opcionais](#).

Muitos métodos COM permitem variação nos tipos de argumento e tipo de retorno, especificando os tipos como `object`. Isso 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 `dynamic` tipo permitirá tratar as ocorrências de `object` em assinaturas com como se fossem do tipo `dynamic` e, portanto, evitar grande parte da conversão. Por exemplo, as seguintes instruções de contrastam como acessar uma célula em uma planilha do Microsoft Office Excel com o tipo `dynamic` e sem o tipo `dynamic`.

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and
// the conversion to Excel.Range are handled by the run-time COM binder.
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

Tópicos relacionados

TÍTULO	DESCRIÇÃO
dinâmico	Descreve o uso da palavra-chave <code>dynamic</code> .
Visão geral do tempo de execução de linguagem dinâmica	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).
Walkthrough: Criando e usando objetos dinâmicos	Fornece instruções passo a passo para criar um objeto dinâmico personalizado e para criar um projeto que acessa uma biblioteca <code>IronPython</code> .

TÍTULO	DESCRIÇÃO
Como acessar objetos de interoperabilidade do Office usando recursos do C#	Demonstra como criar um projeto que usa argumentos nomeados e opcionais, o tipo <code>dynamic</code> e outros aprimoramentos que simplificam o acesso aos objetos de API do Office.

Passo a passo: criar e usar objetos dinâmicos (C# e Visual Basic)

21/01/2022 • 12 minutes to read

Os objetos dinâmicos expõem membros como propriedades e métodos em tempo de execução, em vez de em tempo de compilação. Isso permite 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 que é interpretado em tempo de execução.

Você faz referência a um objeto dinâmico usando a associação tardia. Em C#, você especifica o tipo de um objeto com associação tardia como `dynamic`. No Visual Basic, você especifica o tipo de um objeto de associação tardia como `Object`. Para obter mais informações, consulte [dynamic](#) e [Associação antecipada e tardia](#).

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 passos a passos 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`.

Você pode fazer um desses ou ambos, e se você fizer ambos, a ordem não importa.

Pré-requisitos

- [Visual Studio 2019 versão 16,9 ou uma versão posterior](#) com a carga de **trabalho de desenvolvimento do .net desktop** instalada. O SDK do .NET 5 é instalado automaticamente quando você seleciona essa carga de trabalho.

NOTE

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 a segunda explicação, instale o `IronPython` para .net. Navegue até a respectiva [página de download](#) para

obter a versão mais recente.

Criar um objeto dinâmico personalizado

A primeira instrução 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.

Para criar uma classe dinâmica personalizada

1. Inicie o Visual Studio.
2. Selecione **Criar um novo projeto**.
3. na caixa de diálogo **criar um novo projeto**, selecione C# ou Visual Basic, selecione **aplicativo de Console** e, em seguida, selecione **avançar**.
4. na caixa de diálogo **configurar seu novo projeto**, digite `DynamicSample` para o **nome do Project** e, em seguida, selecione **avançar**.
5. Na caixa de diálogo **informações adicionais**, selecione .NET 5,0 (atual) para a **estrutura de destino** e, em seguida, selecione **criar**.

Quando um novo projeto é criado.

6. Em **Gerenciador de soluções**, clique com o botão direito do mouse no projeto DynamicSample e selecione **Adicionar > classe**. Na caixa **nome**, digite `ReadOnlyFile` e, em seguida, selecione **Adicionar**.

É adicionado um novo arquivo que contém a classe `ReadOnlyFile`.

7. Na parte superior do arquivo `ReadOnlyFile.cs` ou `ReadOnlyFile.vb`, adicione o código a seguir para importar os `System.IO` `System.Dynamic` namespaces e.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

8. O objeto dinâmico personalizado usa um enum para determinar os critérios de pesquisa. Antes da instrução de classe, adicione a seguinte definição de enum.

```

public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}

```

```

Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum

```

9. Atualize a instrução de classe para herdar a classe `DynamicObject`, conforme mostrado no exemplo de código a seguir.

```

class ReadOnlyFile : DynamicObject

```

```

Public Class ReadOnlyFile
    Inherits DynamicObject

```

10. 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`.

```

// 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;
}

```

```

' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub

```

11. 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.

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }
}

return results;
}
```

```

Public Function GetPropertyValue(ByVal propertyName As String,
                               Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                               Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCASE(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCASE(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function

```

12. Após o método `GetPropertyValue`, 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`.

```

// 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;
}

```

```

' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function

```

13. 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 `GetPropertyValue` para recuperar os resultados.

```
// 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 = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```

' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                         ByVal args() As Object,
                                         ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function

```

14. Salve e feche o arquivo.

Para criar um arquivo de texto de exemplo

1. Em 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 clique em Adicionar. Um novo arquivo de texto é adicionado ao projeto.
2. Copie o texto a seguir para o arquivo de *TextFile1.txt*.

```

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

```

3. Salve e feche o arquivo.

Para criar um aplicativo de exemplo que usa o objeto dinâmico personalizado

1. em Gerenciador de Soluções, clique duas vezes no arquivo *program.vb* se você estiver usando Visual Basic ou o arquivo *program.cs* se você estiver usando o Visual C#.
2. Adicione o seguinte código ao `Main` procedimento para criar uma instância da `ReadOnlyFile` classe para o arquivo de *TextFile1.txt*. O código usa associação tardia para chamar membros dinâmicos e recuperar linhas de texto que contêm a cadeia de caracteres "Customer".

```

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);
}

```

```

Dim rFile As Object = New ReadOnlyFile("../..\\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("-----")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next

```

3. Salve o arquivo e pressione Ctrl + F5 para compilar e executar o aplicativo.

Chamar uma biblioteca de linguagem dinâmica

A instrução a seguir cria um projeto que acessa uma biblioteca que é gravada no IronPython de linguagem dinâmica.

Para criar uma classe dinâmica personalizada

1. No Visual Studio, selecione Arquivo > Novo > Projeto.
2. na caixa de diálogo **criar um novo projeto**, selecione C# ou Visual Basic, selecione **aplicativo de Console** e, em seguida, selecione **avançar**.
3. na caixa de diálogo **configurar seu novo projeto**, digite `DynamicIronPythonSample` para o nome do Project e, em seguida, selecione **avançar**.
4. Na caixa de diálogo **informações adicionais**, selecione .NET 5,0 (atual) para a estrutura de destino e, em seguida, selecione **criar**.

Quando um novo projeto é criado.

5. instale o pacote de NuGet do [IronPython](#) .
6. se você estiver usando Visual Basic, edite o arquivo *Program.vb*. Se você estiver usando o Visual C#, edite o arquivo *Program.cs* .
7. Na parte superior do arquivo, adicione o código a seguir para importar os `Microsoft.Scripting.Hosting`
`IronPython.Hosting` namespaces e das bibliotecas de IronPython e o `System.Linq` namespace.

```

using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;

```

```

Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
Imports System.Linq

```

8. 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.
```

```
// 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.");
```

```
' 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")  
Dim py = Python.CreateRuntime()  
Dim random As Object = py.UseFile("random.py")  
Console.WriteLine("random.py loaded.")
```

9. 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.

```
// 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("-----");  
}
```

```
' Initialize an enumerable set of integers.  
Dim items = Enumerable.Range(1, 7).ToArray()  
  
' Randomly shuffle the array of integers by using IronPython.  
For i = 0 To 4  
    random.shuffle(items)  
    For Each item In items  
        Console.WriteLine(item)  
    Next  
    Console.WriteLine("-----")  
Next
```

10. Salve o arquivo e pressione Ctrl + F5 para compilar e executar o aplicativo.

Confira também

- [System.Dynamic](#)
- [System.Dynamic.DynamicObject](#)
- [Usando o tipo dynamic](#)
- [Associação antecipada e tardia](#)
- [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#)

21/01/2022 • 5 minutes to read

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 ao método na classe derivada, o método de classe `override` 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`:

```
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:

```
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:

```
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`:

```
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:

```
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:

```
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:

```
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 , ele tentará corresponder a chamada a um método substituído com o mesmo nome e `Derived` parâmetros compatíveis. Por exemplo:

```
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:

```
((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#](#)
- [O sistema de tipos C#](#)
- [Métodos](#)
- [Herança](#)

Quando usar as palavras-chave override e new (Guia de Programação em C#)

21/01/2022 • 10 minutes to read

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`.

```
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.

```

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`.

```

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.

```

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.

```

// 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`.

```
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`.

```
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`.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Execute o projeto novamente. Observe principalmente as duas últimas linhas da saída a seguir.

```
// 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.

```
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`.

```
// 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.

```

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.

```

// 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`.

```

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`.

```
// 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.

```
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.

```
// 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.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

namespace overrideAndNew
{
    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#](#)
- [O sistema de tipos C#](#)
- [Controle de versão com as palavras-chave override e new](#)
- [base](#)
- [Abstrata](#)

Como substituir o método `ToString` (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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:

```
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](#).

IMPORTANT

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:

```
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.

```
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:

```
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](#)
- [Substituir](#)
- [Virtual](#)
- [Formatar tipos](#)

Membros (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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 Sobrecarga de operador .
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#)

21/01/2022 • 2 minutes to read

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:

```
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:

```
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:

```
// 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:

```
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:

```
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#)

21/01/2022 • 5 minutes to read

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 de 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:

```
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 do .NET, a classe estática contém métodos que executam operações matemáticas, sem nenhum requisito para armazenar ou recuperar dados exclusivos de uma instância específica da `System.Math` classe. Ou seja, você aplica os membros da classe especificando o nome de classe e o nome do método, conforme mostrado no exemplo a seguir.

```
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 para 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.

NOTE

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. Elas não podem herdar de qualquer classe, exceto por [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:

```
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());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;
        }
    }
}
```

```

        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. Métodos e propriedades estáticos não podem acessar campos e eventos não estáticos em seu tipo de conteúdo e não podem acessar uma variável de instância de qualquer 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, `const` os campos podem ser acessados usando a mesma `ClassName.MemberName` notação usada para campos estáticos. Nenhuma instância de objeto é necessária.

O C# não dá suporte a variáveis locais estáticas (ou seja, 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:

```
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:

```
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 no MSIL (Microsoft Intermediate Language), enquanto uma chamada para um método de instância gera uma instrução , que também verifica se há referências de objeto `callvirt` nulo. 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 e Membros estáticos e de instância](#) 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#)

21/01/2022 • 5 minutes to read

Todos os tipos e membros de tipo têm um nível de acessibilidade. O nível de acessibilidade controla se eles podem ser usados de outro código em seu assembly ou em outros assemblies. Um **assembly** é um **.dll** ou **.exe** criado compilando um ou mais arquivos **.cs** em uma única compilação. Use os seguintes modificadores de acesso 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 outro assembly que o referencia. 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 por código no mesmo **class** ou **struct**.
- **protected**: o tipo ou membro pode ser acessado somente por código no mesmo ou em um que **class** é derivado desse **class**.
- **internal**: o tipo ou membro pode ser acessado por qualquer código no mesmo assembly, mas não por outro assembly. Em outras palavras, **internal** tipos ou membros podem ser acessados do 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 dentro de um derivado **class** em outro assembly.
- **private protected**: o tipo ou membro pode ser acessado por tipos derivados do que **class** são declarados dentro de seu assembly que o contém.

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)	✓	✓	✓	✗	✗	✗
Classe não derivada (assembly diferente)	✓	✗	✗	✗	✗	✗

Os exemplos a seguir demonstram como especificar modificadores de acesso em um tipo e membro:

```
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 de seu tipo de conteúdo.

Acessibilidade de classe, registro e struct

Classes, registros e structs declarados diretamente dentro de um namespace (em outras palavras, que não estão aninhados em outras classes ou structs) podem ser `public` ou `internal`. `internal` será o padrão se nenhum modificador de acesso for especificado.

Membros de struct, incluindo classes aninhadas e structs, podem ser declarados `public` `internal` como , ou `private` . Os membros de classe, incluindo classes aninhadas e structs, podem ser `public` , , , ou `protected` `internal` `protected` `internal` `private protected` `private` . Membros de classe e struct, incluindo classes aninhadas e structs, `private` têm acesso por padrão. Tipos aninhados privados não são acessíveis de fora do tipo que o contém.

Classes derivadas e registros derivados não podem ter maior acessibilidade do que seus tipos base. Não é possível declarar uma classe pública `B` que deriva de uma classe interna `A` . Se permitido, ele teria o efeito de tornar público, porque todos os membros ou `A` `protected` de são `internal` `A` acessíveis da classe derivada.

Você pode habilitar outros assemblies específicos para acessar seus tipos internos usando o `InternalsVisibleToAttribute` . Para obter mais informações, consulte [Assemblies amigáveis](#).

Acessibilidade de membro de classe, registro e struct

Membros de classe e registro (incluindo classes aninhadas, registros e structs) podem ser declarados com qualquer um dos seis tipos de acesso. Os membros do struct não podem ser declarados como , ou porque `protected` `protected internal` OS `private protected` structs não são suportados por herança.

Normalmente, a acessibilidade de um membro não é maior que a acessibilidade do tipo que o contém. No entanto, um membro de uma classe interna poderá ser acessível de fora do assembly se o membro implementar métodos de interface ou substituir métodos virtuais que são definidos em uma `public` classe base pública.

O tipo de qualquer campo de membro, propriedade ou evento 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` que retorna `M` uma classe, a menos que `C` também seja `C` `public` . Da mesma forma, você não poderá ter uma `protected` propriedade do tipo se for declarada como `A` `A` `private` .

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 , ou , adicione a palavra-chave apropriada à declaração de membro, conforme `class` mostrado no exemplo a `record` `struct` seguir.

```
// 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

Interfaces declaradas diretamente dentro de um namespace podem ser ou e, assim como classes e `public` `internal` structs, as interfaces são padrão para `internal` acessar. Os membros da interface são, por padrão, porque a finalidade de uma interface é permitir que `public` outros tipos acessem uma classe ou struct. As declarações de membro da interface podem incluir qualquer modificador de acesso. Isso é mais útil para métodos estáticos fornecerem implementações comuns necessárias para todos os implementores de uma classe.

Os membros de enumeração são `public` sempre e nenhum modificador de acesso pode ser aplicado.

Delegados se comportam como classes e structs. Por padrão, eles têm `internal` acesso quando declarados diretamente em um namespace e `private` acesso 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

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Interfaces](#)
- [Privada](#)
- [public](#)
- [interno](#)
- [protected](#)
- [internos protegidos](#)
- [privado protegido](#)
- [class](#)
- [Struct](#)
- [interface](#)

Campos (Guia de Programação em C#)

21/01/2022 • 4 minutes to read

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 struct pode ter campos de instância, campos 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 pode acessar o campo estático somente usando o nome do tipo. Se você acessar o campo estático por um nome de instância, obterá um erro de tempo de compilação [CS0176](#).

Em geral, você só deve usar campos para variáveis que têm acessibilidade particular ou protegida. Os dados que seu tipo expõe para o código do 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*.

Normalmente, os campos armazenam os dados que devem ser acessíveis a mais de um método de tipo e devem ser armazenados por mais tempo do que o tempo de vida de qualquer método único. Por exemplo, um tipo que representa uma data de calendário pode ter três campos de inteiro: 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 Class ou struct especificando o nível de acesso do campo, seguido pelo tipo do campo, seguido pelo nome do campo. Por exemplo:

```

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();
            }
        }
    }

    // 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();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}

```

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:

```
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:

```
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](#).

NOTE

Um inicializador de campo não pode fazer referência a outros campos de instância.

Os campos podem ser marcados como [público](#), [privado](#), [protegido](#), [interno](#), [protegido interno](#) ou [protegido privado](#). 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](#). Isso torna o campo disponível para os chamadores a qualquer momento, mesmo se nenhuma instância do tipo existir. Para obter mais informações, consulte [classes estáticas e membros de classe estática](#).

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 muito 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](#).

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 C#](#)
- [O sistema de tipos C#](#)
- [Usando construtores](#)
- [Herança](#)
- [Modificadores de acesso](#)
- [Classes e membros de classes abstract e sealed](#)

Constantes (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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`. Somente os tipos integrados 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, struct ou matriz que é inicializada uma vez em tempo de run (por exemplo, em um construtor) e, depois disso, não pode ser alterada.

O C# não dá suporte aos métodos `const`, propriedades ou eventos.

O tipo de enumeração permite que você defina constantes nomeadas para tipos internos integrais (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:

```
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.

NOTE

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:

```
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:

```
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:

```
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 abstratas (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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`.

```

// 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:

```
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.

```

// 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 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.

```
// 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 de tipos C#](#)
- [Classes e membros de classes abstract e sealed](#)
- [Propriedades](#)

Como definir constantes em C#

21/01/2022 • 2 minutes to read

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.

NOTE

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

```
using System;

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

- [O sistema de tipos C#](#)

Propriedades (Guia de Programação em C#)

21/01/2022 • 5 minutes to read

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 na verdade elas são métodos realmente especiais chamados *acessadores*. Isso 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 posterior, 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 `set` `init` acessador ou .
- 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.

```

using System;

class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(
                    $"{nameof(value)} must be between 0 and 24.");
            _seconds = value * 3600;
        }
    }
}

class Program
{
    static void Main()
    {
        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.

A partir do C#6, as 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.

```
using System;

public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
    public static void Main()
    {
        var person = new Person("Magnus", "Hedlund");
        Console.WriteLine(person.Name);
    }
}

// The example displays the following output:
//      Magnus Hedlund
```

Começando com o 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 estar presentes. O exemplo a seguir ilustra o uso de definições de corpo de expressão para ambos os acessadores. Observe que a palavra-chave `return` não é usada com o acessador `get`.

```

using System;

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;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem("Shoes", 19.95m);
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95

```

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 adicional. 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 um `get` e um `set` (ou um `get` um `set`) `get` `set` `get` `init` acessador, ambos deverão ser implementados automaticamente. 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, para que os objetos agora sejam inicializados com uma chamada para o construtor sem parâmetros e um [SaleItem inicializador de objeto](#).

```
using System;

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

    public decimal Price
    { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95
```

Seções relacionadas

- [Usando propriedades](#)
- [Propriedades da 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

- [Guia de Programação em C#](#)
- [Usando propriedades](#)
- [Indexadores](#)
- [Palavra-chave get](#)
- [Palavra-chave set](#)

Usando propriedades (Guia de Programação em C#)

21/01/2022 • 8 minutes to read

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 novo 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 posterior, você pode usar um `init` acessador em vez de um `set` acessador 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 realmente 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:

```
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 conhecido, em geral, como o "repositório de backup" da propriedade. É comum as propriedades usarem campos particulares como um 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. A seguir está um acessador `get` que retorna o valor de um campo particular `_name`:

```
class Person
{
    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:

```
Person person = new Person();
//...
System.Console.WriteLine(person.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.

Alterar o estado do objeto usando o acessador `get` é um estilo ruim de programação. Por exemplo, o acessador a seguir produz o efeito colateral de alterar o estado do objeto sempre que o campo `_number` é acessado.

```
private int _number;
public int Number => _number++; // Don't do this
```

O acessador `get` pode ser usado para retornar o valor do campo ou para calculá-lo e retorná-lo. Por exemplo:

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

No segmento de código anterior, se você não atribuir um valor à `Name` propriedade, ele 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`:

```
class Person
{
    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:

```
Person person = new Person();
person.Name = "Joe"; // the set accessor is invoked here

System.Console.WriteLine(person.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 `init` acessador é o mesmo que o código para criar um `set` acessador, exceto que você usa a `init` palavra-chave em vez de `set`. A diferença é que o `init` acessador só pode ser usado 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`. Isso torna a propriedade disponível para chamadores a qualquer momento, mesmo se não existir nenhuma instância da classe. Para obter mais informações, consulte [classes estáticas e membros de classe estática](#).

Uma propriedade pode ser marcada como uma propriedade virtual, usando a palavra-chave `virtual`. Isso habilita as classes derivadas a substituírem 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`. Isso significa que não há 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](#).

NOTE

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.

```
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:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
   Employee number: 108
   Employee name: Claude Vige
*/
```

Exemplo da Propriedade Hidden

Este exemplo demonstra como acessar uma propriedade em uma classe base que é ocultada por outra propriedade que tem o mesmo nome em uma classe derivada:

```

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
{
    static void Main()
    {
        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:

```
public new string Name
```

- A conversão `(Employee)` é usada para acessar a propriedade oculta na classe base:

```
((Employee)m1).Name = "Mary";
```

Para obter mais informações sobre como ocultar membros, consulte o [Modificador new](#).

Exemplo da Propriedade Override

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.

```
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.WriteLine("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.WriteLine("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

- [Guia de programação C#](#)
- [Propriedades](#)
- [Propriedades de interface](#)
- [Propriedades autoimplementadas](#)

Propriedades de interface (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

As propriedades podem ser declaradas em uma [interface](#). O exemplo a seguir declara um acessador de propriedade de interface:

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Normalmente, as propriedades de interface não têm um corpo. Os acessadores indicam se a propriedade é de leitura/gravação, somente leitura ou somente gravação. Ao contrário de classes e structs, a declaração dos acessadores sem um corpo não declara uma [propriedade implementada automaticamente](#). A partir do C# 8.0, 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 não podem 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:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

O exemplo anterior demonstra a [implementação de interface explícita](#). 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:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implementa a propriedade `Name` na interface `IEmployee`, enquanto a seguinte declaração:

```

string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}

```

implementa a propriedade `Name` na interface `ICitizen`.

```

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;
}

```

```

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

```
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 C#](#)
- [Propriedades](#)
- [Usando propriedades](#)
- [Comparação entre propriedades e indexadores](#)
- [Indexadores](#)
- [Interfaces](#)

Restringindo a acessibilidade ao acessador (Guia de Programação em C#)

21/01/2022 • 4 minutes to read

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, isso envolve restringir a acessibilidade do acessador `set` e manter o acessador `get` publicamente acessível. Por exemplo:

```
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 `protégido` ao acessador em si.

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 os modificadores de acessador em uma interface ou em uma implementação de membro de `interface` explícita.
- 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:

```

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:

```

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 é 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` no `DerivedClass` impede o acesso ao acessador e gera um erro quando você atribui a ele.

```
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

- [Guia de Programação em C#](#)
- [Propriedades](#)
- [Indexadores](#)
- [Modificadores de acesso](#)

Como declarar e usar propriedades de gravação de leitura (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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. Isso é feito por meio de *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

```
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;
    }
}

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`. Isso permite que qualquer objeto leia e grave essas propriedades. No entanto, às vezes é desejável excluir um ou os acessadores. Omitir o acessador `set`, por exemplo, torna a propriedade somente leitura:

```

public string Name
{
    get
    {
        return _name;
    }
    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. Isso permite uma sintaxe muito natural na obtenção e configuração do valor de uma propriedade, conforme as instruções a seguir:

```

person.Name = "Joe";
person.Age = 99;

```

Observe que em um método de propriedade `set`, uma variável especial `value` está disponível. Essa variável contém o valor que o usuário especificou, por exemplo:

```

_name = value;

```

Observe a sintaxe normal para incrementar a propriedade `Age` em um objeto `Person`:

```
person.Age += 1;
```

Se métodos `set` e `get` separados fossem usados para modelar propriedades, o código equivalente se pareceria com isto:

```
person.SetAge(person.GetAge() + 1);
```

O método `ToString` é substituído neste exemplo:

```
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

- [Guia de Programação em C#](#)
- [Propriedades](#)
- [O sistema de tipos C#](#)

Propriedades autoimplementadas (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

No C# 3.0 e versões posteriores, 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. No C# 9 e posterior, os acessadores também podem ser `init` declarados como propriedades auto-implementadas.

Exemplo

O exemplo a seguir mostra uma classe simples que tem algumas propriedades autoimplementadas:

```
// This class is mutable. Its data can be modified from
// outside the class.
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;
    }
}
```

Não é possível declarar propriedades auto-implementadas em interfaces. As propriedades auto-implementadas declaram um campo de backing 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.

No C# 6 e versões posteriores, é possível inicializar as propriedades autoimplementadas da mesma forma que os campos:

```
public string FirstName { get; set; } = "Jane";
```

A classe mostrada no exemplo anterior é mutável. O código do cliente pode alterar os valores em objetos após a criação. Em classes complexas que contêm comportamentos significativos (métodos), bem como dados, geralmente é necessário ter propriedades públicas. No entanto, para classes pequenas ou structs que apenas encapsulam 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 `get` acessador (imutável em todos os lugares, exceto o construtor).
- Declare um `get` acessador e um `init` acessador (imutável em todos os lugares, exceto durante a construção do objeto).
- Declare o `set` acessador como `privado` (imutável para consumidores).

Para obter mais informações, [consulte Como implementar uma classe leve com propriedades auto-implementadas](#).

Confira também

- [Propriedades](#)
- [Modificadores](#)

Como implementar uma classe leve com propriedades autoimplementadas (guia de programação C#)

21/01/2022 • 3 minutes to read

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 fazer uma propriedade imutável das seguintes maneiras:

- Declare somente o acessador `Get`, que torna a propriedade imutável em qualquer lugar, exceto no construtor do tipo.
- Declare um acessador `init` em vez de um `set` acessador, o que torna a propriedade configurável somente no construtor ou usando um [inicializador de objeto](#).
- Declare o acessador `set` como [particular](#). A propriedade é configurável dentro do tipo, mas é imutável para os consumidores.

Ao declarar um acessador privado `set`, não é possível usar um inicializador de objeto para inicializar a propriedade. É necessário usar um construtor ou um método de fábrica.

O exemplo a seguir mostra como uma propriedade com somente acessador `get` difere de uma com `Get` e `Private set`.

```
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.

```
// 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#)

21/01/2022 • 10 minutes to read

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 `Main` método é o ponto de entrada para cada aplicativo C# e é chamado pelo Common Language Runtime (CLR) 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.

NOTE

Este artigo discute os métodos nomeados. Para obter 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 de retorno, o nome do método e qualquer parâmetro de método. Juntas, essas partes são a assinatura do método.

IMPORTANT

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:

```
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 de 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:

```

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 do método vs. 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 houver) usado no código de chamada não precisa ser o mesmo que o parâmetro nomeado definido no método. Por exemplo:

```

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 vs. 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 efeito sobre a instância original no método de chamada. Para passar uma instância de tipo de valor por referência, use a `ref` palavra-chave. 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.

Você cria um tipo de referência usando a `class` palavra-chave, como mostra o exemplo a seguir:

```
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` para o método `ModifyObject`:

```
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, começando com o C# 7.0, [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 assinatura de método e a instrução de retorno a seguir indicam que o método retorna uma variável chamada `estDistance` por referência ao chamador.

```
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:

```
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:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
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:

```
ref int distance = 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`, qualquer alteração no conteúdo da matriz é observável por qualquer código que tenha uma referência à matriz, conforme mostrado no exemplo a seguir:

```

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.

NOTE

Um método assíncrono retorna ao chamador quando encontra o primeiro objeto esperado que ainda não está concluído ou chega ao final do método `Async`, o que ocorre primeiro.

Um método assíncrono normalmente tem um tipo de retorno `Task<TResult>` de `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. Começando com o C# 7.0, um método assíncrono pode ter [qualquer tipo de retorno como os de tarefa](#).

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 `Main` método é um exemplo de um método assíncrono que tem um tipo de retorno de `Task`. Ele vai para o `DoSomethingAsync` método e, como é expresso com uma única linha, ele pode omitir as `async` e `await` palavras-chave. 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();`.

```

using System;
using System.Threading.Tasks;

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 métodos assíncronos, consulte [programação assíncrona com tipos de retorno Async e Await e 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 `=>`:

```

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 alcançada, 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>](#), [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 C#](#)
- [O sistema de tipos C#](#)
- [Modificadores de acesso](#)
- [Classes static e membros de classes static](#)
- [Herança](#)
- [Classes e membros de classes abstract e sealed](#)
- [params](#)
- [out](#)
- [ref](#)
- [Passando parâmetros](#)

Funções locais (Guia de Programação em C#)

21/01/2022 • 10 minutes to read

Começando com o C# 7.0, o C# é compatível com *funções locais*. Funções locais são métodos privados 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.

NOTE

Em alguns casos, você pode usar uma expressão lambda para implementar uma funcionalidade que também tem suporte por uma função local. Para uma comparação, consulte [Funções locais versus expressões lambda](#).

Funções locais tornam a intenção do seu código clara. Qualquer pessoa que lê seu código pode 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:

```
<modifiers> <return-type> <method-name> <parameter-list>
```

Você pode usar os seguintes modificadores com uma função local:

- `async`
- `unsafe`
- `static` (no C# 8.0 e posterior). Uma função local estática não pode capturar variáveis locais ou estado de instância.
- `extern` (no C# 9.0 e posterior). Uma função local externa deve ser `static`.

Todas as variáveis locais definidas no membro que o contém, incluindo seus parâmetros de 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`:

```
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:

```
#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 que pode ser anulado.

Funções locais e exceções

Um dos recursos úteis de funções locais é que elas podem permitir que exceções sejam apresentadas imediatamente. Para iteradores de método, 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 `OddSequence` método 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.

```

using System;
using System.Collections.Generic;

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 lançadas quando você recuperar o enumerador, como mostra o exemplo a seguir:

```

using System;
using System.Collections.Generic;

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();
    }

    IEnumerable<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 factorial. Esta é a versão usando uma função local:

```

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:

```

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 nomeadas explicitamente como métodos. As expressões lambda são métodos anônimos e precisam ser atribuídas a variáveis de um tipo, normalmente `Action` ou `Func` tipos. Quando você declara uma função local, o processo é como escrever 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 que elas são `Action` / `Func` atribuídas para determinar os tipos de argumento e retorno. Em funções locais, como a sintaxe é muito semelhante à escrita de um método normal, os tipos de argumento e o tipo de retorno já fazem parte da declaração da função.

A partir do C# 10, algumas expressões lambda têm um tipo *natural*, que permite ao compilador inferir o tipo de retorno e os tipos de parâmetro da expressão lambda.

Atribuição definida

Expressões lambda são objetos que são declarados e atribuídos em tempo de execução. Para que uma expressão lambda seja usada, ela precisa ser definitivamente atribuída: a variável à que ela será atribuída deve ser declarada e a expressão lambda atribuída a `Action` / `Func` dela. 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, elas podem ser referenciadas de qualquer local de código **em que ele está no escopo**; em nosso primeiro exemplo, poderíamos declarar nossa função local acima ou abaixo da instrução e não disparar `LocalFunctionFactorial` `return` 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 refere a mesma expressão lambda.

Implementação como delegado

As expressões lambda são convertidas em delegados quando são declaradas. As funções locais são mais flexíveis, pois podem ser escritas como um método tradicional *ou* como um delegado. As funções locais só são convertidas em delegados ***quando usadas*** como um 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 definida](#) também afetam todas as variáveis capturadas pela função local ou pela expressão lambda. O compilador pode executar uma análise estática que permite que funções locais atribuam definitivamente variáveis capturadas no escopo delimitador. Considere este exemplo:

```

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 delimitante, a função local é implementada como um tipo 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 for convertida em um delegado e nenhuma das variáveis capturadas pela função local for capturada por outras lambdas ou funções locais convertidas em delegados, o compilador poderá evitar alocações de heap.

Considere este exemplo assíncrono:

```

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 do que a versão da expressão lambda.

Se você sabe que sua função local não será convertida em um delegado e nenhuma das variáveis capturadas por ela for capturada por outras lambdas ou funções locais convertidas em delegados, você poderá garantir que sua função local evite ser alocada no heap declarando-a como uma `static` função local. Observe que esse recurso está disponível no C# 8.0 e mais novo.

NOTE

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`.

```
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 `yield` palavra-chave

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.

```
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 `yield return` instrução não é permitida em expressões lambda, consulte [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

- [Métodos](#)

Ref returns e ref locals

21/01/2022 • 7 minutes to read

Começando com C# 7.0, C# dá suporte a valores de referência de devolução. 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.

O que é um valor retornado por referência?

A maioria dos desenvolvedores estão familiarizados com passar um argumento para um método chamado *por referência*. A lista de argumentos de um método chamado inclui uma variável passada por referência. Todas as alterações feitas pelo método chamado ao valor são observadas pelo chamador. 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 muitas vezes é que o código de chamada deve ter acesso a essa variável por meio do alias, inclusive para modificá-lo. Por conseguinte, métodos retornados por referência não podem ter o tipo de retorno `void`.

Há algumas restrições quanto à expressão que um método pode retornar como um valor retornado por referência. As restrições incluem:

- 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. Retornar `null` gera o erro do compilador CS8156, "Uma expressão não pode ser usada neste contexto porque ela não pode ser retornada por referência."

Um método com um retorno ref pode retornar um alias para uma variável cujo valor é atualmente o valor nulo (não imprestável) ou um tipo de valor que pode ser anulado 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`. Violar essa regra gera o erro do compilador CS8156, "Uma expressão não pode ser usada neste contexto porque ela não pode ser retornada por referência."

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.

Definindo um valor retornado ref

Um método que retorna um *valor retornado de referência* deve satisfazer as duas condições a seguir:

- A assinatura do método inclui 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`:

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Consumindo um valor retornado ref

O valor retornado de `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.

Ref locals

Suponha que o método `GetContactInformation` seja declarado como uma referência de retorno:

```
public ref Person GetContactInformation(string fname, string lname)
```

Uma atribuição por valor lê o valor de uma variável e o atribui a uma nova variável:

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

A atribuição anterior declara `p` como uma variável local. O valor inicial é copiado da leitura do valor retornado por `GetContactInformation`. As atribuições futuras para `p` não alterarão o valor da variável retornada por `GetContactInformation`. A variável `p` não é um alias para a variável retornada.

Você declara uma variável *ref/local* para copiar o alias para o valor original. Na atribuição de seguir, `p` é um alias para a variável retornada de `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

O uso subsequente de `p` é o mesmo que usar a variável retornada pelo `GetContactInformation` porque `p` é um alias dessa variável. As alterações em `p` também alteram a variável retornada de `GetContactInformation`.

A palavra-chave `ref` é usada antes da declaração de variável local *e* antes da chamada de método.

Você pode acessar um valor por referência da mesma maneira. Em alguns casos, acessar um valor por referência aumenta o desempenho, evitando uma operação de cópia potencialmente dispendiosa. Por exemplo, a instrução a seguir mostra como é possível definir um valor de local de `ref` que é usado para fazer referência a um valor.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

A palavra-chave `ref` é usada antes da declaração da variável local *e* antes do valor, no segundo exemplo. A

falha ao incluir as palavras-chave `ref` na declaração e na atribuição da variável nos dois exemplos resulta no erro do compilador CS8172, "Não é possível inicializar uma variável por referência com um valor."

Antes do C# 7.3, variáveis locais de referência não podiam ser reatribuídas para se referir a um armazenamento diferente depois de serem inicializadas. Essa restrição foi removida. O exemplo a seguir mostra uma transferência:

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Variáveis locais de referência ainda devem ser inicializadas quando são declaradas.

Ref returns e ref locals: um exemplo

O exemplo a seguir define uma classe `NumberStore` que armazena uma matriz de valores inteiros. O método `FindNumber` retorna por referência o primeiro número maior ou igual ao número passado como um argumento. Se nenhum número for maior ou igual ao argumento, o método retornará o número no índice 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

A exemplo a seguir chama o método `NumberStore.FindNumber` para recuperar o primeiro valor maior ou igual a 16. O chamador então dobra o valor retornado pelo método. A saída do exemplo mostra a alteração refletida no valor dos elementos de matriz da instância `NumberStore`.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence: {store.ToString()}");
// The example displays the following output:
//      Original sequence: 1 3 7 15 31 63 127 255 511 1023
//      New sequence: 1 3 7 15 62 63 127 255 511 1023
```

Sem suporte para valores retornados por referência, essa operação é executada retornando-se o índice do elemento de matriz, juntamente com o respectivo valor. O chamador pode usar esse índice para modificar o valor em uma chamada de método separada. No entanto, o chamador também pode modificar o índice para acessar e possivelmente modificar outros valores de matriz.

A exemplo a seguir mostra como o método `FindNumber` poderia ser reescrito após o C# 7.3 para usar a transferência de local de referência:

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && (numbers[ctr] >= target))
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

Essa segunda versão é mais eficiente com sequências mais longas em cenários em que o número procurado está mais próximo ao final da matriz, pois a matriz é iterada do fim até o início, fazendo com que menos itens sejam examinados.

Confira também

- [ref keyword](#)
- [Escrever código seguro e eficiente](#)

Passando parâmetros (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

No C#, argumentos podem ser passados para parâmetros por valor ou por referência. A passagem por referência permite que métodos, propriedades, indexadores, operadores, construtores e membros da função alterem o valor dos parâmetros e façam essa alteração persistir no ambiente de chamada. Para passar um parâmetro por referência com a intenção de alterar o valor, use a palavra-chave `ref` ou `out`. Para passar por referência com a intenção de evitar a cópia, mas não alterar o valor, use o modificador `in`. Para simplificar, somente a palavra-chave `ref` é usada nos exemplos neste tópico. Para obter mais informações sobre a diferença entre `in`, `ref` e `out`, consulte [in, ref e out](#).

O exemplo a seguir ilustra a diferença entre parâmetros de valor e referência.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

Para obter mais informações, consulte estes tópicos:

- [Passando parâmetros de tipo de valor](#)
- [Passando parâmetros de tipo de referência](#)

Especificação da Linguagem C#

Para obter mais informações, veja as [listas de argumentos](#) 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#](#)
- [Métodos](#)

Passando parâmetros de tipo de valor (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

Uma variável de [tipo de valor](#) contém seus dados diretamente, o que não acontece com uma variável de [tipo de referência](#), que contém uma referência a seus dados. Passar uma variável de tipo de valor para um método por valor significa passar uma cópia da variável para o método. As alterações no parâmetro que ocorrem dentro do método não têm efeito sobre os dados originais armazenados na variável de argumento. Se desejar que o método chamado altere o valor do argumento, é necessário passá-lo por referência, usando a palavra-chave [ref](#) ou [out](#). Você também pode usar a palavra-chave [in](#) para passar um parâmetro de valor por referência, para evitar a cópia e ainda garantir que o valor não seja alterado. Para simplificar, os exemplos a seguir usam [ref](#).

Passando tipos de valor por valor

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`. As alterações que ocorrem dentro do método não têm efeito sobre o valor original da variável.

```
class PassingValByVal
{
    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);
    }
    static void Main()
    {
        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();
    }
}
/* 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`.

Passando tipos de valor por referência

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.

```
class PassingValByRef
{
    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);
    }
    static void Main()
    {
        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();
    }
}
/* 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`.

Trocar valores de tipo

Um exemplo comum de alteração dos valores de argumentos é um método de troca, em que você passa duas variáveis para o método e o método troca seu conteúdo. É necessário passar os argumentos para o método de troca por referência. Caso contrário, você troca cópias locais dos parâmetros dentro do método e nenhuma alteração ocorre no método de chamada. O exemplo a seguir troca valores inteiros.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Quando você chama o método `SwapByRef`, use a palavra-chave `ref` na chamada, conforme mostrado no exemplo a seguir.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

/* Output:
   i = 2  j = 3
   i = 3  j = 2
*/
```

Confira também

- [Guia de Programação em C#](#)
- [Passando parâmetros](#)
- [Passando parâmetros de tipo de referência](#)

Passando parâmetros de tipo de referência (Guia de Programação em C#)

21/01/2022 • 4 minutes to read

Uma variável de um [tipo de referência](#) não contém seus dados diretamente; ela contém uma referência a seus dados. Quando você passa um parâmetro de tipo de referência por valor, é possível alterar os dados que pertencem ao objeto referenciado, como o valor de um membro de classe. No entanto, não é possível alterar o valor da referência em si. Por exemplo, não é possível usar a mesma referência para alocar memória para um novo objeto e fazer com que ele persista fora do bloco. Para fazer isso, passe o parâmetro usando a palavra-chave `ref` ou `out`. Para simplificar, os exemplos a seguir usam `ref`.

Passando tipos de referência por valor

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`.

```
class PassingRefByVal
{
    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]);
    }

    static void Main()
    {
        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]);
    }
}
/* 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`.

Passando tipos de referência por referência

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.

```
class PassingRefByRef
{
    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]);
    }

    static void Main()
    {
        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]);
    }
}
/* 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`.

Trocando duas cadeias de caracteres

Trocar cadeias de caracteres é um bom exemplo de passar parâmetros de tipo de referência por referência. No exemplo, duas cadeias de caracteres, `str1` e `str2`, são inicializadas em `Main` e passadas para o método `SwapStrings` como parâmetros modificados pela palavra-chave `ref`. As duas cadeias de caracteres são trocadas dentro do método e dentro de `Main` também.

```

class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
        // The string parameter is passed by reference.
        // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);    // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
   Inside Main, before swapping: John Smith
   Inside the method: Smith John
   Inside Main, after swapping: Smith John
*/

```

Neste exemplo, os parâmetros precisam ser passados por referência para afetar as variáveis no programa de chamada. Se você remover a palavra-chave `ref` do cabeçalho de método e da chamada de método, nenhuma alteração ocorrerá no programa de chamada.

Para obter mais informações sobre cadeias de caracteres, consulte [cadeia de caracteres](#).

Confira também

- [Guia de Programação em C#](#)
- [Passando parâmetros](#)
- [ref](#)
- [Em](#)
- [out](#)
- [Tipos de referência](#)

Como saber a diferença entre passar uma struct e passar uma referência de classe para um método (guia de programação C#)

21/01/2022 • 2 minutes to read

O exemplo a seguir demonstra como passar um [struct](#) para um método difere de passar uma instância de [classe](#) para um método. No exemplo, ambos os argumentos (struct e instância de classe) são passados por valor e ambos os métodos alteram o valor de um campo do argumento. No entanto, os resultados dos dois métodos não são os mesmos, pois o que é passado ao passar um struct é diferente do que é passado ao passar uma instância de uma classe.

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 da classe que o endereço original e a cópia da referência do endereço. 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.

Exemplo

```

using System;

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";
    }

    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
*/

```

Confira também

- [Guia de programação C#](#)
- [Classes](#)
- [Tipos de estrutura](#)
- [Passando parâmetros](#)

Variáveis locais de tipo implícito (Guia de Programação em C#)

21/01/2022 • 5 minutes to read

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 integrado, um tipo anônimo, um tipo definido pelo usuário ou um tipo definido na biblioteca de classes do .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`:

```
// 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`.

```
for (var x = 1; x < 10; x++)
```

- Em uma instrução de inicialização `foreach`.

```
foreach (var item in list) {...}
```

- Em uma instrução `using`.

```
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`.

```
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 legal: `int i = (i = 20);` mas essa expressão produz 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:

```
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):

```
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.

```
// 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 ajuda a simplificar seu código, mas seu uso deve ser restrito aos casos em que ele é necessário ou quando torna seu código mais fácil de ler. Para obter mais informações sobre quando usar `var` corretamente, consulte a seção [Variáveis locais de tipo implícito](#) no artigo Diretrizes de codificação em 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 e matrizes de tipo implícito em uma expressão de consulta (guia de programação C#)

21/01/2022 • 2 minutes to read

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](#).

```
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.

```
// 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 C#](#)
- [Métodos de Extensão](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [var](#)
- [LINQ em C#](#)

Métodos de extensão (Guia de Programação em C#)

21/01/2022 • 10 minutes to read

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. Para o código de cliente escrito 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 do LINQ que adicionam a funcionalidade de consulta aos [System.Collections.IEnumerable](#) tipos existentes e [System.Collections.Generic.IEnumerable<T>](#). 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. Muitos operadores de consulta padrão usam expressões lambda como parâmetros, mas isso não é um requisito para métodos de extensão. Para obter mais informações, consulte [Expressões Lambda](#).

```
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 por [este](#) modificador. 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](#). Ele é definido dentro de uma classe estática não aninhada não genérica:

```
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` :

```
using ExtensionMethods;
```

E pode ser chamado a partir de um aplicativo usando esta sintaxe:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Você invoca 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 do encapsulamento não está realmente violado. Os métodos de extensão não podem acessar variáveis privadas no tipo que estão sendo estendidas.

A `MyExtensions` classe e o `WordCount` método são `static`, e podem ser acessados como todos os outros `static` Membros. O `WordCount` método pode ser invocado como outros `static` métodos da seguinte maneira:

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

O código C# anterior:

- Declara e atribui um novo `string` nome `s` com um valor de `"Hello Extension Methods"` .
- Chama o `MyExtensions.WordCount` argumento dado `s`

Para obter mais informações, consulte [como implementar e chamar um método de extensão personalizado](#).

Em geral, você provavelmente estará chamando métodos de extensão muito mais frequentemente do que implementar 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:

```
using System.Linq;
```

(Você também pode precisar 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 `IEnumerable<T>` tipos.

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 correspondente, ele se associará a um método de extensão correspondente, se houver.

```
// 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 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 de uso comuns

Funcionalidade de coleção

No passado, era comum criar "classes de coleção" que implementavam a `System.Collections.Generic.IEnumerable<T>` interface para um determinado tipo e uma funcionalidade contida que atuava em coleções desse tipo. Embora não haja nada de errado ao criar esse tipo de objeto de coleção, a mesma funcionalidade pode ser obtida usando uma extensão no `System.Collections.Generic.IEnumerable<T>`. As extensões têm a vantagem de permitir que a funcionalidade seja chamada de qualquer coleção, como uma `System.Array` ou `System.Collections.Generic.List<T>` implementada `System.Collections.Generic.IEnumerable<T>` nesse tipo. Um exemplo disso é usar uma matriz de `Int32`, que pode ser encontrada [anteriormente neste artigo](#).

Layer-Specific funcionalidade

Ao usar uma arquitetura de cebola ou outro design de aplicativo em camadas, é comum ter um conjunto de entidades de domínio ou Transferência de Dados objetos que podem ser usados para se comunicar entre os limites do aplicativo. Esses objetos geralmente não contêm nenhuma funcionalidade ou apenas funcionalidade mínima que se aplica a todas as camadas do aplicativo. Os métodos de extensão podem ser usados para adicionar funcionalidade específica a cada camada de aplicativo sem carregar o objeto com métodos não necessários ou desejados em outras camadas.

```

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 novos 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 `Engine` classe or `Query` para fazer o trabalho de executar uma consulta em um SQL Server que pode ser chamado de vários locais em nosso código. No entanto, podemos estender a `System.Data.SqlClient.SqlConnection` classe usando métodos de extensão para executar essa consulta em

qualquer lugar em que tenhamos uma conexão com um SQL Server. Outros exemplos podem ser adicionar funcionalidade comum à [System.String](#) classe, estender os recursos de processamento de dados dos [System.IO.File](#) objetos e e [System.IO.Stream](#) [System.Exception](#) objetos para a funcionalidade de tratamento de erros específica. Esses tipos de casos de uso são limitados apenas por sua imaginação e bom sentido.

A extensão de tipos predefinidos pode ser difícil com `struct` tipos porque eles são passados por valor para métodos. Isso significa que qualquer alteração na estrutura é feita em uma cópia da estrutura. Essas alterações não serão visíveis depois que o método de extensão sair. A partir do C# 7,2, você pode adicionar o `ref` modificador ao primeiro argumento de um método de extensão. Adicionar o `ref` modificador significa que o primeiro argumento é passado por referência. Isso permite que você escreva métodos de extensão que alteram o estado da estrutura que está sendo estendida.

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 a criação de funcionalidade reutilizável em todo o ecossistema do .NET. Para as ocasiões em que a fonte original não está sob seu controle, quando um objeto derivado é inadequado ou impossível, ou quando a funcionalidade não deve ser exposta além do escopo aplicável, os métodos de extensão são uma opção excelente.

Para obter mais informações sobre tipos derivados, consulte [herança](#).

Ao usar um método de extensão para estender um tipo cujo código-fonte você não está controlando, você corre o risco de que uma alteração na implementação do tipo cause a interrupção 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 contêm métodos de extensão em um único namespace chamado `Extensions`, elas serão colocadas no escopo pela `using Extensions;` diretiva.

Para uma biblioteca de classes que você implemente, não use métodos de extensão para evitar incrementar o número de versão de um assembly. Se você quiser adicionar uma funcionalidade significativa a uma biblioteca para a qual você possui o código-fonte, siga as diretrizes do .NET para o controle de versão do assembly. Para obter mais informações, consulte [Controle de versão do assembly](#).

Confira também

- [Guia de programação 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#)

21/01/2022 • 2 minutes to read

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`.

```

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 C#)

21/01/2022 • 2 minutes to read

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.

```
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 C#](#)
- [Métodos de Extensão](#)

Argumentos nomeados e opcionais (Guia de Programação em C#)

21/01/2022 • 8 minutes to read

O C# 4 apresenta argumentos nomeados e opcionais. *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 fornece argumentos para parâmetros selecionados. Essa funcionalidade facilita muito as chamadas para interfaces COM, como as APIs Microsoft Office Automação.

Argumentos nomeados

Os argumentos nomeados o liberam da correspondência da ordem dos parâmetros nas listas de parâmetros de métodos chamados. O parâmetro para cada argumento pode ser especificado pelo nome do parâmetro. Por exemplo, uma função que imprime detalhes do pedido (como nome do vendedor, número do pedido & nome do produto) pode ser chamada enviando argumentos por posição, na ordem definida pela função.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Se você não se lembrar da ordem dos parâmetros, mas conhecer seus nomes, poderá enviar os argumentos em qualquer ordem.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Os argumentos nomeados também melhoraram a legibilidade do código identificando o que cada argumento representa. No método de exemplo abaixo, o `sellerName` não pode ser nulo ou 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,

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- *começando com o C# 7.2*, sejam usados na posição correta. No exemplo a seguir, o parâmetro `orderNum` está na posição correta, mas não está explicitamente nomeado.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Argumentos posicionais que seguem quaisquer argumentos nomeados fora de ordem são inválidos.

```
// 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.

```
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");      // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                      // C# 7.2 onwards

        // 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 lacunas separadas por vírgulas na lista de argumentos. Por exemplo, no código a seguir, método de instância

`ExampleMethod` está definido com um parâmetro obrigatório e dois opcionais.

```
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.

```
//anExample.ExampleMethod(3, ,4);
```

No entanto, se você souber o nome do terceiro parâmetro, poderá usar um argumento nomeado para realizar a tarefa.

```
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])
```

NOTE

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.

```
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( One , 4),
// 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 parâmetros opcionais não são aplicados corretamente. O primeiro ilustra que um argumento deve ser fornecido para o primeiro parâmetro, que é necessário.

Interfaces COM

Argumentos nomeados e opcionais, juntamente com o suporte para objetos dinâmicos, melhoraram muito a interoperabilidade com APIs COM, como apls de Office Automação.

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 C# 3.0 e versões anteriores, é necessário um argumento para cada parâmetro, como mostrado no exemplo a seguir.

```

// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);

```

No entanto, você pode simplificar muito a chamada para `AutoFormat` usando argumentos nomeados e opcionais, introduzidos no C# 4.0. Argumentos nomeados e opcionais permitem omitir o argumento para um parâmetro opcional se você não quiser alterar o valor padrão do parâmetro. Na chamada a seguir, um valor é especificado para apenas um dos sete parâmetros.

```

// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );

```

Para obter mais informações e exemplos, consulte [Como usar argumentos nomeados e opcionais em programação Office](#) e [Como acessar objetos de Office de interop usando recursos 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 iguais, a preferência vai para um candidato que não tem parâmetros opcionais para os quais os argumentos foram omitidos na chamada. A resolução de sobrecarga geralmente prefere candidatos que têm 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 Office programação (Guia de Programação em C#)

21/01/2022 • 5 minutes to read

Os argumentos nomeados e opcionais, introduzidos em C# 4, 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.

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 querer especificar valores específicos para todos eles. No entanto, sem argumentos nomeados e opcionais, um valor ou um valor de espaço reservado precisa ser fornecido para cada parâmetro. Com argumentos nomeados e opcionais, você especifica valores apenas para os parâmetros que são necessários para seu projeto.

Você deve ter o Microsoft Office Word instalado em seu computador para concluir esses procedimentos.

NOTE

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 criar um novo aplicativo de console

1. Inicie o Visual Studio.
2. No menu **Arquivo**, aponte para **Novo** e clique em **Projeto**.
3. No painel **Templates Categories (Categorias de Modelos)**, expanda **Visual C#** e clique em **Windows**.
4. Observe a parte superior do painel **Modelos** para se certificar de que **.NET Framework 4** é exibido na caixa **Estrutura de Destino**.
5. No painel **Modelos**, clique em **Aplicativo de Console**.
6. Digite um nome para o projeto no campo **Nome**.
7. Clique em **OK**.

O novo projeto aparece no **Gerenciador de Soluções**.

Para adicionar uma referência

1. No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, clique em **Adicionar Referência**. A caixa de diálogo **Adicionar Referência** é exibida.
2. Na página **.NET**, selecione **Microsoft.Office.Interop.Word** na lista **Nome do Componente**.
3. Clique em **OK**.

Para adicionar as diretivas using necessárias

1. No Gerenciador de Soluções, clique com o botão direito do mouse no arquivo *Program.cs* e, em seguida, clique em **Exibir Código**.
2. Adicione as seguintes `using` diretivas à parte superior do arquivo de código:

```
using Word = Microsoft.Office.Interop.Word;
```

Para exibir texto em um documento do Word

1. Na classe `Program` em *Program.cs*, adicione o método a seguir 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.

```
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();
}
```

2. Adicione o seguinte código no final do método para definir onde exibir o texto no documento e qual texto exibir:

```
// 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. . .");
```

Para executar o aplicativo

1. Adicione a seguinte instrução a `Main`:

```
DisplayInWord();
```

2. Pressione CTRL + F5 para executar o projeto. É exibido um documento do Word contendo o texto especificado.

Para alterar o texto para uma tabela

1. 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 simples. O argumento especifica que as vírgulas na cadeia de caracteres de texto em `range` separam as células da tabela.

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

Em versões anteriores do C#, a chamada para requer um argumento de referência para cada `ConvertToTable` parâmetro, conforme mostrado no código a seguir:

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code
// is not part of the solution.
var missing = Type.Missing;
object separator = ",";
range.ConvertToTable(ref separator, ref missing, ref missing,
    ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing,
    ref missing);
```

2. Pressione CTRL + F5 para executar o projeto.

Para fazer experiências com outros parâmetros

1. Para alterar a tabela para que ela tenha uma coluna e três linhas, substitua a última linha em pela instrução a seguir e `DisplayInWord` digite CTRL + F5.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. Para especificar um formato predefinido para a tabela, substitua a última linha em pela instrução a seguir e `DisplayInWord` digite CTRL + F5. O formato pode ser qualquer uma das constantes `WdTableFormat`.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Exemplo

O código a seguir inclui o exemplo completo:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
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);
        }
    }
}

```

Confira também

- [Argumentos nomeados e opcionais](#)

Construtores (guia de programação C#)

21/01/2022 • 2 minutes to read

Sempre que uma [classe](#) ou [struct](#) é criada, o 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 uso de [construtores](#).

Sintaxe do construtor

Um construtor é um método cujo nome é igual ao nome de seu tipo. Sua assinatura do método inclui apenas o nome do método e lista de parâmetros, ele não inclui um tipo de retorno. O exemplo a seguir mostra o construtor para uma classe denominada `Person`.

```
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`.

```
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# Inicializa campos estáticos para 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.

```
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.

```
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 C#](#)
- [O sistema de tipos C#](#)
- [Finalizadores](#)
- [static](#)
- [Por que inicializadores são executados na ordem oposta como construtores? Parte um](#)

Usando construtores (Guia de Programação em C#)

21/01/2022 • 4 minutes to read

Quando uma [classe](#) ou [struct](#) é criado, 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.

```
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`. 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 classe estática](#).

Você pode impedir que uma classe seja instanciada tornando o construtor privado, da seguinte maneira:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E; //2.71828...
}
```

Para obter mais informações, consulte [Construtores particulares](#).

Os construtores de tipos [struct](#) são semelhantes aos construtores de classe, mas [structs](#) não podem conter um construtor sem parâmetros explícito porque um é fornecido automaticamente pelo compilador. Esse construtor inicializa cada campo no `struct` para o [valor padrão](#). No entanto, esse construtor sem parâmetro será invocado apenas se o `struct` for instanciado com `new`. Por exemplo, esse código usa o construtor sem parâmetros para `Int32`, de modo que você tenha certeza de que o inteiro é inicializado:

```
int i = new int();
Console.WriteLine(i);
```

NOTE

A partir do C# 10, um tipo de estrutura pode conter um construtor explícito sem parâmetros. Para obter mais informações, consulte a seção [construtores sem parâmetros e inicializadores de campo](#) do artigo [tipos de estrutura](#).

O código a seguir, no entanto, causa um erro do compilador porque ele não usa `new` e porque ele tenta usar um objeto que não foi inicializado:

```
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:

```
int a = 44; // Initialize the value type...
int b;
b = 33; // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Portanto, não é necessário chamar o construtor sem parâmetros para um tipo de valor.

Ambas as classes e `structs` podem definir construtores que usam parâmetros. 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:

```
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:

```
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:

```
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. Isso significa que as seguintes declarações de construtor são efetivamente iguais:

```
public Manager(int initData)
{
    //Add further instructions here.
}
```

```
public Manager(int initData)
    : 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`:

```
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:

```
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 geralmente 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 C#](#)
- [O sistema de tipos C#](#)
- [Construtores](#)
- [Finalizadores](#)

Construtores de instância (guia de programação C#)

21/01/2022 • 2 minutes to read

Você declara um construtor de instância para especificar o código que é executado quando você cria uma nova instância de um tipo com a [new expressão](#). Para inicializar uma classe [estática](#) ou variáveis estáticas em uma classe não estática, você pode definir um [construtor estático](#).

Como mostra o exemplo a seguir, você pode declarar vários construtores de instância em um tipo:

```
using System;

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, sem parâmetros, construtor chama o segundo construtor com os dois argumentos iguais `0`. Para fazer isso, use a `this` palavra-chave.

Quando você declara um construtor de instância em uma classe derivada, você pode chamar um construtor de uma classe base. Para fazer isso, use a `base` palavra-chave, como mostra o exemplo a seguir:

```

using System;

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 explícitos, o C# fornecerá um construtor sem parâmetros que você pode usar para instanciar uma instância dessa classe, como mostra o exemplo a seguir:

```
using System;

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 campos de instância e propriedades de acordo com os inicializadores correspondentes. Se um campo ou propriedade não tiver nenhum 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 anterior, esse é um Construtor implícito sem parâmetros que produz o [valor padrão](#) de um tipo.
- No C# 10 e posterior, isso 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 [construtores sem parâmetros e inicializadores de campo](#) do artigo [tipos de estrutura](#).

Confira também

- [Guia de programação em C#](#)
- [Classes, estruturas e registros](#)
- [Construtores](#)
- [Finalizadores](#)
- [base](#)
- [this](#)

Construtores particulares (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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:

```
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 classe estática](#).

Exemplo

A seguir, temos um exemplo de uma classe usando um construtor particular.

```

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:

```
// Counter aCounter = new Counter(); // Error
```

Confira também

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Construtores](#)
- [Finalizadores](#)
- [Privada](#)
- [public](#)

Construtores estáticos (Guia de Programação em C#)

21/01/2022 • 5 minutes to read

Um construtor estático é usado para inicializar todos os 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.

```
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;
    }
}
```

Comentários

Construtores estáticos têm as seguintes propriedades:

- Um construtor estático não recebe 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](#) que a primeira instância seja criada ou que todos os membros estáticos declarados nessa classe (não suas classes base) sejam referenciados. Um construtor estático é executado antes de um construtor de instância. O construtor estático de um tipo é chamado quando um método estático atribuído a um evento ou a um delegado é invocado e não quando é atribuído. Se os inicializadores de variável de campo estático estão presentes na classe do construtor estático, eles são executados na ordem textual na qual 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 em Valores padrão de tipos [C#](#).
- Se um construtor estático lançar uma exceção, o runtime não a invocará uma segunda vez e o tipo permanecerá não reinicializado 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 a inspeção do código il (linguagem intermediária).
- 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 é necessário, inicialize campos estáticos na declaração em vez de por meio de um construtor estático para uma otimização de runtime melhor.

- O runtime chama um construtor estático não mais de uma vez em um único domínio de 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 bloquee o thread atual em construtores e inicializadores estáticos. Por exemplo, não aguarde tarefas, threads, alças de espera ou eventos, não adquira bloqueios e não execute operações paralelas de bloqueio, como loops paralelos e consultas `Parallel.Invoke` LINQ paralelas.

NOTE

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`.
- Construtores estáticos também são um local conveniente para impor verificações de tempo de execução no parâmetro de tipo que não podem ser verificados no 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.

```
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());
    }

    // 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: construtores estáticos devem ser privados](#)
- [Inicializadores de módulo](#)

Como escrever um construtor de cópia (guia de programação C#)

21/01/2022 • 2 minutes to read

Os [registros](#) do C# fornecem um construtor de cópia para objetos, mas para classes que você precisa escrever um por conta própria.

Exemplo

No exemplo a seguir, a `Person class` 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.

```

using System;

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](#)
- [Grava](#)
- [Guia de programação C#](#)
- [O sistema de tipos C#](#)
- [Construtores](#)
- [Finalizadores](#)

Finalizadores (Guia de Programação em C#)

21/01/2022 • 4 minutes to read

Os finalizadores (historicamente chamados de destruidores) são usados para executar qualquer limpeza final necessária quando uma instância de classe está sendo coletada pelo coletor de lixo. Na maioria dos casos, você pode evitar escrever um finalizador usando as classes derivadas ou para envolver qualquer alça [System.Runtime.InteropServices.SafeHandle](#) não manuada.

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 sobrecarregados.
- 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`.

```
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.

```
using System;

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:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Esse design significa que o método é chamado recursivamente para todas as instâncias na cadeia de herança, do mais derivado ao `Finalize` menos derivado.

NOTE

Finalizadores vazios não devem ser usados. Quando uma classe contém um finalizador, uma entrada é criada na fila `Finalize`. Essa fila é processada no coletor de lixo. Quando o GC 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; o coletor de lixo decide quando chamá-lo. 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 `Collect` criar problemas de desempenho.

NOTE

Se os finalizadores são executados como parte do encerramento do aplicativo é específico 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 como lixo, a menos que essa limpeza tenha sido suprimida (por uma chamada para o 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, GitHub problema [dotnet/csharpstandard #291](#).

Se você precisar executar a limpeza de forma confiável quando um aplicativo sair, registre um manipulador para o `System.AppDomain.ProcessExit` evento. Esse manipulador garantiria que (ou) fosse chamado para todos os objetos que `IDisposable.Dispose()` `IAsyncDisposable.DisposeAsync()` 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, use ou para garantir que os recursos sejam `Dispose` `DisposeAsync` liberados.

Usar finalizadores para liberar recursos

Em geral, o C# não requer tanto gerenciamento de memória por parte do desenvolvedor quanto linguagens que não se direcionam a um runtime com coleta de lixo. Isso 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 utilizados, 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 da interface que executa a `Dispose` limpeza necessária para o objeto `IDisposable`. Isso pode melhorar consideravelmente o desempenho do aplicativo. Mesmo com esse controle explícito sobre os recursos, o finalizador se torna uma proteção para limpar recursos se a chamada ao `Dispose` método falhar.

Para obter mais informações sobre como limpar recursos, consulte os seguintes artigos:

- [Limpando recursos não gerenciados](#)
- [Implementando um método Dispose](#)

- [Implementando 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 é encerrado.

```
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, consulte 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#)

21/01/2022 • 9 minutes to read

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](#).

```
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;
    }
}
```

```
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 atribua o objeto recém-criado, com suas propriedades atribuídas, à variável na atribuição.

Começando com o C# 6, os inicializadores de objeto podem definir indexadores, além de atribuir campos e propriedades. Considere esta classe `Matrix` básica:

```
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:

```
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:

```
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:

```
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 os inicializadores de objeto 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.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Tipos anônimos habilitam a cláusula em uma expressão de consulta LINQ para transformar objetos da sequência original em objetos cujo valor e forma `select` podem ser diferentes do original. 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.

```
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:

```
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`.

```
select new {p.ProductName, Price = p.UnitPrice};
```

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:

```
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.

```
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.

```
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.

```
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. Antes do C# 6, você podia inicializar dicionários e outros contêineres associativos usando a sintaxe a seguir. Observe que, em vez da sintaxe do indexador, com parênteses e uma atribuição, ele usa um objeto com vários valores:

```
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 `Cats` propriedade de no seguinte `CatOwner` caso:

```
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 receber uma nova lista:

```
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 ao mesmo tempo usando a sintaxe de inicialização omitindo a criação da lista `Cats()`, conforme mostrado a `new List<Cat>` seguir:

```
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 ser adicionado simplesmente aparece entre chaves. O acima é idêntico à escrita:

```
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.

```

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.");
    }
}

```

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](#).

```

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>>>
    {
        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

- [Guia de Programação em C#](#)
- [LINQ em C#](#)
- [Tipos anônimos](#)

Como inicializar objetos usando um inicializador de objeto (guia de programação C#)

21/01/2022 • 2 minutes to read

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 inicializadores de objeto acessando primeiro o construtor de instância sem parâmetros e, em seguida, processando 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 elemento 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:

```
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(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:

```
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 C#](#)
- [Inicializadores de objeto e coleção](#)

Como inicializar um dicionário com um inicializador de coleção (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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.

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`:

```
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 mais internas delimitam o inicializador de objeto para o e as chaves mais externas delimitam o inicializador para o par chave/valor que será adicionado `StudentName` 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#)

21/01/2022 • 2 minutes to read

Um tipo definido dentro de [uma classe, struct](#) ou [interface](#) é chamado de tipo aninhado. Por exemplo

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Independentemente de o tipo externo ser uma classe, interface ou struct, os tipos aninhados assumem como [padrão o privado](#); eles só podem ser acessados de seu tipo de conteúdo. 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".

Além disso, esteja ciente de que tornar um tipo aninhado externamente visível viola a regra de qualidade de código [CA1034](#) "Tipos aninhados não devem estar visíveis".

- Tipos aninhados de um [struct](#) podem ser [públicos](#), [internos](#) ou [particulares](#).

O exemplo a seguir torna a classe `Nested` pública:

```
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:

```
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:

```
Container.Nested nest = new Container.Nested();
```

Confira também

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Modificadores de acesso](#)
- [Construtores](#)
- [Regra CA1034](#)

Classes e métodos partial (Guia de Programação em C#)

21/01/2022 • 7 minutes to read

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:

```
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.

NOTE

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.

```
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:

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

Elas são equivalentes às seguintes declarações:

```
[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:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

Elas são equivalentes às seguintes declarações:

```
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: [!code-csharpAllDefinitionsMustBePartials#7]
- 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: [!code-csharpNestedPartialTypes#8]
- 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:
 - `public`
 - `Privada`
 - `protected`
 - `interno`
 - `Abstrata`
 - `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.

```

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.

```

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 é necessário

para ter uma implementação nos seguintes casos:

- Ele não tem nenhum modificador de acessibilidade (incluindo o padrão [privado](#)).
- Ele retorna [void](#).
- Ele não tem parâmetros [de saída](#).
- Ele não tem nenhum dos modificadores virtuais [aseguir](#), [substituir](#), [lacrado](#), [novo](#) ou [extern](#).

Qualquer método que não esteja em conformidade com todas essas restrições (por exemplo,

`public virtual partial void` método), 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 de 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 que um desenvolvedor decida se deve 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:** 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 escrever 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.

```
// 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 parciais do método em ambas as partes do tipo parcial devem corresponder.
- 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 C#](#)
- [Classes](#)
- [Tipos de estrutura](#)
- [Interfaces](#)
- [partial \(tipo\)](#)

Como retornar subconjuntos de propriedades de elemento em uma consulta (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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:

```
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.

```
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:

```
select new { First = student.FirstName, Last = student.LastName };
```

Se você tentar fazer isso no exemplo anterior, a instrução `Console.WriteLine` também deve ser alterada:

```
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#)

21/01/2022 • 3 minutes to read

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. Este primeiro exemplo define os tipos:

```
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:

```
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 é chamado apenas por meio da interface especificada. Nomeie o membro da classe prefixando-o com o nome da interface e um ponto. Por exemplo:

```

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 está disponível diretamente na classe. Por exemplo:

```

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 usada para resolver casos em que duas interfaces declaram Membros diferentes do mesmo nome, como uma propriedade e um método. Para implementar ambas as interfaces, uma classe deve usar a implementação explícita para a propriedade `P` ou o método `P`, ou ambos, para evitar um erro do compilador. Por exemplo:

```

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 ele está definido. Em vez disso, ele só é acessível quando chamado por meio de uma instância da interface. Se você especificar um modificador de acesso para uma implementação de interface explícita, obterá o erro do compilador [CS0106](#). Para obter mais informações, consulte [interface \(referência C#\)](#).

A partir do [C# 8.0](#), 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 poderá ser acessado 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:

```
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:

```
var sample = new SampleClass();
//sample.Paint(); // "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Qualquer classe que implemente a `IControl` interface pode substituir o `Paint` método padrão, seja como um método público ou como uma implementação de interface explícita.

Confira também

- [Guia de programação C#](#)
- [Programação orientada a objeto](#)
- [Interfaces](#)
- [Herança](#)

Como implementar explicitamente membros de interface (guia de programação C#)

21/01/2022 • 2 minutes to read

Este exemplo declara uma interface, `IDimensions` e uma classe, `Box`, que implementa explicitamente os membros da interface `GetLength` e `GetWidth`. Os membros são acessados por meio da instância `dimensions` da interface.

Exemplo

```

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;

        // 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`:

```

//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:

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());  
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

Confira também

- [Guia de programação C#](#)
- [Programação orientada a objeto](#)
- [Interfaces](#)
- [Como implementar membros de duas interfaces explicitamente](#)

Como implementar explicitamente membros de duas interfaces (guia de programação C#)

21/01/2022 • 2 minutes to read

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

```

// 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`:

```
// 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:

```
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 C#](#)
- [Programação orientada a objeto](#)
- [Interfaces](#)
- [Como implementar membros de interface explicitamente](#)

Delegados (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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:

```
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 chamadas de método ou conectar novo código às classes existentes.

NOTE

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 têm que corresponder exatamente ao tipo delegado. Para obter mais informações, confira [Usando a variação delegados](#).
- Expressões lambda são uma maneira mais concisa de escrever blocos de código em linha. Expressões lambda (em determinados contextos) são compiladas para delegados de tipos. 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](#) [Learning 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#)

21/01/2022 • 5 minutes to read

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 `Del` que pode encapsular um método que usa uma **cadeia de caracteres** como um argumento e retorna **nulo**:

```
public delegate void Del(string message);
```

Um objeto delegado normalmente é construído fornecendo o nome do método que o delegado irá encapsular 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:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Tipos delegados são derivados da **Delegate** classe 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 parâmetro 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 método de exemplo a seguir usa o tipo `Del` como um parâmetro:

```
public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Em seguida, você pode passar o delegado criado acima para esse método:

```
MethodWithCallback(1, 2, handler);
```

e receber a seguinte saída para o 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:

```
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 `Del`.

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:

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del 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:

```
//remove Method1  
allMethodsDelegate -= d1;  
  
// copy AllMethodsDelegate while removing d2  
Del 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:

```
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:

```
delegate void Delegate1();  
delegate void Delegate2();  
  
static void method(Delegate1 d, Delegate2 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 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#)

21/01/2022 • 2 minutes to read

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:

```
// 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.

NOTE

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.

A partir do C# 10, os grupos de métodos com uma única sobrecarga têm um *tipo natural*. Isso significa que o compilador pode inferir o tipo de retorno e os tipos de parâmetro para o tipo delegado:

```
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

```

// 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.

```
// 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 C#](#)
- [Representantes](#)
- [Como combinar delegados \(delegados de multicast\)](#)
- [Eventos](#)

Como combinar delegados (delegados multicast) (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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 um delegado multicast.

Exemplo

```

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;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create 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, insinuar e usar um Delegado (Guia de Programação em C#)

21/01/2022 • 5 minutes to read

No C# 1.0 e versões posteriores, é possível declarar delegados conforme mostrado no exemplo a seguir.

```
// 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}");  
}
```

```
// Create an instance of the delegate.  
Del del1 = new Del(Notify);
```

O C# 2.0 oferece uma maneira mais simples de gravar a declaração anterior, conforme mostrado no exemplo a seguir.

```
// C# 2.0 provides a simpler way to declare an instance of Del.  
Del del2 = Notify;
```

No C# 2.0 e versões posteriores, também é possível usar um método anônimo para declarar e inicializar um [delegado](#), conforme mostrado no exemplo a seguir.

```
// Instantiate Del by using an anonymous method.  
Del del3 = delegate(string name)  
{ Console.WriteLine($"Notification received for: {name}"); };
```

No C# 3.0 e versões posteriores, os delegados também podem ser declarados e instanciados usando uma expressão lambda, conforme mostrado no exemplo a seguir.

```
// 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

```
// 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.

```
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:

```
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:

```
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:

```
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#)

21/01/2022 • 3 minutes to read

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`.

```
type[] arrayName;
```

Exemplo

O exemplo a seguir cria matrizes unidimensionais, multidimensionais e denteadas:

```
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 da matriz

Uma matriz tem as seguintes propriedades:

- Uma matriz pode ser [única](#), [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 do valor padrão

- Para tipos de valor, os elementos de 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 anulável, `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. Você pode usar as propriedades e outros membros da classe que o [Array](#) tem. Um exemplo disso é usar a [Length](#) propriedade 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`:

```
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 [Rank](#) propriedade para exibir o número de dimensões de uma matriz.

```
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](#)
- [Usando foreach com matrizes](#)
- [Passando matrizes como argumentos](#)
- [Matrizes digitadas implicitamente](#)
- [Guia de programação 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#)

21/01/2022 • 2 minutes to read

Você cria uma matriz unidimensional usando o [novo](#) operador especificando o tipo de elemento de matriz e o número de elementos. O exemplo a seguir declara uma matriz de cinco inteiros:

```
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.

As matrizes podem armazenar qualquer tipo de elemento que você especificar, como o exemplo a seguir, que declara uma matriz de cadeias de caracteres:

```
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:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

O código 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:

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Você pode evitar a `new` expressão e o tipo de matriz ao inicializar uma matriz na declaração, conforme mostrado no código a seguir. Isso é chamado de [matriz tipada implicitamente](#):

```
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 `new` operador ao atribuir uma nova matriz a essa variável. Por exemplo:

```
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:

```
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 com 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 para o tipo de elemento. Para obter mais informações sobre tipos de valor e tipos de referência, consulte [tipos de valor](#) e [tipos de referência](#).

Retrieving dados da matriz

Você pode recuperar os dados de uma matriz usando um índice. Por exemplo:

```
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#)

21/01/2022 • 2 minutes to read

As matrizes podem ter mais de uma dimensão. Por exemplo, a declaração a seguir cria uma matriz bidimensional de quatro linhas e duas colunas.

```
int[,] array = new int[4, 2];
```

A declaração a seguir cria uma matriz de três dimensões, 4, 2 e 3.

```
int[,,] array1 = new int[4, 2, 3];
```

Inicialização de Matriz

É possível inicializar a matriz na declaração, conforme mostrado no exemplo a seguir.

```

// 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

```

Você também pode inicializar a matriz sem especificar a classificação.

```
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.

```
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.

```
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 .
```

```
int elementValue = array5[2, 1];
```

O exemplo de código a seguir inicializa os elementos da matriz com valores padrão (exceto em matrizes denteadas).

```
int[,] array6 = new int[10, 10];
```

Confira também

- [Guia de programação C#](#)
- [matrizes](#)
- [Matrizes unidimensionais](#)
- [Matrizes denteadas](#)

Matrizes denteadas (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

Uma matriz denteada é uma matriz cujos elementos são matrizes, possivelmente de tamanhos diferentes. Às vezes, uma matriz denteada é chamada de uma "matriz de matrizes." Os exemplos a seguir mostram como declarar, inicializar e acessar matrizes denteadas.

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:

```
int[][] jaggedArray = new int[3][];
```

Antes de usar `jaggedArray`, seus elementos devem ser inicializados. É possível inicializar os elementos dessa forma:

```
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:

```
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:

```
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:

```
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:

```
// 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 denteadas 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, consulte [matrizes multidimensionais](#).

```
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`):

```
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:

```
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.

```

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 C#](#)
- [matrizes](#)
- [Matrizes unidimensionais](#)
- [Matrizes multidimensionais](#)

Usar foreach com matrizes (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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`:

```
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:

```
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 C#](#)
- [matrizes](#)
- [Matrizes unidimensionais](#)
- [Matrizes multidimensionais](#)
- [Matrizes denteadas](#)

Passando matrizes como argumentos (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

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.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

O código a seguir mostra uma implementação parcial do método de impressão.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

É possível inicializar e passar uma nova matriz em uma etapa, conforme mostrado no exemplo a seguir.

```
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.

```

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 Wed 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.

```

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.

```
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:

```
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.

```
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 C#](#)
- [matrizes](#)
- [Matrizes unidimensionais](#)
- [Matrizes multidimensionais](#)
- [Matrizes denteadas](#)

Matrizes de tipo implícito (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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:

```
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.

```
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 de tipo implícito](#)
- [matrizes](#)
- [Tipos anônimos](#)
- [Inicializadores de objeto e coleção](#)
- [Var](#)
- [LINQ em C#](#)

Cadeias de caracteres (Guia de Programação em C#)

21/01/2022 • 14 minutes to read

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](#).

String 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 fornecido, `string` 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:

```

// 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);

```

Observe que 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 de objetos de cadeia 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:

```

string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello

```

Para obter mais informações sobre como criar novas cadeias de caracteres com base em modificações, como operações de pesquisa e substituição na cadeia original, consulte [como modificar o conteúdo da cadeia de caracteres](#).

Literais de cadeia de caracteres regulares e textuais

Use literais de cadeia de caracteres regulares quando você precisar inserir caracteres de escape fornecidos por C#, conforme mostrado no seguinte exemplo:

```

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 œolian Harp", by Samuel Taylor Coleridge

```

Use cadeias de caracteres textuais para conveniência e para facilitar a leitura quando o texto da cadeia de caracteres contiver caracteres de barra invertida, por exemplo, em caminhos de arquivo. Como as cadeias de caracteres textuais preservam os caracteres de nova linha como parte do texto da cadeia de caracteres, podem ser usadas para inicializar cadeias de caracteres de várias linhas. 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:

```

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."

```

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)	\u{HHHH} (intervalo: 0000-FFFF; exemplo: \u00E7 = "ç")
\U	Sequência de escape Unicode (UTF-32)	\u{0000HHHHHH} (intervalo: 000000- 10FFFF; exemplo: \u0001F47D = "\ud83d\udcbb")
\x	Sequência de escape Unicode semelhante a "\u", exceto pelo comprimento variável	\x{H}{H}{H} (intervalo: 0-FFFF; exemplo: \x00E7 ou \x0E7 \xE7 = "ç")

WARNING

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 ";" 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 "■", que é o ponto de código U+0A1A. Nesses casos, especificar todos os quatro dígitos hexadecimais (por exemplo, `\x00A1`) impedirá qualquer interpretação errônea possível.

NOTE

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 uma cadeia de caracteres cujo conteúdo é determinado dinamicamente no tempo de execução. 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 em tempo de execução. 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.

```
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.

Formatação composta

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.

```

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. Para obter mais informações, consulte [Como pesquisar cadeias de caracteres](#) e [Como modificar o conteúdo da cadeia de caracteres](#).

```

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:

```

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:

```
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:

```
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:

```

static void Main()
{
    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:

```

System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//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:

```

using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            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
            Console.WriteLine();
        }
    }
}

```

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 expressões de consulta LINQ em cadeias de caracteres. Para saber mais, confira [LINQ e cadeias de caracteres](#).

Tópicos Relacionados

TÓPICO	DESCRIÇÃO
Como modificar o conteúdo da 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 <code>String.Split</code> 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.

TÓPICO	DESCRIÇÃO
Interpolação de cadeia 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 cadeia de caracteres	Fornece links para tópicos 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.
Analisando 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 cadeia de caracteres dinâmica usando a classe StringBuilder .
LINQ e cadeias de caracteres	Fornece informações sobre como executar várias operações de cadeia de caracteres usando consultas LINQ.
Guia de Programação em C#	Fornece links para tópicos que explicam as construções de programação em C#.

Como determinar se uma cadeia de caracteres representa um valor numérico (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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.

```
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.

NOTE

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`.

```
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](#)
- [Análise de cadeias de caracteres numéricas](#)
- [Formatar tipos](#)

Indexadores (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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.

```
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.
```

NOTE

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.

```

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:

```

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#)

21/01/2022 • 6 minutes to read

Os indexadores são uma conveniência sintática que permitem que você crie uma [classe](#), [estrutura](#) ou [interface](#) que os aplicativos cliente possam acessar como uma matriz. O compilador gerará uma [Item](#) Propriedade (ou uma propriedade nomeada de forma alternativa [IndexerNameAttribute](#), se 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 horas diferentes durante um período de 24 horas. A classe contém uma [temps](#) matriz 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 cliente; Ele também torna a classe e sua finalidade mais intuitiva para que outros desenvolvedores entendam.

Para declarar um indexador em uma classe ou struct, use a palavra-chave [this](#), como mostra o seguinte exemplo:

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

IMPORTANT

A declaração de um indexador irá gerar automaticamente uma propriedade chamada [Item](#) no objeto. A [Item](#) propriedade não pode ser acessada diretamente da [expressão de acesso de membro](#) de instância. Além disso, se você adicionar sua própria [Item](#) Propriedade a um objeto com um indexador, obterá um [erro do compilador CS0102](#). Para evitar esse erro, use [IndexerNameAttribute](#) 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 valor de indexador não é classificado como uma variável; portanto, não é possível passar um valor de indexador como um parâmetro [ref](#) ou [out](#).

Para fornecer o indexador com um nome que outras linguagens possam usar, use [System.Runtime.CompilerServices.IndexerNameAttribute](#), como mostra o seguinte exemplo:

```

// 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.

```

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.

```

using System;

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 de cadeia de caracteres e inteiros 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.

```

using System;

// 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

```

using System;

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 [System.DayOfWeek](#) enumeração. Um `get` acessador pega um `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.

```

using System;
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

```

using System;

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 C#](#)
- [Indexadores](#)
- [Propriedades](#)

Indexadores em interfaces (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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 um corpo.

A finalidade do acessador é indicar se o indexador é somente leitura, somente leitura ou somente gravação. Você pode fornecer uma implementação para um indexador definido em uma interface, mas isso é raro. Os indexadores normalmente definem uma API para acessar campos de dados e os campos de dados não podem ser definidos em uma interface.

Este é um exemplo de um acessador de indexador de interface:

```
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.

```

// 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;
    }
}

```

```

IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// 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

```

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:

```
string IEmployee.this[int index]
{
}
```

implementa o indexador na interface `IEmployee`, enquanto a seguinte declaração:

```
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#)

21/01/2022 • 2 minutes to read

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 Propriedades Autoimplementadas .	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#)

21/01/2022 • 2 minutes to read

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, consulte [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 .NET, os eventos são baseados no [EventHandler](#) delegado e na [EventArgs](#) classe base.

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 [Learning c# 3,0: conceitos básicos de C# 3,0](#)

Confira também

- [EventHandler](#)
- [Guia de programação C#](#)
- [Representantes](#)
- [Criando manipuladores de eventos no Windows Forms](#)

Como assinar e cancelar a assinatura de eventos (guia de programação C#)

21/01/2022 • 3 minutes to read

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`.

```
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:

```
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:

```
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.

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

Você também pode usar uma [expressão lambda](#) para especificar um manipulador de eventos:

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

Para assinar eventos usando uma função anônima

Se você não precisar cancelar a assinatura de um evento posteriormente, 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.

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

Você não poderá cancelar a assinatura de um evento com facilidade se tiver usado uma função anônima para assiná-lo. Para cancelar a assinatura neste cenário, volte para o código em que você assina o evento, armazene a função anônima em uma variável de delegado e, em seguida, adicione o delegado ao evento. Recomendamos que você não use funções anônimas para assinar eventos se precisar cancelar a assinatura do evento em algum momento posterior em seu código. Para obter mais informações sobre funções anônimas, consulte [expressões 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:

```
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](#)
- [-e-= operadores](#)

- operadores + e =

Como publicar eventos que estão em conformidade com as diretrizes do .NET (guia de programação C#)

21/01/2022 • 3 minutes to read

O procedimento a seguir demonstra como adicionar eventos que seguem o padrão .NET padrão para suas classes e estruturas. Todos os eventos na biblioteca de classes do .NET são baseados no [EventHandler](#) delegado, que é definido da seguinte maneira:

```
public delegate void EventHandler(object sender, EventArgs e);
```

NOTE

.NET Framework 2,0 apresenta uma versão genérica desse delegado, [EventHandler<TEventArgs>](#) . Os exemplos a seguir mostram como usar as duas versões.

Embora eventos em classes que você define possam ser baseados em qualquer tipo delegado válido, até mesmo delegados que retornam um valor, geralmente é recomendável que você baseie seus eventos no padrão .NET usando [EventHandler](#) , conforme mostrado no exemplo a seguir.

O nome [EventHandler](#) pode levar a um pouco de confusão, pois ela não lida realmente com o evento. Os [EventHandler](#) genéricos, e [EventHandler<TEventArgs>](#) são tipos delegados. Um método ou uma expressão lambda cuja assinatura corresponde à definição de delegado é o *manipulador de eventos* e será invocado quando o evento for gerado.

Publicar eventos com base no padrão EventHandler

1. (Pule esta etapa e vá para a etapa 3a se você não precisar enviar dados personalizados com seu evento.)

Declare a classe para seus dados personalizados em um escopo que seja visível para as classes de Publicador e de assinante. Em seguida, adicione os membros necessários para manter seus dados de evento personalizados. Neste exemplo, uma cadeia de caracteres simples é retornada.

```
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 [EventArgs](#) tipo personalizado.

```
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.

```
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.

```
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.

```
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.

```
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 C#](#)
- [Eventos](#)
- [Representantes](#)

Como gerar eventos de classe base em classes derivadas (guia de programação C#)

21/01/2022 • 3 minutes to read

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 usado extensivamente em Windows Forms classes 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.

NOTE

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

```
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 C#](#)
- [Eventos](#)
- [Representantes](#)
- [Modificadores de acesso](#)
- [Criando manipuladores de eventos no Windows Forms](#)

Como implementar eventos de interface (guia de programação C#)

21/01/2022 • 3 minutes to read

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.

```
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 `IShape` ou um `IDrawingObject`.

```

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 C#](#)
- [Eventos](#)
- [Representantes](#)
- [Implementação de interface explícita](#)
- [Como acionar eventos de classe base em classes derivadas](#)

Como implementar acessadores de evento personalizados (guia de programação C#)

21/01/2022 • 2 minutes to read

Um evento é um tipo especial de delegado multicast que só pode ser invocado de dentro da classe que ela 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 desses casos 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 bloquee o evento antes de adicionar ou remover um novo manipulador de eventos.

```
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#)

21/01/2022 • 2 minutes to read

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:

```
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, consulte [Genéricos em Tempo de Execução](#).

Diretrizes para a nomenclatura de parâmetros de tipo

- **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.

```
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 em tipos com parâmetro de tipo de uma letra.

```
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.

```
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.

Confira também

- [System.Collections.Generic](#)
- [Guia de programação C#](#)
- [Genéricos](#)
- [Diferenças entre modelos C++ e genéricos C#](#)

Restrições a parâmetros de tipo (Guia de Programação em C#)

21/01/2022 • 12 minutes to read

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 usar um tipo que não atenda a uma restrição, o compilador emite um erro. Restrições são especificadas usando a palavra-chave contextual `where`. A tabela a seguir lista os vários tipos de restrições:

CONSTRAINT	DESCRIÇÃO
<code>where T : struct</code>	O argumento type deve ser um tipo de valor não anulado . Para obter informações sobre tipos de valor que podem ser anulados, consulte Tipos de valor anulados . Como todos os tipos de valor têm um construtor sem parâmetros acessível, a restrição implica a restrição e <code>struct</code> não pode ser combinada com a <code>new()</code> restrição. Você não pode combinar a <code>struct</code> restrição com a <code>unmanaged</code> restrição.
<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 que pode ser anulado no C# 8.0 ou posterior, deve ser um tipo de referência que não pode ser <code>T</code> anulado.
<code>where T : class?</code>	O argumento type deve ser um tipo de referência, que pode ser anulado ou não anulado. Essa restrição se aplica também a qualquer classe, interface, delegado ou tipo de matriz.
<code>where T : notnull</code>	O argumento type deve ser um tipo que não pode ser anulado. O argumento pode ser um tipo de referência que não pode ser anulado no C# 8.0 ou posterior ou um tipo de valor não anulado.
<code>where T : default</code>	Essa restrição resolve a ambiguidade quando você precisa especificar um parâmetro de tipo não restrito ao substituir um método ou fornecer uma implementação de interface explícita. A <code>default</code> restrição implica o método base sem a <code>class</code> restrição <code>struct</code> ou <code>.</code> . Para obter mais informações, consulte a proposta default de especificação de restrição.
<code>where T : unmanaged</code>	O argumento type deve ser um tipo não anulado e não passível de nulo . A <code>unmanaged</code> restrição implica <code>struct</code> a restrição e não pode ser combinada com as <code>struct</code> restrições ou <code>new()</code> .

CONSTRAINT	DESCRIÇÃO
<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 <code>new()</code> restrição não pode ser combinada com as <code>struct</code> <code>unmanaged</code> restrições e .
<code>where T : <base class name></code>	O argumento de tipo deve ser ou derivar da classe base especificada. Em um contexto que pode ser anulado no C# 8.0 e posterior, deve ser um tipo de referência não anuável derivado da <code>T</code> classe base especificada.
<code>where T : <base class name>?</code>	O argumento de tipo deve ser ou derivar da classe base especificada. Em um contexto que pode ser anulado no C# 8.0 e posterior, pode ser um tipo que pode ser anulado ou não anulado derivado da <code>T</code> classe base especificada.
<code>where T : <interface name></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 que pode ser anulado no C# 8.0 e posterior, deve ser um tipo que não anuável <code>T</code> que implementa a interface especificada.
<code>where T : <interface name>?</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 que pode ser anulado no C# 8.0, pode ser um tipo de referência que pode ser anulado, um tipo de referência que não pode ser anulado ou <code>T</code> um tipo de valor. <code>T</code> pode não ser um tipo de valor que pode ser anulado.
<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 que pode ser anulado, se for um tipo de referência não anulado, deverá ser um tipo de referência que não pode ser <code>U</code> <code>T</code> anulado. Se <code>U</code> for um tipo de referência que pode ser anulado, pode ser <code>T</code> anulado ou não anulado.

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 chamadas de método e operações do tipo de restrição. Se a classe genérica ou o método usar qualquer operação nos membros genéricos além da atribuição simples ou chamar métodos que não são suportados pelo , você aplicará restrições ao parâmetro `System.Object` type. 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.

```

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:

```
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 `==`.

```
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 só sabe que é um tipo de referência no tempo de compilação e deve usar os operadores padrão válidos `T` 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:

```
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 porque não há nenhuma garantia de que o argumento de tipo concreto dará suporte a esses operadores.
- 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:

```
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:

```
//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`

A partir do C# 8.0, você pode usar a restrição para especificar que o argumento de tipo deve ser um tipo de valor não anulado ou um tipo de referência não `notnull` anuável. Ao contrário da maioria das outras restrições, se um argumento de tipo violar a restrição, o compilador gerará um aviso `notnull` em vez de um erro.

A `notnull` restrição tem um efeito somente quando usado em um contexto que pode ser anulado. Se você adicionar a restrição em um contexto anulado, o compilador não gerará avisos ou erros para violações `notnull` da restrição.

restrição de `class`

A partir do C# 8.0, a restrição em um contexto que pode ser anulado especifica que o argumento de tipo deve ser um tipo de referência que não pode ser `class` anulado. Em um contexto que pode ser anulado, quando um argumento de tipo é um tipo de referência que pode ser anulado, o compilador gera um aviso.

restrição de `default`

A adição de tipos de referência que podem ser anuladas complica o uso de `T?` em um tipo ou método genérico. Antes do C# 8, `T?` só podia ser usado quando a `struct` restrição era aplicada a `T`. Nesse contexto, `T?` refere-se ao `Nullable<T>` tipo para `T`. A partir do C# 8, pode ser usado com a restrição ou , mas `T?` um deles deve estar `struct` `class` presente. Quando a `class` restrição foi usada, `T?` refere-se ao tipo de referência que pode ser anulado para `T`. A partir do C# 9, `T?` pode ser usado quando nenhuma restrição é aplicada. Nesse caso, é interpretado da mesma forma que `T?` no C# 8 para tipos de valor e tipos de referência. No entanto, `T` se for uma instância de , será o mesmo que `Nullable<T>` `T? T` . Em outras palavras, ele não se torna `T??` .

Como agora pode ser usado sem a restrição ou , ambiguidades podem surgir em substituições `T?` `class` ou `struct` implementações explícitas de interface. Em ambos os casos, a substituição não inclui as restrições, mas as herda da classe base. Quando a classe base não aplica a restrição ou , as classes derivadas precisam especificar de alguma forma uma substituição se aplica ao método `class` `struct` base sem nenhuma restrição.

É quando o método derivado aplica a `default` restrição. A `default` restrição não esclarece a `class` restrição nem `struct`.

Restrição não gerenciada

A partir do C# 7.3, você pode usar a restrição para especificar que o parâmetro de tipo deve ser um tipo não passível de `unmanaged` `nulo`. 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:

```
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 `unmanaged` restrição implica `struct` a restrição e não pode ser combinada com ela. Como a restrição implica a restrição, a restrição também não pode ser combinada `struct` `new()` com a `unmanaged` `new()` restrição.

Restrições de delegado

Também começando com o C# 7.3, 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, desde que eles são do mesmo tipo:

```
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:

```
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. E `first` são `test` tipos delegados, mas são tipos delegados diferentes.

Restrições de enum

Começando com o C# 7.3, 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.

```
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 para criar uma coleção armazenada em cache e reutilizada em vez de repetir as `EnumNamedValues` 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:

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Classes genéricas](#)
- [nova restrição](#)

Classes genéricas (Guia de Programação em C#)

21/01/2022 • 4 minutes to read

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 uma classe genérica `Node<T>`, o código cliente pode fazer referência à classe, especificando um argumento de tipo para criar um tipo construído fechado (`Node<int>`). Como alternativa, ele pode deixar o parâmetro de tipo não especificado para criar um tipo construído aberto (`Node<T>`), como ao especificar uma classe base genérica. As classes genéricas podem herdar de classes base construídas concretas, fechadas ou abertas:

```

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.

```

//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:

```

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:

```

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:

```

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:

```
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#)

21/01/2022 • 4 minutes to read

Muitas vezes, é útil definir interfaces para classes de coleção genéricas ou para as classes genéricas que representam itens na coleção. Para classes genéricas, prefere-se usar interfaces genéricas, como `IComparable<T>` em vez de `IComparable`, a fim de evitar operações de conversão boxing e unboxing em tipos de valor. A biblioteca de classes do .NET define várias interfaces genéricas para uso com as classes de coleção no `System.Collections.Generic` namespace.

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>`. Isso 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>`.

```
//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:

```

class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}

```

Uma interface pode definir mais de um parâmetro de tipo, da seguinte maneira:

```

interface IDictionary<K, V>
{
}

```

As regras de herança que se aplicam às classes também se aplicam às interfaces:

```

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 de retorno. Na biblioteca de classes do .NET, herda de porque usa apenas no valor `IEnumerable<T>` de retorno de e no `IEnumerable<T> T GetEnumerator Current` getter de propriedade.

Classes concretas podem implementar interfaces construídas fechadas, da seguinte maneira:

```

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:

```

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](#).

Confira também

- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [interface](#)
- [Genéricos](#)

Métodos genéricos (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

Um método genérico é um método declarado com parâmetros de tipo, da seguinte maneira:

```
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:

```
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:

```
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 sobrecarregadas. 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:

```
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.

```
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>`.

```
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 sobrecarregados vários parâmetros de tipo. Por exemplo, todos os seguintes métodos podem ser localizados na mesma classe:

```
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 C#](#)
- [Introdução aos genéricos](#)
- [Métodos](#)

Genéricos e matrizes (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

No C# 2.0 e versões posteriores, 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.

```
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 C#](#)
- [Genéricos](#)
- [matrizes](#)
- [Genéricos](#)

Delegados genéricos (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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:

```
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:

```
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.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

O código que referencia o delegado deve especificar o argumento de tipo da classe recipiente, da seguinte maneira:

```
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](#).

```
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)
    {
        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 C#](#)
- [Introdução aos genéricos](#)
- [Métodos genéricos](#)
- [Classes genéricas](#)
- [Interfaces genéricas](#)
- [Representantes](#)
- [Genéricos](#)

Diferenças entre modelos C++ e genéricos C# (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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 de implementação, a principal diferença é que as substituições de tipo genérico em C# são executadas em tempo de execução e informações de tipo genérico, portanto, são preservadas para objetos instanciados. Para obter mais informações, consulte [Genéricos em tempo de execução](#).

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 C#](#)
- [Introdução aos genéricos](#)
- [Modelos](#)

Genéricos em tempo de execução (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

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:

```
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>`:

```
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:

```
class Customer { }
class Order { }
```

```
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`:

```
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`:

```
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 C# genérica é instanciada usando um tipo de valor ou parâmetro de tipo de referência, a reflexão pode consultá-la em tempo de operação e seu tipo real e seu parâmetro de tipo podem ser verificados.

Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Genéricos](#)

Genéricos e reflexão (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

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, consulte [Genéricos em tempo de execução](#).

no .NET Framework 2,0, vários novos membros foram adicionados à [Type](#) classe para habilitar informações em tempo de execução para tipos genéricos. Consulte a documentação dessas classes para obter mais informações sobre como usar esses métodos e propriedades. 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](#).

NOME DO MEMBRO SYSTEM.TYPe	DESCRÍÇÃO
IsGenericType	Retorna verdadeiro 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.

NOME DO MEMBRO SYSTEM.TYPE	DESCRIÇÃO
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.

NOME DO MEMBRO SYSTEM.REFLECTION.MEMBERINFO	DESCRIÇÃO
IsGenericMethod	Retorna verdadeiro se um método for 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 verdadeiro 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 verdadeiro 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

- [Guia de programação C#](#)
- [Genéricos](#)
- [Reflexão e tipos genéricos](#)
- [Genéricos](#)

Genéricos e atributos (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

Atributos podem ser aplicados a tipos genéricos da mesma forma que a tipos não genéricos. Para obter mais informações sobre a aplicação de atributos, consulte [Atributos](#).

Atributos personalizados são permitidos somente para referenciar tipos genéricos abertos, que são tipos genéricos para os quais nenhum argumento de tipo é fornecido e tipos genéricos construídos fechados, que fornecem argumentos para todos os parâmetros de tipo.

Os exemplos a seguir usam esse atributo personalizado:

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

Um atributo pode referenciar um tipo genérico aberto:

```
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:

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<, >))]
class ClassB { }
```

Um atributo pode referenciar um tipo genérico construído fechado:

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Um atributo que referencia um parâmetro de tipo genérico causará um erro em tempo de compilação:

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
class ClassD<T> { }
```

Um tipo genérico não pode herdar de [Attribute](#):

```
//public class CustomAtt<T> : System.Attribute {} //Error
```

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, consulte [Genéricos e Reflexão](#)

Confira também

- [Guia de programação C#](#)
- [Genéricos](#)
- [Atributos](#)

Sistema de arquivos e o registro (guia de programação C#)

21/01/2022 • 2 minutes to read

Os artigos a seguir mostram como usar o C# e o .NET para executar várias operações básicas em arquivos, pastas e no registro.

Nesta seção

TÍTULO	DESCRIÇÃO
Como iterar em uma árvore de diretório	Mostra como iterar manualmente em uma árvore de diretório.
Como obter informações sobre arquivos, pastas e unidades	Mostra como recuperar informações como tempo de criação e tamanho, sobre arquivos, pastas e unidades.
Como criar um arquivo ou uma pasta	Mostra como criar um novo arquivo ou pasta.
Como copiar, excluir e mover arquivos e pastas (guia de programação C#)	Mostra como copiar, excluir e mover arquivos e pastas.
Como fornecer uma caixa de diálogo de progresso para operações de arquivo	Mostra como exibir uma caixa de diálogo de progresso padrão do Windows para determinadas operações de arquivo.
Como escrever em um arquivo de texto	Mostra como gravar em um arquivo de texto.
Como ler de um arquivo de texto	Mostra como ler de um arquivo de texto.
Como ler um arquivo de texto uma linha de cada vez	Mostra como recuperar o texto do arquivo uma linha por vez.
Como criar uma chave no Registro	Mostra como gravar uma chave no Registro do sistema.

Seções relacionadas

- [Arquivo e e/s de fluxo](#)
- [Como copiar, excluir e mover arquivos e pastas \(guia de programação C#\)](#)
- [Guia de programação C#](#)
- [System.IO](#)

Como iterar por meio de uma árvore de diretórios (Guia de Programação em C#)

21/01/2022 • 7 minutes to read

A expressão "iterar uma árvore de diretório" significa acessar cada arquivo em cada subdiretório aninhado em uma pasta raiz especificada, em qualquer profundidade. Você não precisa necessariamente abrir cada arquivo. Você pode recuperar apenas o nome do arquivo ou subdiretório como um `string`, ou então você pode recuperar informações adicionais na forma de um objeto `System.IO.FileInfo` ou `System.IO.DirectoryInfo`.

NOTE

No Windows, os termos "diretório" e "pasta" são usados de forma intercambiável. A maioria dos textos de documentação e interface do usuário usa o termo "pasta", mas as bibliotecas de classes do .NET usam o termo "diretório".

No caso mais simples, em que você sabe com certeza que tem permissões de acesso a todos os diretórios em uma raiz especificada, é possível usar o sinalizador `System.IO.SearchOption.AllDirectories`. Esse sinalizador retorna todos os subdiretórios aninhados que correspondem ao padrão especificado. O exemplo a seguir mostra como usar o sinalizador.

```
root.GetDirectories(".*", System.IO.SearchOption.AllDirectories);
```

As desvantagem dessa abordagem é que, se qualquer um dos subdiretórios na raiz especificada causar um `DirectoryNotFoundException` ou `UnauthorizedAccessException`, o método inteiro falhará e não retornará nenhum diretório. O mesmo é verdadeiro quando você usa o método `GetFiles`. Se precisar manipular essas exceções em subpastas específicas, você precisa percorrer manualmente a árvore de diretório, conforme mostrado nos exemplos a seguir.

Ao percorrer manualmente uma árvore de diretórios, você pode manipular os arquivos primeiro (*transições de pré-ordem*) ou os subdirecionais primeiro (*transições pós-ordem*). Se você executar uma passagem de pré-pedido, visite os arquivos diretamente sob essa pasta em si e, em seguida, percorrerá toda a árvore sob a pasta atual. A travessia pós-ordem é o contrário, percorrendo toda a árvore abaixo antes de acessar os arquivos da pasta atual. Os exemplos mais adiante neste documento executam a travessia de pré-ordem, mas você pode modificá-los facilmente para executar a travessia pós-ordem.

Outra opção é usar a recursão ou uma passagem baseada em pilha. Os exemplos mais adiante neste documento mostram as duas abordagens.

Se precisar executar uma série de operações em arquivos e pastas, você pode modularizar esses exemplos refatorando a operação em funções separadas que podem ser invocadas usando um único delegado.

NOTE

Sistemas de arquivos NTFS podem conter *Pontos de Nova Análise* na forma de *Pontos de Junção*, *Links Simbólicos* e *Links Físicos*. Métodos do .NET, como `GetFiles` e `GetDirectories`, não retornarão subdiretórios em um ponto de nova análise. Esse comportamento protege contra o risco de entrar em um loop infinito quando dois pontos de nova análise fazem referência um ao outro. De modo geral, você deve ter muito cuidado ao lidar com pontos de nova análise para garantir que arquivos não sejam modificados ou excluídos inadvertidamente. Se precisar ter um controle preciso sobre pontos de nova análise, use a invocação de plataforma ou código nativo para chamar diretamente os métodos apropriados do sistema de arquivos Win32.

Exemplos

O exemplo a seguir mostra como percorrer uma árvore de diretório usando a recursão. A abordagem recursiva é elegante, mas tem o potencial de causar uma exceção de estouro de pilha se a árvore de diretório for grande e profundamente aninhada.

As exceções específicas que são tratadas, bem como as ações específicas que são executadas em cada arquivo ou pasta, são fornecidas apenas como exemplo. Você deve modificar este código para atender às suas necessidades específicas. Consulte os comentários no código para obter mais informações.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
        foreach (System.IO.FileInfo fi in files)
        {
            if (fi.Attributes.ToString() & FileAttributes.Hidden != 0)
                continue;
            if (fi.Attributes.ToString() & FileAttributes.ReadOnly != 0)
                log.Add(fi.FullName);
        }
        foreach (System.IO.DirectoryInfo di in subDirs)
        {
            if (di.Attributes.ToString() & DirectoryAttributes.Hidden != 0)
                continue;
            if (di.Attributes.ToString() & DirectoryAttributes.ReadOnly != 0)
                log.Add(di.FullName);
            WalkDirectoryTree(di);
        }
    }
}
```

```

try
{
    files = root.GetFiles("*.*");
}
// This is thrown if even one of the files requires permissions greater
// than the application provides.
catch (UnauthorizedAccessException e)
{
    // This code just writes out the message and continues to recurse.
    // You may decide to do something different here. For example, you
    // can try to elevate your privileges and access the file again.
    log.Add(e.Message);
}

catch (System.IO.DirectoryNotFoundException e)
{
    Console.WriteLine(e.Message);
}

if (files != null)
{
    foreach (System.IO.FileInfo fi in files)
    {
        // In this example, we only access the existing FileInfo object. If we
        // want to open, delete or modify the file, then
        // a try-catch block is required here to handle the case
        // where the file has been deleted since the call to TraverseTree().
        Console.WriteLine(fi.FullName);
    }

    // Now find all the subdirectories under this directory.
    subDirs = root.GetDirectories();

    foreach (System.IO DirectoryInfo dirInfo in subDirs)
    {
        // Recursive call for each subdirectory.
        WalkDirectoryTree(dirInfo);
    }
}
}
}

```

O exemplo a seguir mostra como iterar em arquivos e pastas em uma árvore de diretório sem o uso de recursão. Essa técnica usa o tipo de coleção genérico [Stack<T>](#), que é uma pilha UEPS (último a entrar, primeiro a sair).

As exceções específicas que são tratadas, bem como as ações específicas que são executadas em cada arquivo ou pasta, são fornecidas apenas como exemplo. Você deve modificar este código para atender às suas necessidades específicas. Consulte os comentários no código para obter mais informações.

```

public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
    }
}

```

```

Stack<string> dirs = new Stack<string>(20);

if (!System.IO.Directory.Exists(root))
{
    throw new ArgumentException();
}
dirs.Push(root);

while (dirs.Count > 0)
{
    string currentDir = dirs.Pop();
    string[] subDirs;
    try
    {
        subDirs = System.IO.Directory.GetDirectories(currentDir);
    }
    // An UnauthorizedAccessException exception will be thrown if we do not have
    // discovery permission on a folder or file. It may or may not be acceptable
    // to ignore the exception and continue enumerating the remaining files and
    // folders. It is also possible (but unlikely) that a DirectoryNotFoundException
    // will be raised. This will happen if currentDir has been deleted by
    // another application or thread after our call to Directory.Exists. The
    // choice of which exceptions to catch depends entirely on the specific task
    // you are intending to perform and also on how much you know with certainty
    // about the systems on which this code will run.
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
        continue;
    }
    catch (System.IO.DirectoryNotFoundException e)
    {
        Console.WriteLine(e.Message);
        continue;
    }

    string[] files = null;
    try
    {
        files = System.IO.Directory.GetFiles(currentDir);
    }

    catch (UnauthorizedAccessException e)
    {

        Console.WriteLine(e.Message);
        continue;
    }

    catch (System.IO.DirectoryNotFoundException e)
    {
        Console.WriteLine(e.Message);
        continue;
    }
    // Perform the required action on each file here.
    // Modify this block to perform your required task.
    foreach (string file in files)
    {
        try
        {
            // Perform whatever action is required in your scenario.
            System.IO.FileInfo fi = new System.IO.FileInfo(file);
            Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
        }
        catch (System.IO.FileNotFoundException e)
        {
            // If file was deleted by a separate application
            // or thread since the call to TraverseTree()
            // then just continue.
        }
    }
}

```

```
        Console.WriteLine(e.Message);
        continue;
    }

    // Push the subdirectories onto the stack for traversal.
    // This could also be done before handing the files.
    foreach (string str in subDirs)
        dirs.Push(str);
}
}
```

Geralmente, é muito demorado testar cada pasta para determinar se seu aplicativo tem permissão para abri-la. Portanto, o exemplo de código apenas coloca essa parte da operação em um bloco `try/catch`. É possível modificar o bloco `catch` para que, quando lhe for negado acesso a uma pasta, você possa tentar elevar as permissões e acessá-la novamente. Como regra, capture apenas as exceções que você puder manipular sem deixar seu aplicativo em um estado desconhecido.

Se você precisar armazenar o conteúdo de uma árvore de diretório, seja na memória ou no disco, a melhor opção é armazenar apenas a propriedade `FullName` (do tipo `string`) para cada arquivo. Você pode, então, usar essa cadeia de caracteres para criar um novo objeto `FileInfo` ou `DirectoryInfo`, conforme necessário ou abra qualquer arquivo que precisar de processamento adicional.

Programação robusta

O código de iteração de arquivo robusto deve levar em conta muitas complexidades do sistema de arquivos. Para saber mais sobre o sistema de arquivos do Windows, confira [Visão geral do NTFS](#).

Confira também

- [System.IO](#)
- [LINQ e diretórios de arquivos](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

Como obter informações sobre arquivos, pastas e unidades (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

No .NET, você pode acessar informações do sistema de arquivos usando as seguintes classes:

- [System.IO.FileInfo](#)
- [System.IO.DirectoryInfo](#)
- [System.IO.DriveInfo](#)
- [System.IO.Directory](#)
- [System.IO.File](#)

As classes [FileInfo](#) e [DirectoryInfo](#) representam um arquivo ou diretório e contêm propriedades que expõem muitos dos atributos de arquivo que têm suporte pelo sistema de arquivos NTFS. Elas também contêm métodos para abrir, fechar, mover e excluir arquivos e pastas. Você pode criar instâncias dessas classes passando uma cadeia de caracteres que representa o nome do arquivo, pasta ou unidade para o construtor:

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

Você também pode obter os nomes das unidades, pastas ou arquivos por meio de chamadas para [DirectoryInfo.GetDirectories](#), [DirectoryInfo.GetFiles](#) e [DriveInfo.RootDirectory](#).

As classes [System.IO.Directory](#) e [System.IO.File](#) fornecem métodos estáticos para recuperar informações sobre arquivos e diretórios.

Exemplo

O exemplo a seguir mostra várias maneiras de acessar informações sobre arquivos e pastas.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles(".*");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }

        // Get the subdirectories directly that is under the root.
    }
}
```

```

// See "How to: Iterate Through a Directory Tree" for an example of how to
// iterate through an entire tree.
System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories(".*");

foreach (System.IO.DirectoryInfo d in dirInfos)
{
    Console.WriteLine(d.Name);
}

// The Directory and File classes provide several static methods
// for accessing files and directories.

// Get the current application directory.
string currentDirName = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(currentDirName);

// Get an array of file names as strings rather than FileInfo objects.
// Use this method when storage space is an issue, and when you might
// hold on to the file name reference for a while before you try to access
// the file.
string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

foreach (string s in files)
{
    // Create the FileInfo object only when needed to ensure
    // the information is as current as possible.
    System.IO.FileInfo fi = null;
    try
    {
        fi = new System.IO.FileInfo(s);
    }
    catch (System.IO.FileNotFoundException e)
    {
        // To inform the user and continue is
        // sufficient for this demonstration.
        // Your application may require different behavior.
        Console.WriteLine(e.Message);
        continue;
    }
    Console.WriteLine("{0} : {1}", fi.Name, fi.Directory);
}

// Change the directory. In this case, first check to see
// whether it already exists, and create it if it does not.
// If this is not appropriate for your application, you can
// handle the System.IO.IOException that will be raised if the
// directory cannot be found.
if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\")) {
    System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");

    System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

    currentDirName = System.IO.Directory.GetCurrentDirectory();
    Console.WriteLine(currentDirName);

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

```

Programação robusta

Quando você processa cadeias de caracteres do caminho especificado pelo usuário, você também deve tratar

exceções para as seguintes condições:

- O nome do arquivo está malformado. Por exemplo, ele contém caracteres inválidos ou somente espaço em branco.
- O nome do arquivo é nulo.
- O nome de arquivo é maior que o comprimento máximo definido pelo sistema.
- O nome de arquivo contém dois-pontos (:).

Se o aplicativo não tem permissões suficientes para ler o arquivo especificado, o método `Exists` retorna `false` independentemente de se um caminho existe, o método não gera uma exceção.

Confira também

- [System.IO](#)
- [Guia de Programação em C#](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

Como criar um arquivo ou pasta (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

Você pode criar uma pasta no seu computador, criar uma subpasta, criar um arquivo na subpasta e gravar dados no arquivo programaticamente.

Exemplo

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
        }
    }
}
```

```

        }

    }
    else
    {
        Console.WriteLine("File \'{0}\' already exists.", fileName);
        return;
    }

    // Read and display the data from your file.
    try
    {
        byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
        foreach (byte b in readBuffer)
        {
            Console.Write(b + " ");
        }
        Console.WriteLine();
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
// Sample output:

// Path to my file: c:\Top-Level Folder\SubFolder\ttxvause.vv0

//0 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 27 28 29
//30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
// 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
//3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}

```

Se a pasta já existir, [.CreateDirectory](#) não fará nada, e nenhuma exceção será gerada. No entanto, [File.Create](#) substituirá um arquivo existente por um novo. O exemplo usa uma instrução `if - else` para impedir que um arquivo existente seja substituído.

Fazendo as alterações a seguir no exemplo, você pode especificar diferentes resultados com base em se já existe um arquivo com um determinado nome. Se esse arquivo não existir, o código criará um. Se esse arquivo existir, o código acrescentará dados a esse arquivo.

- Especifique um nome de arquivo não aleatório.

```

// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";

```

- Substitua a instrução `if - else` pela instrução `using` no código a seguir.

```

using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}

```

Execute o exemplo várias vezes para verificar se os dados são adicionados ao arquivo a cada vez.

Para ver mais valores `FileMode` que você pode tentar, consulte [FileMode](#).

As seguintes condições podem causar uma exceção:

- O nome da pasta está malformado. Por exemplo, ele contém caracteres inválidos ou é somente um espaço em branco (classe [ArgumentException](#)). Use a classe [Path](#) para criar nomes de caminho válidos.
- A pasta pai da pasta a ser criada é somente leitura (classe [IOException](#)).
- O nome da pasta é `null` (classe [ArgumentNullException](#)).
- O nome da pasta é longo demais (classe [PathTooLongException](#)).
- O nome da pasta contém apenas dois-pontos, ":" (classe [PathTooLongException](#)).

Segurança do .NET

Uma instância da classe [SecurityException](#) poderá ser gerada em situações de confiança parcial.

Se você não tiver permissão para criar a pasta, o exemplo gerará uma instância da [UnauthorizedAccessException](#) classe .

Confira também

- [System.IO](#)
- [Guia de Programação em C#](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

Como copiar, excluir e mover arquivos e pastas (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

Os exemplos a seguir mostram como copiar, mover e excluir arquivos e pastas de maneira síncrona usando as classes [System.IO.File](#), [System.IO.Directory](#), [System.IO.FileInfo](#) e [System.IO.DirectoryInfo](#) do namespace [System.IO](#). Esses exemplos não fornecem uma barra de progresso ou qualquer outra interface do usuário. Se você quiser fornecer uma caixa de diálogo de progresso padrão, consulte [Como fornecer uma caixa de diálogo de progresso para operações de arquivo](#).

Use [System.IO.FileSystemWatcher](#) para fornecer eventos que permitirão que você calcule o progresso ao operar em vários arquivos. Outra abordagem é usar a invocação de plataforma para invocar os métodos relacionados ao arquivo relevantes no Shell do Windows. Para obter informações sobre como executar essas operações de arquivo de forma assíncrona, consulte [E/S de arquivo assíncrona](#).

Exemplos

O exemplo a seguir mostra como copiar arquivos e diretórios.

```

// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath = @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder.
        // If the directory already exists, this method does not create a new directory.
        System.IO.Directory.CreateDirectory(targetPath);

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

```

O exemplo a seguir mostra como mover arquivos e diretórios.

```

// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}

```

O exemplo a seguir mostra como excluir arquivos e diretórios.

```

// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

```

```

// Delete a directory and all subdirectories with Directory static method...
if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
{
    try
    {
        System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
    }

    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }
}

// ...or with DirectoryInfo instance method.
System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
// Delete this dir and all subdirs.
try
{
    di.Delete(true);
}
catch (System.IO.IOException e)
{
    Console.WriteLine(e.Message);
}
}
}

```

Confira também

- [System.IO](#)
- [Guia de Programação em C#](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)
- [Como fornecer uma caixa de diálogo de progresso para operações de arquivo](#)
- [E/S de arquivo e fluxo](#)
- [Tarefas comuns de E/S](#)

Como fornecer uma caixa de diálogo de progresso para operações de arquivo (guia de programação C#)

21/01/2022 • 2 minutes to read

Você pode fornecer uma caixa de diálogo padrão que mostra o andamento em operações de arquivos no Windows se você usar o método `CopyFile(String, String, UIOption)` no namespace `Microsoft.VisualBasic`.

NOTE

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 adicionar uma referência no Visual Studio

1. Na barra de menus, escolha **Projeto, Adicionar Referência**.

A caixa de diálogo **Gerenciador de Referências** é exibida.

2. Na área **Assemblies**, escolha **Framework** se ele ainda não estiver escolhido.
3. Na lista de nomes, marque a caixa de seleção **Microsoft.VisualBasic** e, em seguida, escolha o botão **OK** para fechar a caixa de diálogo.

Exemplo

O código a seguir copia o diretório que `sourcePath` especifica, para o diretório que `destinationPath` especifica. Esse código também fornece uma caixa de diálogo padrão que mostra a quantidade estimada de tempo que resta antes da conclusão da operação.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.  
using Microsoft.VisualBasic.FileIO;  
  
class FileProgress  
{  
    static void Main()  
    {  
        // Specify the path to a folder that you want to copy. If the folder is small,  
        // you won't have time to see the progress dialog box.  
        string sourcePath = @"C:\Windows\symbols\";  
        // Choose a destination for the copied files.  
        string destinationPath = @"C:\TestFolder";  
  
        FileSystem.CopyDirectory(sourcePath, destinationPath,  
            UIOption.AllDialogs);  
    }  
}
```

Confira também

- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

Como gravar em um arquivo de texto (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

Neste artigo, há vários exemplos mostrando várias maneiras de gravar texto em um arquivo. Os dois primeiros exemplos usam métodos de conveniência estáticos na classe para gravar cada elemento de `System.IO.File` qualquer e um em um arquivo de `IEnumerable<string>` `string` texto. O terceiro exemplo mostra como adicionar texto a um arquivo quando você precisa processar cada linha individualmente conforme você escreve no arquivo. Nos três primeiros exemplos, você substitui todo o conteúdo existente no arquivo. O exemplo final mostra como anexar texto a um arquivo existente.

Todos esses exemplos gravam literais de cadeia de caracteres em arquivos. Se você quiser formatar o texto gravado em um arquivo, use o método `Format` ou o recurso de [interpolação de cadeia de caracteres](#) do C#.

Gravar uma coleção de cadeias de caracteres em um arquivo

```
using System.IO;
using System.Threading.Tasks;

class WriteAllLines
{
    public static async Task ExampleAsync()
    {
        string[] lines =
        {
            "First line", "Second line", "Third line"
        };

        await File.WriteAllLinesAsync("WriteLines.txt", lines);
    }
}
```

O exemplo de código-fonte anterior:

- Instala uma matriz de cadeia de caracteres com três valores.
- Aguarda uma chamada para `File.WriteAllLinesAsync` a qual:
 - Cria de forma assíncrona um nome de `arquivoWriteLines.txt`. Se o arquivo já existir, ele será substituído.
 - Grava as linhas fornecidas no arquivo.
 - Fecha o arquivo, liberando e despondo automaticamente conforme necessário.

Gravar uma cadeia de caracteres em um arquivo

```

using System.IO;
using System.Threading.Tasks;

class WriteAllText
{
    public static async Task ExampleAsync()
    {
        string text =
            "A class is the most powerful data type in C#. Like a structure, " +
            "a class defines the data and behavior of the data type. ";

        await File.WriteAllTextAsync("WriteText.txt", text);
    }
}

```

O exemplo de código-fonte anterior:

- Instalita uma cadeia de caracteres considerando o literal de cadeia de caracteres atribuído.
- Aguarda uma chamada para [File.WriteAllTextAsync](#) a qual:
 - Cria de forma assíncrona um nome de *arquivoWriteText.txt*. Se o arquivo já existir, ele será substituído.
 - Grava o texto determinado no arquivo.
 - Fecha o arquivo, liberando e despondo automaticamente conforme necessário.

Gravar cadeias de caracteres selecionadas de uma matriz em um arquivo

```

using System.IO;
using System.Threading.Tasks;

class StreamWriterOne
{
    public static async Task ExampleAsync()
    {
        string[] lines = { "First line", "Second line", "Third line" };
        using StreamWriter file = new("WriteLines2.txt");

        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                await file.WriteLineAsync(line);
            }
        }
    }
}

```

O exemplo de código-fonte anterior:

- Instalita uma matriz de cadeia de caracteres com três valores.
- Instalita um com [StreamWriter](#) um caminho de arquivo de *WriteLines2.txt* como uma [declaração using](#).
- Itera por todas as linhas.
- Aguarda condicionalmente uma chamada para , que grava a linha no arquivo quando a [StreamWriter.WriteLineAsync\(String\)](#) linha não contém "Second" .

Anexar texto a um arquivo existente

```
using System.IO;
using System.Threading.Tasks;

class StreamWriterTwo
{
    public static async Task ExampleAsync()
    {
        using StreamWriter file = new("WriteLines2.txt", append: true);
        await file.WriteLineAsync("Fourth line");
    }
}
```

O exemplo de código-fonte anterior:

- Instala uma matriz de cadeia de caracteres com três valores.
- Instala um com um caminho de arquivo deWriteLines2.txt como uma declaração `StreamWriter` `using`, passando para `true` anexar.
- Aguarda uma chamada para `StreamWriter.WriteLineAsync(String)`, que grava a cadeia de caracteres no arquivo como uma linha acrescentada.

Exceções

As seguintes condições podem causar uma exceção:

- `InvalidOperationException`: o arquivo existe e é somente leitura.
- `PathTooLongException`: o nome do caminho pode ser muito longo.
- `IOException`: o disco pode estar cheio.

Há condições adicionais que podem causar exceções ao trabalhar com o sistema de arquivos, é melhor programar de forma defensiva.

Consulte também

- [Guia de Programação em C#](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

Como ler de um arquivo de texto (guia de programação C#)

21/01/2022 • 2 minutes to read

Este exemplo lê o conteúdo de um arquivo de texto usando os métodos estáticos `ReadAllText` e `ReadAllLines` da classe `System.IO.File`.

Para obter um exemplo que usa `StreamReader`, consulte [como ler um arquivo de texto, uma linha por vez](#).

NOTE

Os arquivos usados neste exemplo são criados no tópico [como gravar em um arquivo de texto](#).

Exemplo

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Compilando o código

Copie o código e cole-o em um aplicativo de console em C#.

Se você não estiver usando os arquivos de texto de [como gravar em um arquivo de texto](#), substitua o argumento para `ReadAllText` e `ReadAllLines` pelo caminho e nome de arquivo apropriados no computador.

Programação robusta

As seguintes condições podem causar uma exceção:

- O arquivo não existe ou não existe no local especificado. Verifique o caminho e a ortografia do nome do arquivo.

Segurança do .NET

Não confie no nome de um arquivo para determinar o conteúdo do arquivo. Por exemplo, o arquivo `myFile.cs` pode não ser um arquivo de origem do C#.

Confira também

- [System.IO](#)
- [Guia de programação C#](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

Como ler um arquivo de texto uma linha por vez (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

Este exemplo lê o conteúdo de um arquivo de texto, uma linha por vez, em uma cadeia de caracteres usando o método `ReadLines` da classe `File`. Cada linha de texto é armazenada na cadeia de caracteres `line` e exibida na tela.

Exemplo

```
int counter = 0;

// Read the file and display it line by line.
foreach (string line in System.IO.File.ReadLines(@"c:\test.txt"))
{
    System.Console.WriteLine(line);
    counter++;
}

System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

Compilando o código

Copie o código e cole-o no método `Main` de um aplicativo de console.

Substitua `"c:\test.txt"` pelo nome do arquivo real.

Programação robusta

As seguintes condições podem causar uma exceção:

- O arquivo pode não existir.

Segurança do .NET

Não tome decisões sobre o conteúdo do arquivo com base no nome do arquivo. Por exemplo, o arquivo `myFile.cs` pode não ser um arquivo de origem do C#.

Confira também

- [System.IO](#)
- [Guia de Programação em C#](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

Como criar uma chave no Registro (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

Este exemplo adiciona o par de valores, "Name" e "Isabella", ao Registro do usuário atual, sob a chave "Names".

Exemplo

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

Compilando o código

- Copie o código e cole-o no método `Main` de um aplicativo de console.
- Substitua o parâmetro `Names` pelo nome de uma chave existente diretamente sob o nó `HKEY_CURRENT_USER` do Registro.
- Substitua o parâmetro `Name` pelo nome de um valor que existe diretamente sob o nó `Names`.

Programação robusta

Analise a estrutura do Registro para encontrar um local adequado para a chave. Por exemplo, caso você queira abrir a chave Software do usuário atual e criar uma chave com o nome da empresa. Em seguida, adicione os valores do Registro à chave da empresa.

As seguintes condições podem causar uma exceção:

- O nome da chave é nulo.
- O usuário não tem permissões para criar chaves do Registro.
- O nome da chave excede o limite de 255 caracteres.
- A chave é fechada.
- A chave do Registro é somente leitura.

Segurança do .NET

É mais seguro gravar dados na pasta do usuário — `Microsoft.Win32.Registry.CurrentUser` — em vez de no computador local — `Microsoft.Win32.Registry.LocalMachine`.

Ao criar um valor de Registro, é necessário decidir o que fazer se esse valor já existir. Outro processo, talvez um mal-intencionado, pode já ter criado o valor e tem acesso a ele. Ao colocar dados no valor de Registro, os dados estarão disponíveis para o outro processo. Para impedir isso, use o método

`Overload:Microsoft.Win32.RegistryKey.GetValue` método. Ele retornará null se a chave ainda não existir.

Não é seguro armazenar segredos, como senhas, no Registro como texto sem formatação, mesmo se a chave do Registro estiver protegida por ACL (listas de controle de acesso).

Confira também

- [System.IO](#)
- [Guia de Programação em C#](#)
- [Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)
- [Ler, gravar e excluir do Registro com C#](#)

Interoperabilidade (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

A interoperabilidade permite que você mantenha e aproveite os investimentos existentes em código não gerenciado. O código que é executado sob o controle do CLR (Common Language Runtime) é chamado de *código gerenciado*, e o código que é executado fora do CLR é chamado de *código não gerenciado*. COM, COM+, componentes do C++, componentes do ActiveX e a API do Microsoft Windows são exemplos de código não gerenciado.

O .NET habilita a interoperabilidade com código não tratado por meio de serviços de invocação de plataforma, [System.Runtime.InteropServices](#) namespace, interoperabilidade C++ e interoperabilidade COM (interoperabilidade COM).

Nesta seção

[Visão geral sobre interoperabilidade](#)

Descreve métodos para fins de interoperabilidade entre código gerenciado em C# e código não gerenciado.

[Como acessar objetos de interoperabilidade do Office usando recursos do C#](#)

Descreve os recursos que são introduzidos no Visual C# para facilitar a programação do Office.

[Como usar propriedades indexadas na programação para interoperabilidade COM](#)

Descreve como usar propriedades indexadas para acesso propriedades COM que têm parâmetros.

[Como usar invocação de plataforma para executar um arquivo WAV](#)

Descreve como usar os serviços de invocação de plataforma para reproduzir um arquivo de som .wav no sistema operacional Windows.

[Passo a passo: programação do Office](#)

Mostra como criar uma planilha do Excel e um documento do Word com um link para a planilha.

[Exemplo de classe COM](#)

Demonstra como expor uma classe C# como um objeto COM.

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

- [Marshal.ReleaseComObject](#)
- [Guia de Programação em C#](#)
- [Interoperação com código não gerenciado](#)
- [Passo a passo: programação do Office](#)

Visão geral sobre interoperabilidade (Guia de Programação em C#)

21/01/2022 • 3 minutes to read

O tópico descreve métodos para permitir a interoperabilidade entre código gerenciado e código não gerenciado do C#.

Invocação de plataforma

A *invocação de plataforma* é um serviço que permite ao código gerenciado chamar funções não gerenciadas que são implementadas em DLLs (bibliotecas de vínculo dinâmico), como aquelas na API do Microsoft Windows. Ela localiza e invoca uma função exportada e realiza marshaling dos argumentos (inteiros, cadeias de caracteres, matrizes, estruturas e assim por diante) além do limite de interoperação, conforme necessário.

Para obter mais informações, consulte [Consumindo funções de DLL não gerenciadas](#) e [Como usar a invocação de plataforma para reproduzir um arquivo WAV](#).

NOTE

O [CLR](#) (Common Language Runtime) gerencia o acesso aos recursos do sistema. Chamar código não gerenciado que esteja fora do CLR ignora esse mecanismo de segurança e, portanto, apresenta um risco de segurança. Por exemplo, código não gerenciado pode chamar recursos diretamente em código não gerenciado, ignorando os mecanismos de segurança do CLR. Para obter mais informações, confira [Segurança no .NET](#).

Interoperabilidade C++

Você pode usar a interop do C++, também conhecida como It Just Works (IJW), para envolver uma classe nativa do C++ para que ela possa ser consumida pelo código que é desenvolvido em C# ou em outra linguagem .NET. Para fazer isso, você deve escrever código C++ para encapsular um componente nativo DLL ou COM. Ao contrário de outras linguagens .NET, o Visual C++ tem suporte de interoperabilidade que permite que o código gerenciado e não gerenciado seja localizado no mesmo aplicativo e até mesmo no mesmo arquivo. Então, você compila o código C++ usando a opção do compilador `/clr` para produzir um assembly gerenciado. Finalmente, você adiciona uma referência ao assembly no seu projeto do C# e usa os objetos encapsulados, assim como usaria outras classes gerenciadas.

Expondo componentes COM ao C#

Você pode consumir um componente COM de um projeto do C#. As etapas gerais são as seguintes:

1. Localize um componente COM para usar e registre-o. Use `regsvr32.exe` para registrar ou cancelar o registro de uma DLL do COM.
2. Adicione ao projeto uma referência ao componente COM ou à biblioteca de tipo.

Quando você adiciona a referência, Visual Studio usa o `Tlbimp.exe` (Importador de Biblioteca de [Tipos](#)), que usa uma biblioteca de tipos como entrada, para saída de um assembly de interop do .NET. O assembly, também chamado de RCW (Runtime Callable Wrapper), contém classes gerenciadas e interfaces que encapsulam as classes COM e as interfaces que estão na biblioteca de tipos. O Visual Studio adiciona ao projeto uma referência ao assembly gerado.

3. Crie uma instância de uma classe que esteja definida no RCW. Isso, por sua vez, criará uma instância do objeto COM.
4. Use o objeto da mesma maneira que usa outros objetos gerenciados. Quando o objeto é recuperado pela coleta de lixo, a instância do objeto COM também é liberada da memória.

Para obter mais informações, consulte [Expondo componentes COM para o .NET Framework](#).

Expondo o C# para o COM

Os clientes COM podem consumir tipos do C# que foram expostos corretamente. As etapas básicas para expor os tipos do C# são as seguintes:

1. Adicione atributos de interoperabilidade no projeto do C#.

Você pode tornar visível um assembly COM ao modificar as propriedades do projeto do Visual C#. Para obter mais informações, consulte [Caixa de diálogo de informações do assembly](#).

2. Gere e registre uma biblioteca de tipos COM para ser usada pelo COM.

Você pode modificar as propriedades do projeto do Visual C# para registrar automaticamente o assembly do C# para a interoperabilidade COM. O Visual Studio usa a [Regasm.exe \(ferramenta Assembly Registration\)](#), com a opção de linha de comando `/t1b`, que usa um assembly gerenciado como entrada para gerar uma biblioteca de tipos. Esta biblioteca de tipos descreve os tipos `public` no assembly e adiciona as entradas do Registro para que os clientes COM possam criar classes gerenciadas.

Para obter mais informações, consulte [Expondo componentes do .NET Framework para o COM](#) e [Classe COM de exemplo](#).

Confira também

- [Melhorando o desempenho de interoperabilidade](#)
- [Introdução à interoperabilidade entre COM e .NET](#)
- [Introdução à interoperabilidade COM em Visual Basic](#)
- [Marshaling entre código gerenciado e não gerenciado](#)
- [Interoperação com código não gerenciado](#)
- [Guia de Programação em C#](#)

Como acessar Office de interop (Guia de Programação em C#)

21/01/2022 • 12 minutes to read

O C# tem recursos que simplificam o acesso a Office de API. 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 tópico, você usará os novos recursos para escrever código que cria e exibe uma planilha do Microsoft Office Excel. Em seguida, você irá escrever 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.

NOTE

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 criar um novo aplicativo de console

1. Inicie o Visual Studio.
2. No menu **Arquivo**, aponte para **Novo** e clique em **Projeto**. A caixa de diálogo **Novo Projeto** aparecerá.
3. No painel **Modelos Instalados**, expanda **Visual C#** e, em seguida, selecione **Windows**.
4. Observe a parte superior da caixa de diálogo **Novo Projeto** para verificar se **.NET Framework 4** (ou versão posterior) está selecionado como uma estrutura de destino.
5. No painel **Modelos**, clique em **Aplicativo de Console**.
6. Digite um nome para o projeto no campo **Nome**.
7. Clique em **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, clique em **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 garantir que eles sejam instalados e exibidos. Consulte [How to: Install Office Primary Interop Assemblies](#).
3. Clique em **OK**.

Para adicionar as diretivas using necessárias

1. No Gerenciador de Soluções, clique com o botão direito do mouse no arquivo `Program.cs` e, em seguida, clique em **Exibir Código**.
2. Adicione as seguintes `using` diretivas à parte superior do arquivo de código:

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Para criar uma lista de contas bancárias

1. Cole a seguinte definição de classe em `Program.cs`, na classe `Program`.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Adicione o seguinte código ao método `Main` para criar uma lista `bankAccounts` que contenha duas contas.

```
// 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, novos no C# 4, permitem omitir o argumento para esse parâmetro se você quiser usar o valor padrão do parâmetro. Como nenhum argumento é enviado no código a seguir, `Add` usa o modelo padrão e cria uma nova pasta de trabalho. A instrução equivalente em versões anteriores do C# requer um argumento de espaço reservado: `ExcelApp.Workbooks.Add(Type.Missing)`.

```

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;
}

```

2. Adicione o código a seguir no final de `DisplayInExcel`. O código insere valores nas duas primeiras colunas da primeira linha da planilha.

```

// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";

```

3. 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.

```

var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}

```

4. Adicione o seguinte código no final de `DisplayInExcel` para ajustar as larguras das colunas para adequar o conteúdo.

```

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.

```

((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();

```

O C# 4 e versões posteriores convertem o retornado para automaticamente se o assembly for referenciado pela opção do `Object dynamic` compilador **EmbedInteropTypes** ou, de forma equivalente, se a propriedade **Excel Inserir Tipos de Interop** estiver definida como `true`. `True` é o valor padrão para essa propriedade.

Para executar o projeto

1. Adicione a seguinte linha no final de `Main`.

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

2. Pressione CTRL+F5.

Uma planilha do Excel é exibida contendo os dados das duas contas.

Para adicionar um documento do Word

1. Para ilustrar maneiras adicionais pelas quais o C# 4 e versões posteriores aprimoraram a programação do Office, o código a seguir abre um aplicativo Word e cria um ícone 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 novos recursos apresentados no C# 4 que simplificam as chamadas aos 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. No C# 4.0 e versões posteriores, você poderá omitir argumentos para alguns ou todos os parâmetros se desejar usar os valores padrão. No C# 3.0 e versões anteriores, um argumento deve ser fornecido para cada parâmetro e o argumento deve ser uma variável, pois os parâmetros são parâmetros de referência.

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. No C# 4.0 e versões posteriores, você pode usar argumentos nomeados para esses dois e omitir os outros. Embora esses 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. No C# 3.0 e versões anteriores, você deve fornecer um argumento variável para cada parâmetro de referência.

```
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);  
}
```

No C# 3.0 e versões anteriores da linguagem, o código a seguir mais complexo é necessário.

```

static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to construct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
        ref link,
        ref useDefaultValue,
        ref displayAsIcon,
        ref useDefaultValue,
        ref useDefaultValue,
        ref useDefaultValue);
}

```

2. Adicione a instrução a seguir no final de `Main`.

```

// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();

```

3. Adicione a instrução a seguir no final de `DisplayInExcel`. O método `Copy` adiciona a planilha na área de transferência.

```

// 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();

```

4. 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

1. Melhorias adicionais 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 os tipos necessários e retornados por métodos COM

podem ser representados usando o tipo `dynamic`, em vez de `object`. 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](#).

No C# 4, 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 pois a conversão explícita não é necessária. 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.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. 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**.
3. Se você não conseguir ver a janela **Propriedades**, pressione **F4**.
4. Localize **Inserir Tipos Interop** na lista de propriedades e altere seu valor para **False**. De forma 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

1. Substitua as duas chamadas para `AutoFit` em `DisplayInExcel` pela instrução a seguir.

```
// 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, um argumento é fornecido 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 o nome do parâmetro estiver incluído, conforme mostrado no código a seguir.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Pressione CTRL + F5 para ver o resultado. Outros formatos estão listados na enumeração `XlRangeAutoFormat`.
3. Compare a instrução na etapa 1 com o código a seguir, que mostra os argumentos que são necessários no C# 3.0 ou versões anteriores.

```

// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);

```

Exemplo

O código a seguir mostra um exemplo completo.

```

using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    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](#)
- [Usando o tipo dynamic](#)
- [Argumentos nomeados e opcionais](#)
- [Como usar argumentos nomeados e opcionais na programação do Office](#)

Como usar propriedades indexadas em programação de interoperabilidade COM (guia de programação C#)

21/01/2022 • 2 minutes to read

As *propriedades indexadas* melhoram a maneira na qual as propriedades COM que têm parâmetros são consumidas na programação em C#. As propriedades indexadas trabalham juntamente com outras funcionalidades no Visual C#, como [argumentos nomeados e opcionais](#), um novo tipo ([dinâmico](#)) e [informações de tipo inseridas](#) para melhorar a programação do Microsoft 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.

```
// 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:

```
// Visual C# 2010.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

O exemplo anterior também usa o recurso [argumentos opcionais](#), que permite que você omita `Type.Missing`.

Da mesma forma, para definir o valor da propriedade `value` de um objeto `Range` no C# 3.0 e versões anteriores, são necessários dois argumentos. Um fornece um argumento para um parâmetro opcional que especifica o tipo do valor de intervalo. O outro fornece o valor para a propriedade `Value`. Os exemplos a seguir ilustram essas técnicas. Ambos definem o valor da célula A1 como `Name`.

```
// Visual C# 2008.  
targetRange.set_Value(Type.Missing, "Name");  
// Or  
targetRange.Value2 = "Name";
```

Em vez disso, as propriedades indexadas permitem escrever o código a seguir.

```
// 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 de Office, consulte [como acessar Office objetos de interoperabilidade usando os recursos do C#](#).

```
// 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();
            //CSharp2008();
        }

        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 argumentos nomeados e opcionais na programação do Office](#)
- [Como acessar objetos de interoperabilidade do Office usando recursos do C#](#)
- [Passo a passo: programação do Office](#)

Como usar a invocação de plataforma para reproduzir um arquivo WAV (guia de programação C#)

21/01/2022 • 2 minutes to read

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 reproduzido usando o `PlaySound()` método da biblioteca de `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, clique em **Abrir** para reproduzir o arquivo wave usando a invocação de plataforma. Uma caixa de texto exibe o caminho completo do arquivo selecionado.

A caixa de diálogo **Abrir Arquivos** é filtrada por meio das configurações de filtro para mostrar somente os arquivos que têm uma extensão `.wav`:

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```

using System.Windows.Forms;
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:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            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.
        }
    }
}

```

Compilando o código

1. crie um novo projeto de aplicativo Windows Forms em C# no Visual Studio e nomeie-o **WinSound**.
2. Copie o código acima e cole-o sobre o conteúdo do arquivo *Form1.cs*.
3. Copie o código a seguir e cole-o no arquivo *Form1.designer.cs*, no `InitializeComponent()` método, 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();
```

4. Compile e execute o código.

Confira também

- [Guia de programação C#](#)
- [Visão geral sobre interoperabilidade](#)
- [Um olhar detalhado sobre invocação de plataforma](#)
- [Marshaling de dados com invocação de plataforma](#)

Passo a passo: Programação do Office (C# e Visual Basic)

21/01/2022 • 11 minutes to read

O Visual Studio oferece funcionalidades no C# e no Visual Basic 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. As funcionalidades do Visual Basic incluem propriedades autoimplementadas, instruções em expressões lambda e inicializadores de coleção.

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.

Pré-requisitos

Você deve ter o Microsoft Office Excel e o Microsoft Office Word instalados no computador para concluir esse passo a passo.

NOTE

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 configurar um aplicativo Suplemento do Excel

1. Inicie o Visual Studio.
2. No menu **Arquivo**, aponte para **Novo** e clique em **Projeto**.
3. No painel **Modelos Instalados**, expanda **Visual Basic** ou **Visual C#**, expanda **Office** e, em seguida, clique no ano da versão do produto do Office.
4. No painel **Modelos**, clique em **Excel <version> Add-in**.
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. Clique em **OK**.
8. 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,

clique em **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 garantir que eles sejam instalados e exibidos (consulte [Como instalar Office assemblies de interop primários](#)).

3. Clique em **OK**.

Para adicionar instruções Imports necessárias ou diretivas de uso

1. No Gerenciador de Soluções, clique com o botão direito do mouse no arquivo **ThisAddIn.vb** ou **ThisAddIn.cs** e clique em **Exibir Código**.
2. Adicione as seguintes instruções **Imports** (Visual Basic) ou diretivas **using** (C#) à parte superior do arquivo de código, se elas ainda não estiverem presentes.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

Para criar uma lista de contas bancárias

1. No Gerenciador de Soluções, clique com o botão direito do mouse no nome do projeto, clique em **Adicionar** e clique em **Classe**. Nomeie a classe **Account.vb** se você estiver usando Visual Basic ou **Account.cs**, se você estiver usando C#. Clique em **Adicionar**.
2. Substitua a definição da classe **Account** pelo código a seguir. As definições de classe usam *propriedades autoimplementadas*. Para obter mais informações, consulte [Propriedades autoimplementadas](#).

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. Para criar uma lista que contém duas contas, adicione o código a seguir ao método **bankAccounts ThisAddIn_Startup** em **ThisAddIn.vb** ou **ThisAddIn.cs**. As declarações de lista usam *inicializadores de coleção*. Para obter mais informações, consulte [Inicializadores de coleção](#).

```

var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};

```

```

Dim bankAccounts As New List(Of Account) From {
    New Account With {
        .ID = 345,
        .Balance = 541.27
    },
    New Account With {
        .ID = 123,
        .Balance = -127.44
    }
}

```

Para exportar dados para o Excel

1. No mesmo arquivo, adicione o método a seguir para a classe `ThisAddIn`. O método configura uma planilha do Excel e exporta dados para ela.

```

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();
}

```

```

Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
                   ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub

```

Dois novos recursos do C# são usados neste método. Esses dois recursos já existem no Visual Basic.

- Method `Add` tem um *parâmetro opcional* para especificar um modelo específico. Os parâmetros opcionais, novos no C# 4, permitem omitir o argumento para esse parâmetro se você quiser usar o valor padrão do parâmetro. Como nenhum argumento é enviado no código anterior, `Add` usa o modelo padrão e cria uma nova 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.

```

// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();

```

Em versões anteriores da linguagem, a sintaxe especial a seguir é obrigatória.

```

// In Visual C# 2008, you cannot access the Range, Offset, and Value
// properties directly.
excelApp.get_Range("A1").Value2 = "ID";
excelApp.ActiveCell.get_Offset(1, 0).Select();

```

Não é possível criar propriedades indexadas de sua preferência. O recurso dá suporte apenas ao consumo de propriedades indexadas existentes.

Para obter mais informações, consulte [Como usar propriedades indexadas na programação de interop COM](#).

2. Adicione o seguinte código no final de `DisplayInExcel` para ajustar as larguras das colunas para adequar o conteúdo.

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.
.Columns(1).AutoFit()
.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](#). Isso ocorre automaticamente quando Inserir Tipos de [Interop](#) é definido como seu valor padrão, ou, equivalentemente, quando o assembly é referenciado pela `True` opção do compilador [EmbedInteropTypes](#). O tipo `dynamic` permite a vinculação posterior, já disponível no Visual Basic, e evita a conversão explícita necessária no C# 3.0 e em versões anteriores da linguagem.

Por exemplo, `excelApp.Columns[1]` retorna um `Object` e `AutoFit` é um método [Range](#) do Excel. Sem `dynamic`, você deve converter o objeto retornado em `excelApp.Columns[1]` como uma instância de `Range` antes de chamar o método `AutoFit`.

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

Para obter mais informações sobre como inserir tipos de interoperabilidade, consulte os procedimentos "Para localizar a referência de PIA" e "Para restaurar a dependência de PIA" posteriormente neste tópico. Para obter mais informações sobre `dynamic`, consulte [dynamic](#) ou [Usando o tipo dynamic](#).

Para invocar `DisplayInExcel`

1. 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 com várias linhas que define como os dados deverão ser processados. 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](#).

```
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;
    }
});
```

```

DisplayInExcel(bankAccounts,
    Sub(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 Then
            cell.Interior.Color = RGB(255, 0, 0)
            cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
        End If
    End Sub)

```

2. Para executar o programa, pressione F5. Uma planilha do Excel é exibida contendo os dados das contas.

Para adicionar um documento do Word

1. 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.

```

var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);

```

```

Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)

```

Esse código demonstra vários dos novos recursos no C#: a capacidade para omitir a palavra-chave `ref` em programação COM, argumentos nomeados e argumentos opcionais. Esses recursos já existem no Visual Basic. O [método PasteSpecial](#) tem sete parâmetros, todos definidos como 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 são enviados para indicar 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 Visual C# permite também que você omita a palavra-chave `ref` nesses argumentos.

Para executar o aplicativo

1. 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.

Para limpar o projeto concluído

1. 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.

Para localizar a referência de PIA

1. Execute o aplicativo novamente, mas não clique em **Limpar Solução**.
2. Selecione **Iniciar**. Localize **Microsoft Visual Studio <version>** e abra um 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 **Visual Studio <version>** 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. Observe que os namespaces `Microsoft.Office.Interop.Excel` e `Microsoft.Office.Interop.Word` estão incluídos no assembly. 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 incluídos na lista. Como os tipos do seu projeto precisam ter sido importados para o assembly, referências a um PIA não são necessárias. Isso facilita a implantação. Os PIAs não precisam estar presentes no computador do usuário e como um aplicativo não requer a implantação de uma versão específica de um PIA, os aplicativos podem ser projetados para trabalhar 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. No entanto, isso funcionará apenas se seu código não usar quaisquer APIs que não estejam disponíveis na versão do Office na qual você está trabalhando. Não é sempre claro se uma determinada API estava disponível em uma versão anterior e, por essa razão, não é recomendado trabalhar com versões anteriores do Office.

NOTE

O Office não publicou PIAs antes do Office 2003. Portanto, a única maneira de gerar um assembly de interoperabilidade para o Office 2002 ou versões anteriores é importando referência COM.

6. Feche a janela do manifesto e a janela do assembly.

Para restaurar a dependência de PIA

1. Em **Gerenciador de Soluções**, clique no 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, clique em **Limpar Solução** no menu **Compilar** para limpar o projeto concluído.

Confira também

- [Propriedades autoimplementadas \(Visual Basic\)](#)
- [Propriedades autoimplementadas \(C#\)](#)
- [Inicializadores de coleção](#)
- [Inicializadores de objeto e coleção](#)
- [Parâmetros Opcionais](#)
- [Passando argumentos por posição e nome](#)
- [Argumentos nomeados e opcionais](#)
- [Associação antecipada e tardia](#)
- [dinâmico](#)
- [Usando o tipo dynamic](#)
- [Expressões lambda \(Visual Basic\)](#)
- [Expressões lambda \[C#\]](#)
- [Como usar propriedades indexadas na programação para interoperabilidade COM](#)
- [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](#)
- [COM Interop](#)
- [Interoperabilidade](#)

Exemplo de classe COM (Guia de Programação em C#)

21/01/2022 • 2 minutes to read

A seguir está um exemplo de uma classe que você poderia expor como um objeto COM. Depois que esse código é colocado em um arquivo.cs e adicionado ao 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 Visual 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 na classe que não são declarados nessas interfaces não estarão visíveis para COM, mas estarão 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 na qual os membros são declarados 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 será a interface de classe padrão. Implemente os métodos e propriedades expostos ao COM aqui. Eles devem ser marcados como públicos e devem corresponder às declarações na interface de classe. Além disso, declare aqui os eventos acionados pela classe. Eles devem ser marcados como públicos e devem corresponder às declarações na interface de eventos.

Exemplo

```
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

- [Guia de Programação em C#](#)
- [Interoperabilidade](#)
- [Página de Build, Designer de Projeto \(C#\)](#)

Referência de C#

21/01/2022 • 4 minutes to read

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 do C#

Fornece links para informações sobre caracteres especiais contextuais em C# e seu uso.

Diretivas de 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 C#

Inclui snippets de código que demonstram a causa e a correção de erros do compilador do C# e avisos.

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 – 5ª 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 no formulário de proposta de rascunho. Essas especificações serão refinadas e enviadas para o Comitê de padrões ECMA para revisão formal e Incorporation em uma versão futura do C# Standard.

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 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 o trabalho com `fixed` campos, incluindo o uso de `stackalloc` alocações. Variáveis locais declaradas com a `ref` palavra-chave podem ser reatribuídas para se referirem 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 anuláveis, correspondência de padrões recursivos, métodos de interface padrão, fluxos assíncronos, intervalos e índices, com base em padrões usando e usando declarações, atribuição de União nula e membros de instância `ReadOnly`.

[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, init somente setters, novas expressões com tipo de destino, inicializadores de módulo, extensão de métodos parciais, funções anônimas estáticas, expressões condicionais de tipo de destino, tipos de retorno covariantes, extensão getenumerator em loops foreach, parâmetros de descarte de lambda, atributos em funções locais, inteiros de tamanho nativo, ponteiros de função, supressão de emissão de sinalizador localsinit

[Propostas de especificação C# 10](#)

O C# 10 está disponível com o .NET 6. Os recursos incluem structs de registro, construtores de struct sem parâmetros, funções globais usando diretivas, namespaces com escopo de arquivo, padrões de propriedade estendida, cadeias de caracteres interpoladas, cadeias de caracteres interpoladas constantes, melhorias de lambda, expressão de argumento de chamador, diretivas aprimoradas `#line`, atributos genéricos, análise de atribuição definitiva aprimorada e `AsyncMethodBuilder` substituição.

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 C#](#)

Inclui informações sobre como usar a linguagem de programação do C#.

Controle de versão da linguagem C#

21/01/2022 • 5 minutes to read

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 a estrutura de destino. Você se beneficia do acesso aos recursos de linguagem mais recentes compatíveis com o destino do seu projeto. Essa opção padrão também garante que você não use uma linguagem que exija tipos ou comportamento de tempo de execução não disponível em sua estrutura de destino. Escolher uma versão de idioma mais recente do que o padrão pode causar dificuldade para diagnosticar erros de tempo de compilação e Runtime.

as regras neste artigo se aplicam ao compilador fornecido com o Visual Studio 2019 ou o SDK do .net. Os compiladores do C# que fazem parte da instalação do Visual Studio 2017 ou de versões anteriores do SDK do .NET Core são direcionados ao C# 7.0 por padrão.

O C# 8,0 tem suporte apenas no .NET Core 3. x e em versões mais recentes. Muitos dos recursos mais recentes exigem recursos de biblioteca e tempo de execução introduzidos no .NET Core 3. x:

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

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

O C# 10 tem suporte apenas no .NET 6 e em versões mais recentes.

Padrões

O compilador determina um padrão com base nestas regras:

ESTRUTURA DE DESTINO	VERSION	PADRÃO DA VERSÃO DA LINGUAGEM C#
.NET	6.x	C# 10
.NET	5	C# 9,0
.NET Core	3.x	C# 8,0
.NET Core	2. x	C# 7,3
.NET Standard	2.1	C# 8,0
.NET Standard	2.0	C# 7,3
.NET Standard	1.x	C# 7,3

ESTRUTURA DE DESTINO	VERSION	PADRÃO DA VERSÃO DA LINGUAGEM C#
.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 visualização em qualquer ambiente, sem afetar projetos direcionados a uma versão lançada do .NET Core.

IMPORTANT

Visual Studio 2017 adicionou uma `<LangVersion>latest</LangVersion>` entrada a qualquer arquivo de projeto criado. Isso significava o C# 7,0 quando ele foi adicionado. No entanto, depois de atualizar para Visual Studio 2019, isso significa a última versão lançada, independentemente da estrutura de destino. Esses projetos agora **substituem o comportamento padrão**. Você deve editar o arquivo de projeto e remover esse nó. Em seguida, seu projeto usará a versão do compilador recomendada para sua estrutura de destino.

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](#).
- Configure a [opção de compilador langversion](#).

TIP

Para saber qual versão de idioma você está usando no momento, coloque `#error version` (diferencia maiúsculas de minúsculas) em seu código. Isso faz com que o compilador relate um erro do compilador, CS8304, com uma mensagem contendo a versão do compilador que está sendo usada e a versão do idioma atual selecionada. Consulte [#error \(referência C#\)](#) 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:

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

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

Configurar vários projetos

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

```
<Project>
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
</Project>
```

As compilações em todos os subdiretórios do diretório que contém esse arquivo usarão a versão preview C#. Para obter mais informações, consulte [personalizar sua compilação](#).

Referência à versão da linguagem C#

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

VALOR	SIGNIFICADO
preview	O compilador aceita todas as sintaxes de linguagem válidas da versão prévia mais recente.
latest	O compilador aceita a sintaxe da versão lançada mais recente do compilador (incluindo a versão secundária).
latestMajor (default)	O compilador aceita a sintaxe da versão principal mais recente lançada do compilador.
10.0	O compilador aceita apenas a sintaxe incluída em C# 10 ou inferior.
9.0	O compilador aceita apenas 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 apenas a sintaxe que está incluída em ISO/IEC 23270:2006 C# (2,0).
ISO-1 (ou 1)	O compilador aceita apenas a sintaxe que está incluída em ISO/IEC 23270:2003 C# (1.0/1.2).

Tipos de valor (referência C#)

21/01/2022 • 3 minutes to read

Tipos de [valor](#) e [tipos de referência](#) são as duas principais categorias de tipos C#. Uma variável de um tipo Value 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](#), passando um argumento para um método e retornando um resultado de método, valores de variáveis são copiados. No caso de variáveis de tipo de valor, as instâncias de tipo correspondentes são copiadas. O exemplo a seguir demonstra esse comportamento:

```
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, armazenado 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 tipo de valor de cópia e original tem acesso à mesma instância de tipo de referência. O exemplo a seguir demonstra esse comportamento:

```

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]
    }
}

```

NOTE

Para tornar seu código menos propenso a erros e mais robusto, defina e use tipos de valor imutável. Este artigo usa tipos de valores mutáveis somente para fins de demonstração.

Tipos de tipos de valor e restrições de tipo

Um tipo de valor pode ser um dos dois tipos a seguir:

- 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 opção ou uma combinação de opções

Um [tipo de valor anulável](#) `T?` representa todos os valores de seu tipo de valor subjacente `T` e um valor [nulo](#) adicional. Não é possível 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 [struct restrição](#). A partir do C# 7,3, 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 valores 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 Unicode UTF-16

Todos os tipos simples são tipos de estrutura e diferem de outros tipos de estrutura, pois permitem 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.
- Expressões constantes, cujos operandos são todas constantes dos tipos simples, são avaliadas no momento da compilação.

A partir do C# 7,0, o C# dá suporte a [tuplas de valor](#). 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#)

21/01/2022 • 4 minutes to read

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](#) suportam [operadores aritméticos, lógicos](#), [bita](#) bit, de comparação e [de igualdade](#).

Características dos tipos integrais

O C# é compatível com os seguintes tipos integrais predefinidos:

PALAVRA-CHAVE/TIPO C#	INTERVALO	TAMANHO	TIPO .NET
<code>sbyte</code>	-128 a 127	Inteiro de 8 bits com sinal	System.SByte
<code>byte</code>	0 a 255	Inteiro de 8 bits sem sinal	System.Byte
<code>short</code>	-32.768 a 32.767	Inteiro de 16 bits com sinal	System.Int16
<code>ushort</code>	0 a 65.535	Inteiro de 16 bits sem sinal	System.UInt16
<code>int</code>	-2.147.483.648 a 2.147.483.647	Inteiro assinado de 32 bits	System.Int32
<code>uint</code>	0 a 4.294.967.295	Inteiro de 32 bits sem sinal	System.UInt32
<code>long</code>	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Inteiro assinado de 64 bits	System.Int64
<code>ulong</code>	0 a 18.446.744.073.709.551.615	Inteiro de 64 bits sem sinal	System.UInt64
<code>nint</code>	Depende da plataforma	Inteiro de 32 bits ou 64 bits assinado	System.IntPtr
<code>nuint</code>	Depende da plataforma	Inteiro de 32 bits ou 64 bits sem sinal	System.UIntPtr

Em todas as linhas da tabela, exceto as duas últimas, cada palavra-chave do tipo C# da coluna mais à esquerda é um alias para o 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:

```
int a = 123;
System.Int32 b = 123;
```

Os `nint` tipos e nas duas últimas linhas da tabela são `nuint` inteiros de tamanho nativo. Eles são representados internamente pelos tipos .NET indicados, mas, em cada caso, a palavra-chave e o tipo .NET não

são intercambiáveis. O compilador fornece operações e conversões para e como tipos inteiros que ele `nint` não fornece para os tipos de ponteiro e `nuint` `System.IntPtr` `System.UIntPtr`. Para obter mais informações, consulte [nint](#) e [nuint](#) tipos.

O valor padrão de cada tipo integral é zero, `0`. Cada um dos tipos integrais, exceto os tipos de tamanho nativo, tem constantes e que fornecem o valor mínimo e `MinValue` `MaxValue` máximo desse tipo.

Use a estrutura `System.Numerics.BigInteger` para representar um inteiro com sinal sem nenhum limite superior ou inferior.

Literais inteiros

Literais inteiros podem ser

- *decimal*: sem nenhum prefixo
- *hexadecimal*: com o `0x` `0X` prefixo ou
- *binary*: com o `0b` `0B` prefixo ou (disponível no C# 7.0 e posterior)

O código a seguir demonstra um exemplo de cada um:

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

O exemplo anterior também mostra o uso de como separador de dígitos, que tem suporte a partir `_` do C# 7.0. Você pode usar o separador de dígitos com todos os tipos de literais numéricos.

O tipo de um literal inteiro é determinado por seu sufixo da seguinte maneira:

- Se o literal não tiver sufixo, seu tipo será o primeiro dos seguintes tipos nos quais seu valor pode ser representado: `int` , `uint` `long` `ulong` .

NOTE

Literais são interpretados como valores positivos. Por exemplo, o literal representa o número do tipo, embora tenha a mesma representação `0xFF_FF_FF_FF` de bit que o número do `4294967295` `uint` `-1` `int` tipo. Se você precisar de um valor de um determinado tipo, caste um literal para esse tipo. Use o `unchecked` operador, se um valor literal não puder ser representado no tipo de destino. Por exemplo, `unchecked((int)0xFF_FF_FF_FF)` produz `-1`.

- Se o literal for sufixo `U` ou `u`, seu tipo será o primeiro dos seguintes tipos nos quais seu valor pode ser representado: `uint` `ulong` .
- Se o literal for sufixo `L` ou `l`, seu tipo será o primeiro dos seguintes tipos nos quais seu valor pode ser representado: `long` `ulong` .

NOTE

Você pode usar a letra minúscula `l` como um sufixo. No entanto, isso gera um aviso do compilador porque a letra `l` pode ser confundida com o dígito `1`. Use `L` para maior clareza.

- Se o literal for sufixo `UL` , `UuL` , `uLU` , `Lu` , `1U` , `lu` `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 e o valor representado pelo literal estiver dentro do intervalo do tipo de destino, o valor poderá ser convertido implicitamente em `int` , , , ou `sbyte` `byte` `short` `ushort`

`uint` `ulong` `nint` `nuint` :

```
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 conversão para converter o valor representado por um literal inteiro para o tipo diferente do tipo determinado do literal:

```
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 precisará usar uma expressão [de conversão](#) para executar uma conversão explícita. Para obter mais informações, consulte [Conversões numéricas integrados](#).

Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Tipos integrais](#)
- [Literais inteiros](#)

Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [Tipos de ponto flutuante](#)
- [Cadeias de caracteres de formato numérico padrão](#)
- [Numéricos no .NET](#)

`nint` tipos `nuint` e (referência de C#)

21/01/2022 • 2 minutes to read

A partir do C# 9.0, você pode usar as `nint` `nuint` palavras-chave e para definir *inteiros de tamanho nativo*. Esses são inteiros de 32 bits ao executar em um processo de 32 bits ou inteiros de 64 bits ao executar em um processo de 64 bits. Eles podem ser usados para cenários de interop, bibliotecas de baixo nível e para otimizar o desempenho em cenários em que a matemática de inteiros é amplamente usada.

Os tipos inteiros de tamanho nativo são representados internamente como os tipos .NET `System.IntPtr` e `System.UIntPtr`. Ao contrário de outros tipos numéricos, as palavras-chave não são simplesmente aliases para os tipos. As instruções a seguir não são equivalentes:

```
nint a = 1;  
System.IntPtr a = 1;
```

O compilador fornece operações e conversões para `nint` e `nuint` que são apropriadas para tipos inteiros.

Tamanho inteiro nativo em tempo de run-time

Para obter o tamanho de um inteiro de tamanho nativo em tempo de executar, você pode usar `sizeof()`. No entanto, o código deve ser compilado em um contexto não seguro. Por exemplo:

```
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` `UIntPtr.Size` estáticas e .

MinValue e MaxValue

Para obter os valores mínimo e máximo de inteiros de tamanho nativo em tempo de operação, use e como propriedades estáticas com as palavras-chave e , como `MinValue` `MaxValue` no exemplo a `nint` `nuint` seguir:

```

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

```

Constantes

Você pode usar valores constantes nos seguintes intervalos:

- Para `nint` : [Int32.MinValue](#) para [Int32.MaxValue](#) .
- Para `nuint` : [UInt32.MinValue](#) para [UInt32.MaxValue](#) .

Conversões

O compilador fornece conversões implícitas e explícitas em outros tipos numéricos. Para obter mais informações, consulte [Conversões numéricas integrados](#).

Literais

Não há sintaxe direta para literais inteiros de tamanho nativo. Não há nenhum sufixo para indicar que um literal é um inteiro de tamanho nativo, como `L` para indicar um `long` . Em vez disso, você pode usar conversões implícitas ou explícitas de outros valores inteiros. Por exemplo:

```

nint a = 42
nint a = (nint)42;

```

Membros IntPtr/UIntPtr sem suporte

Os seguintes membros de [IntPtr](#) e não têm suporte para os tipos e [UIntPtr](#) `nint` `nuint` :

- Construtores parametrizados
- [Add\(IntPtr, Int32\)](#)
- [CompareTo](#)
- [Size](#) - Use [sizeof\(\)](#) em vez disso. Embora `nint.Size` não tenha suporte, você pode usar para `IntPtr.Size` obter um valor equivalente.
- [Subtract\(IntPtr, Int32\)](#)
- [ToInt32](#)
- [ToInt64](#)
- [ToPointer](#)
- [Zero](#) - Use 0 em vez disso.

Especificação da linguagem C#

Para obter mais informações, consulte a [especificação da linguagem C#](#) e a seção [Inteiros](#) de tamanho nativo das notas de proposta de recurso do C# 9.0.

Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [Tipos numéricos inteiros](#)
- [Conversões numéricas internas](#)

Tipos numéricos de ponto flutuante (Referência de C#)

21/01/2022 • 4 minutes to read

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	System.Single
<code>double</code>	$\pm 5,0 \times 10^{-324}$ to $\pm 1,7 \times 10^{308}$	~15 a 17 dígitos	8 bytes	System.Double
<code>decimal</code>	$\pm 1,0 \times 10^{-28}$ para $\pm 7,9228 \times 10^{28}$	28 a 29 dígitos	16 bytes	System.Decimal

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:

```
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 `decimal` tipo é apropriado quando o grau necessário de precisão 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, \$1), tarifas de juros (por exemplo, 2,625%) e assim por diante. Os números que são precisos para apenas um dígito decimal são tratados com mais precisão pelo `decimal` tipo: 0,1, por exemplo, podem ser exatamente representados por uma `decimal` instância, enquanto não há `double` ou `float` uma `float` instância ou que represente exatamente 0,1. 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` ao otimizar o desempenho é mais importante do que garantir a precisão. No entanto, qualquer diferença no desempenho passaria despercebida por todos, exceto os aplicativos com uso intensivo de cálculo. Outro motivo possível para evitar `decimal` é minimizar os requisitos de armazenamento. por exemplo, [ML .net](#) usa `float` porque a diferença entre 4 bytes e 16 bytes se soma a conjuntos de dados muito grandes. Para obter mais informações, consulte [System.Decimal](#).

Você pode misturar tipos [integrais](#) e [float](#) os [double](#) tipos e em uma expressão. Nesse caso, os tipos integrais são implicitamente convertidos em um dos tipos de ponto flutuante e, se necessário, o [float](#) tipo é convertido implicitamente em [double](#). A expressão é avaliada como segue:

- Se houver [double](#) tipo na expressão, a expressão será avaliada como [double](#), ou em comparações [bool](#) relacionais e de igualdade.
- Se não houver nenhum [double](#) tipo na expressão, a expressão será avaliada como [float](#), ou em comparações [bool](#) relacionais e de igualdade.

Você também pode misturar tipos integrais e o [decimal](#) tipo em uma expressão. Nesse caso, os tipos integrais são convertidos implicitamente no [decimal](#) tipo e a expressão é avaliada como [decimal](#), ou em comparações [bool](#) relacionais e de igualdade.

Você não pode misturar o [decimal](#) tipo com [float](#) os [double](#) tipos e em uma expressão. Nesse caso, se você quiser executar operações aritméticas, de comparação ou de igualdade, deverá converter explicitamente os operandos de ou para o [decimal](#) tipo, como mostra o exemplo a seguir:

```
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 um literal real é determinado pelo seu sufixo da seguinte maneira:

- O literal sem sufixo ou com o [d](#) [D](#) sufixo or é do tipo [double](#)
- O literal com o [f](#) [F](#) sufixo or é do tipo [float](#)
- O literal com o [m](#) [M](#) sufixo or é do tipo [decimal](#)

O código a seguir demonstra um exemplo de cada um:

```
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 um *separador de dígito*, que tem suporte a partir do C# 7,0. Você pode usar o separador de dígitos com todos os tipos de literais numéricos.

Você também pode usar a notação científica, ou seja, especificar uma parte exponencial de um literal real, como mostra o exemplo a seguir:

```
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 os tipos numéricos de ponto flutuante: de `float` para `double`. No entanto, você pode converter qualquer tipo de ponto flutuante para qualquer outro tipo de ponto flutuante com a [conversão explícita](#). Para obter mais informações, consulte [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 integrais](#)
- [Cadeias de caracteres de formato numérico padrão](#)
- [Numéricos no .NET](#)
- [System.Numerics.Complex](#)

Conversões numéricas internas (referência C#)

21/01/2022 • 4 minutes to read

O C# fornece um conjunto de tipos numéricos de [ponto flutuante](#) e [integral](#). Existe uma conversão entre dois tipos numéricos, implícito ou explícito. 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 os tipos numéricos internos:

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> , OU <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> , OU <code>decimal</code> , OU <code>nuint</code>
<code>long</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>

NOTE

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.

Observe também que

- Qualquer [tipo numérico integral](#) é implicitamente conversível em qualquer [tipo numérico de ponto flutuante](#).

- Não há conversões implícitas para os `byte` tipos e `sbyte`. 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 constante do tipo `int` (por exemplo, um valor representado por um inteiro literal) pode ser convertido implicitamente em `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` ou `nuint`, se estiver dentro do intervalo do tipo de destino:

```
byte a = 13;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a 'byte'
```

Como mostra o exemplo anterior, se o valor da constante não estiver dentro do intervalo do tipo de destino, ocorrerá um erro de compilador [CS0031](#).

Conversões numéricas explícitas

A tabela a seguir mostra as conversões explícitas predefinidas entre os tipos numéricos internos para os quais não há [conversão implícita](#):

DE	PARA
<code>sbyte</code>	<code>byte</code> , <code>ushort</code> , <code>uint</code> OU <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> OU <code>int</code>
<code>long</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>
<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>

DE	PARA
<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>

NOTE

Uma conversão numérica explícita pode resultar em perda de dados ou gerar uma exceção, normalmente um [OverflowException](#).

Observe também 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 que o tipo de destino, o valor de origem será estendido ou estendido em zero para que seja do 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 `double` valor for muito pequeno ou muito grande para caber no `float` tipo, 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 28ª 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.
- Quando você converte `decimal` para `float` ou `double`, o valor de origem é arredondado para o `float` valor mais próximo `double`, respectivamente.

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 C#)

21/01/2022 • 2 minutes to read

A `bool` palavra-chave Type é um alias para o [System.Boolean](#) tipo de estrutura .NET que representa um valor booleano, que pode ser `true` ou `false`.

Para executar operações lógicas com valores do `bool` tipo, use operadores [lógicos booleanos](#). O `bool` tipo é o tipo de resultado dos operadores de [comparação](#) e de [igualdade](#). Uma `bool` expressão pode ser uma expressão condicional de controle nas instruções `if`, `do`, `while` e no operador `?:` [condicional](#).

O valor padrão do `bool` tipo é `false`.

Literais

Você pode usar os `true` `false` literais e para inicializar uma `bool` variável ou para passar um `bool` valor:

```
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 `bool?` tipo anulável, se você precisar dar suporte à lógica de três valores, por exemplo, quando 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 de valor anulável, consulte [tipos de valor anulável](#).

Conversões

O C# fornece apenas duas conversões que envolvem o `bool` tipo. Essas são uma conversão implícita para o tipo anulável correspondente `bool?` e uma conversão explícita do `bool?` tipo. No entanto, o .NET fornece métodos adicionais que você pode usar para converter de ou para o `bool` tipo. Para obter mais informações, consulte a seção [Convertendo de valores booleanos](#) da [System.Boolean](#) página de referência da API.

Especificação da linguagem C#

Para obter mais informações, consulte a seção [tipo bool](#) da [especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [operadores true e false](#)

char (referência de C#)

21/01/2022 • 2 minutes to read

A `char` palavra-chave type é um alias para o tipo de estrutura .NET `System.Char` 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 `char` tipo é , ou `\0` seja, U+0000.

O `char` tipo dá suporte aos [operadores de comparação](#), [igualdade](#), [incremento e decremento](#). Além disso, para `char` os operadores, operadores lógicos aritméticos e bit a bit executam uma operação nos códigos de caractere correspondentes e produzem o resultado do `int` tipo.

O [tipo de cadeia](#) de caracteres representa o texto como uma sequência de `char` valores.

Literais

Você pode especificar um `char` valor com:

- um literal de caractere.
- uma sequência de escape Unicode, que é seguida pela representação hexadecimal de quatro `\u` símbolos de um código de caractere.
- uma sequência de escape hexadecimal, que é `\x` seguida pela representação hexadecimal de um código de caractere.

```
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 transformar o valor de um código de caractere no valor `char` correspondente.

NOTE

No caso de uma sequência de escape Unicode, você deve especificar todos os quatro dígitos hexadecimais. Ou seja, `\u006A` é uma sequência de escape válida, enquanto e não são `\u06A` `\u6A` válidos.

No caso de uma sequência de escape hexadecimal, você pode omitir os zeros à esquerda. Ou seja, as sequências de escape , e `\x006A` `\x06A` são `\x6A` válidas e correspondem ao mesmo caractere.

Conversões

O `char` tipo é implicitamente conversível para os seguintes tipos [integrais](#): , `ushort` , e `int` `uint` `long`

`ulong`. Ele também é implicitamente conversível para os tipos [numéricos](#) de ponto flutuante integrados: `float`, e `double` `decimal`. Ele é explicitamente conversível em `sbyte` `byte` tipos inteiros, `short` e.

Não há conversões implícitas de outros tipos para o `char` tipo. No entanto, [qualquer tipo numérico integral](#) ou de [ponto](#) flutuante é explicitamente conversível em `char`.

Especificação da linguagem C#

Para obter mais informações, consulte a [seção Tipos inteiros](#) da [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 C#)

21/01/2022 • 4 minutes to read

Um *tipo de enumeração* (ou *tipo de enumeração*) 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 `enum` palavra-chave e especifique os nomes dos *membros de enumeração*:

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

Por padrão, os valores constantes associados de membros de enumeração 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:

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

Você não pode 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 por expressão `(E)0`, mesmo se zero não tiver o membro enum 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 bits.

Tipos de Enumeração como Sinalizadores de Bit

Se você quiser que um tipo de enumeração represente uma combinação de opções, defina membros de enum para essas opções, de modo que uma opção 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 de bit](#) `|` ou `&` de opção para combinar opções ou combinações de interseção de opções, respectivamente. Para indicar que um tipo de enumeração declara campos de bits, 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.

```

[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 [System.FlagsAttribute](#) página de referência da API e os [Membros não exclusivos e a seção de atributo flags](#) da página de referência da [System.Enum](#) API.

O tipo System. Enum e a restrição enum

O [System.Enum](#) tipo 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 [System.Enum](#) página de referência da API.

A partir do C# 7,3, 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. Qualquer tipo de enumeração também satisfaz a `struct` restrição, 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.

```

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 [Enum.IsDefined](#) método para determinar se um tipo de enumeração contém um membro enum com determinado valor associado.

Para qualquer tipo de enumeração, existem conversões [boxing](#) e [unboxing](#) de e para o [System.Enum](#) tipo, 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 implícitas de enumeração](#)

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](#)
- [switch expressão](#)
- [switch privacidade](#)

Tipos de estrutura (referência de C#)

21/01/2022 • 12 minutes to read

Um *tipo de estrutura* (ou *tipo de struct*) é um tipo [de valor](#) que pode encapsular dados e funcionalidades relacionadas. Use a `struct` palavra-chave para definir um tipo de estrutura:

```
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})";
}
```

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 de variáveis são copiados na atribuição, passando um argumento para um método e retornando um resultado de método. No caso de uma variável 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 tipos pequenos 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 [booliana](#), 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*.

readonly Struct

A partir do C# 7.2, você usa o modificador para declarar que um tipo `readonly` de estrutura é imutável. Todos os membros de dados `readonly` de um struct devem ser somente leitura da seguinte maneira:

- Qualquer declaração de campo deve ter `readonly` o modificador
- Qualquer propriedade, incluindo aquelas auto-implementadas, deve ser somente leitura. No C# 9.0 e posterior, uma propriedade pode ter `init` um acessador.

Isso garante que nenhum membro de `readonly` um struct modifica o estado do struct. No C# 8.0 e posterior, isso significa que outros membros da instância, exceto construtores, são implicitamente `readonly`.

NOTE

Em um `readonly` struct, um membro de dados de um tipo de referência mutável ainda pode modificar seu próprio estado. Por exemplo, você não pode substituir uma `List<T>` instância, mas pode adicionar novos elementos a ela.

O código a seguir define um struct com setters de propriedade somente `readonly` init, disponíveis no C# 9.0 e

posterior:

```
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})";
}
```

readonly membros da instância

A partir do C# 8.0, você também pode usar o modificador para declarar que um membro da instância não modifica o estado de `readonly` um struct. Se você não puder declarar o tipo de estrutura inteira como , use o modificador para marcar os membros da instância que não modificam o estado `readonly` `readonly` do struct.

Dentro de `readonly` um membro de instância, você não pode atribuir aos campos de instância da estrutura. No entanto, `readonly` um membro pode chamar um não `readonly` membro. Nesse caso, o compilador cria uma cópia da instância de estrutura e chama o não membro `readonly` nessa cópia. Como resultado, a instância da estrutura original não é modificada.

Normalmente, você aplica o `readonly` modificador aos seguintes tipos de membros de instância:

- Métodos:

```
public readonly double Sum()
{
    return X + Y;
}
```

Você também pode aplicar o `readonly` modificador a métodos que substituem métodos declarados em `System.Object`:

```
public readonly override string ToString() => $"({X}, {Y})";
```

- propriedades e indexadores:

```
private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}
```

Se você precisar aplicar o modificador a ambos os acessadores de uma propriedade ou indexador, aplique-o na declaração da propriedade `readonly` ou do indexador.

NOTE

O compilador declara um acessador de uma propriedade auto-implementada como , independentemente da presença do `get` `readonly` modificador em uma declaração de propriedade.

No C# 9.0 e posterior, você pode aplicar o modificador a uma propriedade ou `readonly` indexador com um `init` acessador:

```
public readonly double X { get; init; }
```

Não é possível aplicar o `readonly` modificador a membros estáticos de um tipo de estrutura.

O compilador pode usar o modificador `readonly` para otimizações de desempenho. Para obter mais informações, consulte [Escrever código C# seguro e eficiente](#).

Mutação não estruturativa

A partir do C# 10, você pode usar a expressão para produzir uma cópia de uma instância de tipo de estrutura com as propriedades e campos especificados modificados. `with` Use a [sintaxe do inicializador de objeto](#) para especificar quais membros modificar e seus novos valores, como mostra o exemplo a seguir:

```
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)
}
```

record Struct

A partir do C# 10, você pode definir tipos de estrutura de registro. Os tipos de registro fornecem funcionalidades integrados para encapsular dados. Você pode definir os `record struct` tipos `readonly record struct` e . Um struct de registro não pode ser [ref](#) um struct. Para obter mais informações e exemplos, consulte [Registros](#).

Limitações com o design de um tipo de estrutura

Quando você projeta um tipo de estrutura, tem os mesmos recursos que com um tipo [de classe](#), com as seguintes exceções:

- Não é possível declarar um construtor sem parâmetros. Cada tipo de estrutura já fornece um construtor sem parâmetro implícito que produz [o valor padrão](#) do tipo.

NOTE

A partir do C# 10, você pode declarar um construtor sem parâmetros em um tipo de estrutura. Para obter mais informações, consulte a seção [Construtores sem parâmetros e inicializadores de campo](#).

- Não é possível inicializar um campo de instância ou propriedade em sua declaração. No entanto, você pode inicializar [um campo estático ou const](#) ou uma propriedade estática em sua declaração.

NOTE

A partir do C# 10, você pode inicializar um campo de instância ou propriedade em sua declaração. Para obter mais informações, consulte a seção [Construtores sem parâmetros e inicializadores de campo](#).

- Um construtor de um tipo de estrutura deve inicializar todos os campos de instância do tipo.
- Um tipo de estrutura não pode herdar de outra classe ou tipo de estrutura e não pode ser a base de uma classe. No entanto, um tipo de estrutura pode [implementar interfaces](#).
- Não é possível declarar um [finalizador dentro](#) de um tipo de estrutura.

Construtores sem parâmetros e inicializadores de campo

A partir do C# 10, você pode declarar um construtor de instância sem parâmetros em um tipo de estrutura, como mostra o exemplo a seguir:

```

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 de um tipo de estrutura, que é o valor produzido definindo todos os campos de tipo de valor como seus valores padrão (o padrão de 0 bits) e todos os campos de tipo de referência como `null`. A instância de matriz de tipo de estrutura também ignora um construtor sem parâmetros e produz uma matriz populada com os valores padrão de um tipo de estrutura.

A partir do C# 10, você também pode inicializar um campo de instância ou propriedade em sua declaração, como mostra o exemplo a seguir:

```

public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        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 ()
}

```

Se você não declarar um construtor sem parâmetros explicitamente, um tipo de estrutura fornece um construtor sem parâmetros cujo comportamento é o seguinte:

- Se um tipo de estrutura tiver construtores de instância explícitos ou não tiver inicializadores de campo, um construtor sem parâmetro implícito produzirá o valor padrão de um tipo de estrutura, independentemente dos inicializadores de campo, como mostra o exemplo anterior.
- Se um tipo de estrutura não tiver construtores de instância explícitos e tiver inicializadores de campo, o compilador sintetiza um construtor público sem parâmetros que executa as inicializações de campo especificadas, como mostra o exemplo a seguir:

```

public struct Coords
{
    public double X = double.NaN;
    public double Y = double.NaN;

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

public static void Main()
{
    var p1 = new Coords();
    Console.WriteLine(p1); // output: (NaN, NaN)

    var p2 = default(Coords);
    Console.WriteLine(p2); // output: (0, 0)

    var ps = new Coords[3];
    Console.WriteLine(string.Join(", ", ps)); // output: (0, 0), (0, 0), (0, 0)
}

```

Como mostra o exemplo anterior, a expressão de valor padrão e a instância de matriz ignoram inicializadores

de campo.

IMPORTANT

Quando um inclui inicializadores de campo, mas não inclui nenhum construtor de instância explícito, o construtor público sem parâmetros sintetizado executa os inicializadores de `struct` campo especificados. Se isso incluir um construtor de instância explícita, o construtor sem parâmetros sintetizado `struct` produzirá o mesmo valor que a `default` expressão.

Para obter mais informações, consulte a nota de proposta de proposta de recurso Construtores de `struct` sem parâmetros.

Instação de um tipo de estrutura

No C#, você deve inicializar uma variável declarada antes que ela possa ser usada. Como uma variável de tipo de estrutura não pode ser (a menos que seja uma variável de um tipo de valor anulado), você deve insinuar uma instância do `null` tipo correspondente. Há várias maneiras de fazer isso.

Normalmente, você insinuou um tipo de estrutura chamando um construtor apropriado com o `new` operador . Cada tipo de estrutura tem pelo menos um construtor. Esse é um construtor sem parâmetro implícito, que produz [o valor padrão](#) do tipo. Você também pode usar uma [expressão de valor padrão](#) para produzir o valor padrão de um tipo.

Se todos os campos de instância de um tipo de estrutura são acessíveis, você também pode instancia-lo sem o `new` operador . Nesse caso, você deve inicializar todos os campos de instância antes do primeiro uso da instância. O seguinte exemplo mostra como fazer isso:

```
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 integrados](#), use os literais correspondentes para especificar um valor do tipo .

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. Isso 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 , ou `ref` para indicar que um argumento deve ser passado por `out` `in` referência. Use `ref returns` para retornar um resultado de método por referência. Para obter mais informações, consulte [Escrever código C# seguro e eficiente](#).

ref Struct

A partir do C# 7.2, você pode usar o `ref` modificador na declaração de um tipo de estrutura. Instâncias de um `ref` tipo de struct são alocadas na pilha e não podem escapar para o heap gerenciado. Para garantir isso, o compilador limita o uso de tipos `ref` de struct da seguinte forma:

- Um `ref` struct não pode ser o tipo de elemento de uma matriz.
- Um struct não pode ser um tipo declarado de um campo de uma classe `ref` ou um `ref` struct não.
- Um `ref` struct não pode implementar interfaces.
- Um `ref` struct não pode ser a box para `System.ValueType` ou `System.Object`.
- Um `ref` struct não pode ser um argumento de tipo.
- Uma `ref` variável de struct não pode ser capturada por uma `expressão lambda` ou uma `função local`.
- Uma `ref` variável de struct não pode ser usada em um `async` método. No entanto, você pode usar variáveis de struct em métodos síncronos, por exemplo, em aqueles que retornam `Task` ou `Task<TResult>`.
- Uma `ref` variável de struct não pode ser usada em `iteradores`.

A partir do C# 8.0, você pode definir um `ref` struct descartável. Para fazer isso, verifique se um `ref` struct se ajusta ao [padrão descartável](#). Ou seja, ele tem uma instância ou um método `Dispose` de extensão, que é acessível, sem parâmetros e tem um `void` tipo de retorno.

Normalmente, você define um tipo de struct quando precisa de um tipo que também inclui membros de dados `ref` de `ref` tipos de struct:

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

Para declarar um struct como , combine os modificadores e na declaração de tipo `ref readonly` (o modificador deve vir antes do `readonly` `ref` `readonly` `ref` modificador):

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

No .NET, exemplos de `ref` um struct são `System.Span<T>` e `System.ReadOnlySpan<T>` .

restrição de struct

Você também usa a `struct` palavra-chave na `struct` restrição para especificar que um parâmetro de tipo é um tipo de valor não anulado. Os tipos de `estrutura` e `enumeração` atendem à `struct` restrição .

Conversões

Para qualquer tipo de estrutura (exceto tipos [ref](#) de struct), existem conversões boxing e unboxing para e dos tipos [System.ValueType](#) e [System.Object](#). Também existem conversões boxing e unboxing entre um tipo de estrutura e qualquer interface que ele implementa.

Especificação da linguagem C#

Para obter mais informações, consulte a [seção Structs](#) da [especificação da linguagem C#](#).

Para obter mais informações sobre os recursos introduzidos no C# 7.2 e posterior, consulte as seguintes notas sobre a proposta de recurso:

- [Structs readonly](#)
- [Membros da instância ReadOnly](#)
- [Segurança de tempo de compilação para tipos semelhantes a ref](#)
- [Construtores de struct sem parâmetros](#)
- [Permitir with expressão em structs](#)
- [Structs de registro](#)

Confira também

- [Referência de C#](#)
- [Diretrizes de design – escolhendo entre classe e struct](#)
- [Diretrizes de design – Design de struct](#)
- [O sistema de tipos C#](#)

Tipos de tupla (referência de C#)

21/01/2022 • 8 minutes to read

Disponível no C# 7.0 e posterior, o recurso *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 seus membros de dados:

```
(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.
```

Como mostra o exemplo anterior, para definir um tipo de tupla, especifique tipos de todos os seus 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, como mostra o exemplo a seguir:

```
(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.
```

A partir do C# 7.3, os tipos de tuplas são suportados [por operadores de igualdade](#) e `==` `!=`. Para obter mais informações, consulte a [seção Igualdade de tupla](#).

Os tipos de tupla são [tipos de valor](#); elementos de tupla são campos públicos. Isso torna as tuplas *tipos de valor mutáveis*.

NOTE

O recurso de tuplas requer o tipo e tipos genéricos relacionados (por exemplo, `ValueTuple<T1,T2>`), que estão disponíveis no .NET Core e `System.ValueTuple` .NET Framework 4.7 e posteriores. Para usar tuplas em um projeto destinado .NET Framework 4.6.2 ou anterior, adicione o pacote NuGet ao `System.ValueTuple` projeto.

Você pode definir tuplas com um grande número arbitrário de elementos:

```
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 de tuplas

Um dos casos de uso mais comuns de tuplas é como um tipo de retorno de método. Ou seja, em vez de definir

out parâmetros de método , você pode agrupar resultados de método em um tipo de retorno de tupla, como mostra o exemplo a seguir:

```
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.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}
```

Como mostra o exemplo anterior, você pode trabalhar com a instância de tupla retornada diretamente ou [desconstruir-a](#) em variáveis separadas.

Você também pode usar tipos de tupla em vez de [tipos anônimos](#); por exemplo, em consultas LINQ. Para obter mais informações, consulte [Escolhendo entre tipos anônimos e de tupla](#).

Normalmente, você usa tuplas para agrupar elementos de dados relacionados de forma flexível. Isso geralmente é útil em métodos de utilitários privados e internos. No caso da API pública, considere definir uma classe [ou](#) um tipo [de](#) estrutura.

Nomes de campos de tupla

Você pode especificar explicitamente os nomes de campos de tupla em uma expressão de inicialização de tupla ou na definição de um tipo de tupla, como mostra o exemplo a seguir:

```
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}.");
```

A partir do C# 7.1, se você não especificar um nome de campo, ele poderá ser inferido do nome da variável correspondente em uma expressão de inicialização de tupla, como mostra o exemplo a seguir:

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

Isso é conhecido como 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 do candidato é um nome de membro de um tipo de tupla, por exemplo, `Item3` `ToString`, ou `Rest`.
- O nome do candidato é uma duplicata de outro nome de campo de tupla, explícito ou implícito.

Nesses casos, especifique explicitamente o nome de um campo ou acesse um campo pelo nome padrão.

Os nomes padrão dos campos de tupla `Item1` são, e assim por `Item2` `Item3` diante. Você sempre pode usar o nome padrão de um campo, mesmo quando um nome de campo é especificado explicitamente ou inferido, como mostra o exemplo a seguir:

```
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 igualdade de tupla e atribuição de tupla não levam em conta nomes de campo.

No tempo de compilação, o compilador substitui nomes de campo não padrão com os nomes padrão correspondentes. Como resultado, os nomes de campo especificados explicitamente ou inferidos não estão disponíveis em tempo de executar.

Desconstrução e atribuição de tupla

O C# dá suporte à atribuição entre tipos de tupla que atendem a ambas as seguintes condições:

- 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 que ou implicitamente conversível para o tipo do elemento de tupla à esquerda correspondente

Os valores de elemento de tupla são atribuídos seguindo a ordem dos elementos de tupla. Os nomes dos campos de tupla são ignorados e não atribuídos, como mostra o exemplo a seguir:

```
(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 separadas. Você pode fazer isso de uma das seguintes maneiras:

- Declare explicitamente o tipo de cada variável dentro de parênteses:

```
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.
```

- Use a palavra-chave fora dos parênteses para declarar variáveis de tipo implícito e permitir que o compilador infera seus tipos:

```
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.
```

- Usar variáveis existentes:

```
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.
```

Para obter mais informações sobre a desconstrução de tuplas e outros tipos, consulte [Desconstruindo tuplas e outros tipos](#).

Igualdade de tupla

Começando com o C# 7.3, os tipos de tupla oferecem suporte aos operadores `==` e `!=`. Esses operadores comparam membros do operando esquerdo com os membros correspondentes do operando à direita seguindo a ordem dos elementos de tupla.

```
(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
```

Como mostra o exemplo anterior, as operações `==` e `!=` levam em conta nomes de campos de tupla.

Duas tuplas são comparáveis quando ambas as condições a seguir são atendidas:

- Ambas as tuplas têm o mesmo número de elementos. Por exemplo, `t1 != t2` não compila se `t1` e `t2` têm números diferentes de elementos.
- Para cada posição de tupla, os elementos correspondentes dos operadores de tupla esquerda e direita são

comparáveis com os `==` `!=` operadores e . Por exemplo, `(1, (2, 3)) == ((1, 2), 3)` não é compilado porque `1` não é comparável com `(1, 2)` .

Os `==` `!=` operadores e comparam tuplas de forma de curto-circuito. Ou seja, uma operação é interrompida assim que atende a um par de elementos não iguais ou atinge as extremidades das tuplas. No entanto, antes de qualquer comparação, *todos* os elementos de tupla são avaliados, como mostra o exemplo a seguir:

```
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 out

Normalmente, você refactor um método que tem `out` parâmetros em um método que retorna uma tupla. No entanto, há casos em que um `out` parâmetro pode ser de um tipo de tupla. O exemplo a seguir mostra como trabalhar com tuplas como `out` parâmetros:

```
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 C#, que são apoiadas por tipos, são diferentes `System.ValueTuple` das tuplas representadas por `System.Tuple` tipos. As principais diferenças são as seguintes:

- `System.ValueTuple` tipos são **tipos de valor**. `System.Tuple` tipos são tipos **de referência**.
- `System.ValueTuple` os tipos são mutáveis. `System.Tuple` os tipos são imutáveis.
- Os membros de dados `System.ValueTuple` de tipos são campos. Os membros de dados `System.Tuple` de tipos são propriedades.

Especificação da linguagem C#

Para obter mais informações, consulte as seguintes notas de proposta de recurso:

- Inferir nomes de tupla (também conhecido como inicializadores de projeção de tupla)
- Suporte para `==` e em tipos de `!=` tupla

Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [Escolhendo entre tipos anônimos e de tupla](#)
- [System.ValueTuple](#)

Tipos de valor que podem ser anulados (referência de C#)

21/01/2022 • 8 minutes to read

Um *tipo de valor que pode ser anulado* (`T?`) representa todos os valores de seu tipo de valor subjacente e um valor `nulo` adicional. Por exemplo, você pode atribuir qualquer um dos três valores a seguir a uma `bool?` variável: `true`, `false` ou `null`. Um tipo de valor subjacente (`T`) não pode ser um tipo de valor anulado em si.

NOTE

O C# 8.0 introduz o recurso de tipos de referência que permitem valor nulo. Para obter mais informações, consulte [Tipos de referência que podem ser anulados](#). Os tipos de valor que podem ser anulados estão disponíveis a partir do C# 2.

Qualquer tipo de valor que pode ser anulado é uma instância da estrutura `System.Nullable<T>` genérica. Você pode se referir a um tipo de valor que pode ser anulado com um tipo subjacente em qualquer um dos seguintes formulários (`T`) intercambiáveis: `Nullable<T>` ou `T?`.

Normalmente, você usa um tipo de valor que pode ser anulado quando precisa representar o valor indefinido de um tipo de valor subjacente. Por exemplo, uma variável booleana ou `bool`, só pode ser ou `true` `false`. No entanto, em alguns aplicativos, um valor de variável pode ser indefinido ou ausente. Por exemplo, um campo de banco de dados pode conter ou ou pode não conter nenhum `true` `false` valor, ou seja, `NULL`. Você pode usar o `bool?` tipo nesse cenário.

Declaração e atribuição

Como um tipo de valor é implicitamente conversível para o tipo de valor anulado correspondente, você pode atribuir um valor a uma variável de um tipo de valor que pode ser anulado como faria para seu tipo de valor subjacente. Você também pode atribuir o `null` valor. Por exemplo:

```
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 anulado representa , ou `null` seja, é uma instância cuja `Nullable<T>.HasValue` propriedade retorna `false` .

Exame de uma instância de um tipo de valor anulado

A partir do C# 7.0, você pode usar o operador com um padrão de tipo para examinar uma instância de um tipo de valor que pode ser anulado e recuperar um valor de um `is` tipo `null` subjacente:

```

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 anulado:

- `Nullable<T>.HasValue` indica se uma instância de um tipo de valor anulado tem um valor de seu tipo subjacente.
- `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 exemplo a seguir usa a `.HasValue` propriedade para testar se a variável contém um valor antes de exibi-lo:

```

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 anulado com `null` em vez de usar a `null` propriedade, como mostra o exemplo a seguir:

```

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 que pode ser anulado em um tipo subjacente

Se você quiser atribuir um valor de um tipo de valor anulado a uma variável de tipo de valor não anulada, talvez seja necessário especificar o valor a ser atribuído no lugar de `null`. Use o operador de `??` coalescing nulo para fazer isso (você também pode usar o `Nullable<T>.GetValueOrDefault(T)` método para a mesma finalidade):

```

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 você quiser usar o [valor padrão](#) do tipo de valor subjacente no lugar de , use o `null` método `Nullable<T>.GetValueOrDefault()` .

Você também pode lançar explicitamente um tipo de valor que pode ser anulado em um tipo não anulado, como mostra o exemplo a seguir:

```

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 operação, se o valor de um tipo de valor anulado for , a `null` cast explícita lançará um [InvalidOperationException](#) .

Um tipo de valor não anulado `T` é implicitamente conversível para o tipo de valor anulado `T?` correspondente.

Operadores suspensos

Os operadores unários e binários predefinidos ou quaisquer operadores sobrecarregados com suporte por um tipo de valor também têm suporte pelo tipo de valor anulado `T` `T?` correspondente. Esses operadores, também conhecidos como operadores suspensos, produzem se um ou ambos os operadores são ; caso contrário, o operador usa os valores contidos de seus operadores para calcular `null` `null` o resultado. Por exemplo:

```

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

```

NOTE

Para o tipo , os operadores predefinidos e não seguem as regras descritas nesta seção: o resultado de uma avaliação de operador pode ser não nulo, mesmo se um dos `bool?` `&` `|` operadores 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 são , o resultado é ; caso contrário, os valores contidos de `<` `>` `<=` `>=` `null` `false` operativos 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`

```

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 são , o resultado é , se apenas um dos operandos for , o resultado será ; caso contrário, os valores contidos dos operadores serão `==` `null` `true` `null` `false` comparados.

Para o operador de desigualdade , se ambos os operandos são , o resultado é , se apenas um dos operadores for , o resultado será ; caso contrário, os valores contidos dos operadores serão `!=` `null` `false` `null` `true` 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 anuáveis correspondentes.

Conversão boxing e unboxing

Uma instância de um tipo de valor que pode ser `T?` [anulado](#) é [anulada](#) da seguinte forma:

- Se [HasValue](#) retorna `false` , a referência nula é produzida.
- Se [HasValue](#) retornar , o valor correspondente do tipo de valor subjacente será a `true` `T` box, não a instância de [Nullable<T>](#) .

Você pode unboxar um valor de um tipo de valor para o tipo de valor anulado `T` `T?` correspondente, como mostra o exemplo a seguir:

```

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 que pode ser anulado

O exemplo a seguir mostra como determinar se uma instância representa um tipo de valor que pode ser nulo construído, ou seja, o tipo com [System.Type](#) um parâmetro de tipo especificado [System.Nullable<T>](#) `T` :

```

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 [System.Type](#) instância.

Se você quiser determinar se uma instância é de um tipo de valor anulado, não use o método para fazer com que uma instância seja testada com [Object.GetType](#) Type o código anterior. Quando você chama o método em uma instância de um tipo de valor [Object.GetType](#) anulado, a instância é [anulada](#) para [Object](#). Como a boxing de uma instância não nula de um tipo de valor anulado é equivalente à boxing de um valor do tipo subjacente, retorna uma instância que representa o tipo subjacente de um tipo de valor [GetType](#) Type anulado:

```

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 que pode ser nulo. Como mostra o exemplo a seguir, não é possível distinguir tipos de uma instância de tipo de valor que pode ser anulado e sua instância de tipo subjacente com o [is](#) operador :

```

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?

```

Você pode usar o código apresentado no exemplo a seguir para determinar se uma instância é de um tipo de valor que pode ser anulado:

```

int? a = 14;
Console.WriteLine(IsOfNullableType(a)); // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b)); // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}

```

NOTE

Os métodos descritos nesta seção não são aplicáveis no caso de tipos de referência que podem ser [anulados](#).

Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Tipos anuláveis](#)
- [Operadores suspensos](#)
- [Conversões anuladas implícitas](#)
- [Conversões que podem ser anuladas 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 em C#)

21/01/2022 • 2 minutes to read

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 sua 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](#)
- [gravável](#)

O C# também oferece os seguintes tipos de referência internos:

- [dinâmico](#)
- [object](#)
- [cadeia de caracteres](#)

Confira também

- [Referência do C#](#)
- [Palavras-chave do C#](#)
- [Tipos de ponteiro](#)
- [Tipos de valor](#)

Tipos de referência internos (Referência de C#)

21/01/2022 • 8 minutes to read

O C# tem um número de tipos de referência internos. 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 tipo é convertida em um tipo de valor, é dito `object` que é *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. Isso torna o teste de igualdade de cadeia de caracteres mais intuitivo. Por exemplo:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

Isso exibe "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:

```
string a = "good " + "morning";
```

Isso 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, embora a sintaxe faça com que pareça que você pode fazer isso. 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.

```
string b = "h";
b += "ello";
```

O `[]` [operador](#) pode ser usado para acesso `readonly` a caracteres individuais de uma cadeia de caracteres. Os

valores de índice válidos `0` começam em e devem ser menores que o comprimento da cadeia de caracteres:

```
string str = "test";
char x = str[2]; // x = 's';
```

Da mesma forma, o `[]` operador também pode ser usado para iterar em cada caractere em uma cadeia de caracteres:

```
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 são do tipo `string` e podem ser escritos de duas formas, entre aspas e `@`. Os literais de cadeia de caracteres entre aspas são colocados entre aspas duplas (""):

```
"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.

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
// F
```

NOTE

O código de escape `\udddd` (em que `ddd` é um número de quatro dígitos) representa o caractere Unicode U+ `dddd`. Os códigos de escape Unicode de oito dígitos também são reconhecidos: `\Udddddd`.

Os [literais de cadeia de caracteres textuais](#) começam com `@` e também são colocados entre aspas duplas. Por exemplo:

```
@"good morning" // a string literal
```

A vantagem das cadeias de caracteres textuais é que as sequências de escape *não* são processadas, o que torna mais fácil escrever, por exemplo, um nome de arquivo totalmente qualificado do Windows:

```
@"c:\Docs\Source\a.txt" // rather than "c:\\Docs\\\\Source\\\\a.txt"
```

Para incluir aspas duplas em uma cadeia de caracteres @-quoted, dobre-a:

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

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:

```
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 delegados e a remoção falham com uma exceção de runtime quando os tipos de delegado envolvidos no tempo de execução são diferentes devido à conversão de variantes. O exemplo a seguir demonstra uma situação que falha:

```
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.

```
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 `calli` instrução em vez de iniciar um tipo delegado e chamar o método `Invoke` virtual.

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`.

```
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:

```
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.

```
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.

```

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

```

Confira também

- [Referência de C#](#)
- [Palavras-chave do C#](#)
- [Eventos](#)
- [Usando o tipo dynamic](#)
- [Práticas recomendadas para usar cadeias de caracteres](#)
- [Operações básicas de cadeia de caracteres](#)
- [Criar novas cadeias de caracteres](#)
- [Operadores cast e teste de tipo](#)
- [Como fazer a cast com segurança usando a correspondência de padrões e os operadores as e is](#)

- Passo a passo: Criando e usando objetos dinâmicos
- [System.Object](#)
- [System.String](#)
- [System.Dynamic.DynamicObject](#)

Registros (referência C#)

21/01/2022 • 18 minutes to read

A partir do C# 9, você usa a `record` palavra-chave 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:

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

```
public record Person
{
    public string FirstName { get; init; } = default!;
    public string LastName { get; init; } = default!;
}
```

Você também pode criar tipos de registro com propriedades e campos mutáveis:

```
public record Person
{
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
}
```

Embora os registros possam ser mutáveis, eles são basicamente destinados a oferecer 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 centrada em dados:
 - [Igualdade de valor](#)
 - [Sintaxe concisa para mutação não destrutiva](#)
 - [Formatação interna para exibição](#)
- [Suporte para hierarquias de herança](#)

Você também pode usar [tipos de estrutura](#) para criar tipos centrados em dados que forneçam igualdade de valor e pouco ou nenhum comportamento. No C# 10 e posteriores, você pode definir `record struct` tipos usando parâmetros posicionais ou a sintaxe de propriedade padrão:

```
public readonly record struct Point(double X, double Y, double Z);
```

```
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
    public double Z { get; init; }
}
```

As structs de registro também podem ser mutáveis, as structs de registro posicionais e as estruturas de registro sem parâmetros posicionais:

```
public record struct DataMeasurement(DateTime TakenAt, double Measurement);
```

```
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

Os exemplos anteriores mostram algumas distinções entre registros que são tipos de referência e registros que são tipos de valor:

- A `record` ou `record class` declara um tipo de referência. A `class` palavra-chave 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`. Elas são *mutáveis* em um `record struct`.

O restante deste artigo aborda ambos os `record class` tipos e `record struct`. As diferenças são detalhadas em cada seção. Você deve decidir entre um `record class` e um `record struct` semelhante à decisão entre a `class` e a `struct`. O termo *registro* é usado para descrever o comportamento que se aplica a todos os tipos de registro. `record struct` Ou `record class` é usado para descrever o comportamento que se aplica apenas a tipos struct ou de classe, respectivamente.

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:

```
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 `record` tipos e `readonly record struct` tipos: uma propriedade `somente init`.
 - Para `record struct` tipos: uma propriedade de leitura/gravação.
- 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 `Deconstruct` método com um `out` parâmetro para cada parâmetro posicional fornecido na declaração de registro. O método desconstrói as propriedades definidas usando a sintaxe posicional; Ele ignora as propriedades que são 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 que você aplicar às propriedades do

registro posicional. O exemplo a seguir aplica a `System.Text.Json.Serialization.JsonPropertyNameAttribute` a cada propriedade do `Person` registro. O `property:` destino indica que o atributo é aplicado à propriedade gerada pelo compilador. Outros valores são `field:` aplicar o atributo ao campo e `param:` aplicar o atributo ao parâmetro.

```
/// <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 `<param>` marca para adicionar documentação para os parâmetros do construtor primário.

Se a definição de propriedade autoimplementada gerada não for o que você deseja, você poderá definir sua própria propriedade de mesmo nome. Por exemplo, talvez você queira alterar a acessibilidade ou a imutabilidade ou fornecer uma implementação para o `get` `set` acessador ou. Se você declarar a propriedade em sua origem, deverá inicializá-la a partir do parâmetro posicional do registro. O desconstrutor gerado usará sua definição de propriedade. Por exemplo, os exemplos a seguir declaram as `FirstName` `LastName` Propriedades e de um registro posicional `public`, mas restringem o `Id` parâmetro posicional para `internal`. Você pode usar essa sintaxe para registros e tipos de struct de registro.

```
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:

```
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 posição somente leitura* declaram propriedades somente inicialização. Uma *estrutura de registro posicional* declara as propriedades de leitura/gravação. 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 thread-safe ou se você estiver dependendo de um código hash que permanece o mesmo em uma tabela de hash. No entanto, a imutabilidade não é apropriada para todos os cenários de dados. [Entity Framework Core](#), por exemplo, não dá suporte à atualização com tipos de entidade imutável.

Propriedades somente de inicialização, criadas a partir de parâmetros posicionais (`record class` e `readonly record struct`) ou especificando `init` acessadores, têm *imutabilidade superficial*. Após a inicialização, você não pode alterar o valor das propriedades de tipo de valor ou a referência das propriedades do 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 do tipo de referência (uma matriz, nesse caso) é mutável:

```
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 de tipos de registro são implementados por métodos sintetizados pelo compilador, e nenhum desses métodos compromete a imutabilidade modificando o estado do objeto. A menos que especificado, os métodos sintetizados são gerados para `record` `record struct` declarações, e `readonly record struct`.

Igualdade de valor

Para qualquer tipo que você definir, você pode substituir `Object.Equals(Object)` e sobrecarregar `operator ==`. Se você não substituir `Equals` ou sobrecarregar `operator ==`, o tipo declarado determinará como a igualdade é definida:

- Para `class` tipos, dois objetos são iguais se se referirem ao mesmo objeto na memória.
- Para `struct` tipos, dois objetos são iguais se eles são do mesmo tipo e armazenam os mesmos valores.
- Para `record` tipos, incluindo `record struct` e `readonly record struct`, dois objetos são iguais se eles são do mesmo tipo e armazenam 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, [Entity Framework Core](#) depende da igualdade de referência para garantir que ela use apenas uma instância de um tipo de entidade para o que é conceitualmente uma entidade. Por esse motivo, registros e estruturas de registro não são apropriados para uso como tipos de entidade no Entity Framework Core.

O exemplo a seguir ilustra a igualdade de valor de tipos de registro:

```

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 seguintes métodos:

- Uma substituição de [Object.Equals\(Object\)](#) .

Esse método é usado como base para o [Object.Equals\(Object, Object\)](#) método estático quando ambos os parâmetros são não nulos.

- Um [Equals](#) método virtual cujo parâmetro é o tipo de registro. Esse método implementa [IEquatable<T>](#).
- Uma substituição de [Object.GetHashCode\(\)](#) .
- Substituições de operadores [==](#) e [!=](#) .

Você pode escrever suas próprias implementações para substituir qualquer um desses métodos sintetizados. Se um tipo de registro tiver um método que corresponda à assinatura de qualquer método sintetizado, o compilador não sintetizará esse método.

Se você fornecer sua própria implementação de [Equals](#) em um tipo de registro, forneça também uma implementação de [GetHashCode](#) .

Mutação não destrutiva

Se você precisar copiar uma instância com algumas modificações, poderá usar uma [with](#) expressão para obter uma *mutação não destrutiva*. Uma [with](#) expressão cria 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 objeto](#) para especificar os valores a serem alterados, conforme mostrado no exemplo a seguir:

```

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 `with` expressão pode definir propriedades posicionais ou Propriedades criadas usando a sintaxe de propriedade padrão. As propriedades não posicionais devem ter um `init` `set` acessador ou para serem alteradas em uma `with` expressão.

O resultado de uma `with` expressão 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 terminam com uma referência à mesma instância.

Para implementar esse recurso para `record class` tipos, o compilador sintetiza um método `clone` e um construtor de cópia. O método de clonagem virtual retorna um novo registro inicializado pelo construtor de cópia. Quando você usa uma `with` expressão, o compilador cria um código que chama o método `clone` e, em seguida, define as propriedades que são especificadas na `with` expressão.

Se você precisar de comportamento de cópia diferente, poderá escrever seu próprio construtor de cópia em um `record class`. Se você fizer isso, o compilador não sintetizará um. Torne seu construtor `private` se o registro for `sealed`, caso contrário, o faz `protected`. O compilador não sintetiza um construtor de cópia para `record struct` tipos. Você pode escrever um, mas o compilador não gerará chamadas para ele para `with` expressões. Os valores de `record struct` são copiados na atribuição.

Você não pode substituir o método `clone` e não pode criar um membro nomeado `Clone` em qualquer tipo de registro. O nome real do método `clone` é gerado pelo compilador.

Formatação interna para exibição

Os tipos de registro têm um método gerado pelo compilador `ToString` que exibe os nomes e valores de propriedades públicas e campos. O `ToString` método retorna uma cadeia de caracteres do seguinte formato:

```
<record type name> { <property name> = <value>, <property name> = <value>, ...}
```

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 `record class` tipos, o compilador sintetiza um método virtual `PrintMembers` e uma `ToString` substituição. Em `record struct` tipos, esse membro é `private`. A `ToString` substituição cria um `StringBuilder` objeto 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 um código semelhante ao que a substituição sintetizada contém:

```
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 `PrintMembers` ou a `ToString` substituição. Os exemplos são fornecidos na seção [PrintMembers formatação em registros derivados](#), mais adiante neste artigo. No C# 10 e posteriores, sua implementação do `ToString` pode incluir o `sealed` modificador, que impede o compilador de sintetizar uma `ToString` implementação para quaisquer registros derivados. Você pode fazer isso para criar uma representação de cadeia de caracteres consistente em uma hierarquia de `record` tipos. (Os registros derivados ainda terão um `PrintMembers` método gerado para todas as propriedades derivadas.)

Herança

Esta seção se aplica somente a `record class` tipos.

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 registros derivados

O registro derivado declara parâmetros posicionais para todos os parâmetros no Construtor principal de registro de base. O registro base declara e inicializa essas propriedades. O registro derivado não os oculta, mas só cria e inicializa Propriedades para parâmetros que não são declarados em seu registro base.

O exemplo a seguir ilustra a herança com a sintaxe de propriedade posicional:

```
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 em hierarquias de herança

Esta seção se aplica a `record class` tipos, mas não a `record struct` tipos. Para que duas variáveis de registro sejam iguais, o tipo de tempo de execução deve ser igual. Os tipos das variáveis que a contêm podem ser

diferentes. A comparação de igualdade herdada é ilustrada no exemplo de código a seguir:

```
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 as `Person` variáveis sejam do tipo e `student == student2` retorna `True` embora uma seja uma `Person` variável e uma seja uma `Student` variável. O teste de igualdade depende do tipo de tempo de execução do objeto real, não do tipo declarado da variável.

Para implementar esse comportamento, o compilador sintetiza uma `EqualityContract` propriedade que retorna um `Type` objeto que corresponde ao tipo do registro. O `EqualityContract` habilita os métodos de igualdade para comparar o tipo de tempo de execução 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`.

Ao comparar duas instâncias de um tipo derivado, os métodos de igualdade sintetizados verificam todas as propriedades dos tipos base e derivados para igualdade. O método sintetizado `GetHashCode` usa o `GetHashCode` método de todas as propriedades e campos declarados no tipo base e no tipo de registro derivado.

`with` expressões em registros derivados

O resultado de uma `with` expressão tem o mesmo tipo de tempo de execução que o operando da expressão. 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:

```

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 }
}

```

`PrintMembers` formatação 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 os campos de tipos derivados e base são incluídos na `Tostring` saída, conforme mostrado no exemplo a seguir:

```

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 `PrintMembers` método. Se você fizer isso, use a seguinte assinatura:

- Para um `sealed` registro que deriva de `object` (não declara um registro base):


```
private bool PrintMembers(StringBuilder builder);
```
- Para um `sealed` registro que deriva de outro registro (Observe que o tipo de delimitador é `sealed`, portanto, o método é efetivamente `sealed`): `protected override bool PrintMembers(StringBuilder builder);`
- Para um registro que não é `sealed` derivado de `Object`:


```
protected virtual bool PrintMembers(StringBuilder builder);
```
- Para um registro que não é `sealed` e 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 derivado de `Object` e outro para um tipo de registro que deriva de outro registro:

```

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 }
}

```

NOTE

No C# 10 e posteriores, o compilador irá sintetizar `PrintMembers` em registros derivados, mesmo quando um registro base tiver selado o `ToString` método. Você também pode criar sua própria implementação do `PrintMembers`.

Comportamento de desconstrutor em registros derivados

O `Deconstruct` método 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 de registro base serão desconstruídas, a menos que o objeto seja convertido para o tipo derivado. O exemplo a seguir demonstra como chamar um desconstrutor em um registro derivado.

```

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

Não há nenhuma restrição genérica que exija que um tipo seja um registro. Os registros atendem `class` à `struct` restrição ou. Para fazer uma restrição em uma hierarquia específica de tipos de registro, coloque a restrição no registro base como você faria com uma classe base. Para obter mais informações, consulte [restrições em parâmetros de tipo](#).

Especificação da linguagem C#

Para obter mais informações, consulte a seção [classes](#) da [especificação da linguagem C#](#).

Para obter mais informações sobre os recursos introduzidos no C# 9 e versões posteriores, consulte as seguintes notas de proposta de recurso:

- [Grava](#)
- [Setters somente de inicialização](#)
- [Retornos covariantes](#)

Confira também

- [Referência de C#](#)
- [Diretrizes de design-escolhendo entre classe e estrutura](#)
- [Diretrizes de design – design de struct](#)
- [O sistema de tipos C#](#)
- [with expressão](#)

class (Referência de C#)

21/01/2022 • 2 minutes to read

Classes são declaradas usando a palavra-chave `class`, conforme mostrado no exemplo a seguir:

```
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`.

```

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:

```
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 C#)

21/01/2022 • 3 minutes to read

Uma interface define um contrato. Qualquer um `class` ou `struct` que implemente esse contrato deve fornecer uma implementação dos membros definidos na interface. A partir do C# 8,0, uma interface pode definir uma implementação padrão para membros. Ele também pode definir `static` Membros para fornecer uma única implementação para a funcionalidade comum.

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](#).

Interface de exemplo

```
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 qualquer implementação) dos seguintes membros:

- [Métodos](#)
- [Propriedades](#)
- [Indexadores](#)
- [Eventos](#)

Essas declarações de membro anteriores normalmente não contêm um corpo. A partir do C# 8,0, um membro de interface pode declarar um corpo. Isso é chamado de *implementação padrão*. Os membros com corpos permitem que a interface forneça uma implementação "padrão" para classes e estruturas que não fornecem uma implementação de substituição. Além disso, a partir do C# 8,0, uma interface pode incluir:

- [Constantes](#)
- [Operadores](#)
- [Construtor estático](#).

- [Tipos aninhados](#)
- [Campos, métodos, propriedades, indexadores e eventos estáticos](#)
- Declarações de membro usando a sintaxe de implementação de interface explícita.
- Modificadores de acesso explícitos (o acesso padrão é `public`).

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. Essa regra tem um efeito sutil nas declarações de propriedade. Em uma declaração de interface, o código a seguir não declara uma propriedade implementada automaticamente como faz 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:

```
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 de interface padrão só podem ser acessados por meio de uma instância da interface.

Para obter mais informações sobre implementação de interface 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`.

```

interface IPPoint
{
    // Property signatures:
    int X
    {
        get;
        set;
    }

    int Y
    {
        get;
        set;
    }

    double Distance
    {
        get;
    }
}

class Point : IPPoint
{
    // 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(IPPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPPoint 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, consulte a seção [interfaces](#) da [especificação da linguagem C#](#) e a especificação de recurso para [membros de interface padrão-C# 8,0](#).

[Confira também](#)

- Referência do C#
- Guia de programação C#
- Palavras-chave do C#
- Tipos de referência
- Interfaces
- Usando propriedades
- Usando indexadores

Tipos de referência que podem ser anulados (referência de C#)

21/01/2022 • 6 minutes to read

NOTE

Este artigo aborda tipos de referência que podem ser anulados. Você também pode declarar tipos [de valor que podem ser anulados](#).

Os tipos de referência que podem ser anulados estão disponíveis a partir do C# 8.0, no código que optou por um contexto com valor *nulo*. Os tipos de referência anuáveis, os avisos de análise estática nula e o operador `null-null` são recursos de linguagem opcionais. Todos são desligados por padrão. Um *contexto que pode ser anulado* é controlado no nível do projeto usando configurações de build ou no código usando pragmas.

Em um contexto com valor nulo:

- Uma variável de um tipo de referência deve ser inicializada com não nulo e nunca pode ser atribuído `T` um valor que possa ser `null`.
- Uma variável de um tipo de referência pode ser inicializada com ou atribuída, mas deve ser verificada antes `T?` `null` da `null` desconferência.
- Uma variável do tipo é considerada não nula quando você aplica o operador `m T?` de anulação nula, como em `m!`.

As distinções entre um tipo de referência não anulada e um tipo de referência que pode ser nulo são impostas pela interpretação do compilador das `T T?` regras anteriores. Uma variável do tipo `T` e uma variável do tipo `T?` são representadas pelo mesmo tipo .NET. O exemplo a seguir declara uma cadeia de caracteres que não pode ser anulada e uma cadeia de caracteres que pode ser anulada e, em seguida, usa o operador `null-null` para atribuir um valor a uma cadeia de caracteres não anulada:

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

As `notNull` variáveis `nullable` e são representadas pelo `String` tipo. Como os tipos que permitem valor nulo e que permitem valor nulo são armazenados como o mesmo tipo, há vários locais em que o uso de um tipo de referência que permite valor nulo não é permitido. Em geral, um tipo de referência que pode ser anulado não pode ser usado como uma classe base ou uma interface implementada. Um tipo de referência que pode ser anulado não pode ser usado em nenhuma expressão de teste de tipo ou criação de objeto. Um tipo de referência que pode ser anulado não pode ser o tipo de uma expressão de acesso de membro. Os exemplos a seguir mostram estes constructos:

```

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 anuladas e análise estática

Os exemplos na seção anterior ilustram a natureza dos tipos de referência que podem ser anulados. Tipos de referência que podem ser anulados não são novos tipos de classe, mas sim anotações em tipos de referência existentes. O compilador usa essas anotações para ajudá-lo a encontrar possíveis erros de referência nula em seu código. Não há nenhuma diferença de runtime entre um tipo de referência que não pode ser anulado e um tipo de referência que pode ser anulado. O compilador não adiciona nenhuma verificação de runtime para tipos de referência que não podem ser anulados. Os benefícios estão na análise de tempo de compilação. O compilador gera avisos que ajudam você a encontrar e corrigir possíveis erros nulos em seu código. Você declara sua intenção e o compilador avisa quando seu código viola essa intenção.

Em um contexto habilitado para valor nulo, o compilador executa a análise estática em variáveis de qualquer tipo de referência, que permitem valor nulo e não anuladas. O compilador rastreia *o estado nulo* de cada variável de referência como *não nulo ou talvez nulo*. O estado padrão de uma referência não anulada não é *nulo*. O estado padrão de uma referência que pode ser anulada é *talvez nulo*.

Tipos de referência não anuladas sempre devem ser seguros para desreferência porque seu estado *nulo não é nulo*. Para impor essa regra, o compilador emite avisos se um tipo de referência não anulado não é inicializado para um valor não nulo. As variáveis locais devem ser atribuídas onde elas são declaradas. Cada campo deve ser atribuído a *um valor não nulo*, em um inicializador de campo ou em cada construtor. O compilador emite avisos quando uma referência não anulada é atribuída a uma referência cujo estado é *talvez nulo*. Em geral, uma referência não anulada não é nula e nenhum aviso é emitido quando essas variáveis são desreferenciadas.

NOTE

Se você atribuir uma *expressão talvez nula* a um tipo de referência não anulada, o compilador gerará avisos. Em seguida, o compilador gera avisos para essa variável até que ela seja atribuída a uma *expressão não nula*.

Os tipos de referência que podem ser anulados podem ser inicializados ou atribuídos a `null`. Portanto, a análise estática deve determinar que uma variável não é *nula* antes de ser desreferenciada. Se uma referência que pode ser anulada for determinada como *talvez nula*, atribuir a uma variável de referência não anulada gerará um aviso do compilador. A classe a seguir mostra exemplos desses avisos:

```

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 esta classe:

```

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 idioma para verificações nulas e atribuições para informar sua 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 que esses métodos afetam o estado nulo de uma *variável*. Há atributos que você pode adicionar às suas APIs para informar o compilador sobre a semântica de argumentos e valores de retorno. Esses atributos foram aplicados a muitas APIs comuns nas bibliotecas do .NET Core. Por exemplo, foi atualizado e o compilador interpreta corretamente esse método

[IsNullOrEmpty](#) como uma verificação nula. Para obter mais informações sobre os atributos que se aplicam à análise estática de estado nulo, consulte o artigo sobre [atributos que podem ser anulados](#).

Definindo o contexto que pode ser anulado

Há duas maneiras de controlar o contexto que pode ser anulado. No nível do projeto, você pode adicionar a `<Nullable>enable</Nullable>` configuração do projeto. Em um único arquivo de origem C#, você pode adicionar `#nullable enable` o pragma para habilitar o contexto que permite valor nulo. Consulte o artigo sobre como [definir uma estratégia que pode ser anulada](#). Antes do .NET 6, novos projetos usam o padrão, `<Nullable>disable</Nullable>`. Começando com o .NET 6, novos projetos incluem `<Nullable>enable</Nullable>` o elemento no arquivo de projeto.

Especificação da linguagem C#

Para obter mais informações, consulte as seguintes propostas para a [especificação da linguagem C#](#):

- [Tipos de referência anuláveis](#)
- [Especificação de tipos de referência que podem ser anulados de rascunho](#)

Confira também

- [Referência de C#](#)
- [Tipos de valor anuláveis](#)

void (referência C#)

21/01/2022 • 2 minutes to read

Você usa `void` como o tipo de retorno de um [método](#) (ou uma [função local](#)) para especificar que o método não retorna um valor.

```
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 Referent para declarar um ponteiro para um tipo desconhecido. Para obter mais informações, consulte [Tipos de ponteiros](#).

Você não pode usar `void` como o tipo de uma variável.

Confira também

- [Referência de C#](#)
- [System.Void](#)

var (referência C#)

21/01/2022 • 2 minutes to read

Começando com o C# 3, as variáveis que são declaradas no escopo do método podem ter um "tipo" implícito `var`. Uma variável local de tipo implícito é fortemente tipada, como se você mesmo tivesse declarado o tipo, mas o compilador determina o tipo. As duas declarações a seguir de `i` são funcionalmente equivalentes:

```
var i = 10; // Implicitly typed.  
int i = 10; // Explicitly typed.
```

IMPORTANT

Quando `var` é usado com [tipos de referência anuláveis](#) habilitados, ele sempre implica um tipo de referência anulável mesmo que o tipo de expressão não seja anulável. A análise de estado nulo do compilador protege contra desreferenciar um `null` valor potencial. Se a variável nunca for atribuída a uma expressão que talvez seja nula, o compilador não emitirá nenhum aviso. Se você atribuir a variável a uma expressão que pode ser nula, será preciso testá-la para que ela não seja nula antes de desatribuí-la para evitar avisos.

Um uso comum da `var` palavra-chave é com expressões de invocação de construtor. O uso de `var` permite que você não repita um nome de tipo em uma declaração de variável e instanciação de objeto, como mostra o exemplo a seguir:

```
var xs = new List<int>();
```

A partir do C# 9,0, você pode usar uma [new expressão](#) de tipo de destino como alternativa:

```
List<int> xs = new();  
List<int>? ys = new();
```

Na correspondência de padrões, a `var` palavra-chave é usada em um `var` padrão.

Exemplo

O exemplo a seguir mostra duas expressões de consulta. Na primeira expressão, o uso de `var` é permitido, mas não é necessário, pois o tipo do resultado da consulta pode ser declarado explicitamente como um `IEnumerable<string>`. No entanto, na segunda expressão, `var` permite que o resultado seja uma coleção de tipos anônimos e o nome desse tipo não é acessível, exceto para o próprio compilador. O uso de `var` elimina a necessidade de criar uma nova classe para o resultado. Observe que no exemplo 2, a variável de iteração `foreach` `item` também deve ser implicitamente tipada.

```

// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
    where word[0] == 'g'
    select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
    where cust.City == "Phoenix"
    select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}

```

Confira também

- [Referência de C#](#)
- [Variáveis locais de tipo implícito](#)
- [Relações de tipo em operações de consulta LINQ](#)

Tipos internos (referência C#)

21/01/2022 • 2 minutes to read

A tabela a seguir lista os tipos de [valores](#) internos do C#:

PALAVRA-CHAVE TYPE DO C#	TIPO .NET
<code>bool</code>	<code>System.Boolean</code>
<code>byte</code>	<code>System.Byte</code>
<code>sbyte</code>	<code>System.SByte</code>
<code>char</code>	<code>System.Char</code>
<code>decimal</code>	<code>System.Decimal</code>
<code>double</code>	<code>System.Double</code>
<code>float</code>	<code>System.Single</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>nint</code>	<code>System.IntPtr</code>
<code>nuint</code>	<code>System.UIntPtr</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>

A tabela a seguir lista os tipos de [referência](#) interna do C#:

PALAVRA-CHAVE TYPE DO C#	TIPO .NET
<code>object</code>	<code>System.Object</code>
<code>string</code>	<code>System.String</code>
<code>dynamic</code>	<code>System.Object</code>

Nas tabelas anteriores, cada palavra-chave de tipo C# da coluna esquerda (exceto `Nint` e `nuint` e `Dynamic`) é um

alias para o tipo .NET correspondente. Eles são intercambiáveis. Por exemplo, as declarações a seguir declaram variáveis do mesmo tipo:

```
int a = 123;  
System.Int32 b = 123;
```

Os `nint` e `nuint` tipos e são inteiros de tamanho nativo. Eles são representados internamente pelos tipos do .NET indicados, mas, em cada caso, a palavra-chave e o tipo .NET não são intercambiáveis. O compilador fornece operações e conversões para `nint` e `nuint` como tipos inteiros que ele não fornece para os tipos de ponteiro `System.IntPtr` e `System.UIntPtr`. Para obter mais informações, consulte [nint e nuint tipos](#).

A `void` palavra-chave representa a ausência de um tipo. Você o usa como o tipo de retorno de um método que não retorna um valor.

Confira também

- [Referência de C#](#)
- [Valores padrão de tipos C#](#)

Tipos não gerenciados (referência em C#)

21/01/2022 • 2 minutes to read

Um tipo é um **tipo não gerenciado**, se for qualquer um dos seguintes tipos:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` ou `bool`
- Qualquer tipo de [Enumeração](#)
- Qualquer tipo de [ponteiro](#)
- Qualquer tipo de [struct](#) definido pelo usuário que contém campos de tipos não gerenciados somente e, em C# 7,3 e anterior, não é um tipo construído (um tipo que inclui pelo menos um argumento de tipo)

A partir do C# 7,3, você pode usar a `unmanaged` restrição para especificar que um parâmetro de tipo é um tipo não gerenciado não-ponteiro e não anulável.

A partir do C# 8,0, um tipo struct *construído* que contém campos de tipos não gerenciados também é não gerenciado, como mostra o exemplo a seguir:

```
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
```

Uma estrutura genérica pode ser a fonte de tipos construídos não gerenciados e não gerenciados. O exemplo anterior define uma struct genérica `Coords<T>` e apresenta os exemplos de tipos construídos não gerenciados. O exemplo de não é um tipo não gerenciado `Coords<object>`. Não é não gerenciado porque tem os campos do `object` tipo, que não são gerenciados. Se você quiser que *todos* os tipos construídos sejam tipos não gerenciados, use a `unmanaged` restrição na definição de uma estrutura genérica:

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

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#)

21/01/2022 • 2 minutes to read

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 <i>valor nulo</i> de um tipo de valor que pode ser nulo.

Expressões de valor padrão

Use o `default` operador para produzir o valor padrão de um tipo, como mostra o exemplo a seguir:

```
int a = default(int);
```

A partir do C# 7.1, você pode usar o `default` literal para inicializar uma variável com o valor padrão de seu tipo:

```
int a = default;
```

Construtor sem parâmetros de um tipo de valor

Para um tipo de valor, o construtor sem parâmetro implícito também produz o valor padrão do tipo, como mostra o exemplo a seguir:

```
var n = new System.Numerics.Complex();
Console.WriteLine(n); // output: (0, 0)
```

Em tempo de operação, se a instância representar um tipo de valor, você poderá usar o método para invocar o

construtor sem parâmetros para obter o [System.Type Activator.CreateInstance\(Type\)](#) valor padrão do tipo.

NOTE

No C# 10 e posterior, um tipo de estrutura (que é um tipo de valor) pode ter um construtor sem parâmetros explícito que pode produzir um valor não padrão do tipo. Portanto, é recomendável usar `default` o operador ou o literal para produzir o valor padrão de um `default` tipo.

Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Valores padrão](#)
- [Construtores padrão](#)

Confira também

- [Referência de C#](#)
- [Construtores](#)

Palavras-chave C#

21/01/2022 • 2 minutes to read

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 incluam `@` 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.

resume
como
base
bool
break
byte
case
catch
char
check
class
const
continua
decimal
default
delegate
do
double
senão
enumeração

event
explicita
extern
false
disso
fixado
float
for
foreach
goto
if
localiza
Em
int
interface

interno
is
proprietário
longo

namespace
novo
null
object
operator
out
substituição
params
pessoal
protected
public
leitura
ref
exibir
sbyte
sealed
short
sizeof
stackalloc

static
cadeia de caracteres
struct
switch
this
throw
true
Tente
typeof
uint
ulong
unchecked
UNSAFE
ushort
usando
virtuaisLUNs
livre
volatile
mesmo

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
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

nonnull
nuint
on
or
orderby
partial (tipo)
partial (método)
Registro
remove
select

set
nãomanaged (convenção de chamada de ponteiro de função)
nãomanaged (restrição de tipo genérico)
value
Var
when (condição de filtro)
where (restrição de tipo genérico)
where (cláusula de consulta)
Com
Rendimento

Confira também

- Referência de C#

Modificadores de acesso (Referência de C#)

21/01/2022 • 2 minutes to read

Os modificadores de acesso são palavras-chave usadas para especificar a acessibilidade declarada de um membro ou de um tipo. Esta seção apresenta os quatro modificadores de acesso:

- `public`
- `protected`
- `internal`
- `private`

Os seis seguintes níveis de acessibilidade podem ser especificados usando os modificadores de acesso:

- `public`: O acesso não é restrito.
- `protected`: O acesso é limitado à classe que a contém ou aos tipos derivados da classe que a contém.
- `internal`: O acesso é limitado ao assembly atual.
- `protected internal`: O acesso é limitado ao assembly ou tipos atuais derivados da classe que a contém.
- `private`: O acesso é limitado ao tipo recipiente.
- `private protected`: O acesso é limitado à classe que a contém ou aos tipos derivados da classe que a contém dentro do assembly atual.

Esta seção também apresenta o seguinte:

- **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

- [Referência do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Palavras-chave de acesso](#)
- [Modificadores](#)

Níveis de acessibilidade (Referência de C#)

21/01/2022 • 2 minutes to read

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. Disponível desde o C# 7.2.

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
<code>enum</code>	<code>public</code>	Nenhum

MEMBROS DE	ACESSIBILIDADE DE MEMBRO PADRÃO	ACESSIBILIDADE DECLARADA PERMITIDA DO MEMBRO
<code>class</code>	<code>private</code>	<code>public</code> <code>protected</code> <code>internal</code> <code>private</code> <code>protected internal</code> <code>private protected</code>
<code>interface</code>	<code>public</code>	<code>public</code> <code>protected</code> <code>internal</code> <code>private</code> * <code>protected internal</code> <code>private protected</code>
<code>struct</code>	<code>private</code>	<code>public</code> <code>internal</code> <code>private</code>

* Um `interface` membro com `private` acessibilidade deve ter uma implementação padrão.

A acessibilidade de um tipo aninhado depende de seu [domínio de acessibilidade](#), que é determinado pela acessibilidade declarada do membro e o domínio de acessibilidade do tipo que o contém imediatamente. 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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Domínio de acessibilidade](#)
- [Restrições ao uso de níveis de acessibilidade](#)
- [Modificadores de acesso](#)
- [público](#)
- [pessoal](#)

- [protected](#)
- [interno](#)

Domínio de acessibilidade (Referência de C#)

21/01/2022 • 2 minutes to read

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 pela referência a um membro inacessível, remova os comentários um de cada vez.

```
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
```

```

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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Restrições ao uso de níveis de acessibilidade](#)
- [Modificadores de acesso](#)
- [público](#)
- [pessoal](#)
- [protected](#)
- [interno](#)

Restrições ao uso de níveis de acessibilidade (Referência em C#)

21/01/2022 • 2 minutes to read

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`:

```
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.
Operadores	O tipo de retorno e os tipos de parâmetro de um operador devem ser, pelo menos, tão acessíveis quanto o próprio operador.

CONTEXTO	COMENTÁRIOS
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.

```

// 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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Domínio de acessibilidade](#)
- [Níveis de acessibilidade](#)
- [Modificadores de acesso](#)
- [público](#)
- [pessoal](#)
- [protected](#)
- [interno](#)

internal (Referência de C#)

21/01/2022 • 2 minutes to read

A palavra-chave `internal` é um [modificador de acesso](#) para tipos e membros de tipo.

Esta página aborda o acesso `internal`. A `internal` palavra-chave também faz parte do [protected internal](#) modificador de acesso.

Tipos ou membros internos são acessíveis somente em arquivos no mesmo assembly, como neste exemplo:

```
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.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
    public static int intM = 0;
}
```

```
// 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.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
    internal static int intM = 0;
}
```

```
// 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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)
- [public](#)
- [pessoal](#)
- [protected](#)

private (Referência de C#)

21/01/2022 • 2 minutes to read

A palavra-chave `private` é um modificador de acesso de membro.

Esta página aborda o acesso `private`. A `private` palavra-chave também faz parte do `private protected` modificador de acesso.

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:

```
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. (Consulte [Propriedades](#) para obter mais informações.)

```

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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)
- [public](#)
- [protected](#)
- [interno](#)

protected (Referência de C#)

21/01/2022 • 2 minutes to read

A palavra-chave `protected` é um modificador de acesso de membro.

NOTE

Esta página aborda o acesso `protected`. A `protected` palavra-chave também faz parte `protected internal` dos `private protected` modificadores de acesso e.

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:

```
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.

```
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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)
- [public](#)
- [pessoal](#)
- [interno](#)
- [Questões de segurança de palavras-chave virtuais internas](#)

public (Referência de C#)

21/01/2022 • 2 minutes to read

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:

```
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`.

```
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 [protegido](#), 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 do C#](#)
- [Guia de programação C#](#)
- [Modificadores de acesso](#)
- [Palavras-chave do C#](#)

- Modificadores de acesso
- Níveis de acessibilidade
- Modificadores
- pessoal
- protected
- interno

protected internal (referência do C#)

21/01/2022 • 2 minutes to read

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:

```
// 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;
        }
    }
}
```

```
// 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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)
- [público](#)
- [pessoal](#)
- [interno](#)
- [Questões de segurança de palavras-chave virtuais internas](#)

private protected (referência do C#)

21/01/2022 • 2 minutes to read

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](#).

NOTE

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:

```
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;
    }
}
```

```
// 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` [InternalsVisibleToAttribute](#) contiver um que `Assembly2` nomeia, a classe derivada terá acesso `DerivedClass2` aos membros `private protected` declarados em `BaseClass`. [InternalsVisibleTo](#) torna `private protected` os membros 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](#)
- [Privada](#)
- [interno](#)
- [Questões de segurança de palavras-chave virtuais internas](#)

abstract (Referência de C#)

21/01/2022 • 3 minutes to read

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`:

```
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 contem 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:

```
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:

```
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`.

```

// 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:

```
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](#)
- [Substituir](#)
- [Palavras-chave do C#](#)

async (Referência de C#)

21/01/2022 • 4 minutes to read

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`:

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

Se você for novo na programação assíncrona ou não entender `await` como um método assíncrono usa o operador para fazer um trabalho de execução potencialmente longa sem bloquear o thread do chamador, leia a introdução na programação assíncrona com [async aguarde](#). O código a seguir é encontrado dentro de um método assíncrono e chama o método [HttpClient.GetStringAsync](#):

```
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:

```
<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`.

```
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
```

IMPORTANT

Para obter mais informações sobre tarefas e o código que é executado enquanto aguarda uma tarefa, consulte [Programação assíncrona com `async` e `await`](#). Para ver um exemplo de console completo que usa elementos semelhantes, consulte [Processar tarefas assíncronas conforme elassão 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.

- Começando com o C# 7.0, qualquer tipo que tenha um método acessível `GetAwaiter`. 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.

A partir do C# 7.0, 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](#)

const (Referência de C#)

21/01/2022 • 2 minutes to read

Use a palavra-chave `const` para declarar um campo constante ou um local constante. 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:

```
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:

```
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 um local ou 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 as constantes de tipos de referência são `string` e uma referência nula.

A declaração constante pode declarar constantes múltiplas como:

```
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:

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

NOTE

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

```
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
*/
```

Este exemplo demonstra como usar constantes como variáveis locais.

```
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, 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 do C#](#)

- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)
- [Leitura](#)

evento (referência C#)

21/01/2022 • 2 minutes to read

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 `EventHandler<TEventArgs>` tipo de delegado genérico e como assinar um evento e criar um método manipulador de eventos, consulte [como publicar eventos que estão em conformidade com as diretrizes do .NET](#).

```
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 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 [público](#), [privado](#), [protegido](#), [interno](#), [interno protegido](#) ou [protegido privado](#). 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
<code>static</code>	Torna o evento disponível para chamadores a qualquer momento, mesmo se não existir nenhuma instância da classe.	Classes static e membros de classes static

PALAVRA-CHAVE	DESCRIÇÃO	PARA OBTER MAIS INFORMAÇÕES
virtuaisLUNs	Permite que classes derivadas substituam o comportamento do evento, usando a palavra-chave <code>override</code> .	Herança
sealed	Especifica que, para classes derivadas, o evento não é mais virtual.	
resume	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 [classe estáticas e membros de classe estática](#).

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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [adicionar](#)
- [remove](#)
- [Modificadores](#)
- [Como combinar delegados \(delegados de multicast\)](#)

extern (Referência de C#)

21/01/2022 • 2 minutes to read

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:

```
[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.

```
//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`:

```
// 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`:

```
// 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:

```
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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)

in (modificador genérico) (Referência de C#)

21/01/2022 • 2 minutes to read

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.

```
// 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.

```
// 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#)

21/01/2022 • 3 minutes to read

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 está visível, pois ela já estaria oculta se fosse marcada como ou, em alguns casos, `internal`. Embora você possa ocultar `public` membros ou sem usar o `protected` `new` modificador, você obterá 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:

```
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.

```
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.

```

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:

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#)

21/01/2022 • 2 minutes to read

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.

```
// 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.

NOTE

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.

```
// 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](#)

override (referência de C#)

21/01/2022 • 3 minutes to read

O modificador `override` é necessário para estender ou modificar a implementação abstrata ou virtual de um método, propriedade, indexador ou evento herdado.

No exemplo a seguir, a classe deve fornecer uma implementação de substituído `Square` de `GetArea` porque é `GetArea` herdada da classe `Shape` abstrata:

```
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 `override` método fornece uma nova implementação do método herdado de uma classe base. O método que é substituído por uma declaração `override` é conhecido como o método base substituído. Um `override` método deve ter a mesma assinatura que o método base substituído. A partir do C# 9.0, os métodos `override` suportam tipos de retorno covariantes. Em particular, o tipo de retorno de `override` um método pode derivar do tipo de retorno do método base correspondente. No C# 8.0 e anteriores, os tipos de retorno de um método e o método `override` base substituído devem ser os mesmos.

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. A partir do C# 9.0, as propriedades de substituição somente leitura suportam tipos de retorno covariantes. A propriedade substituído deve ser `virtual`, `abstract` ou `override`.

Para obter mais informações sobre como usar a palavra-chave , consulte Controle de versão com as palavras-chave `Override` e `New` e Saber quando usar `override` [Override e New Keywords](#). 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.

```

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, consulte a [seção Substituir métodos](#) da [especificação da linguagem C#](#).

Para obter mais informações sobre tipos de retorno covariantes, consulte a nota [de proposta de recurso](#).

Confira também

- [Referência de C#](#)
- [Herança](#)
- [Palavras-chave de C#](#)
- [Modificadores](#)
- [Abstrata](#)
- [Virtual](#)
- [new \(modificador\)](#)
- [Polimorfismo](#)

readonly (Referência de C#)

21/01/2022 • 4 minutes to read

A `readonly` palavra-chave é um modificador que pode ser usado em quatro contextos:

- Em uma [declaração de campo](#), indica que a atribuição ao campo só pode ocorrer como parte da declaração ou em um construtor na mesma `readonly` classe. Um campo `readonly` pode ser atribuído e reatribuído várias vezes na declaração de campo e no construtor.

Um `readonly` campo não pode ser atribuído após a 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 os dados da instância do campo são modificados por meio do campo somente leitura.

WARNING

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 o aviso [CA2104](#): "Não declarar tipos de referência mutáveis somente leitura".

- Em uma `readonly struct` definição de `readonly` tipo, indica que o tipo de estrutura é imutável. Para obter mais informações, consulte a [readonly](#) seção `struct` do artigo [Tipos de estrutura](#).
- Em uma declaração de membro de instância dentro de um tipo de estrutura, indica que um membro da instância `readonly` não modifica o estado da estrutura. Para obter mais informações, consulte a [readonly](#) seção [membros da](#) instância do artigo [Tipos de estrutura](#).
- Em um `ref readonly` [retorno de método](#), o modificador indica que o método retorna uma referência e as gravações não são `readonly` permitidas para essa referência.

Os `readonly struct` `ref readonly` contextos e foram adicionados no C# 7.2. `readonly` Membros do `struct` foram adicionados no C# 8.0

Exemplo de campo `readonly`

Neste exemplo, o valor do campo não pode ser alterado no método , mesmo que ele tenha atribuído um valor no `year` `ChangeYear` construtor de classe:

```
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:

```
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 contextos nos quais é válido passar um `readonly` campo como um parâmetro `out` ou `ref`.

NOTE

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, embora um campo seja uma constante de tempo de compilação, o campo pode ser usado para constantes de tempo de executar, como `const` no exemplo a seguir:

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```

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:

```
p2.y = 66; // Error
```

Você obterá a mensagem de erro do compilador:

Um campo somente leitura não pode ser atribuído a (exceto em um construtor ou em um inicializador de variável)

Exemplo de retorno de readonly de ref

O `readonly` modificador 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 `readonly` modificador para indicar que os chamadores não podem modificar a origem:

```

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:

- [struct readonly ref e readonly](#)
- [membros de struct readonly](#)

Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)
- [const](#)
- [Fields](#)

sealed (Referência de C#)

21/01/2022 • 2 minutes to read

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`.

```
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`.

```
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](#).

```

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](#)
- [Substituir](#)
- [Virtual](#)

static (Referência de C#)

21/01/2022 • 3 minutes to read

Esta página aborda a `static` palavra-chave do modificador. A `static` palavra-chave também faz parte da [using static](#) diretiva.

Use o modificador `static` para declarar um membro estático que pertença ao próprio tipo, em vez de um objeto específico. O `static` modificador pode ser usado para declarar `static` classes. Em classes, interfaces e structs, você pode adicionar o `static` modificador a campos, métodos, propriedades, operadores, eventos e construtores. O `static` modificador não pode ser usado com indexadores ou finalizadores. Para obter mais informações, consulte [classes estáticas e membros de classe estática](#).

A partir do C# 8,0, você pode adicionar o `static` modificador a uma [função local](#). Uma função local estática não pode capturar variáveis locais ou estado de instância.

A partir do C# 9,0, você pode adicionar o `static` modificador a uma [expressão lambda](#) ou a um [método anônimo](#). Um método lambda ou anônimo estático não pode capturar variáveis locais ou estado de instância.

Exemplo-classe estática

A seguinte classe é declarada como `static` e contém apenas métodos `static`:

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/ }
}
```

Uma constante ou declaração de tipo é implicitamente um `static` membro. Um `static` membro não pode ser referenciado por meio de uma instância. Em vez disso, ele é referenciado por meio do nome do tipo. Por exemplo, considere a seguinte classe:

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

Para fazer referência ao `static` membro `x`, use o nome totalmente qualificado, `MyBaseC.MyStruct.x`, a menos que o membro seja acessível do mesmo escopo:

```
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 `static` campo.

Não é possível usar `this` para referenciar `static` métodos ou acessadores de propriedade.

Se a `static` palavra-chave for aplicada a uma classe, todos os membros da classe deverão ser `static`.

Classes, interfaces e `static` classes podem ter `static` construtores. Um `static` Construtor é chamado em algum ponto entre quando o programa é iniciado e a classe é instanciada.

NOTE

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 `static` os membros, 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 uma instância de funcionário. Em vez disso, eles pertencem à classe de funcionários como um todo. Eles devem ser declarados como `static` membros da classe.

Exemplo-campo e método estáticos

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. Este programa lê o número atual de funcionários do teclado.

```

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.WriteLine("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 `static` campo usando outro `static` campo que ainda não está declarado. Os resultados serão indefinidos até que você atribua explicitamente um valor ao `static` campo.

```
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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)
- [usando diretiva estática](#)
- [Classes static e membros de classes static](#)

unsafe (Referência de C#)

21/01/2022 • 2 minutes to read

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`:

```
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:

```
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:

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

Para compilar código não seguro, você deve especificar a opção do compilador [AllowUnsafeBlocks](#). O código não seguro não é verificável pelo Common Language Runtime.

Exemplo

```
// 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#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Instrução fixed](#)
- [Código não seguro e ponteiros](#)
- [Buffers de tamanho fixo](#)

virtual (Referência de C#)

21/01/2022 • 3 minutes to read

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:

```
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:

```

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";
            }
        }
    }
}

```

As propriedades virtuais se comportam como métodos virtuais, exceto pelas diferenças na sintaxe de declaração e invocação.

- 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.

```
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.

```
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](#)
- [Abstrata](#)
- [Substituir](#)
- [new \(modificador\)](#)

volatile (Referência de C#)

21/01/2022 • 3 minutes to read

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](#).

NOTE

Em um sistema multiprocessador, uma operação de leitura volátil não garante a obtenção do valor mais recente gravado nesse local de memória por qualquer processador. Da mesma forma, uma operação de gravação volátil não garante que o valor escrito seria 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 multi-threaded a esses tipos de campos, use os membros da [Interlocked](#) classe ou proteja o acesso usando a `lock` instrução.

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`.

```
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.

```
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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)
- [instrução Lock](#)
- [Interlocked](#)

Palavras-chave de instrução (Referência de C#)

21/01/2022 • 2 minutes to read

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 jump	<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>
Verificado e desmarcado	<code>checked</code> , <code>unchecked</code>
instrução fixed	Fixo
instrução lock	<code>lock</code>
instrução yield	<code>yield</code>

Confira também

- [Referência de C#](#)
- [Declarações](#)
- [Palavras-chave do C#](#)

throw (Referência de C#)

21/01/2022 • 3 minutes to read

Indica a ocorrência de uma exceção durante a execução do programa.

Comentários

A sintaxe de `throw` é:

```
throw [e];
```

em que `e` é uma instância de uma classe derivada de [System.Exception](#). O exemplo a seguir usará a instrução `throw` para gerar um [IndexOutOfRangeException](#) se o argumento passado para um método chamado `GetNumber` não corresponder a um índice válido de uma matriz interna.

```
using System;

namespace Throw2
{
    public class NumberGenerator
    {
        int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

        public int GetNumber(int index)
        {
            if (index < 0 || index >= numbers.Length)
            {
                throw new IndexOutOfRangeException();
            }
            return numbers[index];
        }
    }
}
```

Os chamadores do método, então, usam um bloco `try-catch` ou `try-catch-finally` para tratar a exceção gerada. O exemplo a seguir trata a exceção gerada pelo método `GetNumber`.

```
using System;

public class Example
{
    public static void Main()
    {
        var gen = new NumberGenerator();
        int index = 10;
        try
        {
            int value = gen.GetNumber(index);
            Console.WriteLine($"Retrieved {value}");
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
        }
    }
}

// The example displays the following output:
//      IndexOutOfRangeException: 10 is outside the bounds of the array
```

Como gerar novamente uma exceção

`throw` também pode ser usado em um bloco `catch` para gerar novamente uma exceção tratada em um bloco `catch`. Nesse caso, `throw` não usa um operando de exceção. Ele é mais útil quando um método passa um argumento de um chamador para algum outro método de biblioteca e o método de biblioteca gera uma exceção que deve ser passada para o chamador. O exemplo a seguir gera novamente um [NullReferenceException](#) que é gerado ao tentar recuperar o primeiro caractere de uma cadeia de caracteres não inicializada.

```

using System;

namespace Throw
{
    public class Sentence
    {
        public Sentence(string s)
        {
            Value = s;
        }

        public string Value { get; set; }

        public char GetFirstCharacter()
        {
            try
            {
                return Value[0];
            }
            catch (NullReferenceException e)
            {
                throw;
            }
        }
    }

    public class Example
    {
        public static void Main()
        {
            var s = new Sentence(null);
            Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
        }
    }
}
// The example displays the following output:
//     Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//         at Sentence.GetFirstCharacter()
//         at Example.Main()

```

IMPORTANT

Também é possível usar a sintaxe `throw` e `catch` em um bloco `try` para instanciar uma nova exceção que você passa para o chamador. Nesse caso, o rastreamento de pilha da exceção original, que fica disponível na propriedade `StackTrace`, não é preservado.

A expressão `throw`

Começando com o C# 7.0, o `throw` pode ser usado como uma expressão, bem como uma instrução. Isso permite que uma exceção seja gerada em contextos que não tinham suporte anteriormente. Estão incluídos:

- **o operador condicional.** O exemplo a seguir usará uma expressão `throw` para gerar um `ArgumentException` se uma matriz de cadeia de caracteres vazia for passada para um método. Antes do C# 7.0, essa lógica precisava aparecer em uma instrução `if / else`.

```
private static void DisplayFirstNumber(string[] args)
{
    string arg = args.Length >= 1 ? args[0] :
        throw new ArgumentException("You must supply an argument");
    if (Int64.TryParse(arg, out var number))
        Console.WriteLine($"You entered {number:F0}");
    else
        Console.WriteLine($"{arg} is not a number.");
}
```

- [o operador de união nula](#). No exemplo a seguir, uma expressão `throw` será usada com um operador de união nula para gerar uma exceção se a cadeia de caracteres atribuída a uma propriedade `Name` for `null`.

```
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 exemplo a seguir ilustra um método com corpo de expressão que gera um [InvalidOperationException](#) porque uma conversão para um valor [DateTime](#) não tem suporte.

```
DateTime ToDateTime(IFormatProvider provider) =>
    throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

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#](#)
- [try-catch](#)
- [Palavras-chave do C#](#)
- [Como gerar exceções explicitamente](#)

try-catch (Referência de C#)

21/01/2022 • 9 minutes to read

A instrução try-catch consiste em um bloco `try` seguido por uma ou mais cláusulas `catch`, que especificam os manipuladores para diferentes exceções.

Quando uma exceção é lançada, o CLR (Common Language Runtime) procura a instrução `catch` que trata essa exceção. Se o método em execução 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 exibirá uma mensagem de exceção sem tratamento para o usuário e interromperá a execução do programa.

O bloco `try` contém o código protegido que pode causar a exceção. O bloco é executado até que uma exceção seja lançada ou ele seja concluído com êxito. Por exemplo, a tentativa a seguir de converter um objeto `null` gera a exceção `NullReferenceException`:

```
object o2 = null;
try
{
    int i2 = (int)o2;    // Error
}
```

Embora a cláusula `catch` possa ser usada sem argumentos para capturar qualquer tipo de exceção, esse uso não é recomendado. Em geral, você deve capturar apenas as exceções das quais você sabe se recuperar. Portanto, você sempre deve especificar um argumento de objeto derivado de `System.Exception`. O tipo de exceção deve ser o mais específico possível para evitar a aceitação incorreta de exceções que o manipulador de exceção não é realmente capaz de resolver. Dessa forma, prefira exceções concretas em vez do tipo `Exception` base. Por exemplo:

```
catch (InvalidOperationException e)
{
    // recover from exception
}
```

É possível usar mais de uma cláusula `catch` específica na mesma instrução try-catch. Nesse caso, a ordem das cláusulas `catch` é importante porque as cláusulas `catch` são examinadas em ordem. Capture as exceções mais específicas antes das menos específicas. O compilador gerará um erro se você ordenar os blocos `catch` de forma que um bloco posterior nunca possa ser alcançado.

Usar argumentos `catch` é uma maneira de filtrar as exceções que deseja manipular. Use também um filtro de exceção que examina melhor a exceção para decidir se ela deve ser manipulada. Se o filtro de exceção retornar falso, a pesquisa por um manipulador continuará.

```
catch (ArgumentException e) when (e.ParamName == "...")
{
    // recover from exception
}
```

Os filtros de exceção são preferíveis em relação à captura e relançamento (explicados abaixo) porque os filtros deixam a pilha intacta. Se um manipulador posterior despeja a pilha, você pode ver de onde a exceção

originalmente veio, em vez de apenas o último lugar em que ela foi relançada. Um uso comum de expressões de filtro de exceção é o registro em log. Crie um filtro que sempre retorna falso e que também gera um log; você pode registrar exceções conforme elas ocorrem sem precisar manipulá-las e gerá-las novamente.

Uma instrução `throw` pode ser usada em um bloco `catch` para relançar a exceção que foi capturada pela instrução `catch`. O exemplo a seguir extrai informações de origem de uma exceção `IOException` e, em seguida, lança a exceção para o método pai.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

Você pode capturar uma exceção e lançar uma exceção diferente. Quando fizer isso, especifique a exceção capturada como a exceção interna, como mostrado no exemplo a seguir.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

Você pode também relançar uma exceção quando uma determinada condição for verdadeira, conforme mostrado no exemplo a seguir.

```
catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

Também é possível usar um filtro de exceção para obter um resultado semelhante em um modo geralmente mais limpo (além de não modificar a pilha, conforme explicado anteriormente neste documento). O exemplo a seguir tem um comportamento semelhante para chamadores como no exemplo anterior. A função gera a `InvalidCastException` novamente para o chamador quando `e.Data` é `null`.

```
catch (InvalidCastException e) when (e.Data != null)
{
    // Take some action.
}
```

De dentro de um bloco `try`, initialize somente as variáveis que são declaradas nele. Caso contrário, uma exceção pode ocorrer antes da conclusão da execução do bloco. Por exemplo, no exemplo de código a seguir, a variável `n` é inicializada dentro do bloco `try`. Uma tentativa de usar essa variável fora do bloco `try` na instrução `Write(n)` gerará um erro de compilador.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.WriteLine(n);
}
```

Para obter mais informações sobre `catch`, consulte [try-catch-finally](#).

Exceções em métodos assíncronos

Um método assíncrono é marcado por um modificador `async` e geralmente contém uma ou mais expressões ou instruções `await`. Uma expressão `await` aplica o operador `await` a um `Task` ou `Task<TResult>`.

Quando o controle atinge um `await` no método assíncrono, o progresso no método é suspenso até que a tarefa aguardada seja concluída. Quando a tarefa for concluída, a execução poderá ser retomada no método. Para obter mais informações, consulte [Programação assíncrona com async e await](#).

A tarefa concluída para a qual `await` é aplicada pode estar em um estado de falha devido a uma exceção sem tratamento no método que retorna a tarefa. Aguardar a tarefa gera uma exceção. Uma tarefa também poderá terminar em um estado cancelado se o processo assíncrono que a retorna for cancelado. Aguardar uma tarefa cancelada lança um `OperationCanceledException`.

Para capturar a exceção, aguarde a tarefa em um bloco `try` e capture a exceção no bloco `catch` associado. Para obter um exemplo, confira a seção [Exemplo de método assíncrono](#).

Uma tarefa pode estar em um estado de falha porque ocorreram várias exceções no método assíncrono esperado. Por exemplo, a tarefa pode ser o resultado de uma chamada para `Task.WhenAll`. Quando você espera uma tarefa, somente uma das exceções é capturada e não é possível prever qual exceção será capturada. Para obter um exemplo, confira a seção [Exemplo de Task.WhenAll](#).

Exemplo

No exemplo a seguir, o bloco `try` contém uma chamada para o método `ProcessString` que pode causar uma exceção. A cláusula `catch` contém o manipulador de exceção que apenas exibe uma mensagem na tela. Quando instrução `throw` é chamada de dentro de `ProcessString`, o sistema procura a instrução `catch` e exibe a mensagem `Exception caught`.

```

class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException(paramName: nameof(s), message: "parameter can't be null.");
        }
    }

    public static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}

/*
Output:
System.ArgumentNullException: Value cannot be null.
at TryFinallyTest.Main() Exception caught.
*/

```

Exemplo de dois blocos catch

No exemplo a seguir, dois blocos catch são usados e a exceção mais específica, que vem primeiro, é capturada.

Para capturar a exceção menos específica, você pode substituir a instrução `throw` em `ProcessString` pela seguinte instrução: `throw new Exception()`.

Se você colocar o bloco catch menos específica primeiro no exemplo, a seguinte mensagem de erro aparecerá:

`A previous catch clause already catches all exceptions of this or a super type ('System.Exception').`

```

class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException(paramName: nameof(s), message: "Parameter can't be null");
        }
    }

    public static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/

```

Exemplo de método assíncrono

O exemplo a seguir ilustra o tratamento de exceção para métodos assíncronos. Para capturar uma exceção que lança uma tarefa assíncrona, coloque a expressão `await` em um bloco `try` e capture a exceção em um bloco `catch`.

Remova a marca de comentário da linha `throw new Exception` no exemplo para demonstrar o tratamento de exceção. A propriedade `IsFaulted` da tarefa é definida para `True`, a propriedade `Exception.InnerException` da tarefa é definida para a exceção e a exceção é capturada em um bloco `catch`.

Remova a marca de comentário da linha `throw new OperationCanceledException` para demonstrar o que acontece quando você cancela um processo assíncrono. A propriedade `IsCanceled` da tarefa é definida para `true` e a exceção é capturada no bloco `catch`. Em algumas condições que não se aplicam a este exemplo, a propriedade `IsFaulted` da tarefa é definida para `true` e `IsCanceled` é definido para `false`.

```

public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted: " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted: False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted: True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted: False

```

Exemplo de Task.WhenAll

O exemplo a seguir ilustra a manipulação de exceção em que várias tarefas podem resultar em várias exceções. O bloco `try` aguarda a tarefa que é retornada por uma chamada para [Task.WhenAll](#). A tarefa é concluída quando as três tarefas às quais WhenAll se aplica são concluídas.

Cada uma das três tarefas causa uma exceção. O bloco `catch` itera por meio de exceções, que são encontradas na propriedade `Exception.InnerExceptions` da tarefa que foi retornada por [Task.WhenAll](#).

```

public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerException)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task

```

Especificação da linguagem C#

Para obter mais informações, confira a seção [A instrução try](#) da [Especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Instruções try, throw e catch \(C++\)](#)
- [Jogar](#)
- [try-finally](#)
- [Como gerar exceções explicitamente](#)

try-finally (Referência de C#)

21/01/2022 • 3 minutes to read

Usando um bloco `finally`, você pode limpar todos os recursos alocados em um bloco `try` e pode executar código mesmo se uma exceção ocorrer no bloco `try`. Normalmente, as instruções de um bloco `finally` são executadas quando o controle deixa uma instrução `try`. A transferência de controle pode ocorrer como resultado da execução normal, da execução de uma instrução `break`, `continue`, `goto` ou `return`, ou da propagação de uma exceção para fora da instrução `try`.

Dentro de uma exceção tratada, é garantido que o bloco `finally` será executado. No entanto, se a exceção não for tratada, a execução do bloco `finally` depende de como a operação de desenrolamento da exceção é disparada. Isso, por sua vez, depende da configuração do seu computador. Os únicos casos em `finally` que as cláusulas não são executados envolvem um programa sendo interrompido imediatamente. Um exemplo disso seria quando é lançado [InvalidOperationException](#) devido à corrupção das instruções IL. Na maioria dos sistemas operacionais, a limpeza razoável de recursos ocorrerá como parte da interrupção e descarregamento do processo.

Normalmente, quando uma exceção sem tratamento encerra um aplicativo, não é importante se o bloco `finally` é executado. No entanto, se você tiver instruções em um bloco `finally` que devem ser executadas mesmo nesse caso, uma solução é adicionar um bloco `catch` à instrução `try - finally`. Como alternativa, você pode detectar a exceção que pode ser gerada no bloco `try` de uma instrução `try - finally` em posição superior na pilha de chamadas. Ou seja, você pode detectar a exceção no método que chama o método que contém a instrução `try - finally` ou no método que chama esse método ou em qualquer método na pilha de chamadas. Se a exceção não for detectada, a execução do bloco `finally` dependerá do sistema operacional escolher disparar uma operação de desenrolamento de exceção.

Exemplo

No exemplo a seguir, uma instrução de conversão inválida causa uma exceção `System.InvalidCastException`. A exceção não é tratada.

```
public class ThrowTestA
{
    public static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

No exemplo a seguir, uma exceção do método `TryCast` ocorre em um método mais para cima na pilha de chamadas.

```

public class ThrowTestB
{
    public static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine(
                "Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
    // Output:
    // In the finally block in TryCast, i = 123.

    // Catching the System.InvalidCastException exception triggers the finally block.

    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}

```

Para obter mais informações sobre `finally`, consulte [try-catch-finally](#).

C# também contém a [instrução using](#), que fornece uma funcionalidade semelhante para objetos [IDisposable](#) em uma sintaxe conveniente.

Especificação da linguagem C#

Para obter mais informações, confira a seção [A instrução try](#) da [Especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)

- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Instruções try, throw e catch \(C++\)](#)
- [Jogar](#)
- [try-catch](#)
- [Como gerar exceções explicitamente](#)

try-catch-finally (Referência de C#)

21/01/2022 • 2 minutes to read

Um uso comum de `catch` e `finally` juntos é obter e usar recursos em um bloco `try`, lidar com circunstâncias excepcionais em um bloco `catch` e liberar os recursos no bloco `finally`.

Para obter mais informações e exemplos sobre como lançar exceções, consulte [try-catch](#) e [Lançando exceções](#).

Para obter mais informações sobre o bloco `finally`, consulte [try-finally](#).

Exemplo

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }
        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

Especificação da linguagem C#

Para obter mais informações, confira a seção [A instrução try](#) da [Especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Instruções try, throw e catch \(C++\)](#)
- [Jogar](#)
- [Como gerar exceções explicitamente](#)
- [Instrução using](#)

Contexto verificado e não verificado (Referência de C#)

21/01/2022 • 2 minutes to read

Instruções C# podem ser executadas em contexto marcado ou desmarcado. Em um contexto marcado, o estouro aritmético gera uma exceção. Em um contexto não verificado, o estouro aritmético é ignorado, e o resultado é truncado descartando todos os bits de ordem superior que não se encaixam no tipo de destino.

- **verificado** Especificar o contexto verificado.
- **não verificado** Especificar o contexto não verificado.

As seguintes operações são afetadas pela verificação de estouro:

- Expressões que usam os seguintes operadores predefinidos em tipos integrais:

`++`, `--`, unário `-`, `+`, `-`, `*`, `/`

- Conversões numéricas explícitas entre tipos integrais ou de `float` ou `double` para um tipo integral.

Se nem nem for especificado, o contexto padrão para expressões não constantes (expressões avaliadas em tempo de executar) será definido pelo valor da opção do `checked` ou `unchecked` compilador

CheckForOverflowUnderflow. Por padrão, o valor dessa opção é removido e as operações aritméticas são executadas em um contexto não verificado.

Para expressões de constante (expressões que podem ser totalmente avaliadas no tempo de compilação), o contexto padrão sempre é verificado. A menos que uma expressão de constante seja explicitamente colocada em um contexto não verificado, estouros que ocorrem durante a avaliação do tempo de compilação da expressão causam erros de tempo de compilação.

Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Palavras-chave de instrução](#)

checked (Referência de C#)

21/01/2022 • 2 minutes to read

A palavra-chave `checked` é usada para habilitar explicitamente a verificação estouro para conversões e operações aritméticas de tipo integral.

Por padrão, uma expressão que contém somente valores constantes causa um erro do compilador se a expressão produzir um valor fora do intervalo do tipo de destino. Se a expressão contiver um ou mais valores não constantes, o compilador não detectará o estouro. Avaliar a expressão atribuída a `i2` no exemplo a seguir não causa um erro do compilador.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

Por padrão, essas expressões não constantes não são verificados quanto ao estouro em tempo de execução e não geram exceções de estouro. O exemplo anterior exibe -2,147,483,639 como a soma de dois números inteiros positivos.

A verificação de estouro pode ser habilitada por opções do compilador, configuração do ambiente ou uso da palavra-chave `checked`. Os exemplos a seguir demonstram como usar uma expressão `checked` ou um bloco `checked` para detectar o estouro produzido pela soma anterior no tempo de execução. Os dois exemplos geram uma exceção de estouro.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

A palavra-chave `unchecked` pode ser usada para impedir a verificação de estouro.

Exemplo

Este exemplo mostra como usar `checked` para habilitar a verificação de estouro em tempo de execução.

```

class OverFlowTest
{
    // Set maxValue to the maximum value for integers.
    static int maxValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT: " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT: " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
            CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
            UncheckedMethod());
    }
    /*
Output:
CHECKED and CAUGHT: System.OverflowException: Arithmetic operation resulted
in an overflow.
at ConsoleApplication1.OverFlowTest.CheckedMethod()

CHECKED output value is: 0
UNCHECKED output value is: -2147483639
*/
}

```

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#](#)
- [Contexto verificado e não verificado](#)
- [Desmarcada](#)

unchecked (Referência de C#)

21/01/2022 • 2 minutes to read

A palavra-chave `unchecked` é usada para suprimir a verificação estouro para conversões e operações aritméticas de tipo integral.

Em um contexto não verificado, se uma expressão produzir um valor que está fora do intervalo do tipo de destino, o estouro não será sinalizado. Por exemplo, como o cálculo no exemplo a seguir é realizado em uma expressão ou bloco `unchecked`, o fato de o resultado ser muito grande para um inteiro é ignorado e `int1` recebe a atribuição do valor -2.147.483.639.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Se o ambiente `unchecked` for removido, ocorrerá um erro de compilação. O estouro pode ser detectado em tempo de compilação porque todos os termos da expressão são constantes.

Expressões que contêm termos não constantes não são verificadas por padrão em tempo de compilação e tempo de execução. Consulte [checked](#) para obter informações sobre como habilitar um ambiente verificado.

Como a verificação de estouro leva tempo, o uso de código não verificado em situações em que não há nenhum risco de estouro pode melhorar o desempenho. No entanto, se o estouro for uma possibilidade, um ambiente verificado deverá ser usado.

Exemplo

Esse exemplo mostra como usar a palavra-chave `unchecked`.

```

class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}

```

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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Contexto verificado e não verificado](#)
- [check](#)

Instrução fixed (Referência de C#)

21/01/2022 • 3 minutes to read

A instrução `fixed` impede que o coletor de lixo faça a realocação de uma variável móvel. A instrução `fixed` é permitida somente em um contexto **não seguro**. Você também pode usar a palavra-chave `fixed` para criar **buffers de tamanho fixo**.

A instrução `fixed` define um ponteiro para uma variável gerenciada e "fixa" essa variável durante a execução da instrução. Os ponteiros móveis gerenciados são úteis apenas em um contexto `fixed`. Sem um contexto `fixed`, a coleta de lixo poderia realocar as variáveis de forma imprevisível. O compilador do C# só permite que você atribua um ponteiro a uma variável gerenciada em uma instrução `fixed`.

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

Você pode inicializar um ponteiro usando uma matriz, uma cadeia de caracteres, um buffer de tamanho fixo ou o endereço de uma variável. O exemplo a seguir ilustra o uso de endereços de variáveis, matrizes e sequências de caracteres:

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

Começando com o C# 7.3, a instrução `fixed` opera em tipos adicionais além de cadeias de caracteres, matrizes, buffers de tamanho fixo ou variáveis não gerenciadas. Qualquer tipo que implementa um método chamado `GetPinnableReference` pode ser anexado. O `GetPinnableReference` deve retornar uma variável `ref` para um tipo [não gerenciado](#). Os tipos .NET `System.Span<T>` e `System.ReadOnlySpan<T>` introduzidos no .NET Core 2.0 usam esse padrão e podem ser anexados. Isso é mostrado no exemplo a seguir:

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
        1,
        1, 1,
        1, 2, 1,
        1, 3, 3, 1,
        1, 4, 6, 4, 1,
        1, 5, 10, 10, 5, 1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

Se você estiver criando tipos que devem participar desse padrão, consulte [Span<T>.GetPinnableReference\(\)](#) para obter um exemplo da implementação do padrão.

Vários ponteiros podem ser inicializados em uma instrução quando eles são do mesmo tipo:

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

Para inicializar ponteiros de tipos diferentes, basta aninhar instruções `fixed`, conforme mostrado no exemplo a seguir.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

Depois que o código na instrução é executado, todas as variáveis fixadas são desafixadas e ficam sujeiras à coleta de lixo. Portanto, não aponte para essas variáveis fora da instrução `fixed`. As variáveis declaradas na instrução `fixed` estão no escopo dessa instrução, facilitando isto:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    ...
}
// ps and pd are no longer in scope here.
```

Os ponteiros inicializados em instruções `fixed` são variáveis somente leitura. Se você quiser modificar o valor do ponteiro, será necessário declarar uma segunda variável de ponteiro e modificá-la. A variável declarada na instrução `fixed` não pode ser modificada:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

É possível alocar memória na pilha, local que não está sujeito à coleta de lixo e, portanto, não precisa ser fixado. Para fazer isso, use uma `stackalloc` expressão.

Especificação da linguagem C#

Para saber mais, confira a seção [A instrução corrigida](#) na [Especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Inseguro](#)
- [Tipos de ponteiro](#)
- [Buffers de tamanho fixo](#)

Parâmetros de método (Referência de C#)

21/01/2022 • 2 minutes to read

Os parâmetros declarados para um método sem `in`, `ref` nem `out` são passados para o método chamado pelo valor. Esse valor pode ser alterado no método, mas o valor alterado não será mantido quando o controle passá-lo de volta para o procedimento de chamada. É possível alterar esse comportamento usando uma palavra-chave de parâmetro de método.

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#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)

params (Referência de C#)

21/01/2022 • 2 minutes to read

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 não for uma matriz unidimensional, ocorrerá o erro do `params` compilador [CS0225](#).

Ao chamar um método com um `params` parâmetro, 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`.

```

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#)

21/01/2022 • 6 minutes to read

A `in` palavra-chave 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.

```
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.

NOTE

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.

O modificador de parâmetro `in` está disponível no C# 7.2 e posteriores. As versões anteriores geravam o erro de compilador `CS8107` ("o recurso 'referências somente leitura' não está disponível no C# 7.0. Use a linguagem na versão 7.2 ou posteriores"). Confira como configurar a versão de linguagem do compilador em [Selecionar a versão da linguagem C#](#).

Embora os modificadores de parâmetro , e sejam considerados parte de uma assinatura, os membros declarados em um único tipo não podem diferir somente na assinatura `in` `out` por e `ref` `in` `ref` `out` . Portanto, os métodos não poderão ser sobre carregados 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:

```
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 sobrecarga com base na presença de `in` é permitida:

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Regras de resolução de sobrecarga

Você pode entender as regras de resolução de sobrecarga 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 declararam 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:

```

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:

```

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.

NOTE

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 `in` modificador, 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).

Você pode saber mais sobre o `in` modificador, como ele difere de e no artigo `ref` em Escrever código eficiente `out` [seguro](#).

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#)

21/01/2022 • 9 minutes to read

A `ref` palavra-chave indica que um valor é passado por referência. Ela é usada em quatro contextos diferentes:

- 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 um corpo de membro, para indicar que um valor retornado por referência é armazenado localmente como uma referência que o chamador pretende modificar. Ou para indicar que uma variável local acessa outro valor por referência. Para obter mais informações, veja [Locais de referência](#).
- Em uma `struct` declaração, para declarar um `ref struct` ou um `readonly ref struct`. Para obter mais informações, consulte a `ref` seção `struct` do artigo [Tipos 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. O método chamado pode substituir o objeto ao qual o parâmetro `ref` se refere. Nesse caso, a variável local do chamador ou o elemento array refere-se ao novo objeto quando o método retorna.

NOTE

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 `ref` omitir ao fazer uma chamada COM.)

```
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 difere 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.

```
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 sobreescritos quando um método tem um parâmetro `ref`, ou o outro tem um parâmetro que é passado por valor, conforme mostrado no `ref` `in` exemplo a `out` seguir.

```
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.

Propriedades não são variáveis. Eles são métodos e não podem ser passados para `ref` parâmetros.

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 `out` palavra-chave não pode ser usada no primeiro argumento de um método de extensão.
- A palavra-chave 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 para ser `ref` um struct.
- A `in` palavra-chave não pode ser usada, a menos que o primeiro argumento seja um struct. A `in` palavra-chave não pode ser usada em nenhum tipo genérico, mesmo quando restrita 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`.

```

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 é refletida no estado do objeto no método de chamada.

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.

```
public ref decimal GetCurrentPrice()
```

- Entre o token `return` e a variável retornada em uma instrução `return` no método. Por exemplo:

```
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 [ref local](#).

Aqui está um exemplo de retorno de ref mais completo, mostrando a assinatura do método e o corpo do método.

```
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 pode declarar o valor retornado como para retornar o valor por referência e impor que o código de chamada não [ref readonly](#) pode modificar o valor retornado. O método de chamada pode evitar copiar o valor retornado, armazenar o valor em uma variável [ref readonly](#) local.

Para ver um exemplo, consulte [A ref returns e ref locals example](#).

Ref locals

Uma variável de ref local é usada para fazer referência a valores retornados usando [return ref](#). Uma variável local ref não pode ser inicializada para um valor retornado que não seja ref. Em outras palavras, o lado direito da inicialização deve ser uma referência. Todas as modificações ao valor do ref local são refletidas no estado do objeto cujo método retornou o valor por referência.

Você define um ref local usando a [ref](#) palavra-chave em dois locais:

- Antes da declaração de variável.
- Imediatamente antes da chamada para o método que retorna o valor por referência.

Por exemplo, a instrução a seguir define um valor de ref local que é retornado por um método chamado [GetEstimatedValue](#):

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

Você pode acessar um valor por referência da mesma maneira. Em alguns casos, acessar um valor por referência aumenta o desempenho, evitando uma operação de cópia potencialmente dispendiosa. Por exemplo, a instrução a seguir mostra como definir uma variável local ref usada para referenciar um valor.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

Em ambos os exemplos, a palavra-chave deve ser usada em ambos os locais ou o compilador gera o erro CS8172, "Não é possível inicializar uma variável por referência com [ref](#) um valor".

A partir do C# 7.3, a variável de iteração da instrução pode ser uma variável local ref [foreach](#) local ou ref readonly. Para saber mais, confira o artigo [Instrução foreach](#).

Também começando com o C# 7.3, você pode reatribuir uma variável local ref local ou ref readonly com o operador [de atribuição ref](#).

Locais somente leitura de referência

Um ref readonly local é usado para se referir aos valores retornados por um método ou propriedade que tem `ref readonly` em sua assinatura e usa `return ref`. Uma variável combina as propriedades de uma variável local com uma variável: é um alias para o armazenamento ao que está atribuída e não `ref readonly` `ref` pode ser `readonly` modificada.

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`.

```
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.

```
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

- [Escrever código seguro e eficiente](#)
- [Ref returns e ref locals](#)
- [Expressão condicional ref](#)
- [Passando parâmetros](#)
- [Parâmetros de método](#)
- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)

Modificador de parâmetro out (Referência de C#)

21/01/2022 • 4 minutes to read

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:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);      // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

NOTE

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 sobre carregados 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:

```
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:

```
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 `out` palavra-chave não pode ser usada no primeiro argumento de um método de extensão.
- A `ref` palavra-chave 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 `in` palavra-chave não pode ser usada, a menos que o primeiro argumento seja um struct. A `in` palavra-chave não pode ser usada em nenhum tipo genérico, mesmo quando restrita a um struct.

Declarando parâmetros `out`

Declarar um método com argumentos `out` é uma solução clássica para retornar vários valores. A partir do C# 7.0, considere as [tuplas de valor](#) para cenários semelhantes. O exemplo a seguir usa `out` para retornar três variáveis com uma única chamada de método. O terceiro argumento é atribuído a NULL. Isso permite que os métodos retornem valores opcionalmente.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, 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`

No C# 6 e em versões anteriores, você precisa declarar uma variável em uma instrução separada antes de passá-lo como um argumento `out`. 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.

```
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
```

Começando com o C#7.0, você pode declarar a variável `out` na lista de argumentos da chamada de método em vez de declará-la em uma declaração de variável separada. Isso produz um código mais compacto e legível, além de impedir que você atribua acidentalmente 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](#).

```
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.

```
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

- [Referência do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Parâmetros de método](#)

namespace

21/01/2022 • 2 minutes to read

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.

```
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` no escopo do arquivo permitem declarar que todos os tipos em um arquivo estão em um único namespace. As declarações de namespace no escopo do arquivo estão disponíveis com o C# 10. O exemplo a seguir é semelhante ao exemplo anterior, mas usa uma declaração de namespace com escopo de arquivo:

```
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 no escopo do arquivo não podem incluir declarações de namespace adicionais. Não é possível declarar um namespace aninhado ou um segundo namespace com escopo de arquivo:

```

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 no escopo do 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` :

```

namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}

```

O exemplo a seguir mostra como chamar um método estático em um namespace aninhado.

```
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 no escopo do arquivo, consulte a [especificação de recurso](#).

Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [Usando](#)
- [usando estático](#)
- [Qualificador de alias de namespace](#) `::`
- [Namespaces](#)

using (Referência de C#)

21/01/2022 • 2 minutes to read

A palavra-chave `using` tem dois usos principais:

- A instrução `using` define um escopo no final do qual um objeto será descartado.
- A diretiva `using` cria um alias para um namespace ou importa tipos definidos em outros namespaces.

Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Namespaces](#)
- [extern](#)

usando diretiva

21/01/2022 • 10 minutes to read

A diretiva permite que você use tipos definidos em um namespace sem especificar o namespace totalmente qualificado desse tipo. Em sua forma básica, a diretiva importa todos os tipos de um único namespace, conforme mostrado no exemplo a seguir:

```
using System.Text;
```

Você pode aplicar dois modificadores a uma diretiva :

- O `global` modificador tem o mesmo efeito que adicionar a mesma diretiva a cada arquivo de origem em seu projeto. Esse modificador foi introduzido no C# 10.
- O modificador importa os membros e tipos aninhados de um único tipo em vez de importar todos os tipos em um `static` namespace. Esse modificador foi introduzido no C# 6.0.

Você pode combinar os dois modificadores para importar os membros estáticos de um tipo em todos os arquivos de origem em seu projeto.

Você também pode criar um alias para um namespace ou um tipo com uma *diretiva using alias*.

```
using Project = PC.MyCompany.Project;
```

Você pode usar o `global` modificador em uma diretiva *using alias*.

NOTE

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*, consulte [using Statement](#).

O escopo de uma diretiva `using` sem `global` o modificador é o arquivo no qual ele aparece.

A diretiva `using` pode aparecer:

- No início de um arquivo de código-fonte, antes de qualquer declaração de namespace ou tipo.
- Em qualquer namespace, mas antes de todos os namespaces ou tipos declarados nesse namespace, a menos que o modificador seja usado, nesse caso, a diretiva deverá aparecer antes de todos os namespaces e declarações `global` 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 não lhe dá acesso a nenhum namespace aninhado 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 a uma diretiva significa que o uso é aplicado a todos `global` `using` os arquivos na compilação (normalmente um projeto). A `global using` diretiva foi adicionada no C# 10. Sua sintaxe é:

```
global using <fully-qualified-namespace>;
```

em que *namespace totalmente qualificado* é o nome totalmente qualificado do namespace cujos tipos podem ser referenciados sem especificar o namespace.

Uma *diretiva using global* pode aparecer no início de qualquer arquivo de código-fonte. Todas `global using` as diretivas em um único arquivo devem aparecer antes de:

- Todas `using` as diretivas sem o `global` modificador.
- Todas as declarações de namespace e tipo no arquivo.

Você pode adicionar `global using` diretivas a qualquer arquivo de origem. Normalmente, você vai querer mantê-los em um único local. A ordem das `global using` diretivas não importa, seja em um único arquivo ou entre arquivos.

O `global` modificador pode ser combinado com o `static` modificador. O `global` modificador pode ser aplicado a uma diretiva *de alias using*. Em ambos os casos, o escopo da diretiva é 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 em seu projeto:

```
global using static System.Math;
```

Você também pode incluir globalmente um namespace adicionando um `<using>` item ao arquivo de projeto, por exemplo, `<using Include="My.Awesome.Namespace" />`. Para obter mais informações, consulte [<Using> O item](#).

IMPORTANT

Os modelos C# para .NET 6 usam *instruções de nível superior*. Seu aplicativo pode não corresponder ao código neste artigo, se você já tiver atualizado para as visualizações do .NET 6. Para obter mais informações, consulte o artigo sobre [novos modelos C# gerar 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 `using static` diretiva nomeia um tipo cujos membros estáticos e tipos aninhados você pode acessar sem especificar um nome de tipo. A diretiva `using static` foi introduzida no C# 6. Sua sintaxe é:

```
using static <fully-qualified-type-name>;
```

O é o nome do tipo cujos membros estáticos e tipos aninhados podem ser `<fully-qualified-type-name>` referenciados sem especificar um nome de tipo. Se você não fornecer um nome de tipo totalmente qualificado

(o nome completo do namespace junto com o nome do tipo), o C# gerará o erro do compilador CS0246:"O nome do tipo ou namespace 'type/namespace' não pôde ser encontrado (você não encontrou uma diretiva using ou uma referência de assembly?)".

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:

```
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 definição a seguir de `Circle` uma classe faz referência a muitos membros da classe `Math`.

```
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); }
    }
}
```

Eliminando a necessidade de referenciar explicitamente a classe sempre que um membro é `Math` referenciado, a `using static` diretiva produz código mais limpo:

```
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. Membros herdados não são importados. Você pode importar de qualquer tipo nomeado com uma `using static` diretiva , incluindo Visual Basic módulos. 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 para o escopo para 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 sobreposição 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.

```

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 possibilitaria chamar o `TryParse(String, Double)` método sem especificar um nome de tipo. No entanto, usar sem um nome de tipo cria um código menos acessível, pois se torna necessário verificar as diretivas para determinar qual método do tipo numérico `TryParse` `using static` é `TryParse` chamado.

`using static` também se aplica a `enum` tipos. Ao adicionar `using static` com a enum, o tipo não é mais necessário para usar os membros de enum.

```
using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
{
    public static void Main()
    {
        Color color = Green;
    }
}
```

usando alias

Crie uma diretiva de alias `using` para tornar mais fácil a qualificação de um identificador para um namespace ou tipo. Em qualquer diretiva, o namespace ou tipo totalmente qualificado deve `using` ser usado independentemente das `using` diretivas que vêm antes dele. Nenhum alias `using` pode ser usado na declaração de uma diretiva `using`. Por exemplo, o exemplo a seguir gera um erro do compilador:

```
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:

```
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 alias de uso 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:

```

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 Visual Basic `My` dados

O namespace (no Visual Basic) fornece acesso fácil e intuitivo a várias classes .NET, permitindo que você escreva código que interage com o computador, o aplicativo, as configurações, os recursos e assim

[Microsoft.VisualBasic.MyServices](#) `My` 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` em seu 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:

```
// 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 de uso global*, consulte a especificação de recurso [de usos globais – C# 10](#).

Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Namespaces](#)
- [Instrução using](#)

Instrução using (Referência de C#)

21/01/2022 • 4 minutes to read

Fornece uma sintaxe conveniente que garante o uso correto de objetos [IDisposable](#). A partir do C# 8,0, a `using` instrução garante o uso correto de [IAsyncDisposable](#) objetos.

Exemplo

O exemplo a seguir mostra como usar a instrução `using`.

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line."

using (var reader = new StringReader(manyLines))
{
    string? item;
    do
    {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
```

A partir do C# 8,0, você pode usar a seguinte sintaxe alternativa para a `using` instrução que não exige chaves:

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line."

using var reader = new StringReader(manyLines);
string? item;
do
{
    item = reader.ReadLine();
    Console.WriteLine(item);
} while (item != null);
```

Comentários

[File](#) e [Font](#) são exemplos de tipos gerenciados que acessam recursos não gerenciados (nesse caso, identificadores de arquivo de caso e contextos de dispositivo). Há muitos outros tipos de recursos não gerenciados e tipos de biblioteca de classes que os encapsula. Todos esses tipos devem implementar a [IDisposable](#) interface ou a [IAsyncDisposable](#) interface.

Quando o tempo de vida de um objeto `IDisposable` é limitado a um único método, você deve declará-lo e instanciá-lo na instrução `using`. A instrução `using` chama o método `Dispose` no objeto da forma correta e (quando você o usa como mostrado anteriormente) ele também faz com que o objeto em si saia do escopo assim que `Dispose` é chamado. Dentro do `using` bloco, o objeto é somente leitura e não pode ser modificado ou reatribuído. Se o objeto for implementado [IAsyncDisposable](#) em vez de [IDisposable](#), a `using` instrução

chamará o [DisposeAsync](#) e [awaits](#) o retornado [ValueTask](#). Para obter mais informações sobre [IAsyncDisposable](#), consulte [implementar um método DisposeAsync](#).

A `using` instrução garante que `Dispose` (ou `DisposeAsync`) seja chamado mesmo se ocorrer uma exceção dentro do `using` bloco. Você pode obter o mesmo resultado colocando o objeto dentro de um `try` bloco e, em seguida `Dispose`, chamando (ou `DisposeAsync`) em um `finally` bloco; na verdade, é assim que a `using` instrução é convertida pelo compilador. O exemplo de código anterior se expande para o seguinte código em tempo de compilação (observe as chaves extras para criar o escopo limitado para o objeto):

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line."

{
    var reader = new StringReader(manyLines);
    try
    {
        string? item;
        do
        {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while (item != null);
    }
    finally
    {
        reader?.Dispose();
    }
}
```

A sintaxe mais recente da `using` instrução é convertida em código semelhante. O `try` bloco é aberto onde a variável é declarada. O `finally` bloco é adicionado ao fechamento do bloco delimitador, normalmente no final de um método.

Para obter mais informações sobre a `try` - `finally` instrução, consulte o artigo [Experimente-finally](#).

Várias instâncias de um tipo podem ser declaradas em uma única `using` instrução, conforme mostrado no exemplo a seguir. Observe que você não pode usar variáveis de tipo implícito (`var`) ao declarar várias variáveis em uma única instrução:

```

string numbers = @"One
Two
Three
Four.";
string letters = @"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
       right = new StringReader(letters))
{
    string? item;
    do
    {
        item = left.ReadLine();
        Console.WriteLine(item);
        Console.WriteLine("    ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}

```

Você pode combinar várias declarações do mesmo tipo usando a nova sintaxe introduzida com C# 8 também, conforme mostrado no exemplo a seguir:

```

string numbers = @"One
Two
Three
Four.";
string letters = @"A
B
C
D.";

using StringReader left = new StringReader(numbers),
      right = new StringReader(letters);
string? item;
do
{
    item = left.ReadLine();
    Console.WriteLine(item);
    Console.WriteLine("    ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while (item != null);

```

Você pode instanciar o objeto de recurso e, em seguida, passar a variável para a `using` instrução, mas essa não é uma prática recomendada. Nesse caso, após o controle sair do bloco `using`, o objeto permanecerá no escopo, mas provavelmente não terá acesso a seus recursos não gerenciados. Em outras palavras, ele não é mais totalmente inicializado. Se você tentar usar o objeto fora do bloco `using`, corre o risco de causar o lançamento de uma exceção. Por esse motivo, é melhor instanciar o objeto na `using` instrução e limitar seu escopo ao `using` bloco.

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.;

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do
    {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
// reader is in scope here, but has been disposed
```

Para obter mais informações sobre como descartar objetos `IDisposable`, veja [Usando objetos que implementam `IDisposable`](#).

Especificação da linguagem C#

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

Confira também

- [Referência do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Diretiva de uso](#)
- [Coleta de lixo](#)
- [Usando objetos que implementam `IDisposable`](#)
- [Interface `IDisposable`](#)
- [instrução using em C# 8,0](#)

extern alias (Referência de C#)

21/01/2022 • 2 minutes to read

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.

NOTE

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 Visual Studio, os aliases 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 aliases de global para `GridV1` e `GridV2`, respectivamente.

Use esses aliases da mesma maneira acima

```
extern alias GridV1;  
  
extern alias GridV2;
```

Agora você pode criar um alias para um namespace ou um tipo *usando a diretiva alias*. Para obter mais informações, consulte [usando a diretiva](#).

```
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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Operador::](#)
- [Referências \(opções do compilador C#\)](#)

Restrição new (Referência em C#)

21/01/2022 • 2 minutes to read

A restrição `new` especifica que um argumento de tipo em uma declaração de classe 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:

```
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:

```
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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Genéricos](#)

where (restrição de tipo genérico) (Referência de C#)

21/01/2022 • 5 minutes to read

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. As restrições podem especificar interfaces, classes base ou exigir que um tipo genérico seja uma referência, um valor ou um tipo não gerenciado. Eles declaram recursos que o argumento de tipo deve ter.

Por exemplo, você pode declarar uma classe genérica, `AGenericClass`, de modo que o parâmetro de tipo `T` implementa a interface `IComparable<T>`:

```
public class AGenericClass<T> where T : IComparable<T> { }
```

NOTE

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 esse 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`. Antes do C# 7.3, `Enum`, `Delegate`, e `MulticastDelegate` também não foram permitidos como restrições de classe base. O exemplo a seguir mostra os tipos que agora podem ser especificados como classe base:

```
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 no C# 8,0 e posterior, a nulidade do tipo de classe base é imposta. Se a classe base for não 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 for não 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`:

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

Em um contexto anulável no C# 8,0 e posterior, a `class` restrição requer que um tipo seja um tipo de referência

não anulável. Para permitir tipos de referência anuláveis, use a `class?` restrição, que permite tipos de referência anuláveis e não anuláveis.

A `where` cláusula pode incluir a `notnull` restrição. A `notnull` restrição limita o parâmetro de tipo para tipos não anuláveis. O tipo pode ser um tipo de `valor` ou um tipo de referência não anulável. A `notnull` restrição está disponível a partir do C# 8,0 para o código compilado em um `nullable enable` contexto. Ao contrário de outras restrições, se um argumento de tipo violar a `notnull` restrição, o compilador gerará um aviso em vez de um erro. Os avisos são gerados apenas em um `nullable enable` contexto.

A adição de tipos de referência anuláveis introduz uma possível ambiguidade no significado de `T?` métodos genéricos. Se `T` é 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 `default` restrição resolve essa ambiguidade. Você o adicionará quando uma classe base ou interface declarar duas sobrecargas de um método, uma que especifica a `struct` restrição e outra que não tenha a `struct` `class` restrição ou aplicada:

```
public abstract class B
{
    public void M<T>(T? item) where T : struct { }
    public abstract void M<T>(T? item);

}
```

Você usa a `default` restrição para especificar que a classe derivada substitui o método sem a restrição em sua classe derivada ou implementação de interface explícita. Ele só é válido em métodos que substituem métodos base ou implementações de interface explícitas:

```
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 { }
}
```

IMPORTANT

Declarações genéricas que incluem a `notnull` restrição podem ser usadas em um contexto alheio anulável, mas o compilador não impõe a restrição.

```
#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`:

```
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 \(\)](#) permite que o compilador saiba que qualquer argumento de tipo fornecido deve ter um construtor sem parâmetros acessível. Por exemplo:

```
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:

```
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:

```
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:

```
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 do C#
- Guia de programação C#
- Introdução aos genéricos
- nova restrição
- Restrições a parâmetros de tipo

base (Referência de C#)

21/01/2022 • 2 minutes to read

A palavra-chave `base` é usada para acessar membros de classe base de dentro de uma classe derivada:

- 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.

Um acesso de classe base é permitido somente em um construtor, em um método de instância ou em um acessador de propriedade de instância.

É um erro usar a palavra-chave `base` de dentro de um método estático.

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, as classes base `Person` e a classe derivada `Employee` têm um método chamado `Getinfo`.

Usando a palavra-chave `base`, é possível chamar o método `Getinfo` na classe base, de dentro da classe derivada.

```
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.

```

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#)

21/01/2022 • 2 minutes to read

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.

NOTE

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:

```
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:

```
CalcTax(this);
```

- Para declarar indexadores, por exemplo:

```
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.

```

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: mp
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

- [Referência do C#](#)
- [Guia de programação C#](#)

- Palavras-chave do C#

- base

- Métodos

null (Referência de C#)

21/01/2022 • 2 minutes to read

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. Tipos de valor comuns não podem ser nulos, exceto para [tipos de valor anulados](#).

O exemplo a seguir demonstra alguns comportamentos da `null` palavra-chave :

```

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 C#)

21/01/2022 • 2 minutes to read

A `bool` palavra-chave Type é um alias para o [System.Boolean](#) tipo de estrutura .NET que representa um valor booleano, que pode ser `true` ou `false`.

Para executar operações lógicas com valores do `bool` tipo, use operadores [lógicos booleanos](#). O `bool` tipo é o tipo de resultado dos operadores de [comparação](#) e de [igualdade](#). Uma `bool` expressão pode ser uma expressão condicional de controle nas instruções `if`, `do`, `while` e no operador `?:` [condicional](#).

O valor padrão do `bool` tipo é `false`.

Literais

Você pode usar os `true` `false` literais e para inicializar uma `bool` variável ou para passar um `bool` valor:

```
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 `bool?` tipo anulável, se você precisar dar suporte à lógica de três valores, por exemplo, quando 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 de valor anulável, consulte [tipos de valor anulável](#).

Conversões

O C# fornece apenas duas conversões que envolvem o `bool` tipo. Essas são uma conversão implícita para o tipo anulável correspondente `bool?` e uma conversão explícita do `bool?` tipo. No entanto, o .NET fornece métodos adicionais que você pode usar para converter de ou para o `bool` tipo. Para obter mais informações, consulte a seção [Convertendo de valores booleanos](#) da [System.Boolean](#) página de referência da API.

Especificação da linguagem C#

Para obter mais informações, consulte a seção [tipo bool](#) da [especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [operadores true e false](#)

default (referência de C#)

21/01/2022 • 2 minutes to read

Você pode usar a `default` palavra-chave nos seguintes contextos:

- Para especificar o caso padrão na `switch` instrução.
- Como o `operador padrão ou literal` para produzir o valor padrão de um tipo.
- Como a `default` restrição de tipo em uma substituição de método genérico ou implementação explícita de interface.

Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)

Palavras-chave contextuais (referência C#)

21/01/2022 • 2 minutes to read

Uma palavra-chave contextual é usada para fornecer um significado específico no código, mas não é uma palavra reservada no C#. As seguintes palavras-chave contextuais são apresentadas nesta seção:

PALAVRA-CHAVE	DESCRIÇÃO
adicionar	Define um acessador de evento personalizado que é invocado quando o código de cliente assina o evento.
and	Cria um padrão que corresponde quando os dois padrões aninhados correspondem.
Async	Indica que o método modificado, expressão lambda ou método anônimo é assíncrono.
await	Suspende um método assíncrono até que uma tarefa esperada seja concluída.
dinâmico	Define um tipo de referência, que habilita operações nas quais ele ocorre, para ignorar a verificação de tipo em tempo de compilação.
get	Define um método do acessador de uma propriedade ou um indexador.
geral	Alias do namespace global que, de outra forma, não tem nome.
init	Define um método do acessador de uma propriedade ou um indexador.
nint	Define um tipo de dados Integer de tamanho nativo.
not	Cria um padrão que corresponde quando o padrão negado não corresponde.
nuint	Define um tipo de dados inteiro sem sinal de tamanho nativo.
or	Cria um padrão que corresponde quando qualquer um dos padrões aninhados corresponde.
parcial	Define classes parciais, structs e interfaces ao longo da mesma unidade de compilação.
gravável	Usado para definir um tipo de registro.
remove	Define um acessador de evento personalizado que é invocado quando o código de cliente cancela a assinatura do evento.

PALAVRA-CHAVE	DESCRIÇÃO
<code>set</code>	Define um método do acessador de uma propriedade ou um indexador.
<code>value</code>	Usado para definir acessadores e para adicionar ou remover manipuladores de eventos.
<code>var</code>	Permite que o tipo de uma variável declarada no escopo do método seja determinado pelo compilador.
<code>when</code>	Especifica uma condição de filtro para um bloco <code>catch</code> ou o rótulo <code>case</code> de uma instrução <code>switch</code> .
<code>where</code>	Adiciona restrições a uma declaração genérica. (Consulte também where).
<code>proporcionar</code>	Usada em um bloco iterador para retornar um valor para o objeto enumerador ou para sinalizar o final da iteração.

Todas as palavras-chave de consulta introduzidas no C# 3.0 também são contextuais. Para obter mais informações, consulte [Palavras-chave de consulta \(LINQ\)](#).

Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [Operadores e expressões C#](#)

add (Referência de C#)

21/01/2022 • 2 minutes to read

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 ver o exemplo completo, [consulte Como implementar eventos de interface](#).

```
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#)

21/01/2022 • 2 minutes to read

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.

```
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. Começando com o C# 7.0, 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.

```
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.

```
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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Propriedades](#)

init (referência C#)

21/01/2022 • 2 minutes to read

No C# 9 e posterior, a `init` palavra-chave define um método de *acessador* em uma propriedade ou um indexador. Um setter somente `init` atribui um valor à propriedade ou ao elemento do indexador somente durante a construção do objeto. Para obter mais informações e exemplos, consulte [Propriedades, Propriedades autoimplementadas e Indexadores](#).

O exemplo a seguir define um `get` e um `init` acessador para uma propriedade chamada `Seconds`. Ela usa um campo particular chamado `_seconds` para dar suporte ao valor da propriedade.

```
class InitExample
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        init { _seconds = value; }
    }
}
```

Geralmente, o acessador `init` consiste em uma única instrução que retorna um valor, como no exemplo anterior. Você pode implementar o `init` acessador como um membro expression-aptô para. O exemplo a seguir implementa os acessadores `get` e `init` como membros com corpo de expressão.

```
class InitExampleExpressionBodied
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        init => _seconds = value;
    }
}
```

Para casos simples em que os acessadores `get` e `init` 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.

```
class InitExampleAutoProperty
{
    public double Hours { 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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)
- [Propriedades](#)

tipo parcial (Referência em C#)

21/01/2022 • 2 minutes to read

Definições de tipo parcial permitem que a definição de uma classe, struct, interface ou registro seja dividida em vários arquivos.

Em *File1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

Em *File2.cs*, a declaração:

```
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)

21/01/2022 • 2 minutes to read

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 **parcial**.
- As assinaturas em ambas as partes do tipo parcial devem ser correspondentes.

A `partial` palavra-chave não é permitida em construtores, finalizadores, operadores sobreescritos, declarações de propriedade ou declarações de evento.

Um método parcial não é necessário para ter uma implementação nos seguintes casos:

- Ele não tem nenhum modificador de acessibilidade (incluindo o padrão **privado**).
- Ele retorna **void**.
- Ele não tem parâmetros **de** saída.
- Ele não tem nenhum dos modificadores virtuais **aseguir**, **substituir**, **lacrado**, **novo** ou **extern**.

Qualquer método que não esteja em conformidade com todas essas restrições (por exemplo, `public virtual partial void` método), deve fornecer uma implementação.

O exemplo a seguir mostra um método parcial definido em duas partes de uma classe parcial:

```
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, um regex pode ser definido usando o seguinte padrão:

```
[RegexGenerated("(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#)

21/01/2022 • 2 minutes to read

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 ver o exemplo completo, [consulte Como implementar eventos de interface](#).

```
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#)

21/01/2022 • 2 minutes to read

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.

```
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. Começando com o C# 7.0, 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.

```
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.

```
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#)

21/01/2022 • 2 minutes to read

Use a `when` palavra-chave contextual para especificar uma condição de filtro nos seguintes contextos:

- Na instrução `catch` de um bloco `try/catch` ou `try/catch/finally`.
- Como um `case guard` na `switch` instrução.
- Como uma `proteção de caso` na `switch` expressão.

when em uma instrução catch

Começando com o C# 6, `when` pode ser usado em uma instrução `catch` para especificar uma condição que deve ser verdadeira para o manipulador para uma exceção específica a ser executada. Sua sintaxe é:

```
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.

```
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#)

21/01/2022 • 2 minutes to read

A palavra-chave contextual `value` é usada no `set` acessador em declarações de [Propriedade](#) e [indexador](#). 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.

```
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, consulte os artigos sobre [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 do C#](#)
- [Guia de programação C#](#)
- [Palavras-chave do C#](#)

yield (Referência de C#)

21/01/2022 • 5 minutes to read

Ao usar a palavra-chave contextual `yield` em uma instrução, você indica que o método, o operador ou o acessador `get` em que ela é exibida é um iterador. Usar `yield` para definir um iterador elimina a necessidade de uma classe adicional explícita (a classe que mantém o estado de uma enumeração, consulte `IEnumerator<T>` para obter um exemplo) ao implementar o padrão `IEnumerable` e `IEnumerator` para um tipo de coleção personalizado.

O exemplo a seguir mostra as duas formas de instrução `yield`.

```
yield return <expression>;  
yield break;
```

Comentários

Você usa uma instrução `yield return` para retornar cada elemento individualmente.

A sequência retornada de um método iterador pode ser consumida usando uma instrução `foreach` ou uma consulta LINQ. Cada iteração do loop `foreach` chama o método iterador. Quando uma instrução `yield return` é atingida no método iterador, `expression` é retornado e o local atual no código é retido. A execução será reiniciada desse local na próxima vez que a função iteradora for chamada.

Quando o iterador retorna um `System.Collections.Generic.IAsyncEnumerable<T>`, essa sequência pode ser consumida de forma assíncrona usando uma instrução `Await foreach`. A iteração do loop é análoga à `foreach` instrução. A diferença é que cada iteração pode ser suspensa para uma operação assíncrona antes de retornar a expressão para o próximo elemento.

Você pode usar uma instrução `yield break` para terminar a iteração.

Para obter mais informações sobre iteradores, consulte [Iteradores](#).

Métodos de iterador e acessadores get

A declaração de um iterador deve atender aos seguintes requisitos:

- O tipo de retorno deve ser um dos seguintes tipos:
 - `IAsyncEnumerable<T>`
 - `IEnumerable<T>`
 - `IEnumerable`
 - `IEnumerator<T>`
 - `IEnumerator`
- A declaração não pode ter nenhum parâmetro `in`, `ref` ou `out`.

O tipo `yield` de um iterador que retorna `IEnumerable` ou `IEnumerator` é `object`. Se o iterador retornar `IEnumerable<T>` ou `IEnumerator<T>`, deverá haver uma conversão implícita do tipo da expressão na `yield return` instrução para o parâmetro de tipo genérico.

Você não pode incluir uma instrução `yield return` ou `yield break` em:

- Expressões lambda e métodos anônimos.

- Métodos que contêm blocos inseguros. Para obter mais informações, consulte [unsafe](#).

Tratamento de exceções

Uma instrução `yield return` não pode estar localizada em um bloco try-catch. Uma instrução `yield return` pode estar localizada no bloco try de uma instrução try-finally.

Uma instrução `yield break` pode estar localizada em um bloco try ou em um bloco catch, mas não em um bloco finally.

Se o `foreach` `await foreach` corpo ou (fora do método iterador) lançar uma exceção, um `finally` bloco no método iterador será executado.

Implementação técnica

O código a seguir retorna uma `IEnumerable<string>` de um método iterador e itera através de seus elementos.

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
    ...
}
```

A chamada a `MyIteratorMethod` não executa o corpo do método. Em vez disso, a chamada retorna `IEnumerable<string>` na variável `elements`.

Em uma iteração do loop `foreach`, o método `MoveNext` é chamado para `elements`. Essa chamada executará o corpo de `MyIteratorMethod` até que a próxima instrução `yield return` seja atingida. A expressão retornada pela instrução `yield return` determina não apenas o valor da variável `element` para o consumo do corpo do loop, mas também a propriedade `Current` de `elements`, que é uma `IEnumerable<string>`.

Em cada iteração subsequente do loop `foreach`, a execução do corpo do iterador continuará de onde parou, parando novamente quando atingir uma instrução `yield return`. O loop `foreach` é concluído quando o fim do método iterador ou uma instrução `yield break` é atingida.

O código a seguir retorna uma `IAsyncEnumerable<string>` de um método iterador e itera através de seus elementos.

```
IAsyncEnumerable<string> elements = MyAsyncIteratorMethod();
await foreach (string element in elements)
{
    // ...
}
```

Em uma iteração do loop `await foreach`, o método `IAsyncEnumerator<T>.MoveNextAsync` é chamado para `elements`. O `System.Threading.Tasks.ValueTask<TResult>` retorno `MoveNext` é concluído quando o próximo `yield return` é atingido.

Em cada iteração subsequente do loop `await foreach`, a execução do corpo do iterador continuará de onde parou, parando novamente quando atingir uma instrução `yield return`. O loop `await foreach` é concluído quando o fim do método iterador ou uma instrução `yield break` é atingida.

Exemplos

O exemplo a seguir contém uma instrução `yield return` dentro de um loop `for`. Cada iteração do corpo da

instrução `foreach` no método `Main` cria uma chamada à função iteradora `Power`. Cada chamada à função iteradora prossegue para a próxima execução da instrução `yield return` que ocorre durante a próxima iteração do loop `for`.

O tipo de retorno do método iterador é [IEnumerable](#) que é um tipo de interface de iterador. Quando o método iterador é chamado, ele retorna um objeto enumerável que contém as potências de um número.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

O exemplo a seguir demonstra um acessador `get` que é um iterador. No exemplo, cada instrução `yield return` retorna uma instância de uma classe definida pelo usuário.

```

public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { 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 do C#](#)
- [Guia de programação C#](#)
- [foreach, in](#)
- [Iteradores](#)

Palavras-chave de consulta (Referência de C#)

21/01/2022 • 2 minutes to read

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>&&</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#)

21/01/2022 • 6 minutes to read

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`.

```
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 uma 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.

```

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
Beebe Score: 91
*/

```

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.

```
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#)

21/01/2022 • 3 minutes to read

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.

```
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

Em uma única cláusula, você pode especificar quantos predicados for necessário usando `where` os operadores `&&` e `||`. No exemplo a seguir, a consulta especifica dois predicados para selecionar apenas os números pares que são menores que cinco.

```

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.

```

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. Esse é um benefício da forte verificação de tipo 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#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)

Cláusula select (Referência de C#)

21/01/2022 • 6 minutes to read

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.

```
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 `select` cláusula e o tipo da *variável de consulta* (`studentQuery1`, `studentQuery2` e assim por diante).

```
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

```

```

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 studentQuery/: 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
 * ClaireO@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 do C#](#)
- [Palavras-chave de consulta \(LINQ\)](#)
- [cláusula from](#)
- [partial \(método\) \(Referência do C#\)](#)
- [Tipos anônimos](#)
- [LINQ em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)

Cláusula group (Referência de C#)

21/01/2022 • 8 minutes to read

A cláusula `group` retorna uma sequência de objetos `IGrouping< TKey, TElement >` que contêm 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:

```
// 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:

```
// 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:

```
// 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:

```
// 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`.

```

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 em expressões de consulta com segurança, consulte [tratar exceções em expressões de consulta](#).

```
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.

```
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, consulte [como implementar uma classe leve com propriedades implementadas automaticamente](#) e [como consultar arquivos duplicados em uma árvore de diretórios](#). 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.

```

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.

```

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#)

21/01/2022 • 2 minutes to read

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.

```
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#)

21/01/2022 • 2 minutes to read

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.)

```

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.

```

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

```

Garcia, David
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 do C#](#)
- [Palavras-chave de consulta \(LINQ\)](#)
- [LINQ em C#](#)
- [Cláusula group](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)

Cláusula join (Referência de C#)

21/01/2022 • 10 minutes to read

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.

```
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.

```
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:

```
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 `join` em combinação com uma junção de grupo para especificar um elemento padrão do lado direito a ser produzido se um elemento do lado esquerdo não tiver `DefaultIfEmpty` nenhuma combinação. 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:

```
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 `equals` palavra-chave só pode ser usada em `join` uma cláusula e difere do operador de algumas maneiras `==` importantes. Ao comparar cadeias de caracteres, `equals` tem uma sobrecarga a ser comparada por valor e o operador usa igualdade de `==` referência. Quando ambos os lados da comparação têm variáveis de cadeia de caracteres `equals` `==` idênticas e atingirão o mesmo resultado: `true`. Isso 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 internamento de cadeia de caracteres para obter mais informações. Outra diferença importante é a comparação nula: é avaliada como `false` com o operador `, em vez do operador null equals null que a avalia como equals == true`. Por fim, o comportamento de scoping é diferente: com `,` a chave esquerda consome a sequência de origem externa e a chave direita consome `equals` a origem interna. A origem externa somente está no escopo

no lado esquerdo de `equais` 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 LINQ, as operações de junção são executadas em coleções de objetos. As coleções de objetos não podem ser "unidas" exatamente da mesma forma que duas tabelas relacionais. No LINQ, `join` cláusulas explícitas só são necessárias quando duas sequências de origem não estão vinculadas 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 de LINQ to SQL, consulte [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.

```
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
```

```
readies      J
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 de junção](#)
- [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#)

21/01/2022 • 2 minutes to read

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`.

```

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#)

21/01/2022 • 2 minutes to read

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](#).

```
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#)

21/01/2022 • 2 minutes to read

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](#).

```
IEnumerable<string> sortDescendingQuery =  
    from vegetable in vegetables  
    orderby vegetable descending  
    select vegetable;
```

Confira também

- [Referência do C#](#)
- [LINQ em C#](#)
- [ascending](#)

on (Referência de C#)

21/01/2022 • 2 minutes to read

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`.

```
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#)

21/01/2022 • 2 minutes to read

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`.

```
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#)

21/01/2022 • 2 minutes to read

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.

```
var query = from student in students
            group student by student.LastName[0];
```

Confira também

- [LINQ em C#](#)

in (Referência de C#)

21/01/2022 • 2 minutes to read

A `in` palavra-chave 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 do c# (referência C#)

21/01/2022 • 5 minutes to read

O C# fornece vários operadores. Muitos deles têm suporte dos [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 operações lógicas com `bool` operandos
- [Operadores de tecla e bit de alternância](#) que executam operações de bit ou de Shift com operandos dos tipos inteiros
- [Operadores de igualdade](#) que verificam se os operandos são iguais ou não

Normalmente, você pode [sobrepor](#) 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-los em expressões complexas usando operadores. A [precedência](#) de operador e a [Associação](#) 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:

```
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 `void` chamada de método 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:

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

Aqui estão alguns outros tipos de expressões que o C# fornece:

- [Expressões de cadeia de caracteres interpoladas](#) que fornecem uma sintaxe conveniente para criar cadeias formatadas:

```

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:

```

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#:

```

var scores = new[] { 90, 97, 78, 68, 85 };
IQueryable<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:

```

var a = 2 + 2 * 2;
Console.WriteLine(a); // output: 6

```

Use parênteses para alterar a ordem de avaliação imposta pela precedência do operador:

```

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], <code>x?.y</code> , <code>x?[y]</code> , x++, x--, xl, novo, typeof, Checked, desmarcado, padrão, nameof, delegate, sizeof, stackalloc, x->y	Primário
+ x, -x, ! x, ~ x, + + x, --x, ^ x, (T) x, Await, &x, * x, true e false	Unário

OPERADORES	CATEGORIA OU NOME
x..y	Intervalo
switch, com	switch e with expressões
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, é, como	Teste de tipo e relacional
x == y, x != y	Igualitário
x & y	AND lógico booliano ou AND lógico bit a bit
x ^ y	XOR lógico booliano ou XOR lógico bit a bit
x y	OR lógico booliano ou OR lógico bit a bit
x && y	AND condicional
x y	OR condicional
x?? lar	Operador de coalescência nula
c ? t : f	Operador condicional
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x = y, x ^= y, x <= y, x >= y, x?? = y, =>	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. Exceto para os operadores de **atribuição** e os **operadores de União nula**, todos os operadores binários são associativos à esquerda. 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, os operadores de União nula e o **operador ?: condicional** são associativos à direita. Por exemplo, `x = y = z` é avaliado como `x = (y = z)`.

Use parênteses para alterar a ordem de avaliação imposta pela associação de operador:

```
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. No entanto, alguns operadores avaliam os operandos condicionalmente. Ou seja, o valor do operando mais à esquerda de tal operador define if (ou quais) outros operandos devem ser avaliados. Esses operadores são os operadores lógicos condicional `and` (`&&`) e `or` (`||`), os [operadores de União nula](#) `??` e `??=`, os [operadores condicionais](#) `NULL` `?.` 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#)

21/01/2022 • 10 minutes to read

Os operadores a seguir executam operações aritméticas com operadores de tipos numéricos:

- Operadores unários `++` (incremento), `--` (decremento), `+` (mais) e `-` (menos)
- 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 `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`, `+=` e `-=`. Quando os operadores são de outros tipos integrais (`int`, `uint`, `long`, `ulong`), seus valores são convertidos no tipo `int`, que também é o tipo de resultado `sbyte`, `byte` de uma `short`, `ushort`, `char` ou `int` operação. Quando os operadores são de tipos integrais ou de ponto flutuante diferentes, seus valores são convertidos no tipo que contém 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 integrais e de ponto flutuante e o tipo `char`.

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:

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i++);
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:

```
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 `--` decrementa 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:

```
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:

```
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.

```
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:

```
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:

```
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`:

```
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:

```
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 obter mais informações sobre conversões entre tipos numéricos, consulte [Conversões numéricas integrados](#).

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 que o do operando à esquerda, conforme mostra o seguinte exemplo:

```
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.

NOTE

Esse método de computação do restante é análogo ao usado para os operadores inteiros, mas diferente da especificação do IEEE 754. Se você precisar da operação restante que está em conformidade com a especificação do IEEE 754, use o [Math.IEEEremainder](#) método .

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:

```
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:

```
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 concatenação de cadeia de caracteres e combinação de delegados.

Para obter mais informações, consulte o [artigo + Operadores e .](#)

Operador de subtração -

O operador de subtração `-` subtrai o operando à direita do operando à esquerda:

```
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 remoção de delegado. Para obter mais informações, consulte o [artigo - Operadores e .](#)

Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

```
x op= y
```

é equivalente a

```
x = x op y
```

exceto que `x` é avaliado apenas uma vez.

O seguinte exemplo demonstra o uso da atribuição composta com operadores aritméticos:

```

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:

```

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

```

Você também usa os `+=` `-=` operadores e para assinar e cancelar a assinatura de [um evento](#), respectivamente. Para obter mais informações, [consulte 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.

```

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 ver a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência do](#)

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](#).

No caso de um estouro aritmético de inteiros, um contexto de verificação de estouro, que pode ser [verificado ou não verificado](#), controla 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 [verificadas e não verificadas](#), você pode usar os operadores `checked` e `unchecked` para controlar o contexto de verificação de estouro, no qual uma expressão é avaliada:

```
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:

```
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](#) e a divisão por zero sempre gera uma [DivideByZeroException](#).

Erros de arredondamento

Devido a limitações gerais da representação de ponto flutuante de números reais e aritmética de ponto

flutuante, podem ocorrer erros de arredondados 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:

Para obter mais informações, consulte comentários nas páginas de referência [System.Double](#), [System.Single](#) ou [System.Decimal](#).

Capacidade de sobrecarga do operador

Um tipo definido pelo usuário pode sobrecarregar os operadores aritméticos unários (, e) e `++` `--` `+` `-` binários (, `*` , 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.

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 (referência do C#)

21/01/2022 • 8 minutes to read

Os operadores a seguir executam operações lógicas com operandos `bool` :

- Operador unário `!` (negação lógica) .
- Operadores Binary `&` (and lógico), `|` (OR lógico) e `^` (exclusivo lógico) . Esses operadores sempre avaliam os dois operandos.
- Operadores Binary `&&` (and lógico condicional and) e `||` (OR lógico condicional) . Esses operadores avaliam o operando à direita apenas se for necessário.

Para operandos dos [tipos numéricos inteiros](#), os `&` `|` operadores, e `^` executam operações lógicas de bit-nte.

Para obter mais informações, veja [Operadores bit a bit e shift](#).

Operador de negação lógica !

O operador de prefixo unário `!` computa 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` :

```
bool passed = false;
Console.WriteLine(!passed); // output: True
Console.WriteLine(!true); // output: False
```

A partir do C# 8,0, o operador de sufixo unário `!` é o [operador NULL-tolerante](#).

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 esquerdo for avaliado `false` , de modo que o resultado da operação seja `false` independente 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:

```
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 operandos dos tipos numéricos inteiros, o `&` operador computa a lógica e os opera bits de bit e de seus operando. 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 `bool` operandos, o `^` operador computa o mesmo resultado que o operador de desigualdade `!=`.

```
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 operandos dos tipos numéricos inteiros, o `^` operador computa o bit lógico Exclusive ou 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 esquerdo for avaliado `true`, de modo que o resultado da operação seja `true` independente 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:

```
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 dos tipos numéricos inteiros, o `|` operador computa o bit lógico ou 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`:

```
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`:

```
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 `bool?` operandos, os operadores `&` (and lógico) e `|` (lógico) oferecem suporte à lógica de três valores, da seguinte maneira:

- O `&` operador só produzirá `true` se ambos os operandos forem avaliados como `true`. Se `x` ou `y` for avaliada como `false`, `x & y` produz `false` (mesmo que outro operando seja avaliado como `null`). Caso contrário, o resultado de `x & y` é `null`.
- O `|` operador só produzirá `false` se ambos os operandos forem avaliados como `false`. Se `x` ou `y` for avaliada como `true`, `x | y` produz `true` (mesmo que outro operando seja avaliado como `null`). Caso contrário, o resultado de `x | y` é `null`.

A tabela a seguir apresenta essa semântica:

X	A	X&Y	X Y
true	true	true	true
true	false	false	true
true	nulo	null	true
false	true	false	true
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 que é definido para operandos de um tipo de valor também pode ser usado com operandos do tipo de valor anulável correspondente. Tal operador produz `null` se qualquer um de seus operandos é avaliado como `null`. No entanto, os `&` `|` operadores e podem produzir não nulo mesmo se um dos operandos for avaliado como `null`. Para obter mais informações sobre o comportamento do operador com tipos de valores anuláveis, consulte a seção [operadores levantados](#) do artigo [tipos de valores anuláveis](#).

Você também pode usar os `!` `^` operadores e com `bool?` operandos, como mostra o exemplo a seguir:

```
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 dão suporte a `bool?` operandos.

Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

```
x op= y
```

é equivalente a

```
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:

```
bool test = true;
test &= false;
Console.WriteLine(test); // output: False

test |= true;
Console.WriteLine(test); // output: True

test ^= false;
Console.WriteLine(test); // output: True
```

NOTE

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:

```

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 sobre [operadores do c#](#).

Capacidade de sobrecarga do operador

Um tipo definido pelo usuário pode [sobrecarregar](#) os `!`, `&`, `|` 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#)

21/01/2022 • 8 minutes to read

Os operadores a seguir executam operações bit a bit ou shift com os operadores dos tipos numéricos [integrais](#) ou o [tipo char](#):

- Operador Unário `~` ([complemento bit a bit](#))
- Operadores `<<` binários ([deslocamento para a esquerda](#)) `>>` e ([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 `&`, `|` e `^` operadores, e também são definidos para os operadores do `bool` tipo. 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:

```
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 obter informações sobre como o operador à direita define a contagem de deslocamentos, consulte 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:

```

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: 1100100100000000000000000010001
// After: 10010000000000000000000000100010000

```

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:

```

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 obter informações sobre como o operador à direita define a contagem de deslocamentos, consulte 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:

```

uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x >> 2;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 1001
// After: 10

```

As posições vazias de bit de ordem superior são definidas com base no tipo do operando à esquerda da seguinte maneira:

- Se o operando esquerdo for do tipo `int` ou `long`, o operador de deslocamento para a direita executará uma mudança aritmética: o valor do bit mais significativo (o bit de sinal) do operando esquerdo será propagado para as posições de bit vazias de ordem `int` `long` alta. 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.

```

int a = int.MinValue;
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

int b = a >> 3;
Console.WriteLine($"After: {Convert.ToString(b, toBase: 2)}");
// Output:
// Before: 10000000000000000000000000000000
// After: 11110000000000000000000000000000

```

- Se o operand à esquerda for do tipo ou , o operador de deslocamento para a direita executará um deslocamento lógico: as posições de bit vazias de ordem alta serão sempre `uint` `ulong` definidas como zero.

```

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), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After: 10000000000000000000000000000000

```

Operador AND lógico &

O operador calcula o AND lógico bit a bit `&` de seus operadores integrais:

```

uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000

```

Para `bool` os operadores, o `&` operador calcula o AND lógico de seus operadores. O operador `&` unário é o [operador address-of](#).

Operador OR exclusivo lógico ^

O operador calcula o OR exclusivo lógico bit a bit, também conhecido como XOR lógico bit a `^` bit, de seus operadores integrais:

```

uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100

```

Para `bool` os operadores, o `^` operador calcula o OR exclusivo lógico de seus operadores.

Operador OR lógico |

O operador calcula o OR lógico bit a bit `|` de seus operadores integrais:

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

Para `bool` os operadores, o `|` operador calcula o OR lógico de seus operadores.

Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

```
x op= y
```

é equivalente a

```
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:

```
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: 11110000

a = INITIAL_VALUE;
a <<= 2;
Display(a); // output: 1111100000

a = INITIAL_VALUE;
a >>= 4;
Display(a); // output: 1111

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 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:

```

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:

```

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 ver a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência do operador](#) do artigo [Operadores C#](#).

Contagem de deslocamento dos operadores de deslocamento

Para os operadores `shift` e `, o tipo do operand à direita deve ser ou um tipo que tenha uma << >> int conversão numérica implícita predefinida para int.`

Para as expressões `x << count` e `x >> count`, a contagem real de deslocamento depende do tipo de `x` da seguinte maneira:

- Se o tipo de `for` ou `, a contagem de deslocamentos será definida pelos cinco bits de ordem baixa do operand x int à uint direita. Ou seja, a contagem de deslocamentos é calculada a partir de count & 0x1F (ou count & 0b_1_1111).`
- Se o tipo de `for` ou `, a contagem de deslocamentos será definida pelos seis bits de ordem baixa do operand x long à ulong 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:

```

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

```

NOTE

Como mostra o exemplo anterior, o resultado de uma operação de deslocamento pode ser diferente de zero, mesmo se o valor do operand à direita for maior que o número de bits no operand esquerdo.

Operadores lógicos de enumeração

Os `~`, `&` operadores `|`, , e também são `^` suportados por qualquer tipo de enumeração. Para os operadores 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)`.

Normalmente, você usa operadores lógicos bit a bit com um tipo de enumeração definido com o atributo Flags. 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` sobrepor o operador `<<` ou `>>`, o tipo do operando à esquerda deverá ser `T` e o tipo do operando à direita deverá ser `int`.

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](#)

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Operadores lógicos booleanos](#)

Operadores de igualdade (Referência de C#)

21/01/2022 • 5 minutes to read

Os `==` operadores (igualdade) `!=` e (desigualdade) verificam se seus operadores são iguais ou não.

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:

```
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
```

NOTE

Para os `==` operadores, `<`, `,`, `>`, `<=`, `=`, `>=` e, se qualquer um dos operadores não for um número (ou), o `Double.NaN` resultado da operação será `Single.NaN` `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](#).

Começando com C# 7.3, os operadores `==`, `!=`, e são compatíveis com as `tuplas` de C#. Para obter mais informações, consulte a [seção Igualdade de tupla](#) do artigo [Tipos de tupla](#).

Igualdade de tipos de referência

Por padrão, dois operadores de tipo de referência não registro serão iguais se se referirem ao mesmo objeto:

```

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 sobrepor o operador `==`. Se um tipo de referência sobrepor 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

Disponível 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 operadores de registro são iguais quando ambos são ou valores correspondentes de todos os campos e propriedades `null` auto-implementadas são iguais.

```

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, no caso de membros de tipo de referência não registro, 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:

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower()); // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3); // output: False
```

Essa é uma comparação ordinal que faz a troca de minúsculas. Para obter mais informações sobre a comparação de cadeias de caracteres, confira [Como comparar cadeias de caracteres no C#](#).

Igualdade de delegado

Dois [operadores delegados](#) do mesmo tipo de tempo de run time são iguais quando ambos são ou suas listas de invocação têm o mesmo tamanho e têm entradas iguais em cada `null` posição:

```
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#](#).

Os delegados produzidos pela avaliação de [expressões lambda](#) semanticamente idênticas não são iguais, como mostra o exemplo a seguir:

```
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; caso contrário, `false`.

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 `!=`:

```
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 deverá sobrepor o outro.

Um tipo de registro não pode sobrepor explicitamente `==` os `!=` operadores e . Se você precisar alterar o comportamento dos operadores e para o tipo de registro `==` `!=` , `T` implemente o método com a seguinte `IEquatable<T>.Equals` assinatura:

```
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#)

21/01/2022 • 2 minutes to read

A comparação `<` (menor que), `>` (maior que), `<=` (menor que ou igual) e `>=` (maior que ou igual), também conhecida como operadores relacionais, comparam seus operandos. Esses operadores têm suporte de todos os tipos numéricos **integral** e **de ponto flutuante**.

NOTE

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 dos `char` operandos, 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 `==` e `!=` operadores 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`:

```
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`:

```
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 <=

O operador `<=` retornará `true` se o operando à esquerda for menor ou igual ao operando à direita, caso contrário, `false`:

```
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`:

```
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

Um tipo definido pelo usuário pode **sobrecarregar** os `<`, `>`, `<=` e `>=`.

Se um tipo sobrecarregar um dos operadores `<` ou `>`, ele deverá sobrecarregar tanto `<` quanto `>`. Se um tipo sobrecarregar um dos operadores `<=` ou `>=`, ele deverá sobrecarregar 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 de acesso de membro e expressões (referência C#)

21/01/2022 • 9 minutes to read

Você pode usar os seguintes operadores e expressões ao acessar um membro de tipo:

- `.` (acesso de membro): para acessar um membro de um namespace ou de um tipo
- `[]` (elemento de matriz ou acesso de indexador): para acessar um elemento de matriz ou um indexador de tipo
- `?.` and `?[]` (operadores condicionais nulos): para executar uma operação de acesso de elemento ou membro somente se um operando não for nulo
- `()` (invocação): para chamar um método acessado ou invocar um delegado
- `^` (índice de fim): para indicar que a posição do elemento é do final de uma sequência
- `..` (Range): para especificar um intervalo de índices que você pode usar para obter um intervalo de elementos de sequência

Expressão de acesso de 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 em um namespace, como mostra o exemplo a seguir de uma `using` diretiva :

```
using System.Collections.Generic;
```

- Use `.` para formar um *nome qualificado* para acessar um tipo dentro de um namespace, como mostra o código a seguir:

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

Use uma `using` diretiva para tornar o uso de nomes qualificados opcional.

- Use `.` para acessar **membros de tipo**, estático e não-estático, como mostra o código a seguir:

```
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:

```
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 [Dictionary<TKey,TValue>](#) tipo .net para demonstrar o acesso ao indexador:

```
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 de matriz, que devem ser inteiros, os parâmetros do indexador podem ser declarados como sendo de 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](#):

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Operadores condicionais nulos ?. e ?[]

Disponível no C# 6 e posterior, um operador NULL-Conditional aplica um [acesso de membro](#), `?.` ou [acesso de elemento](#), `?[]`, operação para seu operando somente se esse operando for avaliado como não nulo; caso contrário, retornará `null`. Isto é

- Se for `a` avaliada como `null`, o resultado de `a?.x` ou `a?[]` é `null`.
- Se for `a` avaliada como não nula, o resultado `a?.x` ou `a?[]` será o mesmo que o resultado de `a.x` ou `a[]`, respectivamente.

NOTE

Se `a.x` ou `a[]` lançar uma exceção, `a?.x` ou `a?[]` geraria a mesma exceção para não `null` `a`. Por exemplo, se `a` for uma instância de matriz não nula e `x` estiver fora dos limites de `a`, o `a?[]` geraria 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`:

```
A?.B?.Do(C);
A?.B?[C];
```

Se `A` pode ser nulo, `B` mas `C` não seria nulo se um não for nulo, você só precisará aplicar o operador NULL-Conditional a `A`:

```
A?.B.C();
```

No exemplo anterior, `B` não é avaliado e `c()` não é chamado se `A` for nulo. No entanto, se o acesso de membro encadeado for interrompido, por exemplo, entre parênteses como em `(A?.B).c()`, o curto-circuito não ocorrerá.

Os exemplos a seguir demonstram o uso `?.` dos `?[]` operadores e:

```
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
```

```

using System;
using System.Collections.Generic;
using System.Linq;

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 União nula para especificar uma expressão alternativa a ser avaliada, caso o resultado de uma operação condicional nula seja `null`.

Se `a.x` ou `a[x]` for de um tipo de valor não anulável `T` `a?.x` ou `a?[]` for do tipo de valor anulável correspondente `T?`. Se você precisar de uma expressão do tipo `T`, aplique o operador de União nula `??` a uma expressão condicional nula, como mostra o exemplo a seguir:

```

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, o `numbers?.Length < 2` será avaliado como `false` quando `numbers` é `null`.

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:

```
PropertyChanged?.Invoke(...)
```

Esse código é equivalente ao código a seguir que você usaria no C# 5 ou anterior:

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(...);
}
```

Essa é uma forma thread-safe de garantir que apenas um não nulo `handler` seja invocado. Como as instâncias de delegado são imutáveis, nenhum thread pode alterar o objeto referenciado pela `handler` variável local. Em particular, se o código executado por outro thread cancelar a assinatura do `PropertyChanged` evento e `PropertyChanged` se tornar `null` antes de `handler` ser invocado, o objeto referenciado por `handler` permanecerá inalterado. O `?.` operador avalia seu operando à esquerda não mais de uma vez, garantindo que ele não possa ser alterado para `null` depois de ser verificado como não nulo.

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:

```
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.

Índice do operador end ^

Disponível em C# 8,0 e posterior, o `^` operador indica a posição do elemento do final de uma sequência. Para uma sequência de comprimento `length`, `^n` aponta para o elemento com offset `length - n` do 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.

```

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, Expression `^e` é do [System.Index](#) tipo. Na expressão `^e`, o resultado de `e` deve ser implicitamente conversível para `int`.

Você também pode usar o `^` operador com o [operador Range](#) para criar um intervalo de índices. Para obter mais informações, consulte [índices e intervalos](#).

Operador de intervalo..

Disponível em C# 8,0 e posterior, o `..` operador especifica o início e o término de um intervalo de índices como operandos. O operando à esquerda é um início *inclusivo* de um intervalo. O operando de lado direito é uma extremidade *exclusiva* de um intervalo. Qualquer um dos operandos pode ser um índice do início ou do final de uma sequência, como mostra o exemplo a seguir:

```

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, Expression `a..b` é do [System.Range](#) tipo. Na expressão `a..b`, os resultados de `a` e `b` devem ser conversíveis implicitamente para `int` ou [Index](#).

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`

```
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));
```

Para obter mais informações, consulte [índices e intervalos](#).

Capacidade de sobrecarga do operador

Os `.`, `()`, `^` e `..` operadores, 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](#)
- [NULL-acesso de membro condicional](#)
- [Expressões de invocação](#)

Para obter mais informações sobre índices e intervalos, consulte a [Nota de proposta de recurso](#).

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [?? \(operador de união nula\)](#)
- [operador::](#)

Operadores de teste de tipo e expressão de cast (referência de C#)

21/01/2022 • 6 minutes to read

Você pode usar os seguintes operadores e expressões para executar a verificação de tipo ou conversão de tipo:

- **operador is**: verifique se o tipo de tempo de run time de uma expressão é compatível com um determinado tipo
- **operador as**: converter explicitamente uma expressão em um determinado tipo se seu tipo de tempo de run-time for compatível com esse tipo
- **expressão cast**: executar uma conversão explícita
- **Operador typeof**: obter a `System.Type` instância de um tipo

Operador is

O `is` operador verifica se o tipo de tempo de run time de um resultado de expressão é compatível com um determinado tipo. A partir do C# 7.0, o `is` operador 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

```
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 `is` operador retorna quando um resultado de expressão não é `true`, nulo e qualquer uma das seguintes condições é verdadeira:

- O tipo de tempo de execução de um resultado de expressão é `T`.
- O tipo de tempo de execução de um resultado de expressão deriva do tipo `T`, implementa a interface ou existe outra conversão de referência implícita `T` `T` dele para `T`.
- O tipo de tempo de run time de um resultado de expressão é **um** tipo de valor que pode ser anulado com o tipo subjacente e `T` é `Nullable<T>.HasValue` `true`.
- Existe **uma conversão boxing ou unboxing** do tipo de tempo de run-time de um resultado de expressão para o tipo `T`.

O operador `is` não considera conversões definidas pelo usuário.

O exemplo a seguir demonstra que o operador retorna se o tipo de tempo de run-time de um resultado de expressão deriva de um determinado tipo, ou seja, existe uma conversão de referência `is true` entre tipos:

```

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 exemplo a seguir mostra que o operador leva em conta conversões boxing e unboxing, mas não `is` considera [conversões numéricas](#):

```

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

A partir do C# 7.0, o `is` operador também testa um resultado de expressão em relação a um padrão. O exemplo a seguir mostra como usar um padrão [de declaração](#) para verificar o tipo de tempo de run-time de uma expressão:

```

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, consulte [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 `as` operador retornará `null`. Ao contrário [de uma expressão de cast](#), o operador nunca lança uma `as` exceção.

A expressão da forma

```
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

```
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 cast](#).

O exemplo a seguir demonstra o uso do operador `as`:

```
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
}
```

NOTE

Como mostrado no exemplo anterior, você precisa comparar o resultado da expressão `as` com `null` para verificar se uma conversão foi bem-sucedida. A partir do C# 7.0, 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:

```
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:

```

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 requer anotações de metadados. Os exemplos incluem os seguintes tipos:

- `dynamic`
- `string?` (ou qualquer tipo de referência que anulável)

Esses tipos não são representados diretamente nos metadados. Os tipos incluem atributos que descrevem o tipo subjacente. Em ambos os casos, você pode usar o tipo subjacente. Em vez `dynamic` de, você pode usar `object`. Em vez `string?` de, você pode usar `string`.

Você também pode usar o `typeof` operador com tipos genéricos `nobound`. 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:

```

Console.WriteLine(typeof(Dictionary<, >));
// Output:
// System.Collections.Generic.Dictionary`2[TKey, TValue]

```

Uma expressão não pode ser um argumento do `typeof` operador. Para obter a `System.Type` instância do tipo de tempo de run-time de um resultado de expressão, use o método `Object.GetType`.

Teste de tipo com o operador `typeof`

Use o operador para verificar se o tipo de tempo de run time do resultado da expressão `typeof` corresponde exatamente a um determinado tipo. O exemplo a seguir demonstra a diferença entre a verificação de tipo feita com o `typeof` operador e o operador `is`:

```

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 `is`, `as` operadores, `typeof` e não podem ser sobreescritos.

Um tipo definido pelo usuário não pode sobreescalar 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](#)
- [O operador typeof](#)

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Como fazer a cast com segurança usando a correspondência de padrões e os operadores is e as](#)
- [Generics in .NET \(Genéricos no .NET\)](#)

Operadores de conversão definidos pelo usuário (Referência de C#)

21/01/2022 • 2 minutes to read

Um tipo definido pelo usuário pode definir uma conversão implícita ou explícita personalizada de outro tipo ou para outro.

Conversões implícitas não requerem que uma sintaxe especial seja invocada e podem ocorrer em uma variedade de situações, por exemplo, em atribuições e invocações de método. Conversões implícitas C# predefinidos sempre são bem-sucedidas e nunca lançam 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.

Conversões definidas pelo usuário não são consideradas pelos operadores `is` e `as`. 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:

```
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
    }
}
```

Use também a palavra-chave `operator` para sobrepor um operador C# predefinido. Para obter mais informações, consulte [Sobrecarga de 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#](#)
- [Sobrecarga de 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 a ponteiro (referência do C#)

21/01/2022 • 7 minutes to read

Você pode usar os operadores a seguir para trabalhar com ponteiros:

- Operador Unário `&` ([endereço de](#)) : para obter o endereço de uma variável
- Operador Unário `*` ([indicação de ponteiro](#)) : para obter a variável apontada por um ponteiro
- Os `->` [operadores \(acesso de membro\)](#) `[]` e [\(acesso a elemento\)](#)
- Operadores `+` `-` aritméticos `++` `,`, e `--`
- Operadores `==` [de comparação](#) `, , , ! = < > , <= >=`

Para obter informações sobre tipos de ponteiros, veja [Tipos de ponteiro](#).

NOTE

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:

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(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 "corrigir" ou "fixar", com uma [fixed](#) instrução. O endereço obtido é válido somente dentro do bloco de uma `fixed` instrução . O exemplo a seguir mostra como usar uma `fixed` instrução e o `&` operador :

```

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 inteiros.

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.

```

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 for um ponteiro do tipo e for um membro `x` `T*` `y` acessível do tipo `T`, uma expressão do formulário

`x->y`

é equivalente a

`(*x).y`

O exemplo a seguir demonstra o uso do operador `->`:

```

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 `[]`:

```

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 `stackalloc` expressão aloca um bloco de memória na pilha.

NOTE

O operador de acesso de elemento de ponteiro não verifica se há erros fora dos limites.

Não é possível usar `[]` para acesso de elemento de ponteiro com uma expressão do tipo `void*`.

Você também pode usar o `[]` operador para acesso de elemento de matriz ou [indexador](#).

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

Você não pode 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 `sizeof` operador obtém o tamanho de um tipo em bytes.

O exemplo a seguir demonstra o uso do operador `+` com um ponteiro:

```
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:

```
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 decremeno `--` 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:

```
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 ver a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência do operador](#) do artigo [Operadores C#](#).

Capacidade de sobrecarga do operador

Um tipo definido pelo usuário não pode sobrecarregar operadores relacionados a 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#](#)
- [Tipos de ponteiro](#)
- [Palavra-chave unsafe](#)
- [palavra-chave fixed](#)
- [stackalloc](#)
- [Operador sizeof](#)

Operadores de atribuição (referência C#)

21/01/2022 • 2 minutes to read

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 à direita, ou seja, uma expressão do formulário

```
a = b = c
```

é avaliada como

```
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:

```
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
```

Operador de atribuição ref

Começando pelo C# 7.3, você pode usar o operador de atribuição `ref = ref` para reatribuir uma variável [ref local](#) ou [ref readonly local](#). O exemplo a seguir demonstra o uso do operador de atribuição `ref`:

```
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 caso do operador de atribuição de referência, ambos os operandos devem ser do mesmo tipo.

Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

```
x op= y
```

é equivalente a

```
x = x op y
```

exceto que `x` é avaliado apenas uma vez.

A atribuição composta é suportada por operadores aritméticos, lógicos e bit a bit.

Atribuição de União nula

A partir do C# 8.0, você pode usar o operador de atribuição de União nula `?=?` para atribuir o valor de seu operando à direita para seu operando à esquerda somente se o operando à esquerda for avaliado como `null`. Para obter mais informações, consulte [?? e ?? = artigo de operadores](#).

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 `op=` operador, se existir, também será sobrecarregado implicitamente.

Especificação da linguagem C#

Saiba mais na seção [Operadores de atribuição](#) na [Especificação da linguagem C#](#).

Para obter mais informações sobre o operador de atribuição de referência `= ref`, consulte a [Nota de proposta de recurso](#).

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [ref keyword](#)

Expressões lambda (referência C#)

21/01/2022 • 15 minutes to read

Você usa 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:

```
(input-parameters) => expression
```

- [Instrução lambda](#) que tem um bloco de instrução como corpo:

```
(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 exemplo a seguir, a expressão lambda `x => x * x`, que especifica um parâmetro denominado `x` e retorna o valor de `x` quadrado, é atribuída a uma variável de um tipo delegado:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

As lambdas de expressão também podem ser convertidas para os tipos de [árvore de expressão](#), como mostra o exemplo a seguir:

```
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. Você também pode usar expressões lambda ao escrever [LINQ em C#](#), como mostra o exemplo a seguir:

```
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 `Queryable.Select` método na `System.Linq.Queryable` classe, por exemplo, em 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 criado dos 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:

```
(input-parameters) => expression
```

O corpo de um lambda de expressão pode consistir em uma chamada de método. No entanto, se você estiver criando [árvores de expressão](#) que são avaliadas fora do contexto do CLR (Common Language Runtime) do .NET, como no SQL Server, você não deve usar chamadas de método em expressões lambda. Os métodos não terão significado fora do contexto do CLR (Common Language Runtime) do .NET.

Lambdas de instrução

Uma instrução lambda é semelhante a uma expressão lambda, exceto que suas instruções são colocadas entre chaves:

```
(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.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Você não pode usar lambdas de instrução para criar árvores de expressão.

Parâmetros de entrada de uma expressão lambda

Você coloca os parâmetros de entrada de uma expressão lambda entre parênteses. Especifique parâmetros de entrada zero com parênteses vazios:

```
Action line = () => Console.WriteLine();
```

Se uma expressão lambda tiver apenas um parâmetro de entrada, os parênteses serão opcionais:

```
Func<double, double> cube = x => x * x * x;
```

Dois ou mais parâmetros de entrada são separados por vírgulas:

```
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:

```
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, você pode usar os [Descartes](#) para especificar dois ou mais parâmetros de entrada de uma expressão lambda que não são usados na expressão:

```
Func<int, int, int> constant = (_, _) => 42;
```

Os parâmetros de descarte lambda podem ser úteis quando você usa uma expressão lambda para [fornecer um manipulador de eventos](#).

NOTE

Para compatibilidade com versões anteriores, se apenas um único parâmetro de entrada for nomeado `_`, em seguida, dentro de 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`.

```

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:

```

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 métodos assíncronos, consulte [programação assíncrona com Async e Await](#).

Expressões lambda e tuplas

A partir do C# 7.0, 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.

```

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 nomeados `Item1`, `Item2` e assim por diante. No entanto, você pode definir uma tupla com componentes nomeados, como é feito no exemplo a seguir.

```

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 obter mais informações sobre tuplas C#, consulte [tipos de tupla](#).

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. `Func` delegados são úteis para encapsular expressões definidas pelo usuário que são aplicadas a cada elemento em um conjunto de dados de origem. Por exemplo, considere o seguinte tipo delegado `Func<T,TResult>`:

```
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:

```

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`:

```

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:

```

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 `numbers` matriz até encontrar um número cujo valor seja menor do que sua posição ordinal na matriz:

```

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 em expressões de consulta, como mostra o exemplo a seguir:

```

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 `Customer` objeto, o que significa que você tem acesso a seus métodos e propriedades:

```
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 não tem um tipo porque o Common Type System não tem um conceito intrínseco de "expressão lambda". No entanto, às vezes é conveniente falar informalmente do "tipo" de uma expressão lambda. Esse "tipo" informal refere-se ao tipo delegado ou [Expression](#) ao tipo para o qual a expressão lambda é convertida.

A partir do C# 10, uma expressão lambda pode ter um *tipo natural*. Em vez de forçá-lo 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:

```
var parse = (string s) => int.Parse(s);
```

O compilador pode inferir `parse` ser um `Func<string, int>`. O compilador escolhe um disponível `Func` ou `Action` delegado, se houver um adequado. Caso contrário, ele sintetiza um tipo delegado. Por exemplo, o tipo delegado será sintetizado se a expressão lambda tiver `ref` parâmetros. Quando uma expressão lambda tem um tipo natural, ela pode ser atribuída a um tipo menos explícito, como [System.Object](#) ou [System.Delegate](#):

```
object parse = (string s) => int.Parse(s); // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Grupos de métodos (ou seja, nomes de método sem listas de parâmetros) com exatamente uma sobrecarga têm um tipo natural:

```
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>](#), com o tipo de delegado natural usado como o argumento para o parâmetro de tipo:

```
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:

```
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:

```
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

funciona:

```
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 parênteser os parâmetros de entrada:

```
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Atributos

Começando com o 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:

```
Func<string, int> parse = [Example(1)] (s) => int.Parse(s);
var choose = [Example(2)][Example(3)] object (bool b) => b ? 1 : "two";
```

Você também pode adicionar atributos aos parâmetros de entrada ou ao valor de retorno, como mostra o exemplo a seguir:

```
var sum = ([Example(1)] int a, [Example(2), Example(3)] int b) => a + b;
var inc = [return: Example(1)] (int s) => s++;
```

Como mostram os exemplos anteriores, você deve incluir parênteses nos parâmetros de entrada ao adicionar atributos a uma expressão lambda ou a seus parâmetros.

IMPORTANT

As expressões lambda são invocadas por meio do tipo delegado subjacente. Isso é diferente de métodos e funções locais. O método do delegado `Invoke` não verifica atributos na expressão lambda. Os atributos não têm nenhum efeito quando a expressão lambda é chamada. Os atributos em expressões lambda são úteis para análise de código e podem ser descobertos por meio de reflexão. Uma consequência dessa decisão é que o `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 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:

```

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}");
    }
}

```

As seguintes regras se aplicam ao escopo variável em expressões lambda:

- Uma variável capturada não será coletada por lixo até que o delegado que faz referência se torne qualificado 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**, **refou** **out** diretamente do 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**, **Breakou** **continue** se o destino dessa instrução de salto estiver fora do bloco de 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 `static` modificador a uma expressão lambda para impedir a captura não intencional de variáveis locais ou o estado de instância pelo lambda:

```
Func<double, double> square = static x => x * x;
```

Um lambda estático não pode capturar variáveis locais ou estado de instância de escopos delimitadores, mas pode referenciar membros estáticos e definições de constante.

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 em C# 9,0 e posterior, consulte as seguintes notas de proposta de recurso:

- [Parâmetros de descarte de lambda \(C# 9,0\)](#)
- [Funções anônimas estáticas \(C# 9,0\)](#)
- [Melhorias de lambda \(C# 10\)](#)

Confira também

- [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 de XQuery](#)
- [exemplos de LINQ 101](#)

Padrões (referência de C#)

21/01/2022 • 15 minutes to read

O C# introduziu a correspondência de padrões no C# 7.0. Desde então, cada versão principal do C# estende os recursos de correspondência de padrões. As seguintes expressões e instruções C# são compatíveis com a correspondência de padrões:

- `is` Expressão
- `switch` Declaração
- `switch expression` (introduzido no C# 8.0)

Nesses constructos, você pode corresponder a uma expressão de entrada com qualquer um dos seguintes padrões:

- **Padrão de declaração**: para verificar o tipo de tempo de run-time de uma expressão e, se uma corresponder for bem-sucedida, atribua um resultado de expressão a uma variável declarada. Introduzido no C# 7.0.
- **Padrão de tipo** : para verificar o tipo de tempo de run-time de uma expressão. Introduzido no C# 9.0.
- **Padrão constante**: para testar se um resultado de expressão é igual a uma constante especificada. Introduzido no C# 7.0.
- **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 os campos de uma expressão corresponderem a padrões aninhados. Introduzido no C# 8.0.
- **Padrão posicional**: para desconstruir um resultado de expressão e testar se os valores resultantes corresponderem a padrões aninhados. Introduzido no C# 8.0.
- **var pattern**: para corresponder a qualquer expressão e atribuir seu resultado a uma variável declarada. Introduzido no C# 7.0.
- **Padrão de descarte**: para corresponder a qualquer expressão. Introduzido no C# 8.0.

Padrões lógicos, de propriedade e posicionais são padrões recursivos. Ou seja, eles podem conter *padrões aninhados*.

Para ver o exemplo de como usar esses padrões para criar um algoritmo orientado a dados, consulte Tutorial: Usar a correspondência de padrões para criar [algoritmos orientados a tipos e orientados a dados](#).

Padrões de tipo e declaração

Você usa padrões de tipo e declaração para verificar se o tipo de tempo de run time 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 convertido, como mostra o exemplo a seguir:

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower()); // output: hello, world!
```

A partir do C# 7.0, um padrão de declaração com tipo corresponde a uma expressão quando um resultado de expressão é não nulo e qualquer uma das `T` seguintes condições é verdadeira:

- O tipo de tempo de execução de um resultado de expressão é `T`.
- O tipo de tempo de execução de um resultado de expressão deriva do tipo , implementa a interface ou existe outra conversão de referência implícita `T` `T` dele para `T`. O exemplo a seguir demonstra dois casos quando essa condição é verdadeira:

```
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 para o método , o primeiro padrão corresponde a um valor de argumento porque o tipo de tempo de `GetSourceLabel` run-time do argumento deriva do tipo `int[]` `Array`. Na segunda chamada para o método , o tipo de tempo de execução do argumento não deriva do tipo , mas `GetSourceLabel` `List<T>` implementa a interface `Array` `ICollection<T>`.

- O tipo de tempo de run-time de um resultado de expressão é um tipo de valor que pode ser [anulado](#) com o tipo subjacente `T`.
- Existe [uma conversão boxing ou unboxing](#) do tipo de tempo de run-time de um resultado de expressão para o tipo `T`.

O exemplo a seguir demonstra as duas últimas condições:

```
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:

```

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:

```

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 run-time atende a qualquer uma das condições listadas acima.

Para obter mais informações, consulte as [seções Padrão de declaração](#) e Padrão de tipo das notas de proposta de recurso.

Padrão de constante

A partir do C# 7.0, você usa um padrão constante para testar se um resultado de expressão é igual a uma constante especificada, como mostra o exemplo a seguir:

```

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 [inteiro ou](#) de ponto flutuante
- um [caractere ou](#) um literal de cadeia [de caracteres](#)
- um valor booliana [true](#) ou [false](#)
- um [valor de enum](#)
- o nome de um campo [const declarado](#) ou local
- [null](#)

Use um padrão constante para verificar [null](#) se há , como mostra o exemplo a seguir:

```
if (input is null)
{
    return;
}
```

O compilador garante que nenhum operador de igualdade sobreescrito pelo `==` usuário seja invocado quando a expressão `x is null` é avaliada.

A partir do C# 9.0, você pode usar um padrão de constante negada para verificar se há não nulo, como mostra o exemplo a seguir:

```
if (input is not null)
{
    // ...
}
```

Para obter mais informações, consulte [a seção Padrão constante](#) da nota de proposta de 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:

```
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 [inteiro](#), [ponto flutuante](#), [char](#) ou [tipo de enum](#).

Para verificar se um resultado de expressão está em um determinado intervalo, match-lo com um padrão [conjuntivo](#) `and`, como mostra o exemplo a seguir:

```
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 ou falhar ao converter para o tipo de uma constante por uma conversão anuável ou unboxing, um padrão relacional não corresponderá `null` a uma expressão.

Para obter mais informações, consulte a [seção Padrões relacionais](#) da nota de proposta de recurso.

Padrões lógicos

A partir do C# 9.0, você usa os combinadores de padrão `not`, e para criar os seguintes padrões `and` `or` 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` para verificar `null` se uma expressão não é nula:

```
if (input is not null)
{
    // ...
}
```

- *Conjuntivo* `and` padrão que corresponde a uma expressão quando ambos os padrões corresponderem à expressão. O exemplo a seguir mostra como você pode combinar `padrões relacionais` para verificar se um valor está em um determinado intervalo:

```
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",
};
```

- *Disjuntive* `or` padrão que corresponde a uma expressão quando um dos padrões corresponde à expressão, como mostra o exemplo a seguir:

```
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 de verificação

A lista a seguir ordena combinadores de padrão começando da precedência mais alta para a mais baixa:

- `not`
- `and`

- or

Para especificar explicitamente a precedência, use parênteses, como mostra o exemplo a seguir:

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

NOTE

A ordem na qual os padrões são verificados é indefinida. Em tempo de executar, os padrões aninhados à direita de or e padrões podem ser and verificados primeiro.

Para obter mais informações, consulte [a seção Combinadores de padrão](#) da nota de proposta de recurso.

Padrão de propriedade

A partir do C# 8.0, você usa um padrão de propriedade para corresponder propriedades ou campos de uma expressão a padrões aninhados, como mostra o exemplo a seguir:

```
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 run time e uma declaração de variável a um padrão de propriedade, como mostra o exemplo a seguir:

```
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 um padrão aninhado. Use um padrão de propriedade para corresponder partes de dados com padrões aninhados, como mostra o exemplo a seguir:

```
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: combinador de padrões or e tipos de registro.

A partir do C# 10, você pode referenciar propriedades aninhadas ou campos dentro de um padrão de propriedade. Por exemplo, você pode refator o método do exemplo anterior no seguinte código equivalente:

```
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 de recurso e a observação proposta de proposta de recursos Padrões de [propriedade](#) estendida.

Padrão posicional

A partir do C# 8.0, você usa um padrão posicional para desconstruir um resultado de expressão e corresponder aos valores resultantes em relação aos padrões aninhados correspondentes, como mostra o exemplo a seguir:

```
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 uma expressão contém o método [Deconstruct](#), 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 a várias entradas em vários padrões, como mostra o exemplo a seguir:

```
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 em um padrão [Deconstruct](#) posicional, como mostra o exemplo a seguir:

```

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 operação e uma declaração de variável, como mostra o exemplo a seguir:

```

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 `Deconstruct` método .

- Use um [padrão de propriedade](#) dentro de um padrão posicional, como mostra o exemplo a seguir:

```

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:

```

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 um padrão aninhado.

Para obter mais informações, consulte [a seção Padrão posicional](#) da nota de proposta de recurso.

var Padrão

A partir do C# 7.0, você usa um padrão para corresponder `var` a qualquer expressão, incluindo e atribui seu

resultado a uma nova variável local, como mostra o exemplo `null` a seguir:

```
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 `var` padrão é útil quando você precisa de uma variável temporária dentro de uma expressão booleana para manter o resultado de cálculos intermediários. Você também pode usar um padrão quando precisar executar verificações adicionais em caso de proteção de uma expressão ou `var` instrução, como mostra o exemplo a seguir:

```
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, o tipo de uma variável declarada é o tipo de tempo de compilação da expressão que é `var` corresponder ao padrão.

Para obter mais informações, consulte a [seção Padrão var](#) da nota de proposta de recurso.

Padrão de descarte

A partir do C# 8.0, você usa um padrão de descarte para corresponder a qualquer expressão, incluindo `_`, como mostra o exemplo a seguir:

```

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,
};

```

No exemplo anterior, um padrão de descarte é usado para manipular qualquer valor inteiro que não tenha o membro `null` correspondente da `DayOfWeek` enumeração. Isso garante que uma `switch` expressão no exemplo trata de todos os valores de entrada possíveis. Se você não usar um padrão de descarte em uma expressão e nenhum dos padrões da expressão corresponde a uma entrada, o `switch` runtime [lançará uma exceção](#). O compilador gerará um aviso se uma `switch` expressão não tratar todos os valores de entrada possíveis.

Um padrão de descarte não pode ser um padrão em `is` uma expressão ou uma `switch` instrução. Nesses casos, para corresponder a qualquer expressão, use um `var` padrão com um descarte: `var _`.

Para obter mais informações, consulte a [seção Descartar padrão](#) da nota de proposta de 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:

```

if (input is not (float or double))
{
    return;
}

```

Especificação da linguagem C#

Para obter mais informações, consulte as seguintes notas de proposta de recurso:

- [Correspondência de padrões para c# 7.0](#)
- [Correspondência de padrões recursivos \(introduzido no C# 8.0\)](#)
- [Alterações de correspondência de padrões para o C# 9.0](#)
- [Padrões de propriedade estendidos \(C# 10\)](#)

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Tutorial: Usar a correspondência de padrões para criar algoritmos orientados por tipo e orientados a dados](#)

Operadores + e += (referência de C#)

21/01/2022 • 2 minutes to read

Os operadores + e += são suportados pelos tipos numéricos integrais e de ponto flutuante, o tipo de 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 operadores são do tipo cadeia de caracteres , o operador concatena as representações de cadeia de caracteres de seus operandos (a representação de cadeia de caracteres de é uma cadeia + de caracteres null vazia):

```
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 partir do C# 6, a interpolação de cadeia de caracteres fornece uma maneira mais conveniente de formatar cadeias de caracteres:

```
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 os espaços reservados também são cadeias de caracteres constantes.

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 + :

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab(); // output: ab
```

Para executar a remoção do 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

```
x += y
```

é equivalente a

```
x = x + y
```

exceto que `x` é avaliado apenas uma vez.

O exemplo a seguir demonstra o uso do operador `+=`:

```
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 [sobrepor](#) o operador `+`. Quando um operador `+` binário é sobrepor, o operador `+=` também é implicitamente sobrepor. Um tipo definido pelo usuário não pode sobrepor 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 -= (referência do C#)

21/01/2022 • 3 minutes to read

Os operadores e são suportados pelos tipos numéricos integrais e flutuantes e `-` `-=` tipos delegados integrados.

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`.

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("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 apropriada da lista de invocação do operando à esquerda, o resultado da operação será o operando à esquerda. Por exemplo, a remoção de um delegado que não faz parte do delegado multicast não tem consequências, e o delegado multicast permanece inalterado.

```

Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("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.WriteLine("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 pela 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.

```

Action a = () => Console.WriteLine("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

```
x -= y
```

é equivalente a

```
x = x - y
```

exceto que `x` é avaliado apenas uma vez.

O exemplo a seguir demonstra o uso do operador `-=`:

```
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, [consulte 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 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 ?: (referência do C#)

21/01/2022 • 3 minutes to read

O operador condicional , também conhecido como operador condicional ternary, avalia uma expressão booliana e retorna o resultado de uma das duas expressões, dependendo se a expressão booliana é avaliada como ou , como mostra o exemplo `?:` a `true` `false` seguir:

```
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:

```
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 do tipo de destino. Ou seja, se um tipo de destino de uma expressão condicional for conhecido, os tipos de e deverão ser implicitamente conversíveis para o tipo de destino, como mostra o exemplo a `consequent` `alternative` seguir:

```
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) ou no `var` C# 8.0 e anterior, o tipo de e deve ser o mesmo ou deve haver uma conversão implícita de um tipo para o `consequent` `alternative` outro:

```
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

```
a ? b : c ? d : e
```

é avaliada como

```
a ? b : (c ? d : e)
```

TIP

Você pode usar o seguinte dispositivo mnemônico para se lembrar de como o operador condicional é avaliado:

```
is this condition true ? yes : no
```

Expressão condicional ref

A partir do C# 7.2, uma [variável local](#) ref local ou [ref readonly](#) pode ser atribuída condicionalmente com uma expressão ref condicional. Você também pode usar uma expressão ref condicional como um valor [de retorno de referência](#) ou como um argumento de [ref](#) método.

A sintaxe de uma expressão ref condicional é a seguinte:

```
condition ? ref consequent : ref alternative
```

Como o operador condicional original, uma expressão ref condicional avalia apenas uma das duas expressões:

`consequent` ou `alternative`.

No caso de uma expressão ref condicional, o tipo de `consequent` e deve ser o `alternative` mesmo. As expressões ref condicionais não são de tipo de destino.

O exemplo a seguir demonstra o uso de uma expressão ref condicional:

```
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
```

Operador condicional e uma instrução `if`

O uso do operador condicional `if` em vez de uma instrução pode resultar em código mais conciso nos casos em que você precisa calcular condicionalmente um valor. O exemplo a seguir demonstra duas maneiras de classificar um inteiro como negativo ou não negativo:

```
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#](#).

Para obter mais informações sobre os recursos adicionados no C# 7.2 e posterior, consulte as seguintes notas sobre a proposta de recurso:

- [Expressões ref condicionais \(C# 7.2\)](#)
- [Expressão condicional com tipo de destino \(C# 9.0\)](#)

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Instrução if](#)
- [Operadores ?. e ?\[\]](#)
- [?? E?? = operadores](#)
- [ref keyword](#)

! Operador (null-null) (referência de C#)

21/01/2022 • 2 minutes to read

Disponível no C# 8.0 e posterior, o operador de pós-sufixo unário é o operador `!` null-null-suppression ou null-suppression. Em um contexto [deanotação](#) que permite valor nulo habilitado, você usa o operador de anulação nula para declarar que a expressão de um tipo de referência não é: `x null x!`. O operador de `!` prefixo unário é o [operador de negação lógica](#).

O operador de anulação nula não tem nenhum efeito em tempo de operação. Ela afeta apenas a análise de fluxo estático do compilador alterando o estado nulo da expressão. Em tempo de executar, `x!` a expressão é avaliada como o resultado da expressão subjacente `x`.

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 de anulação nula está em testar a lógica de validação de argumento. Por exemplo, considere a seguinte classe:

```
#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 do MSTest](#), você pode criar o seguinte teste para a lógica de validação no construtor:

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Sem o operador null-operator, o compilador gera o seguinte aviso para o código anterior:

`Warning CS8625: Cannot convert null literal to non-nullable reference type`. Usando o operador null-operator, você informa ao compilador que a passagem é esperada e `null` não deve ser avisada.

Você também pode usar o operador null-operator quando você definitivamente sabe que uma expressão não pode ser, mas o compilador não consegue `null` reconhecer isso. No exemplo a seguir, se o método retornar, seu argumento não será e `IsValid` `true` você poderá `null` desreferenciar com segurança:

```

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 null-operator, o compilador gera o seguinte aviso para o `p.Name` código:

`Warning CS8602: Dereference of a possibly null reference`.

Se você puder modificar o método, poderá usar o atributo `IsValid NotNullWhen` para informar ao compilador que um argumento do método não pode ser quando o `IsValid` método retorna `null true`:

```

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 null-operator porque o compilador tem informações suficientes para descobrir que não pode estar `p null` dentro da `if` instrução. Para obter mais informações sobre os atributos que permitem que você forneça informações adicionais sobre o estado nulo de uma variável, consulte Atualizar APIs com atributos para definir expectativas [nulas](#).

Especificação da linguagem C#

Para obter mais informações, consulte a seção [O operador de anulação](#) nula do rascunho da especificação de tipos de referência que podem ser [anuladas](#).

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Tutorial: Design com tipos de referência que podem ser anulados](#)

?? E ?? Operadores = (referência de C#)

21/01/2022 • 2 minutes to read

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.

Disponível no C# 8.0 e posterior, o operador de atribuição de coalescing nulo atribuirá o valor de seu operand à direita ao seu operand esquerdo somente se o operador esquerdo for avaliada como ??= null . O operador ??= não avaliará o operando do lado direito se o operando esquerdo for avaliado como não nulo.

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers)); // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers)); // output: 5 0
Console.WriteLine(a); // output: 0
```

O operand esquerdo do operador deve ??= ser uma variável, uma propriedade ou um elemento indexador.

No C# 7.3 e anteriores, o tipo do operand esquerdo do operador deve ser um tipo de referência ou um tipo de valor que pode ser ?? anulado. A partir do C# 8.0, esse requisito é substituído pelo seguinte: o tipo do operand esquerdo dos operadores e não pode ser um tipo de valor não ?? ??= anulado. Em particular, começando com o C# 8.0, você pode usar os operadores de coalização nula com parâmetros de tipo irrestrictos:

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

Os operadores de coalescing nulos são associativos à direita. Ou seja, expressões do formulário

```
a ?? b ?? c
d ??= e ??= f
```

são avaliados como

```
a ?? (b ?? c)
d ??= (e ??= f)
```

Exemplos

Os ?? ??= operadores e podem ser úteis nos seguintes cenários:

- Em expressões com operadores condicionais nulos ?. e ?[], você pode usar o operador para fornecer uma expressão alternativa a ser avaliada caso o resultado da expressão com operações condicionais nulas seja ?? null :

```
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 trabalha com tipos de valor que podem ser anulados e precisa fornecer um valor de um tipo de valor subjacente, use o operador para especificar o valor a ser especificado caso um valor de tipo que ?? anulável seja null :

```
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.

- A partir do C# 7.0, você pode usar uma expressão como o operand direito do operador para tornar o código de verificação de argumento throw mais ?? conciso:

```
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.

- Começando com o C# 8.0, você pode usar o ??= operador para substituir o código do formulário

```
if (variable is null)
{
    variable = expression;
}
```

pelo código a seguir:

```
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 coalescing nulo da [especificação da linguagem C#](#).

Para obter mais informações sobre o ??= operador , consulte a nota de proposta de [recurso](#).

Confira também

- Referência de C#
- Operadores e expressões C#
- Operadores ?. e ?[]
- Operador ?:

Operador => (referência do C#)

21/01/2022 • 2 minutes to read

O token tem suporte em 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 => de expressão.

Operador lambda

Em expressões lambda, o operador lambda separa os parâmetros de entrada no 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:

```
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 declarações de tipo. Se você precisar especificar o tipo de parâmetros de entrada, deverá fazer isso para cada parâmetro, como mostra o exemplo a seguir:

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product); // output: 280
```

O exemplo a seguir mostra como definir uma expressão lambda sem parâmetros de entrada:

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

Para obter mais informações, consulte [Expressões Lambda](#).

Definição de corpo da expressão

Uma definição de corpo da expressão tem a seguinte sintaxe geral:

```
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 `void` tipo de retorno ou
- É um:

- Construtor
- Finalizer
- Acessador de propriedade `set` 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`:

```
public override string ToString() => $"{fname} {lname}".Trim();
```

É uma versão abreviada da seguinte definição de método:

```
public override string ToString()
{
    return $"{fname} {lname}".Trim();
}
```

Há suporte para definições de corpo de expressão para métodos, operadores e propriedades somente leitura a partir do C# 6. Há suporte para definições de corpo de expressão para construtores, finalizadores e acessadores de propriedade e indexador a partir do C# 7.0.

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, consulte a [seção Expressões de função anônima](#) da [especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)

Operador :: (referência do C#)

21/01/2022 • 2 minutes to read

Use o qualificador de alias de namespace :: para acessar um membro de um namespace com alias. Você só pode usar :: o qualificador entre dois identificadores. O identificador à esquerda pode ser qualquer um dos seguintes aliases:

- Um alias de namespace criado com uma diretiva using alias:

```
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 extern.
- 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:

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

NOTE

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 um membro de um namespace com alias. No entanto, . o token também é usado para acessar um membro de 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 (referência de C#)

21/01/2022 • 3 minutes to read

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 é aplicado ao operador que representa uma operação já concluída, ele retorna o resultado da operação imediatamente sem suspensão do `await` método delimitar. O operador `await` não bloqueia o thread que avalia o método assíncrono. Quando o `await` operador suspende o método assíncrono delimitador, o controle retorna 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.

```
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://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocs MainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}

// Output similar to:
// DownloadDocs MainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocs MainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

O exemplo anterior usa o [método assíncrono `Main`](#), que é possível a partir do C# 7.1. Para obter mais informações, confira a seção [Operador await no método Main](#).

NOTE

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`. Dentro de um método assíncrono, não é possível usar o operador no corpo de uma função síncrona, dentro do bloco de uma instrução lock e em um contexto não `await` seguro.

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. Para obter mais informações sobre o tratamento de exceções, confira a seção [Exceções em métodos assíncronos](#) do artigo [Instrução try-catch](#).

As `async` `await` palavras-chave estão disponíveis no C# 5 e posterior.

Fluxos assíncronos e descartáveis

A partir do C# 8.0, você pode trabalhar com fluxos e descartáveis assíncronos.

Use a `await foreach` instrução para consumir um fluxo assíncrono de dados. Para obter mais `foreach` informações, consulte a seção `statement` do artigo [Instruções de iteração](#) e a seção [Fluxos assíncronos](#) do artigo [Novidades no C# 8.0](#).

Você usa a instrução para trabalhar com um `await using` objeto descartável assíncrono, ou seja, um objeto de um tipo que implementa uma `IAsyncDisposable` interface. Para obter mais informações, consulte a seção [Using async descartável](#) do artigo [Implementar um método DisposeAsync](#).

Operador `await` no método `Main`

A partir do C# 7.1, o método `Main`, que é o ponto de entrada do aplicativo, pode retornar ou , permitindo que ele seja assíncrono para que você possa usar o operador em `Task` `Task<int>` seu `await` 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 obter informações sobre como selecionar a versão do idioma, consulte [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](#)

- Assincronia detalhada
- Passo a passo: acessando a Web usando async e await
- Tutorial: Gerar e consumir fluxos assíncronos usando o C# 8.0 e o .NET Core 3.0

expressões de valor padrão (referência de C#)

21/01/2022 • 2 minutes to read

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 `default` palavra-chave como o rótulo de caso padrão dentro de 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:

```
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

A partir do C# 7.1, você pode usar o literal `default` para produzir o valor padrão de um tipo quando o compilador pode 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 `return` instrução ou como uma expressão em um membro apto para expressão.

O exemplo a seguir mostra o uso do literal `default`:

```
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) ]
```

Especificação da linguagem C#

Para saber mais, confira a seção [Expressões de valor padrão](#) da [Especificação da linguagem C#](#).

Para obter mais informações sobre o literal `default`, confira a [nota da proposta do recurso](#).

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Valores padrão de tipos C#](#)
- [Generics in .NET \(Genéricos no .NET\)](#)

operador delegate (referência do C#)

21/01/2022 • 2 minutes to read

O operador `delegate` cria um método anônimo que pode ser convertido em um tipo delegado:

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4)); // output: 7
```

NOTE

Começando com o C# 3, as expressões lambda fornecem uma maneira mais concisa e expressiva de criar uma função anônima. Use o `operador =>` para construir uma expressão lambda:

```
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:

```
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 os [Descartes](#) para especificar dois ou mais parâmetros de entrada de um método anônimo que não são usados pelo método:

```
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 único parâmetro for nomeado `_`, `_` será tratado como o nome desse parâmetro dentro de um método anônimo.

Além de começar com o C# 9,0, você pode usar o `static` modificador na declaração de um método anônimo:

```
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 estado de instância de escopos delimitadores.

Você também usa a palavra-chave `delegate` para declarar um [tipo delegado](#).

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 de>](#)

operador is (referência C#)

21/01/2022 • 2 minutes to read

O `is` operador verifica se o resultado de uma expressão é compatível com um determinado tipo. Para obter informações sobre o operador de teste de tipo `is`, consulte a seção [is Operator](#) do artigo [Type-Test and Cast Operators](#).

A partir do C# 7,0, você também pode usar o `is` operador para corresponder uma expressão a um padrão, como mostra o exemplo a seguir:

```
static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

No exemplo anterior, o `is` operador corresponde a uma expressão em relação a um [padrão de propriedade](#) (disponível em c# 8,0 e posterior) com [constantes](#) aninhadas e [relacionais](#) (disponíveis em c# 9,0 e posterior) padrões.

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:

```
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:

```
if (input is null)
{
    return;
}
```

Quando você corresponde a uma expressão `null`, o compilador garante que nenhum operador ou sobrecarga de usuário `==` `!=` seja invocado.

- 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:

```
if (result is not null)
{
    Console.WriteLine(result.ToString());
}
```

NOTE

Para obter a lista completa de padrões com suporte pelo `is` operador, consulte [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 a correspondência de padrões para criar algoritmos orientados a dados e baseados em tipos](#)
- [Operadores cast e teste de tipo](#)

expressão nameof (referência C#)

21/01/2022 • 2 minutes to read

Uma `nameof` expressão produz o nome de uma variável, tipo ou membro como a constante de cadeia de caracteres:

```
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
```

Como o exemplo anterior mostra, no caso de um tipo e um namespace, o nome produzido não é **totalmente qualificado**.

No caso de [identificadores textuais](#), o `@` caractere não é a parte de um nome, como mostra o exemplo a seguir:

```
var @new = 5;
Console.WriteLine(nameof(@new)); // output: new
```

Uma `nameof` expressão é avaliada em tempo de compilação e não tem nenhum efeito no tempo de execução.

Você pode usar uma `nameof` expressão para tornar o código de verificação de argumento mais passível de manutenção:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");
}
```

Uma `nameof` expressão está disponível no C# 6 e posterior.

Especificação da linguagem C#

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

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [CallerArgumentExpression](#)

operador new (Referência em C#)

21/01/2022 • 3 minutes to read

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`:

```
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:

```
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
```

A partir do C# 9.0, as expressões de invocação de Construtor são de tipo de destino. Ou seja, se um tipo de destino de uma expressão for conhecido, você poderá omitir um nome de tipo, como mostra o exemplo a seguir:

```
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 de tipo de destino `new`.

Se um tipo de destino de uma `new` expressão for desconhecido (por exemplo, ao usar a `var` palavra-chave),

você 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:

```
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:

```
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:

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Destrução 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. Instâncias de tipos de referência são destruídas pelo [coletor de lixo](#) em algum momento não especificado depois que a última referência a eles é removida.

Para instâncias de tipo que contêm recursos não gerenciados, por exemplo, um identificador de arquivo, é recomendável empregar uma limpeza determinística para garantir que os recursos que eles contêm 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`, consulte a [Nota de proposta de](#)

recurso.

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Inicializadores de objeto e coleção](#)

Operador sizeof (referência em C#)

21/01/2022 • 2 minutes to read

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`:

```

using System;

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#)

21/01/2022 • 3 minutes to read

Uma `stackalloc` expressão 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. Você não pode liberar explicitamente a memória alocada com `stackalloc`. Um bloco de memória alocado de pilha não está sujeito à coleta de lixo e não precisa ser fixado com uma `fixed` instrução.

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

- Começando com o C# 7.2 ou `System.Span<T>`, como mostra o exemplo a `System.ReadOnlySpan<T>` seguir:

```
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:

```
int length = 1000;
Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
```

A partir do C# 8.0, você pode usar uma expressão dentro de outras expressões sempre que uma variável ou é permitida, como mostra o `stackalloc` exemplo a `Span<T> ReadOnlySpan<T>` seguir:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

NOTE

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:

```
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 `stackalloc` expressão somente 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 código a seguir:

```
const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new
byte[inputLength];
```

NOTE

Como a quantidade de memória disponível na pilha depende do ambiente no qual o código é executado, seja conservadora quando você definir o valor de limite real.

- Evite usar `stackalloc` loops de dentro. Aloce o bloco de memória fora de um loop e reutiliza-o dentro do loop.

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

A partir do C# 7.3, você pode usar a sintaxe do inicializador de matriz para definir o conteúdo da memória recém-alocada. O seguinte exemplo demonstra várias maneiras de fazer isso:

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

Na expressão `stackalloc T[E]`, deve ser um tipo `T` [não gerenciado](#) e deve ser avaliada como `E` 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 Alocação](#) de pilha da especificação da linguagem C# e a observação da proposta de recurso Permitir em `stackalloc` [contextos](#) aninhados.

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Operadores relacionados a ponteiro](#)
- [Tipos de ponteiro](#)
- [Tipos relacionados a memória e extensão](#)
- [Dos e Dons de stackalloc](#)

expressão switch (referência de C#)

21/01/2022 • 3 minutes to read

A partir do C# 8.0, você usa a expressão para avaliar uma única expressão de uma lista de expressões candidatas com base em uma combinação de padrões com uma `switch` expressão de entrada. Para obter informações sobre a `switch` instrução que dá `switch` suporte à semântica -like `switch` em um contexto de instrução, consulte a seção statement do [artigo Instruções de seleção](#).

O exemplo a seguir demonstra uma expressão, que converte valores de um que representa as direções visuais em um mapa online para as `switch` `enum` direções cardinais correspondentes:

```
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 `switch` expressão:

- Uma expressão seguida pela `switch` palavra-chave. No exemplo anterior, é o parâmetro `direction` de método.
- Os `switch` braços da expressão, separados por vírgulas. Cada `switch` arm de expressão contém um padrão, uma proteção de `caso opcional`, o token e uma `=>` expressão.

No exemplo anterior, uma expressão `switch` usa os seguintes padrões:

- Um [padrão constante](#): para manipular os valores definidos da `Direction` enumeração.
- Um [padrão de descarte](#): para manipular qualquer valor inteiro que não tenha o membro correspondente da `Direction` enumeração (por exemplo, `(Direction)10`). Isso torna a `switch` expressão [exaustiva](#).

IMPORTANT

Para obter informações sobre os padrões com suporte pela `switch` expressão e mais exemplos, consulte [Padrões](#).

O resultado de uma expressão é o valor da expressão do primeiro arm de expressão cujo padrão corresponde à expressão de entrada e cuja proteção de caso, se `switch switch` presente, é avaliada como `true`. Os `switch` braços da expressão são avaliados em ordem de texto.

O compilador gera um erro quando um arm de expressão inferior não pode ser escolhido porque um arm de expressão superior `switch` corresponde a todos os seus `switch` valores.

Proteção de caso

Um padrão pode não ser suficientemente expressiva para especificar a condição para a avaliação da expressão de um arm. Nesse caso, você pode usar um proteção de caso. Essa é uma condição adicional que deve ser atendida junto com um padrão de combinação. Um case guard deve ser uma expressão booliana. Especifique um case guard após `when` a palavra-chave que segue um padrão, como mostra o exemplo a seguir:

```
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 var [aninhados](#).

Expressões de opção não exaustivas

Se nenhum dos padrões `switch` de uma expressão corresponde a um valor de entrada, o runtime lança 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`. O compilador gerará um aviso se uma `switch` expressão não tratar todos os valores de entrada possíveis.

TIP

Para garantir que uma `switch` expressão trata todos os valores de entrada possíveis, forneça um arm de expressão com um padrão de `switch` [descarte](#).

Especificação da linguagem C#

Para obter mais informações, consulte a [switch](#) seção expressão da nota de proposta de recurso.

Confira também

- Referência de C#
- Operadores e expressões C#
- Padrões
- Tutorial: Usar a correspondência de padrões para criar algoritmos orientados por tipo e orientados a dados
- [switch](#) Declaração

Operadores true e false (referência do C#)

21/01/2022 • 2 minutes to read

O `true` operador retorna o valor `bool true` para indicar que seu operando é definitivamente verdadeiro. O `false` operador retorna o `bool false` valor para indicar que seu operando é definitivamente falso. Não há garantia de que os operadores `true` e `false` se complementarão. 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.

TIP

Use o `bool?` tipo, se você precisar dar 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 `true` operadores e `false` definido sobrecarregar o operador lógico `or` `|` ou o operador and lógico `&` de uma determinada maneira, o operador lógico condicional `or` `||` 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 `&` forma que o `&&` operador também possa ser avaliado para os operandos desse tipo.

```

using System;

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 método `GetFuelLaunchStatus` 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:

```
Getting fuel launch status...
Wait!
```

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)

expressão with (referência C#)

21/01/2022 • 3 minutes to read

Disponível em C# 9,0 e posterior, uma `with` expressão produz uma cópia de seu operando com as propriedades e os campos especificados modificados:

```
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 = "1.19" };
        Console.WriteLine($"original apples: {apples}");
        var saleApples = apples with { Price = "0.79" };
        Console.WriteLine($"sale apples: {saleApples}");

    }
}
```

Como mostra o exemplo anterior, você usa a sintaxe do [inicializador de objeto](#) para especificar quais membros modificar e seus novos valores.

No C# 9,0, um operando do lado esquerdo de uma `with` expressão deve ser de um [tipo de registro](#). A partir do C# 10, um operando à esquerda de uma `with` expressão também pode ser de um tipo de [estrutura](#) ou de um [tipo anônimo](#).

O resultado de uma `with` expressão tem o mesmo tipo de tempo de execução que o operando da expressão, como mostra o exemplo a seguir:

```

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 de tipo de referência, somente a referência a uma instância de membro é copiada quando um operando é copiado. A cópia e o operando original têm acesso à mesma instância de tipo de referência. O exemplo a seguir demonstra esse comportamento:

```

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*. Esse 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 `with` expressão, 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:

```

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 de estrutura.

Especificação da linguagem C#

Para obter mais informações, consulte as seções a seguir da [proposta de recurso de registros observação](#):

- [with expressão](#)
- [Copiar e clonar Membros](#)

Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Grava](#)
- [Tipos de estrutura](#)

Sobrecarga de operador (referência de C#)

21/01/2022 • 4 minutes to read

Um tipo definido pelo usuário pode sobrestrar um operador C# predefinido. Ou seja, um tipo pode fornecer a implementação personalizada de uma operação caso um ou ambos os operadores sejam desse tipo. A seção [Operadores sobrestragáveis](#) mostra quais operadores do C# podem ser sobrestrados.

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 sobrestraga alguns dos [operadores aritméticos](#):

```

using System;

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` para `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 fornece informações sobre capacidade de sobrecarga de operadores do C#:

OPERADORES	CAPACIDADE DE SOBRECARGA
<code>+x, -x, !x, ~x, ++, --, true, false</code>	Esses operadores unários podem ser sobrecarregados.
<code>x + y, x - y, x * y, x / y, x % y, x & y, x y, x ^ y, x > << y, x > y, x == y, x != y, x < y, x > y, x = <= y, x > y</code>	Esses operadores binários podem ser sobrecarregados. Determinados operadores devem ser sobrecarregados em pares; para obter mais informações, consulte a observação após esta tabela.
<code>x && y, x y</code>	Operadores lógicos condicionais não podem ser sobrecarregados. No entanto, se um <code>true</code> ou <code>false</code> tipo com os operadores sobrecarregados e também sobrecarregar o operador ou de uma determinada maneira, o operador ou, respectivamente, poderá ser avaliado para os <code>&</code> , <code> </code> , <code>&&</code> ou <code> </code> operadores desse tipo. Para obter mais informações, veja a seção Operadores lógicos condicionais definidos pelo usuário na especificação da linguagem C# .
<code>um[i], a?[i]</code>	O acesso de elemento não é considerado um operador que pode ser sobrecarregado, mas você pode definir um indexador .
<code>(T)x</code>	O operador cast não pode ser sobrecarregado, mas você 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 .
<code>+=, -=, . . . , =, *=, /=, %=, &=, ^=, <<=, >>=</code>	Operadores de atribuição compostos não podem ser sobrecarregados explicitamente. No entanto, quando um operador binário estiver sobrecarregado, o operador de atribuição composto correspondente, se houver, também estará implicitamente sobrecarregado. Por exemplo, <code>+=</code> é avaliado usando <code>+</code> , que pode ser sobrecarregado.
<code>^x, x = y, x.y, x?.y, c ? t : f, x ?? y, x..y, x->y, =>, f(x), como, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with</code>	Esses operadores não podem ser sobrecarregados.

NOTE

Os operadores de comparação precisam ser sobrecarregados em pares. Ou seja, se o operador de um par está sobrecarregado, o outro operador precisa estar sobrecarregado também. Esses pares são os seguintes:

- Operadores `==` e `!=`
- Operadores `<` e `>`
- Operadores `<=` e `>=`

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 sobre carregados sempre são estáticos em C#?](#)

Instruções de iteração (referência de C#)

21/01/2022 • 7 minutes to read

As instruções a seguir executam repetidamente uma instrução ou um bloco de instruções:

- A `for` instrução: executa seu corpo enquanto uma expressão booliana especificada é avaliada como `true`.
- A `foreach` instrução: enumera os elementos de uma coleção e executa seu corpo para cada elemento da coleção.
- A `do` instrução: executa condicionalmente seu corpo uma ou mais vezes.
- A `while` instrução: executa condicionalmente seu corpo zero ou mais vezes.

A qualquer momento dentro do corpo de uma instrução de iteração, você pode sair do loop usando a instrução `break` ou ir 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 booliana especificada é avaliada como `true`. O exemplo a seguir mostra `for` a instrução que executa seu corpo enquanto um contador inteiro é menor que três:

```
for (int i = 0; i < 3; i++)
{
    Console.WriteLine(i);
}
// Output:
// 012
```

O exemplo anterior mostra os elementos da `for` instrução :

- A *seção do inicializador* executada apenas uma vez, antes de inserir o loop. Normalmente, você declara e inicializa uma variável de loop local nessa seção. A variável declarada não pode ser acessada de fora da `for` instrução .

A *seção inicializador* no exemplo anterior declara e inicializa uma variável de contador de inteiros:

```
int i = 0
```

- A *seção de condição* que determina se a próxima iteração no loop deve ser executada. Se for avaliada como `true`, a próxima iteração será executada; caso contrário, o loop será fechado. A *seção condition* deve ser uma expressão booliana.

A *seção condição* no exemplo anterior verifica se um valor de contador é menor que três:

```
i < 3
```

- A *seção iterador* que define o que acontece após cada execução do corpo do loop.

A *seção do iterador* no exemplo anterior incrementa o contador:

```
i++
```

- O corpo do loop, que deve ser uma instrução ou 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 você não declarar uma variável de loop na seção inicializador, poderá usar zero ou mais expressões da lista anterior na seção inicializador também. O exemplo a seguir 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:

```
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 `for` instrução são opcionais. Por exemplo, o código a seguir define o `for` loop infinito:

```
for ( ; ; )
{
    //...
}
```

A instrução `foreach`

A instrução executa uma instrução ou um bloco de instruções para cada elemento em uma instância do tipo que implementa a interface ou , como mostra o `foreach` exemplo a [System.Collections.IEnumerable](#) [System.Collections.Generic.IEnumerable<T>](#) seguir:

```
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 `foreach` instrução não está limitada a esses tipos. Você pode usá-lo com uma instância de qualquer tipo que atenda às seguintes condições:

- Um tipo tem o método público sem `GetEnumerator` parâmetros. A partir do C# 9.0, o `GetEnumerator` método

pode ser o método de extensão de um [tipo](#).

- O tipo de retorno do `GetEnumerator` método tem a propriedade pública e o método público sem `Current` `MoveNext` parâmetros cujo tipo de retorno é `bool`.

O exemplo a seguir usa `foreach` a instrução com uma instância `System.Span<T>` do tipo , que não implementa nenhuma interface:

```
Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
foreach (int number in numbers)
{
    Console.WriteLine($"{number} ");
}
// Output:
// 3 14 15 92 6
```

A partir do C# 7.3, se a propriedade do enumerador retornar um valor de retorno de referência (em que é o tipo de um elemento de coleção), você poderá declarar uma variável de iteração com o modificador ou , como mostra o exemplo `Current` `ref T` a `T` `ref` `ref readonly` seguir:

```
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 estiver vazia, o corpo da `foreach` `foreach` instrução não será executado e ignorado.

await foreach

A partir do C# 8.0, você pode usar a instrução para consumir um fluxo assíncrono de dados, ou seja, o tipo de coleção que implementa `await foreach` a `IAsyncEnumerable<T>` interface. Cada iteração do loop pode ser suspensa enquanto o próximo elemento é recuperado de forma assíncrona. O exemplo a seguir mostra como usar a `await foreach` instrução :

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

Você também pode usar a `await foreach` instrução com uma instância de qualquer tipo que atenda às seguintes condições:

- Um tipo tem o método público sem `GetAsyncEnumerator` parâmetros. Esse método pode ser o método de extensão [de um tipo](#).
- O tipo de retorno do método tem a propriedade pública e o método público sem parâmetros cujo tipo de retorno é , ou qualquer outro tipo a espera cujo método `awaiter` retorna `GetAsyncEnumerator` `Current` um `MoveNextAsync` `Task<bool>` `ValueTask<bool>` `GetResult` `bool` valor.

Por padrão, os elementos de fluxo são processados no contexto capturado. Se você quiser desabilitar a captura

do contexto, use o método `TaskAsyncEnumerableExtensions.ConfigureAwait` de extensão. Para obter mais informações sobre contextos de sincronização e capturar o contexto atual, consulte Consumindo o padrão assíncrono baseado em tarefa. Para obter mais informações sobre fluxos assíncronos, consulte a seção [Fluxos assíncronos](#) do artigo [Novidades no C# 8.0](#).

Tipo de uma variável de iteração

Você pode usar a `var` palavra-chave para permitir que o compilador infera o tipo de uma variável de iteração na instrução `foreach`, como mostra o código a seguir:

```
foreach (var item in collection) { }
```

Você também pode especificar explicitamente o tipo de uma variável de iteração, como mostra o código a seguir:

```
IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }
```

Na forma anterior, o tipo de um elemento de coleção deve ser implicitamente ou explicitamente conversível no tipo `T` `V` de uma variável de iteração. Se uma conversão explícita de `T` para falhar em tempo de `V` executar, a `foreach` instrução lançará um [InvalidCastException](#). Por exemplo, se for um tipo de classe não lacrado, poderá ser qualquer tipo de interface, mesmo aquele `T` `V` que não `T` implementa. Em tempo de execução, o tipo de um elemento de coleção pode ser aquele que deriva de `T` e, na verdade, implementa `V`. Se esse não for o caso, um [InvalidCastException](#) será lançado.

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. Isso é diferente de um loop [while](#), que executa zero ou mais vezes.

O exemplo a seguir mostra o uso da `do` instrução :

```
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. Isso é diferente de um loop `do`, que é executado uma ou mais vezes.

O exemplo a seguir mostra o uso da `while` instrução :

```
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#](#):

- A `for` instrução
- A `foreach` instrução
- A `do` instrução
- A `while` instrução

Para obter mais informações sobre os recursos adicionados no C# 8.0 e posterior, confira as seguintes notas sobre a proposta de recurso:

- [Fluxos assíncronos \(C# 8.0\)](#)
- [Suporte `GetEnumerator` de extensão para `foreach` loops \(C# 9.0\)](#)

Confira também

- [Referência de C#](#)
- [Usando foreach com matrizes](#)
- [Iteradores](#)

Instruções de seleção (referência de C#)

21/01/2022 • 5 minutes to read

As instruções a seguir selecionam instruções a executar de várias instruções possíveis com base no valor de uma expressão:

- A `if` instrução: seleciona uma instrução a ser executada com base no valor de uma expressão booliana.
- A `switch` instrução: seleciona uma lista de instrução a ser executada com base em uma combinação de padrões com uma expressão.

A instrução `if`

Uma `if` instrução pode ser qualquer uma das duas formas a seguir:

- Uma instrução com uma parte seleciona uma das duas instruções a executar com base no valor de uma expressão `if` `else` booliana, como mostra o exemplo a seguir:

```
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 `if` instrução sem uma parte executará seu corpo somente se uma expressão booliana for avaliada como `true`, como mostra o exemplo a seguir:

```
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.Write("Warning: not acceptable value! ");  
    }  
  
    Console.WriteLine($"The measurement value is {value}");  
}
```

Você pode `if` aninhar instruções para verificar várias condições, como mostra o exemplo a seguir:

```

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 seleciona uma lista de instrução a ser executada com base em uma combinação de padrões com uma expressão de match, como mostra o exemplo a `switch` seguir:

```

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 `switch` instrução usa os seguintes padrões:

- Um [padrão relacional](#): para comparar um resultado de expressão com uma constante.
- Um [padrão constante](#): para testar se um resultado de expressão é igual a uma constante.

IMPORTANT

Para obter informações sobre os padrões com suporte da `switch` instrução, consulte [Padrões](#).

O exemplo anterior também demonstra o `default` caso. O `default` caso especifica instruções a ser executadas quando uma expressão de match não corresponder a nenhum outro padrão de caso. Se uma expressão de match não corresponder a nenhum padrão de caso e não houver `default` nenhum caso, o controle se enquadra em uma `switch` instrução.

Uma instrução executa a lista de instrução na primeira seção `switch` cujo padrão de caso corresponde a uma expressão de match e cujo case guard , se `switch` presente, é avaliada como `true` . Uma `switch` instrução avalia os padrões de caso na ordem de texto de cima para baixo. O compilador gera um erro quando uma `switch` instrução contém um caso inacessível. Esse é um caso que já é tratado por um caso superior ou cujo padrão é impossível de corresponder.

NOTE

O `default` caso pode aparecer em qualquer lugar dentro de uma `switch` instrução . Independentemente de sua posição, o caso sempre será avaliado por último e somente se todos os outros padrões de caso `default` não corresponderem.

Você pode especificar vários padrões de caso para uma seção de `switch` uma instrução, como mostra o exemplo a seguir:

```
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;
    }
}
```

Em uma `switch` instrução , o controle não pode passar de uma seção de opção para a próxima. Como mostram os exemplos nesta seção, normalmente você usa a instrução no final de cada seção `switch` para passar `break` o controle de uma `switch` instrução. Você também pode usar as [instruções return](#) e [throw](#) para passar o controle de uma `switch` instrução . Para imitar o comportamento de fall-through e passar o controle para outra seção de opção, você pode usar a [goto](#) instrução.

Em um contexto de expressão, você pode usar a expressão para avaliar uma única expressão de uma lista de expressões candidatas com base em uma combinação de padrões com uma expressão. `switch`

Proteção de caso

Um padrão de caso pode não ser suficientemente expressiva para especificar a condição para a execução da seção switch. Nesse caso, você pode usar um *case guard*. Essa é uma condição adicional que deve ser atendida junto com um padrão de combinação. Um case guard deve ser uma expressão booleana. Especifique um case guard após `when` a palavra-chave que segue um padrão, como mostra o exemplo a seguir:

```
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.

Suporte à versão do idioma

A `switch` instrução dá suporte à correspondência de padrões começando com o C# 7.0.

No C# 6 e anterior, você usa a `switch` instrução com as seguintes limitações:

- Uma expressão de match deve ser de um dos seguintes tipos: `char`, `string`, `bool`, um tipo numérico integral ou um tipo de `enum`.
- Somente expressões constantes são permitidas em `case` rótulos.

Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- A `if` instrução
- A `switch` instrução

Para obter mais informações sobre os recursos introduzidos no C# 7.0 e posterior, confira as seguintes notas sobre a proposta de recurso:

- [Instrução Switch \(Correspondência de padrões para C# 7.0\)](#)

Confira também

- [Referência de C#](#)
- [Operador condicional `?:`](#)
- [Operadores lógicos](#)
- [Padrões](#)
- [switch Expressão](#)

Instruções jump (referência de C#)

21/01/2022 • 6 minutes to read

As instruções a seguir transferem incondicionalmente o controle:

- A `break` instrução: encerra a instrução ou instrução de iteração delimitador `switch` mais próxima.
- A `continue` instrução: inicia uma nova iteração da instrução de iteração delimitada mais próxima.
- A `return` instrução: encerra a execução da função na qual ela aparece e retorna o controle para o chamador.
- A `goto` instrução: transfere o controle para uma instrução que é marcada por um rótulo.

Para obter informações sobre a instrução que lança uma exceção e transfere incondicionalmente o controle `throw` também, consulte [throw](#).

A instrução `break`

A `break` instrução encerra a instrução de iteração delimitador mais próxima (ou seja, `for`, ou `foreach`, `while` ou `do` loop) ou `switch` instrução. A `break` instrução transfere o controle para a instrução que segue a instrução terminada, se for o caso.

```
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 encerra apenas o loop mais interno que o `break` contém, como mostra o exemplo a seguir:

```

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 dentro de um loop, uma instrução no final de uma seção switch transfere o controle `switch` somente para fora da `break` `switch` instrução . O loop que contém a `switch` instrução não é afetado, como mostra o exemplo a seguir:

```

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 inicia uma nova iteração da instrução de `continue` [iteração](#) delimitada mais próxima (ou seja, , , ou loop), como mostra o `for` exemplo a `foreach` `while` `do` seguir:

```
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 encerra a execução da função na qual ela aparece e retorna o controle e o resultado da função, se for o `return` caso, para o chamador.

Se um membro da função não computar um valor, você usará a instrução `return` sem expressão, como mostra o exemplo a seguir:

```
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 no início. Se um membro da função não contém a `return` instrução , ele é encerrado depois que sua última instrução é executada.

Se um membro da função computar um valor, você usará a `return` instrução com uma expressão, como mostra o exemplo a seguir:

```

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 tem uma expressão, essa expressão deve ser implicitamente conversível para o tipo de retorno de um membro de função, a menos que `return` seja [assíncrona](#). No caso de uma função, a expressão deve ser implicitamente conversível para o argumento de tipo de ou , o que for o tipo `async Task<TResult>` de retorno da `ValueTask<TResult>` função. Se o tipo de retorno de `async` uma função for ou , você usará a `Task ValueTask` `return` instrução sem expressão.

Por padrão, a `return` instrução retorna o valor de uma expressão. A partir do C# 7.0, você pode retornar uma referência a uma variável. Para fazer isso, use a `return` instrução com a `ref` palavra-chave, como mostra o exemplo a seguir:

```

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.");
}

```

Para obter mais informações sobre retornos de `ref`, consulte [Ref returns](#) e [ref locals](#).

A instrução `goto`

A `goto` instrução transfere o controle para uma instrução marcada por um rótulo, como mostra o exemplo a seguir:

```

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 `goto` instrução para sair de um loop aninhado.

TIP

Ao trabalhar com loops aninhados, considere a possibilidade de refatorar loops separados em métodos separados. Isso pode levar a um código mais simples e acessível sem a `goto` instrução.

Você também pode usar a instrução `break` na instrução para transferir o controle para uma seção `switch` com um rótulo de `goto` caso constante, como mostra o exemplo a seguir: `switch`

```

using System;

public enum CoffeChoice
{
    Plain,
    WithMilk,
    WithIceCream,
}

public class GotoInSwitchExample
{
    public static void Main()
    {
        Console.WriteLine(CalculatePrice(CoffeChoice.Plain)); // output: 10.0
        Console.WriteLine(CalculatePrice(CoffeChoice.WithMilk)); // output: 15.0
        Console.WriteLine(CalculatePrice(CoffeChoice.WithIceCream)); // output: 17.0
    }

    private static decimal CalculatePrice(CoffeChoice choice)
    {
        decimal price = 0;
        switch (choice)
        {
            case CoffeChoice.Plain:
                price += 10.0m;
                break;

            case CoffeChoice.WithMilk:
                price += 5.0m;
                goto case CoffeChoice.Plain;

            case CoffeChoice.WithIceCream:
                price += 7.0m;
                goto case CoffeChoice.Plain;
        }
        return price;
    }
}

```

Na `switch` instrução, você também pode usar a instrução `goto default;` para transferir o controle para a seção switch com o `default` rótulo.

Se um rótulo com o nome determinado não existir no membro da função atual ou se a instrução não estiver dentro do escopo do rótulo, ocorrerá um erro em tempo de `goto` compilação. Ou seja, você não pode usar a instrução para transferir o controle do membro da função atual ou para qualquer escopo `goto` aninhado, por exemplo, um `try` bloco.

Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- A `break` instrução
- A `continue` instrução
- A `return` instrução
- A `goto` instrução

Confira também

- [Referência de C#](#)
- [yield Declaração](#)

instrução lock (referência em C#)

21/01/2022 • 2 minutes to read

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` está no formato

```
lock (x)
{
    // Your code...
}
```

em que `x` é uma expressão de um [tipo de referência](#). Ela é precisamente equivalente a

```
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 um bloco `try...finally`, o bloqueio será liberado mesmo se uma exceção for gerada dentro do corpo de uma instrução `lock`.

Não é possível usar o [operador 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 os seguintes itens como objetos de bloqueio:

- `this`, uma vez que pode ser usado pelos chamadores como um bloqueio.
- Instâncias [Type](#), pois 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 o 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.

```
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 1000
    }
}
```

```
// 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](#)

Caracteres especiais de C#

21/01/2022 • 2 minutes to read

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 textual.
- `$`, o caractere de cadeia de caracteres interpolada.

Esta seção inclui apenas os tokens que não são operadores. Consulte a seção [operadores](#) para todos os operadores.

Confira também

- [Referência do C#](#)
- [Guia de programação C#](#)

\$ – interpolação de cadeia de caracteres (referência de C#)

21/01/2022 • 6 minutes to read

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. Esse recurso está disponível a partir do C# 6.

A interpolação de cadeia de caracteres fornece uma sintaxe mais acessível e conveniente para formatar cadeias de caracteres. É mais fácil de ler do que a [formatação de composição de cadeia de caracteres](#). Compare o exemplo a seguir que usa ambos os recursos para produzir a mesma saída:

```
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 `$`. Você não pode ter nenhum espaço em branco entre o `$` e o que inicia `$` um literal de cadeia de `"` caracteres.

A estrutura de um item com uma expressão de interpolação é da seguinte maneira:

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Os elementos entre colchetes são opcionais. A seguinte tabela descreve cada elemento:

ELEMENTO	DESCRIÇÃO
<code>interpolationExpression</code>	A expressão que produz um resultado a ser formatado. A representação de cadeia de <code>null</code> caracteres de é String.Empty .
<code>alignment</code>	A expressão constante cujo valor define o número mínimo de caracteres na 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 Componente de alinhamento .

ELEMENTO	DESCRIÇÃO
<code>formatString</code>	Uma cadeia de caracteres de formato compatível com o tipo do resultado da expressão. Para obter mais informações, consulte Componente da cadeia de caracteres de formato .

O exemplo a seguir usa os componentes opcionais de formatação descritos acima:

```
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 os espaço reservados devem ser cadeias de caracteres constantes. Em outras palavras, cada *expressão de interpolação* deve ser uma cadeia de caracteres e deve ser uma constante de tempo de compilação.

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 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:

```
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 textual interpolada começa com `$` o caractere seguido pelo caractere `@`. Para obter mais informações sobre cadeias de caracteres textual, consulte os [artigos de cadeia de caracteres](#) e [identificador textual](#).

NOTE

A partir do C# 8.0, você pode usar os tokens e em qualquer ordem: e são cadeias de `$` `@` `$@"..."` `@$"..."` caracteres textual interpoladas válidas. Em versões anteriores do C#, `$` o token deve aparecer antes do `@` token.

Conversões implícitas e como especificar a `IFormatProvider` implementação

Há três conversões implícitas de uma cadeia de caracteres interpolada:

1. Conversão de uma cadeia de caracteres interpolada em uma [String](#) instância. A cadeia de caracteres é o resultado da resolução de cadeia de caracteres interpolada. Todos os itens de expressão de interpolação são substituídos por representações de cadeia de caracteres formatadas corretamente de seus 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.
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:

```
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.
```

Recursos adicionais

Se você for novo na interpolação de cadeia de caracteres, confira o tutorial Interativo Interpolação de cadeia de caracteres em C#. Você também pode verificar outro [tutorial interpolação de cadeia de caracteres em 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 satisfaz o padrão de manipulador de cadeia de caracteres *interpolado*. Um *manipulador de cadeia de caracteres interpolado* é um tipo personalizado que converte a cadeia de caracteres interpolada em uma cadeia de caracteres. Um manipulador de cadeia de caracteres interpolado é um cenário avançado, normalmente usado por motivos de desempenho. Você pode aprender sobre os requisitos para criar um manipulador de cadeia de caracteres interpolado na especificação de linguagem para [melhorias de cadeia de caracteres interpoladas](#). Você pode criar um seguindo o tutorial do manipulador de cadeia de caracteres [interpolado](#) na seção Novidades em C#. No .NET 6, quando você usa uma cadeia de caracteres interpolada para um argumento do tipo , a cadeia de caracteres `string` interpolada é processada pelo [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#) .

NOTE

Um efeito colateral dos manipuladores de cadeia de caracteres interpolados é que um manipulador personalizado, incluindo , pode não avaliar todas as expressões usadas como espaço reservados na cadeia de caracteres interpolada em todas [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#) as condições. Isso significa que os efeitos colaterais nessas expressões podem não ocorrer.

Especificação da linguagem C#

Para obter mais informações, consulte a seção [cadeias de caracteres interpoladas](#) da [especificação da linguagem C#](#).

Confira também

- [Referência de C#](#)
- [Caracteres especiais de C#](#)
- [Cadeias de caracteres](#)
- [Cadeias de caracteres de formato numérico padrão](#)
- [Formatação composta](#)
- [String.Format](#)

@ (Referência de C#)

21/01/2022 • 3 minutes to read

O caractere especial `@` serve como um identificador textual. Ele pode ser usado das seguintes maneiras:

1. Para habilitar palavras-chave de C# a serem usadas 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`.

```
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!
```

2. 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. Apenas uma sequência de escape de aspas (`""`) não é interpretada literalmente; ela produz uma 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.

```
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.

```
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"
```

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`. Veja [CS1614](#) para obter mais informações.

```
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 do C#](#)

Atributos de nível de assembly interpretados pelo compilador C#

21/01/2022 • 2 minutes to read

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:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Os atributos globais aparecem no código-fonte após quaisquer diretivas de nível superior `using` e antes de qualquer tipo, módulo ou declaração de namespace. Os atributos globais podem aparecer em vários arquivos de origem, mas os arquivos devem ser compilados em uma única passagem de compilação. Visual Studio adiciona atributos globais ao arquivo `AssemblyInfo.cs` em projetos .NET Framework. Esses atributos não são adicionados aos 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, o Visual Studio IDE na [caixa de diálogo informações do assembly](#) ou o 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
AssemblyVersionAttribute	Especifica a versão de um assembly.
AssemblyCultureAttribute	Especifica a qual cultura o assembly dá suporte.
AssemblyFlagsAttribute	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
AssemblyProductAttribute	Especifica um nome de produto para um manifesto de assembly.
AssemblyTrademarkAttribute	Especifica uma marca registrada para um manifesto do assembly.
AssemblyInformationalVersionAttribute	Especifica uma versão informativa para um manifesto do assembly.
AssemblyCompanyAttribute	Especifica um nome de empresa para um manifesto do assembly.
AssemblyCopyrightAttribute	Define um atributo personalizado que especifica os direitos autorais para um manifesto do assembly.
AssemblyFileVersionAttribute	Define um número de versão específico para o recurso de versão de arquivo do Win32.
CLSCompliantAttribute	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
AssemblyTitleAttribute	Especifica um título de assembly para um manifesto de assembly.
AssemblyDescriptionAttribute	Especifica uma descrição de assembly para um manifesto de assembly.
AssemblyConfigurationAttribute	Especifica uma configuração de assembly (como varejo ou depuração) para um manifesto do assembly.
AssemblyDefaultAliasAttribute	Define um alias amigável padrão para um manifesto do assembly

Determinar informações do chamador usando atributos interpretados pelo compilador C#

21/01/2022 • 4 minutes to read

Usando atributos de informações, você obtém informações sobre o chamador para um método. Você Obtém o caminho do arquivo do código-fonte, o número da 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
CallerFilePathAttribute	O caminho completo do arquivo de origem que contém o chamador. O caminho completo é o caminho no momento da compilação.	String
CallerLineNumberAttribute	Número de linha no arquivo de origem do qual o método é chamado.	Integer
CallerMemberNameAttribute	Nome do método ou nome da propriedade do chamador.	String
CallerArgumentExpressionAttribute	Representação da cadeia de caracteres da expressão do argumento.	String

Essas informações ajudam no rastreamento e na depuração e ajudam você a criar ferramentas de diagnóstico. O exemplo a seguir mostra como usar atributos de informações do chamador. Em cada chamada para o `TraceMessage` método, as informações do chamador são inseridas para os argumentos para os parâmetros opcionais.

```
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 a 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 na IL (linguagem intermediária) no momento da 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`.

AS CHAMADAS OCORREM EM	RESULTADO DE NOME DE MEMBRO
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

Você usa o [System.Runtime.CompilerServices.CallerArgumentExpressionAttribute](#) quando deseja que a expressão seja passada como um argumento. As bibliotecas de diagnóstico podem querer fornecer mais detalhes sobre as *expressões* passadas aos 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:

```
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:

```
public void Operation(Action func)
{
    Utilities.ValidateArgument(nameof(func), func is not null);
    func();
}
```

A expressão usada para `condition` é injetada pelo compilador no `message` argumento. Quando um desenvolvedor chama `Operation` um `null` argumento, a seguinte mensagem é armazenada no `ArgumentException` :

```
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 o 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:

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence, int frequency,
[CallerArgumentExpression("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;
    }
}
```

Você pode chamar esse método da seguinte maneira:

```
sample = Enumerable.Range(0, 10).Sample(100);
```

O exemplo anterior lançaria um [ArgumentException](#) cuja mensagem é o seguinte texto:

```
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 interpretada pelo compilador C#

21/01/2022 • 15 minutes to read

Em um contexto habilitado para valor nulo, 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, habilitando um [System.NullReferenceException](#). Esses atributos fornecem ao compilador informações semânticas sobre o estado nulo de argumentos, valores de retorno e membros de objeto com base no estado de argumentos e valores de retorno. 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 que permitem valor nulo e como usá-los.

Vamos começar com um exemplo. Imagine sua biblioteca tem a API a seguir para recuperar uma cadeia de caracteres de recurso. Esse método foi originalmente escrito antes do C# 8.0 e das anotações que podem ser anuladas:

```
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 *estado nulo* desses parâmetros:

- Os chamadores não devem passar `null` como o argumento para `key`.
- Os chamadores podem passar uma variável cujo valor `null` é como o argumento para `message`.
- Se o `TryGetMessage` método retornar `true`, o valor de `message` não será `null`. Se o valor de retorno `false`, o valor de `message` é `null`.

A regra para `key` pode ser expressa de forma sucinta no C# 8.0: deve ser um tipo de referência não anulável. O `message` parâmetro é mais complexo. Ele permite uma variável que é como o argumento, mas garante, em caso `null` de êxito, que o `out` argumento não seja `null`. Para esses cenários, você precisa de um vocabulário mais rico para descrever as expectativas. O `NotNullWhen` atributo descrito abaixo descreve o estado *nulo* para o argumento usado para o `message` parâmetro.

NOTE

Adicionar esses atributos fornece ao compilador mais informações sobre as regras para sua API. Quando a chamada de código é compilada em um contexto habilitado para valor nulo, o compilador avisará os chamadores quando violarem essas regras. Esses atributos não habilitam mais verificações em sua implementação.

ATRIBUTO	CATEGORIA	SIGNIFICADO
AllowNull	Pré-condição	Um parâmetro, campo ou propriedade que não pode ser anulado pode ser nulo.
DisallowNull	Pré-condição	Um parâmetro, campo ou propriedade que pode ser anulada nunca deve ser nulo.
BeNull	Pós-condição	Um parâmetro não anulado, campo, propriedade ou valor de retorno pode ser nulo.
NotNull	Pós-condição	Um parâmetro, campo, propriedade ou valor de retorno que pode ser anulado nunca será nulo.
BeNullWhen	Pós-condição condicional	Um argumento não anulado pode ser nulo quando o método retorna o valor <code>bool</code> especificado.
NotNullWhen	Pós-condição condicional	Um argumento anulado não será nulo quando o método retornar o valor <code>bool</code> especificado.
NotNullIfNotNull	Pós-condição condicional	Um valor de retorno, propriedade ou argumento não será nulo se o argumento para o parâmetro especificado não for nulo.
MemberNotNull	Métodos auxiliares de propriedade e método	O membro listado não será nulo quando o método retornar.
MemberNotNullWhen	Métodos auxiliares de propriedade e método	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 lança uma exceção.
DoesNotReturnIf	Código inacessível	Esse método ou propriedade nunca retornará se o parâmetro <code>bool</code> associado tiver o valor especificado.

As descrições anteriores são uma referência rápida ao que cada atributo faz. As seções a seguir descrevem o comportamento e o significado desses atributos mais detalhadamente.

Pré-condições: `AllowNull` e `DisallowNull`

Considere uma propriedade de leitura/gravação que nunca retorna `null` porque ela tem um valor padrão razoável. Os chamadores `null` passam para o acessador `set` ao defini-lo com 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:

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

Quando você compila o código anterior em um contexto anulado, tudo está bem. Depois de habilitar tipos de referência que permitem valor nulo, `ScreenName` a propriedade se torna uma referência que não permite valor nulo. Isso está correto para o `get` acessador: ele nunca retorna `null`. Os chamadores não precisam verificar a propriedade retornada para `null`. Mas agora definir a propriedade `null` como gera um aviso. Para dar suporte a esse tipo de código, adicione [System.Diagnostics.CodeAnalysis.AllowNullAttribute](#) o atributo à propriedade , conforme mostrado no código a seguir:

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

Talvez seja necessário adicionar uma `using` diretiva para usar este e outros atributos [System.Diagnostics.CodeAnalysis](#) discutidos neste artigo. O atributo é aplicado à propriedade , não ao `set` acessador. O `AllowNull` atributo especifica as *pré-condições* e aplica-se somente a argumentos. O `get` acessador tem um valor de retorno, mas nenhum parâmetro. Portanto, `AllowNull` o atributo se aplica somente ao `set` acessador.

O exemplo anterior demonstra o que procurar ao adicionar o `AllowNull` atributo em um argumento:

1. O contrato geral para essa variável é que ela não deve ser , portanto, você deseja um tipo de referência `null` que não anulável.
2. Há cenários para um chamador passar como o argumento, embora eles `null` não sejam o uso mais comum.

Na maioria das vezes, você precisará desse atributo para propriedades ou `in` `out` argumentos , `ref` e . O atributo é a melhor opção quando uma variável normalmente não é nula, mas você precisa `AllowNull` permitir `null` como uma pré-condição.

Contraste que com cenários para usar : você usa esse atributo para especificar que um argumento de um tipo de referência que pode ser anulado `DisallowNull` não deve ser `null` . Considere uma propriedade em `null` que é o valor padrão, mas os clientes só podem defini-la como um valor não nulo. Considere o seguinte código:

```
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 de que o poderia `ReviewComment` ser , mas não pode ser definido como `null null` . Depois que esse código for anulado, você poderá expressar esse conceito mais claramente para os chamadores usando o `System.Diagnostics.CodeAnalysis.DisallowNullAttribute` :

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

Em um contexto que pode ser anulado, `ReviewComment` `get` o acessador pode retornar o valor padrão de `null` . O compilador avisa que ele deve ser verificado antes do acesso. Além disso, ele avisa os chamadores de que, embora possa ser , os chamadores não devem `null` defini-lo explicitamente como `null` . O `DisallowNull` atributo também especifica uma *pré-condição*, que não afeta o `get` acessador. Você usa o `DisallowNull` atributo ao observar estas características sobre:

1. A variável pode estar `null` em cenários principais, geralmente quando instaurou pela primeira vez.
2. A variável não deve ser definida explicitamente como `null` .

Essas situações são comuns no código que originalmente era *nulo*. Pode ser que as propriedades do objeto sejam definidas em duas operações de inicialização distintas. Pode ser que algumas propriedades sejam definidas somente depois que algum trabalho assíncrono é concluído.

Os atributos e permitem que você especifique que as pré-condições em variáveis podem não corresponder às anotações que permitem valor nulo `AllowNull` `DisallowNull` 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 que não permite valor nulo pode ser nulo.
- `DisallowNull`:um argumento que permite valor nulo nunca deve ser nulo.

Pós-condições: `MaybeNull` e `NotNull`

Suponha que você tenha um método com a seguinte assinatura:

```
public Customer FindCustomer(string lastName, string firstName)
```

Você provavelmente escreveu um método como este para retornar `null` quando o nome buscado não foi encontrado. O `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 de retorno como um tipo de referência que pode ser anulado especifica claramente a intenção dessa API:

```
public Customer? FindCustomer(string lastName, string firstName)
```

Por motivos [abordados em Nulidade de genéricos](#), essa técnica pode não produzir a análise estática que corresponde à api. Você pode ter um método genérico que segue um padrão semelhante:

```
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 quando um item não é encontrado adicionando a `null` `MaybeNull` anotação ao retorno do método:

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

O código anterior informa os chamadores de que o valor de *retorno pode realmente ser nulo*. Ele também informa ao compilador que o método pode retornar uma expressão, mesmo que o `null` tipo não seja anulado. Quando você tem um método genérico que retorna uma instância de seu parâmetro de tipo, , você pode expressar que ele nunca retorna `T null` usando o atributo `NotNull` .

Você também pode especificar que um valor de retorno ou um argumento não é nulo, mesmo que o tipo seja um tipo de referência que pode ser anulado. O método a seguir é um método auxiliar que lança se seu primeiro argumento for `null` :

```
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 forma:

```
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 `value` parâmetro tem a garantia de que não é nulo. No entanto, é aceitável chamar com `ThrowWhenNull` uma referência nula. Você pode fazer um tipo de referência que pode `value` ser anulado e adicionar a `NotNull` pós-condição à declaração de parâmetro:

```
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 , mas o argumento tem a garantia de que nunca será nulo se o método retornar sem lançar uma `null` exceção.

Especifique postcondições incondicionais usando os seguintes atributos:

- `MaybeNull`:um valor de retorno não anulado pode ser nulo.
- `NotNull`:um valor de retorno que pode ser anulado nunca será nulo.

Pós-condições condicionais: `NotNullWhen` `MaybeNullWhen` , e

`NotNullIfNotNull`

Você provavelmente está familiarizado com o `string` método `String.IsNullOrEmpty(String)` . Esse método retorna quando `true` o argumento é nulo ou uma cadeia de caracteres vazia. É uma forma de verificação nula: os chamadores não precisarão verificar nulo o argumento se o método retornar `false` . Para tornar um método como esse anulado de conhecimento, você definiria o argumento como um tipo de referência que pode ser anulado e adicionaria o `NotNullWhen` atributo :

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

Isso informa ao compilador que qualquer código em que o valor de retorno é `false` não precisa de verificações nulas. A adição do atributo informa a análise estática do compilador que executa a verificação nula necessária: quando ele retorna , o `IsNullOrEmpty` `false` argumento não é `null` .

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

O `String.IsNullOrEmpty(String)` método 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 quanto a valores nulos. O compilador não reconhecerá métodos de verificação nulos personalizados 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 marcada como nula.

Outro uso para esses atributos é o `Try*` padrão . As pós-condições dos `ref` `out` argumentos e são comunicadas por meio do valor de retorno. Considere esse método mostrado anteriormente (em um contexto desabilitado que pode ser anulado):

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

O método anterior segue uma idiom típica do .NET: o valor de retorno indica se foi definido como o valor encontrado ou, se nenhuma mensagem é encontrada, com o `message` valor padrão. Se o método retornar `true` , o valor de não será `message` nulo; caso contrário, o método define `message` como nulo.

Em um contexto habilitado para valor nulo, você pode comunicar essa idiom usando o `NotNullWhen` atributo . Quando você anota parâmetros para tipos de referência que podem ser anulados, faça `message` um e adicione um `string?` atributo:

```
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 é conhecido como não `message` nulo quando `TryGetMessage` retorna `true` . Você deve anotar métodos semelhantes em sua base de código da mesma maneira: os argumentos podem ser iguais a e são conhecidos como não nulos quando o `null` método retorna `true` .

Há um atributo final que talvez você também precise. Às vezes, o estado nulo de um valor de retorno depende do estado nulo de um ou mais argumentos. Esses métodos retornarão um valor não nulo sempre que

determinados argumentos não são `null`. Para anotar corretamente esses métodos, use o `NotNullIfNotNull` atributo. Considere o seguinte método:

```
string GetTopLevelDomainFromFullUrl(string url)
```

Se o `url` argumento não for nulo, a saída não será `null`. Depois que as referências que permitem valor nulo são habilitadas, você precisa adicionar mais anotações se sua API pode aceitar um argumento nulo. Você pode anotar o tipo de retorno conforme mostrado no código a seguir:

```
string? GetTopLevelDomainFromFullUrl(string? url)
```

Isso também funciona, mas geralmente força os chamadores a implementar verificações `null` extras. O contrato é que o valor de retorno seria `null` somente quando o argumento fosse `url null`. Para expressar esse contrato, você anotaria esse método, conforme mostrado no código a seguir:

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url)
```

O valor de retorno e o argumento foram anotados com o `?` que indica que qualquer um pode ser `null`. O atributo esclarece ainda mais que o valor de retorno não será nulo quando `url` o argumento não for `null`.

Você especifica pós-condições condicionais usando estes atributos:

- [MaybeNullWhen](#):um argumento não anulado pode ser nulo quando o método retorna o valor `bool` especificado.
- [NotNullWhen](#):um argumento anulado não será nulo quando o método retornar o valor `bool` especificado.
- [NotNullIfNotNull](#):um valor de retorno não será nulo se o argumento para o parâmetro especificado não for nulo.

Métodos auxiliares: `MemberNotNull` e `MemberNotNullWhen`

Esses atributos especificam sua intenção quando você refactorou 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 anulados tenham sido inicializados antes que cada construtor retorne. No entanto, o compilador C# não acompanha as atribuições de campo por meio de todos os métodos auxiliares. O compilador emite um aviso quando os campos não são inicializados diretamente no construtor, mas `CS8618` em um método auxiliar. Adicione o [MemberNotNullAttribute](#) a uma declaração de método e especifique os campos que são inicializados para um valor não nulo no método . Por exemplo, considere o exemplo a seguir:

```

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 `MemberNotNull` o construtor de atributo.

O `MemberNotNullWhenAttribute` tem um argumento `bool`. Você usa em situações em que o método auxiliar retorna um que indica se o método `MemberNotNullWhen` auxiliar `bool` inicializou campos.

Parar análise anulada quando o método chamado lança

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 booliana.

No primeiro caso, você pode adicionar o `DoesNotReturnAttribute` atributo à declaração de método. A análise de *estado* nulo do compilador não verifica nenhum código em um método que segue uma chamada para um método anotado com `DoesNotReturn`. Considere este método:

```

[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 nenhum aviso após a chamada para `FailFast`.

No segundo caso, você adiciona o `System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute` atributo a um parâmetro boolana do método . Você pode modificar o exemplo anterior da seguinte forma:

```

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, o compilador não executa nenhuma análise de estado `DoesNotReturnIf` nulo após esse método.

Resumo

IMPORTANT

A documentação oficial acompanha a versão mais recente do C#. Estamos escrevendo no momento para o C# 9.0. Dependendo da versão do C# que você está usando, vários recursos podem não estar disponíveis. A versão C# padrão para seu projeto é baseada na estrutura de destino. Para obter mais informações, consulte [padrões de controle de versão da linguagem C#](#).

Adicionar tipos de referência que permitem valor nulo fornece um vocabulário inicial para descrever suas expectativas de APIs para variáveis que podem ser `null`. Os atributos fornecem um vocabulário mais rico para descrever o estado nulo das 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.

Ao atualizar bibliotecas para um contexto que pode ser anulado, adicione esses atributos para orientar os usuários de suas APIs para o uso correto. Esses atributos ajudam você a descrever totalmente o estado nulo de argumentos e valores de retorno.

- [AllowNull](#):um campo, parâmetro ou propriedade que não permite valor nulo pode ser nulo.
- [DisallowNull](#):um campo, parâmetro ou propriedade que permite valor nulo nunca deve ser nulo.
- [MaybeNull](#):um campo, parâmetro, propriedade ou valor de retorno não anulado pode ser nulo.
- [NotNull](#):um campo, parâmetro, propriedade ou valor de retorno que pode ser nulo nunca será nulo.
- [MaybeNullWhen](#):um argumento não anulado pode ser nulo quando o método retorna o valor `bool` especificado.
- [NotNullWhen](#):um argumento anulado não será nulo quando o método retornar o valor `bool` especificado.
- [NotNullIfNotNull](#):um parâmetro, propriedade ou valor de retorno não será nulo se o argumento para o parâmetro especificado não for nulo.
- [DoesNotReturn](#):um método ou propriedade nunca retorna. Em outras palavras, ele sempre lança uma exceção.
- [DoesNotReturnIf](#):esse método ou propriedade nunca retornará se o parâmetro `bool` associado tiver o valor especificado.

Atributos diversos interpretados pelo compilador C#

21/01/2022 • 10 minutes to read

Os atributos,,, `Conditional` `Obsolete` `AttributeUsage` `AsyncMethodBuilder` `InterpolatedStringHandler` e `ModuleInitializer` podem ser aplicados a elementos em seu código. Eles adicionam significado semântico a esses elementos. O compilador usa esses significados semânticos para alterar sua saída e relatar possíveis erros por desenvolvedores usando 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 a seguir, `Conditional` é aplicado a um método para habilitar ou desabilitar a exibição de informações de diagnóstico específicas do programa:

```
#define TRACE_ON
using System;
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 `TRACE_ON` identificador não estiver definido, a saída de rastreamento não será exibida. Explore você mesmo na janela interativa.

O `Conditional` atributo é frequentemente usado com o `DEBUG` identificador para habilitar os recursos de rastreamento e log para compilações de depuração, mas não em compilações de versão, conforme mostrado no exemplo a seguir:

```
[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 `void` tipo de retorno. Usar o `Conditional` é mais limpo, mais elegante e menos propenso a erros do que os métodos de circunscrição dentro de `#if...#endif` blocos.

Se um método tiver vários `Conditional` atributos, o compilador incluirá chamadas para o método se um ou mais símbolos condicionais forem definidos (os símbolos são vinculados logicamente usando o operador OR). No exemplo a seguir, a presença de um `A` ou `B` resulta em uma chamada de método:

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Usando `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.

```
[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 `Obsolete` atributo marca um elemento de código como não é 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 `Obsolete` atributo 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:

```
using System;

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 o primeiro argumento para o construtor de 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 `[Obsolete]` atributo pode ser usado sem argumentos, mas incluir uma explicação o que usar em vez disso é recomendado.

No C# 10, você pode usar a interpolação de cadeia de caracteres constante e o `[nameof]` operador para garantir que os nomes correspondam:

```
public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}
```

Atributo `AttributeUsage`

O `[AttributeUsage]` atributo determina como uma classe de atributo personalizado 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:

```
[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:

```
[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:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Do C# 7.3 em diante, 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:

```
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:

```
[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:

```
[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 `field:` especificador para adicionar um atributo ao campo de apoio de uma [propriedade implementada automaticamente](#). Ou você pode usar o `field:` `property:` `param:` especificador ou para aplicar um atributo a qualquer um dos elementos gerados a partir de um registro posicional. Para obter um exemplo, consulte [sintaxe posicional para definição de propriedade](#).

Atributo `AsyncMethodBuilder`

A partir do C# 7, você adiciona o [System.Runtime.CompilerServices.AsyncMethodBuilderAttribute](#) atributo 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 `AsyncMethodBuilder` atributo pode ser aplicado a um tipo que:

- Tem um `GetAwaiter` método acessível.
- O objeto retornado pelo `GetAwaiter` método implementa a [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#) interface.

O construtor para o `AsyncMethodBuilder` atributo especifica o tipo do Construtor associado. O construtor deve implementar os seguintes membros acessíveis:

- Um `Create()` método estático que retorna o tipo do construtor.
- Uma `Task` Propriedade legível que retorna o tipo de retorno assíncrono.
- Um `void SetException(Exception)` método que define a exceção quando uma tarefa falha.
- Um `void SetResult()` `void SetResult(T result)` método ou que marca a tarefa como concluída e, opcionalmente, define o resultado da tarefa
- Um `Start` método com a seguinte assinatura de API:

```
void Start<TStateMachine>(ref TStateMachine stateMachine)
    where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

- Um `AwaitOnCompleted` método com a seguinte assinatura:

```
public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine
stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.INotifyCompletion
    where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

- Um `AwaitUnsafeOnCompleted` método com a seguinte assinatura:

```
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 método assíncrono 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 posteriores, o `AsyncMethodBuilder` atributo pode ser aplicado a um método assíncrono para substituir o Construtor desse tipo.

Atributos `InterpolatedStringHandler` e

`InterpolatedStringHandlerArguments`

Começando com o C# 10, você usa esses atributos para especificar que um tipo é um *manipulador de cadeia de caracteres interpolado*. A biblioteca do .NET 6 já inclui

`System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` cenários em que você usa uma cadeia de caracteres interpolada como o argumento para um `string` parâmetro. Você pode ter outras instâncias em que deseja controlar como as cadeias de caracteres interpoladas são processadas. Você aplica o `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute` ao tipo que implementa seu manipulador. Você aplica os `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute` parâmetros to do construtor do tipo.

Você pode aprender mais sobre a criação de um manipulador de cadeia de caracteres interpolado na especificação de recurso C# 10 para [aprimoramentos de cadeia de caracteres interpolados](#).

Atributo `ModuleInitializer`

Começando com o C# 9, o `ModuleInitializer` atributo marca um método que o tempo de execução chama quando o assembly é carregado. `ModuleInitializer` é um alias para `ModuleInitializerAttribute`.

O `ModuleInitializer` atributo só pode ser aplicado a um método que:

- É estático.
- Não tem parâmetros.

- Retorna `void`.
- É acessível no módulo que o contém, ou `internal` seja, ou `public`.
- Não é um método genérico.
- Não está contido em uma classe genérica.
- Não é uma função local.

O `ModuleInitializer` atributo pode ser aplicado a vários métodos. Nesse caso, a ordem na qual o runtime os chama é determinística, mas não especificada.

O exemplo a seguir ilustra o uso de vários métodos de inicializador de módulo. Os `Init1` métodos e são `Init2` executados antes de e cada um adiciona `Main` uma cadeia de caracteres à propriedade `Text`. Portanto, `Main` quando é executado, a propriedade já tem `Text` cadeias de caracteres de ambos os métodos de inicializador.

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

```
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. Os 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`

A partir do C# 9, o atributo impede que o compilador desfira o sinalizador ao `SkipLocalsInit .locals init` emitir para metadados. O atributo 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 `SkipLocalsInit` a um assembly. `SkipLocalsInit` é um alias para [SkipLocalsInitAttribute](#).

O `.locals init` sinalizador faz com que o CLR initialize todas as variáveis locais declaradas em um método para seus valores padrão. 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, a inicialização zero extra

pode ter um impacto mensurável no desempenho em alguns cenários, como quando você usa `stackalloc` para alocar uma matriz na pilha. Nesses casos, você pode adicionar o `SkipLocalsInit` atributo. Se aplicado a um método diretamente, o atributo afetará esse método e todas as suas funções aninhadas, incluindo lambdas e funções locais. Se aplicado a um tipo ou módulo, ele afeta todos os métodos aninhados dentro dele. 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 de memória alocada em pilha não reinicializada).

O exemplo a seguir ilustra o efeito do `SkipLocalsInit` atributo em um método que usa `stackalloc`. O método exibe o que estava na memória quando a matriz de inteiros foi alocada.

```
[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
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

Para experimentar esse código por conta própria, defina a `AllowUnsafeBlocks` opção do compilador em seu arquivo `.csproj`:

```
<PropertyGroup>
    ...
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

Confira também

- [Attribute](#)
- [System.Reflection](#)
- [Atributos](#)
- [Reflexão](#)

Código não seguro, tipos de ponteiro e ponteiros de função

21/01/2022 • 14 minutes to read

A maior parte do código C# que você escreve é "código de segurança verificável". *Código seguro verificável* significa que as ferramentas .net podem verificar se 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, cria objetos gerenciados.

O C# dá suporte a um `unsafe` contexto, no qual você pode escrever código não *verificável*. Em um `unsafe` contexto, 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 de compilador `AllowUnsafeBlocks`.

Tipos de ponteiro

Em um contexto sem segurança, um tipo pode ser um tipo de ponteiro, além de um tipo de valor, ou um tipo de referência. Uma declaração de tipo de ponteiro usa uma das seguintes formas:

```
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.

Tipos de ponteiro não herdam de `objeto` e não existem conversões entre tipos de ponteiro e `object`. Além disso, boxing e unboxing não dão suporte a ponteiros. No entanto, você pode converter entre diferentes tipos de ponteiro e tipos de ponteiro e tipos inteiros.

Quando você declara vários ponteiros na mesma declaração, escreve o asterisco (`*`) junto apenas com o tipo subjacente. Ele não é usado como um prefixo para cada nome de ponteiro. Por exemplo:

```
int* p1, p2, p3; // Ok  
int *p1, *p2, *p3; // Invalid in C#
```

Um ponteiro não pode apontar para uma referência ou para uma `struct` que contém referências, pois uma referência de objeto pode ser coletada como lixo, mesmo se um ponteiro estiver apontando para ele. O coletor de lixo não acompanha se 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 Char.
- `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:

```
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. Esses exemplos devem ser compilados com a opção de compilador [AllowUnsafeBlocks](#) definida.

```

// 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
*/

```

Não é possível aplicar o operador de indireção a 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>-></code>	Acessa um membro de um struct através de um ponteiro.

OPERADOR/INSTRUÇÃO	USE
[]	Indexa um ponteiro.
&	Obtém o endereço de uma variável.
++ e --	Incrementa e decrementa ponteiros.
+ e -	Executa aritmética de ponteiros.
== , != , < , > , <= e >=	Compara ponteiros.
stackalloc	Aloca memória na pilha.
fixed privacidade	Corrigé temporariamente uma variável para que seu endereço possa ser encontrado.

Para obter mais informações sobre operadores relacionados a ponteiros, consulte [operadores relacionados a ponteiros](#).

Qualquer tipo de ponteiro pode ser convertido implicitamente em um `void*` tipo. Qualquer tipo de ponteiro pode ser atribuído ao valor `null`. Qualquer tipo de ponteiro pode ser explicitamente convertido em qualquer outro tipo de ponteiro usando uma expressão de conversão. Você também pode converter qualquer tipo integral para um tipo de ponteiro ou qualquer tipo de ponteiro para um tipo integral. Essas conversões exigem uma conversão explícita.

O exemplo a seguir converte um `int*` para 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.

```
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

No C#, você pode usar a instrução `fixed` para criar um buffer com uma matriz de tamanho fixo em uma estrutura de dados. Buffers de tamanho fixo são úteis quando você escreve métodos que interoperam com fontes de dados de outras linguagens ou plataformas. A matriz fixa 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`.

```
private fixed char name[30];
```

No código seguro, um struct C# que contém uma matriz não contém os elementos da matriz. O struct contém uma referência para os elementos em vez disso. Você pode inserir uma matriz de tamanho fixo em uma `struct` quando ela é usada em um bloco de código [não seguro](#).

O tamanho dos itens a seguir `struct` não depende do número de elementos na matriz, pois `pathName` é uma referência:

```
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. É possível uma instrução `fixed` para estabelecer um ponteiro para o 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.

```
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 [caracteres](#) de tamanho fixo sempre ocupam 2 bytes por caractere, independentemente da codificação. Esse tamanho de matriz é o mesmo mesmo

quando buffers de caracteres são empacotados para métodos de API ou structs 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, que estão disponíveis a partir do C# 7.3.

Outra matriz de tamanho fixo comum é a matriz `bool`. Os elementos em uma `bool` matriz sempre têm um tamanho de 1 byte. `bool` as matrizes não são apropriadas para a criação de matrizes de bits ou buffers.

Buffers de tamanho fixo são compilados com o [System.Runtime.CompilerServices.UnsafeValueTypeAttribute](#), que instrui o Common Language Runtime (CLR) de que um tipo contém uma matriz não gerenciada que potencialmente pode exceder. A memória alocada usando `stackalloc` também habilita automaticamente os recursos de detecção de estouro de buffer no CLR. O exemplo anterior mostra como um buffer de tamanho fixo poderia existir em um `unsafe struct`.

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

O C# gerado pelo compilador `Buffer` para é atribuído da seguinte forma:

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

Buffers de tamanho fixo diferem das matrizes regulares das seguintes maneiras:

- Só pode ser usado em um `unsafe` contexto.
- Podem 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]`. Você não pode 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 fixados quando o bloco `fixed` for concluído. Como o `Copy` método neste exemplo usa a `unsafe` palavra-chave, 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. Esse recurso está disponível começando com o C# 7.3.

```

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();
    }

    // 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();
    }

    // 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();
    }

    // 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");
}

```

```

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 `delegate` tipos para definir objetos de ponteiro de função segura. Invocar um delegado envolve a instância de um tipo derivado de e fazer uma `SystemDelegate` chamada de método virtual para seu `Invoke` método. Essa chamada virtual usa a `callvirt` instrução IL. Em caminhos de código críticos de desempenho, usar a `calli` instrução IL é mais eficiente.

Você pode definir um ponteiro de função usando a `delegate*` sintaxe. O compilador chamará a função usando a instrução em vez de `calli` iniciar um `delegate` objeto e chamar `Invoke`. O código a seguir declara dois métodos que usam um `delegate` ou um para combinar dois objetos do mesmo `delegate*` tipo. O primeiro método usa um `System.Func<T1,T2,TResult>` tipo delegado. O segundo método usa uma `delegate*` declaração com os mesmos parâmetros e tipo de retorno:

```

public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static T UnsafeCombine<T>(delegate*<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

```

O código a seguir mostra como você declararia uma função local estática e invocaria o método `UnsafeCombine` usando um ponteiro para essa função local:

```

static int localMultiply(int x, int y) => x * y;
int product = UnsafeCombine(&localMultiply, 3, 4);

```

O código anterior ilustra várias das regras na função acessadas como um ponteiro de função:

- Os ponteiros de função só podem ser declarados em um `unsafe` contexto.
- Métodos que levam um `delegate*` (ou retornam `delegate*` um) só podem ser chamados em um `unsafe` contexto.
- O `&` operador para obter o endereço de uma função é permitido somente em `static` funções. (Essa regra se aplica a funções membro e funções locais).

A sintaxe tem paralelos com a declaração de `delegate` tipos 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 assume o endereço do método.

Você pode especificar a convenção de chamada para um `delegate*` usando as palavras-chave `managed` e `unmanaged`. Além disso, para `unmanaged` ponteiros de função, você pode especificar a convenção de chamada. As declarações a seguir mostram exemplos de cada uma. A primeira declaração usa a `managed` convenção de chamada, que é o padrão. Os três próximos usam uma `unmanaged` convenção de chamada. Cada especifica uma das convenções de chamada ECMA 335: `Cdecl`, `Stdcall`, ou `Fastcall` `Thiscall`. As últimas declarações usam a convenção `unmanaged` de chamada, instruindo o CLR a escolher a convenção de chamada padrão para a plataforma. O CLR escolherá a convenção de chamada em tempo de executar.

```
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 de ponteiro de função para C# 9.0](#).

Especificação da linguagem C#

Para obter mais informações, consulte o [capítulo Código não seguro](#) da [especificação da linguagem C#](#).

Diretivas de pré-processador do C#

21/01/2022 • 14 minutes to read

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 C e C++, você não pode usar essas diretivas para criar macros. Uma diretiva de pré-processador deve ser a única instrução em uma linha.

Contexto que pode ser anulado

A `#nullable` diretiva de pré-processador define o contexto de *anotação que pode ser anulado* e o *contexto de aviso que pode ser anulado*. Essa diretiva controla se as anotações que anuáveis 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 `#nullable` diretiva 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 o substitua ou até o final do arquivo de origem.

O efeito das diretivas é o seguinte:

- `#nullable disable` : define a anotação anuável e os contextos de aviso como *desabilitados*.
- `#nullable enable` : define os contextos de aviso e anotação que permitem valor nulo *como habilitados*.
- `#nullable restore` : restaura a anotação anuável 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 que permite valor nulo *como habilitado*.
- `#nullable restore annotations` : restaura o contexto de anotação que pode ser anulado para as configurações do projeto.
- `#nullable disable warnings` : define o contexto de aviso que pode ser anulado como *desabilitado*.
- `#nullable enable warnings` : define o contexto de aviso que permite valor nulo *como habilitado*.
- `#nullable restore warnings` : restaura o contexto de aviso que pode ser anulado 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 em se o símbolo especificado estiver definido.
- `#else` : fecha a compilação condicional anterior e abre uma nova compilação condicional se o símbolo especificado anterior não estiver definido.
- `#endif` : fecha a compilação condicional anterior.

Quando o compilador C# encontra uma diretiva, seguida eventualmente por uma diretiva , ele compila o código entre as diretivas somente se o `#if` `#endif` símbolo especificado for definido. Ao contrário de C e C++, você não pode atribuir um valor numérico a um símbolo. A `#if` instrução em C# é booliana e testa apenas se o símbolo foi definido ou não. Por exemplo:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Você pode usar os `==` (operadores (igualdade)) e `!=` (desigualdade) para testar os `bool` valores ou `true` `false`. `true` significa que o símbolo está 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 `,,`, `,` e `,`, permite incluir ou excluir código com base na existência de `#else` `#elif` um ou mais `#endif` `#define` `#undef` símbolos. A compilação condicional pode ser útil ao compilar código para um build de depuração ou ao compilar para uma configuração específica.

Uma diretiva condicional que começa com `#if` uma diretiva deve ser terminada explicitamente com uma diretiva `#endif`. A diretiva `#define` permite definir um símbolo. Usando o símbolo como a expressão passada para a `#if` diretiva, a expressão é avaliada como `true`. Você também pode definir um símbolo com a opção do compilador **DefineConstants**. Você pode indevidamente 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 não entra em conflito com `#define` 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 será avaliada se nem as expressões de diretiva anteriores nem as `#elif` `#if` anteriores, `#elif` opcionais, são avaliadas como `true`. Se uma `#elif` expressão for avaliada como `true`, o compilador avaliará todo o código entre o e a `true` próxima diretiva `#elif` condicional. Por exemplo:

```
#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, de modo que, se nenhuma das expressões nas diretivas anteriores ou (opcional) for avaliada como `true`, o compilador avaliará todo o código entre e o próximo `#if` `#elif` `true` `#else` `#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 `#if` diretiva.

O sistema de build também está ciente de símbolos predefinidos de pré-processador que representam **estruturas de destino** diferentes em projetos no estilo SDK. Eles são úteis ao criar aplicativos que podem ser destinados a mais de uma versão do .NET.

FRAMEWORKS DE DESTINO	SÍMBOLOS	SÍMBOLOS ADICIONAIS DISPONÍVEIS NO .NET 5 + SDK
.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 , NET6_0 , NET6_0_ANDROID , NET6_0_IOS , NET6_0_MACOS , NET6_0_MACCATALYST , NET6_0_TVOS , NET6_0_WINDOWS , NET5_0 , NETCOREAPP , NETCOREAPP3_1 , NETCOREAPP3_0 , NETCOREAPP2_2 , NETCOREAPP2_1 , NETCOREAPP2_0 , NETCOREAPP1_1 , NETCOREAPP1_0	NET6_0_OR_GREATER , NET6_0_ANDROID_OR_GREATER , NET6_0_IOS_OR_GREATER , NET6_0_MACOS_OR_GREATER , NET6_0_MACCATALYST_OR_GREATER , NET6_0_TVOS_OR_GREATER , NET6_0_WINDOWS_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

NOTE

- Os símbolos sem versão são definidos independentemente da versão que você está direcionando.
- Os símbolos específicos da versão são definidos apenas para a versão que você está direcionando.
- Os <framework>_OR_GREATER símbolos 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:
`NET_2_0 .. NET_2_0_OR_GREATER` `NET_1_1_OR_GREATER` e `NET_1_0_OR_GREATER` .

NOTE

Para projetos tradicionais no estilo não 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 `DEBUG` as `TRACE` constantes e . 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 em um arquivo e, em `MYTEST` seguida, testar os valores dos `MYTEST` `DEBUG` símbolos e . A saída deste exemplo depende se você criou o projeto no modo de configuração de Depuração ou Versão.

```
#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:

```
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 a seguir para definir ou indevidamente símbolos para compilação condicional:

- `#define` : defina um símbolo.
- `#undef` : desefine um símbolo.

Use `#define` para definir um símbolo. Quando você usar o símbolo como a expressão passada para a diretiva , a expressão será avaliada como `#if` , como mostra o exemplo a `true` seguir:

```
#define VERBOSE

#if VERBOSE
    Console.WriteLine("Verbose output version");
#endif
```

NOTE

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. Você pode testar o símbolo com `#if` ou `#elif`. Você também pode usar o [ConditionalAttribute](#) para executar uma compilação condicional. Você pode definir um símbolo, mas não pode 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 indevidamente um símbolo com `#undef`.

Definindo regiões

Você pode definir regiões de código que podem ser recolhidos em um contorno usando as duas diretivas de pré-processador a seguir:

- `#region` : inicie uma região.
- `#endregion` : encerrar uma região.

`#region` permite especificar um bloco de código que pode ser expandido ou ressalvado ao usar o recurso de estrutura `de` destaque do editor de códigos. Em arquivos de código mais longos, é conveniente fechar ou ocultar uma ou mais regiões para que você possa se concentrar na parte do arquivo no qual você está trabalhando no momento. O exemplo a seguir mostra como definir uma região:

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Um `#region` bloco deve ser encerrado com uma diretiva `#endregion`. Um `#region` bloco não pode se sobrepor a um `#if` bloco. No entanto, `#region` um bloco pode ser aninhado em um bloco e um bloco pode ser `#if` `#if` aninhado em um `#region` bloco.

Informações de erro e aviso

Você instrui o compilador a gerar erros e avisos do compilador definido pelo usuário e controlar informações de linha usando as seguintes diretivas:

- `#error` : gere um erro do compilador com uma mensagem especificada.
- `#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:

```
#error Deprecated code in this method.
```

NOTE

O compilador trata de uma maneira especial e relata um erro `#error version` do compilador, CS8304, com uma mensagem contendo o compilador e as versões de linguagem usados.

`#warning` permite gerar um aviso do compilador [CS1030](#) de nível um de um local específico no código. Por exemplo:

```
#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.

```
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:

```
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 oculta as linhas sucessivas do depurador, de forma que, quando o desenvolvedor passa pelo código, todas as linhas entre uma e a próxima diretiva (supondo que não seja outra diretiva) serão `#line hidden`.

sobresserdas. `#line hidden` `#line` `#line hidden` Essa opção também pode ser usada para permitir que o ASP.NET diferencie entre o código gerado pelo computador e definido pelo usuário. Embora ASP.NET seja o consumidor principal desse recurso, é provável que mais geradores de origem façam uso dele.

Uma `#line hidden` diretiva não afeta nomes de arquivo ou números de linha no relatório de erros. 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.

A partir do C# 10, você pode usar uma nova forma da `#line` diretiva :

```
#line (1, 1) - (5, 60) 10 "partial-class.g.cs"
/*34567*/int b = 0;
```

Os componentes deste formulário são:

- `(1, 1)` : a linha inicial e a coluna para o primeiro caractere na linha que segue a diretiva . Neste exemplo, a próxima linha seria relatada como linha 1, coluna 1.
- `(5, 60)` : a linha final e a coluna para a região marcada.
- `10` : o deslocamento de coluna para que `#line` a diretiva entre em vigor. Neste exemplo, a 10^a coluna seria relatada como coluna um. É aí que a declaração `int b = 0;` começa. Esse campo é opcional. Se omitida, a diretiva entra em vigor na primeira coluna.
- `"partial-class.g.cs"` : o nome do arquivo de saída.

O exemplo anterior geraria o seguinte aviso:

```
partial-class.g.cs(1,5,1,6): warning CS0219: The variable 'b' is assigned but its value is never used
```

Após o remapeamento, a variável `b`, está na primeira linha, no caractere seis.

DSLs (linguagens específicas do domínio) normalmente usam esse formato para fornecer um mapeamento melhor do arquivo de origem para a saída C# gerada. Para ver mais exemplos desse formato, consulte 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, você não pode usar para `#pragma` criar instruções de pré-processamento personalizadas.

- `#pragma warning` : Habilita ou desabilita avisos.
- `#pragma checksum` : gere uma verificação.

```
#pragma pragma-name pragma-arguments
```

Em `pragma-name` que é o nome de um pragma reconhecido e são os `pragma-arguments` argumentos específicos do pragma.

`#pragma warning`

O `#pragma warning` pode habilitar ou desabilitar determinados avisos.

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Em `warning-list` que é uma lista separada por vírgulas de números de aviso. 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.

NOTE

Para localizar números de aviso no Visual Studio, compile o projeto e, em seguida, procure os números de aviso na janela de Saída.

O `disable` entra em vigor a partir da próxima linha do arquivo de origem. O aviso é restaurado na linha após o `restore`. Se não houver nenhum no arquivo, os avisos serão restaurados para seu estado padrão na primeira linha de qualquer arquivo posterior `restore` na mesma compilação.

```
// 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.

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Em que é o nome do arquivo que requer monitoramento de alterações ou atualizações, é o GUID (Identificador Global Exclusivo) do algoritmo de hash e é a cadeia de caracteres de `"filename"` `"{guid}"` dígitos hexadecimais que representam os bytes da caixa de `"checksum_bytes"` verificação. Deve ser um número par de dígitos hexadecimais. Um número ímpar de dígitos resulta em um aviso de tempo de compilação e a diretiva é ignorada.

O depurador do Visual Studio usa uma soma de verificação para certificar-se de sempre encontrar a 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 ASP.NET, porque a verificação computada é para o arquivo de origem

gerado, em vez do 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 no arquivo, ele calculará a verificação e grava o valor

`#pragma checksum` no arquivo PDB.

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

Opções do compilador de C#

21/01/2022 • 2 minutes to read

Esta seção descreve as opções interpretadas pelo compilador C#. As opções são agrupadas em artigos separados com base no que eles controlam, por exemplo, recursos de idioma, geração de código e saída. Use o Sumário para navegar entre eles.

Como definir opções

Há duas maneiras diferentes de definir opções de compilador em projetos .NET:

- *Em seu * arquivo. csproj*

você pode adicionar MSBuild propriedades para qualquer opção de compilador em seu arquivo *. csproj em 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 trecho de arquivo de projeto a seguir define a `LangVersion` propriedade.

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

para obter mais informações sobre como definir opções em arquivos de projeto, consulte o artigo [MSBuild propriedades para projetos do SDK do .net](#).

- *usando as páginas de propriedades de Visual Studio*

Visual Studio fornece páginas de propriedades para editar propriedades de compilação. para saber mais sobre eles, consulte [gerenciar propriedades do projeto e da solução-Windows](#) ou [gerenciar propriedades do projeto e da solução-Mac](#).

projetos de .NET Framework

IMPORTANT

esta seção aplica-se somente a projetos .NET Framework.

além dos mecanismos descritos acima, você pode definir opções de compilador usando dois métodos adicionais para projetos de .NET Framework:

- **argumentos de linha de comando para projetos de .NET Framework:** os projetos de .NET Framework usam `csc.exe` em vez de `dotnet build` criar projetos. você pode especificar argumentos de linha de comando para `csc.exe` para projetos .NET Framework.
- **páginas ASP.NET compiladas:** .NET Framework projetos usam uma seção do arquivo `web.config` para compilar páginas. para o novo sistema de compilação e ASP.NET Core projetos, as opções são obtidas do arquivo de projeto.

a palavra para algumas opções de compilador alteradas de `csc.exe` e .NET Framework projetos para o novo sistema de MSBuild. A nova sintaxe é usada em toda esta seção. Ambas as versões são listadas na parte superior de cada página. Por `csc.exe`, todos os argumentos são listados seguindo a opção e dois-pontos. Por exemplo, a `-doc` opção seria:

Você pode invocar o compilador 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* da linha de comando. Todas as opções do compilador estão disponíveis em duas formas: **-option** e **/option**. em projetos da web .NET Framework, você especifica opções para a compilação do código-behind no arquivo *web.config* . Para obter mais informações, consulte [<compiler> 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 Visual Studio](#).

o arquivo executável *csc.exe* geralmente está localizado na pasta Microsoft. NET\Framework \ <*Version*> sob o 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 linguagem

21/01/2022 • 7 minutes to read

As opções a seguir controlam como o compilador interpreta os recursos de linguagem. A nova MSBuild nova sintaxe é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **CheckForOverflowUnderflow:** / `-checked` gera verificações de estouro.
- **AllowUnsafeBlocks:** / `-unsafe` permita o código 'unsafe'.
- **DefineConstants:** / `-define` definir símbolos de compilação condicional.
- **LangVersion:** / `-langversion` especifique a versão do idioma, como (versão principal mais `default` recente) ou `latest` (versão mais recente, incluindo versões secundárias).
- **Anulado:** / `-nullable` habilita o contexto que permite valor nulo ou avisos que permitem valor nulo.

Checkforoverflowunderflow

A opção **CheckForOverflowUnderflow** especifica se uma instrução aritmética de inteiro que resulta em um valor que está fora do intervalo do tipo de dados causa uma exceção de tempo de run-time.

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

Uma instrução aritmética de inteiro que está no escopo de uma palavra-chave ou não está sujeita ao efeito da `checked` `unchecked` opção **CheckForOverflowUnderflow**. Se uma instrução aritmética de inteiro que não está no escopo de uma palavra-chave ou resulta em um valor fora do intervalo do tipo de dados `checked` `unchecked` e **CheckForOverflowUnderflow** é, essa instrução causa uma exceção em tempo de `true` operação. Se **CheckForOverflowUnderflow** for `false`, essa instrução não causará uma exceção em tempo de executar. O valor padrão dessa opção é; a `false` verificação de estouro está desabilitada.

Allowunsafeblocks

A opção do compilador **AllowUnsafeBlocks** permite que o código que usa a palavra-chave `unsafe` seja compilado. O valor padrão dessa opção é, o `false` que significa que o código não seguro não é permitido.

```
<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 programa.

```
<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 a diretiva `#define` pré-processador, 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 `-define` opção, uma diretiva em um arquivo não tem efeito sobre 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.

NOTE

A diretiva C# não permite que um símbolo seja dado a um valor, como em `#define` linguagens como 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.

```
<LangVersion>9.0</LangVersion>
```

Os seguintes valores são válidos:

VALOR	SIGNIFICADO
<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> (<code>default</code>)	O compilador aceita a sintaxe da versão principal mais recente lançada do compilador.
<code>10.0</code>	O compilador aceita apenas a sintaxe incluída em C# 10 ou inferior.
<code>9.0</code>	O compilador aceita apenas 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.

VALOR	SIGNIFICADO
6	O compilador aceita somente a sintaxe incluída no C# 6.0 ou inferior.
5	O compilador aceita somente a sintaxe incluída no C# 5.0 ou inferior.
4	O compilador aceita somente a sintaxe incluída no C# 4.0 ou inferior.
3	O compilador aceita somente a sintaxe incluída no C# 3.0 ou inferior.
ISO-2 (ou 2)	O compilador aceita apenas a sintaxe que está incluída em ISO/IEC 23270:2006 C# (2,0).
ISO-1 (ou 1)	O compilador aceita apenas a sintaxe que está incluída em 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 no [versão da linguagem C#](#).

Os metadados referenciados pelo aplicativo C# não estão sujeitos à opção do compilador **LangVersion**.

Como cada versão do compilador C# contém extensões para a especificação de linguagem, o **LangVersion** não dá a você a funcionalidade equivalente de uma versão anterior do compilador.

Além disso, embora as atualizações de versão do C# geralmente coincidam com as versões .NET Framework principais, a nova sintaxe e os recursos não estão necessariamente vinculados a essa versão específica da estrutura. Embora as novas funcionalidades definitivamente exijam uma nova atualização do compilador que também é liberada junto com a revisão do C#, cada funcionalidade específica tem seus 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 da [configuração de LangVersion](#) usada, use a versão atual do Common Language Runtime para criar seu .exe ou .dll. Uma exceção são assemblies amigos [e ModuleAssemblyName](#), que funcionam em `-langversion:ISO-1`.

Para outras maneiras de especificar a versão da linguagem C#, consulte [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		Não disponível no momento
C# 6.0	Link	Especificação da Linguagem C# Versão 6 – Rascunho não oficial: .NET Foundation
C# 5.0	Baixar PDF	Padrão ECMA-334 – 5ª Edição
C# 3.0	Baixar DOC	Especificação da Linguagem C# Versão 3.0: Microsoft Corporation

VERSÃO	LINK	DESCRIÇÃO
C# 2.0	Baixar PDF	Padrão ECMA-334 – 4ª Edição
C# 1.2	Baixar DOC	Especificação da Linguagem C# Versão 1.2: Microsoft Corporation
C# 1.0	Baixar DOC	Especificação da Linguagem C# Versão 1.0: Microsoft Corporation

Versão mínima do SDK necessária para dar suporte a todos os recursos de linguagem

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# 10	Microsoft Visual Studio/Ferramentas de Build 2022 ou SDK do .NET 6
C# 9.0	Microsoft Visual Studio/Ferramentas de Build 2019, versão 16.8 ou SDK do .NET 5
C# 8.0	Microsoft Visual Studio/Ferramentas de Build 2019, versão 16.3 ou SDK do .NET Core 3.0
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
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 pacote .NET Framework o compilador 1.0

Nullable

A opção **Nullable** permite especificar o contexto anulável. O valor padrão dessa opção é `disable`.

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

O argumento deve ser um de `enable`, `disable`, `warnings` ou `annotations`. O `enable` argumento habilita o contexto anulável. Especificar `disable` desativará o contexto anulável. Ao fornecer o `warnings` argumento, o contexto de aviso anulável é habilitado. Ao especificar o `annotations` argumento, o contexto de anotação anulável é habilitado.

Flow análise é usada para inferir a nulidade das 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 omitidas condicionalmente. Por exemplo, `Debug.Assert` no modo de versão.

A invocação de métodos anotados com os seguintes atributos também afetará a análise do fluxo:

- Pré-condições simples: `AllowNullAttribute` e `DisallowNullAttribute`
- Pós-condições simples: `MaybeNullAttribute` e `NotNullAttribute`
- Pós-condições condicionais: `MaybeNullWhenAttribute` e `NotNullWhenAttribute`
- `DoesNotReturnIfAttribute` (por exemplo, `DoesNotReturnIf(false)` para `Debug.Assert`) e `DoesNotReturnAttribute`
- `NotNullIfNotNullAttribute`
- Pós-condições do membro: `MemberNotNullAttribute(String)` e `MemberNotNullAttribute(String[])`

IMPORTANT

O contexto anulável global não se aplica a 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. Em `editorconfig`, especifique `generated_code = true` em uma seção que se aplica a esse arquivo.
2. Put `<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 comentário deve ser o primeiro elemento no 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 optar por usar a `#nullable` diretiva de pré-processador.

Opções do compilador C# que controlam a saída do compilador

21/01/2022 • 9 minutes to read

As opções a seguir controlam a geração de saída do compilador.

MSBUILD	CSC.EXE	Descrição
DocumentationFile	-doc:	Gere o arquivo de documento XML de <code>///</code> comentários.
OutputAssembly	-out:	Especifique o arquivo de assembly de saída.
Platformtarget	-platform:	Especifique a CPU da plataforma de destino.
ProduceReferenceAssembly	-refout:	Gere um assembly de referência.
Targettype	-target:	Especifique o tipo do assembly de saída.

DocumentationFile

A opção **DocumentationFile** permite que você coloque comentários de documentação em um arquivo XML. Para saber mais sobre como documentar seu código, consulte Marcas [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.

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

O arquivo de código-fonte que contém instruções Main ou de nível superior é produzido primeiro no XML. Geralmente, você deseja usar o arquivo .xml gerado com [o IntelliSense](#). O .xml de arquivo deve ser o mesmo que o nome do assembly. O .xml arquivo deve estar no mesmo diretório que o assembly. Quando o assembly é referenciado em um projeto Visual Studio, o `arquivo.xml` é encontrado também. Para obter mais informações sobre como gerar comentários de código, consulte [Fornecendo comentários de código](#). A menos que você compile com , conterá marcas e especificando o nome do arquivo que contém o `<TargetType:Module>` manifesto do assembly para o arquivo de `file <assembly> </assembly>` saída. Para exemplos, consulte [Como usar os recursos de documentação XML](#).

NOTE

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 a 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.

```
<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, MSBuild usará o nome do projeto para especificar o nome do assembly de saída. Projetos de estilo antigo usam as seguintes regras:

- Um .exe terá seu nome do arquivo de código-fonte que contém o `Main` método ou instruções de nível superior.
- 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 amigo](#).

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. Você pode especificar essa opção somente para projetos destinados .NET Framework 4.5 ou posterior.
- O **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** são executados no CLR de 32 bits em execução em WOW64.
- Uma DLL compilada com o **anycpu** é executada no mesmo CLR que o processo no qual ele é carregado.
- Os executáveis compilados com o **anycpu** são executados no CLR de 64 bits.
- Executáveis compilados com **anycpu32bitpreferred** são executados no CLR de 32 bits.

A configuração **anycpu32bitpreferred** é válida somente para arquivos executáveis (.EXE) e requer .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](#).

Você pode definir a opção `PlatformTarget` na página **Propriedades** de build do 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 x86 `dotnet.exe` ou o x64 `dotnet.exe`.

Para aplicativos autossuporte, `dotnet publish` a etapa pacotes o executável para o RID de configuração.

ProduceReferenceAssembly

A opção `ProduceReferenceAssembly` especifica um caminho de arquivo em que o assembly de referência deve ser produzido. Ela é traduzida para `metadataPeStream` na API de Eit. O `filepath` especifica o caminho para o assembly de referência. Geralmente, ele deve corresponder ao assembly principal. A convenção recomendada (usada pelo MSBuild) é colocar o assembly de referência em uma subpasta "ref/" em relação ao assembly primário.

```
<ProduceReferenceAssembly>filepath</ProduceReferenceAssembly>
```

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. Eles incluem declarações para todos os membros que são significativos ao referenciar um assembly em ferramentas de build. Assemblies de referência excluem todas as implementações de membro e declarações de membros privados. Esses membros não têm nenhum 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.

TargetType

A opção do compilador `TargetType` pode ser especificada em um dos seguintes formulários:

- `library`: para criar uma biblioteca de códigos. `library` é o valor padrão.
- `exe`: para criar um .exe arquivo.
- `módulo` para criar um módulo.
- `winexe` para criar um Windows programa.
- `winmdobj` para criar um arquivo `.winmdobj` intermediário.
- `appcontainerexe` para criar um arquivo .exe para aplicativos Windows 8.x Store.

NOTE

Para .NET Framework destinos, a menos que você especifique o módulo , essa opção faz com que um manifesto .NET Framework assembly seja colocado em um arquivo de saída. Para obter mais informações, consulte [Assemblies no .NET](#) e [Atributos comuns](#).

```
<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, poderá indicar que todo ou parte do código é compatível com CLS com o `CLSCompliantAttribute` atributo .

biblioteca

A opção de biblioteca faz com que o compilador crie uma DLL (biblioteca de vínculo dinâmico) em vez de um arquivo executável (EXE). A DLL será criada com a extensão .dll dados. A menos que especificado de outra forma com a opção [OutputAssembly](#), o nome do arquivo de saída recebe o nome do primeiro arquivo de entrada. Ao criar um arquivo de .dll, um `Main` método não é necessário.

exe

A opção `exe` faz com que o compilador crie um executável (exe), aplicativo de console. O arquivo executável será criado com a extensão .exe. Use [winexe](#) para criar um executável de programa de Windows. A menos que especificado de outra forma com a opção [OutputAssembly](#), o nome do arquivo de saída usa o nome do arquivo de entrada que contém o ponto de entrada (método `Main` ou instruções de nível superior). Apenas um ponto de entrada é necessário nos arquivos de código-fonte que são compilados em um arquivo de .exe. A opção de compilador [StartupObject](#) permite especificar qual classe contém o `Main` método, em casos em que seu código tem mais de uma classe com um `Main` método.

module

Essa opção faz com que o compilador não gere um manifesto do assembly. Por padrão, o arquivo de saída criado pela compilação com essa opção terá uma extensão de .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 ao 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 faz referência `internal` a tipos em outro módulo, ambos os módulos devem ser incorporados a um manifesto do assembly, com [AddModules](#). não há suporte para a criação de um módulo no ambiente de desenvolvimento Visual Studio.

winexe

a opção [winexe](#) faz com que o compilador crie um executável (EXE), Windows programa. O arquivo executável será criado com a extensão .exe. um programa de Windows é um que fornece uma interface do usuário da biblioteca do .net ou com as APIs do Windows. Use o `exe` para criar um aplicativo de console. A menos que especificado de outra forma com a opção [OutputAssembly](#), o nome do arquivo de saída usa o nome do arquivo de entrada que contém o `Main` método. Um e apenas um `Main` método é necessário nos arquivos de código-fonte que são compilados em um arquivo de .exe. A opção [StartupObject](#) permite especificar qual classe contém o `Main` método, em casos em que seu código tem mais de uma classe com um `Main` método.

winmdobj

se você usar a opção [winmdobj](#), o compilador criará um arquivo intermediário .winmdobj que poderá ser convertido em um arquivo binário Windows Runtime (.winmd). O arquivo .winmd pode então ser consumido por programas JavaScript e C++, além de programas de linguagem gerenciada.

A configuração [winmdobj](#) indica para o compilador que um módulo intermediário é necessário. o arquivo .winmdobj pode ser alimentado por meio da [WinMDExp](#) ferramenta de exportação para produzir um arquivo de metadados Windows (.winmd). o arquivo .winmd contém o código da biblioteca original e os metadados de winmd que são usados pelo JavaScript ou C++ e pelo Windows Runtime. A saída de um arquivo que é compilado usando a opção de compilador [winmdobj](#) é usada somente como entrada para a ferramenta de exportação WimMDExp. O próprio arquivo .winmdobj não é referenciado diretamente. A menos que você use a opção [OutputAssembly](#), o nome do arquivo de saída usa o nome do primeiro arquivo de entrada. Um `Main` método não é necessário.

appcontainerexe

se você usar a opção de compilador [appcontainerexe](#), o compilador criará um arquivo Windows executável (.exe) que deve ser executado em um contêiner de aplicativo. essa opção é equivalente a [-target:winexe](#), mas é projetada para aplicativos da loja Windows 8. x.

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 usa o nome do arquivo de entrada que contém o `Main` método.

Opções do compilador C# que especificam entradas

21/01/2022 • 5 minutes to read

As opções a seguir controlam as entradas do compilador. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe *csc.exe* mais antiga é mostrada em `code style`.

- / Referências `-reference` ou `-references` : metadados de referência do arquivo de assembly ou arquivos especificados.
- **AddModules:** / `-addmodule` adicione um módulo (criado com `target:module`) a este assembly.)
- **EmbedInteropTypes:** / `-link` inserir metadados dos arquivos de assembly de interop especificados.

Referências

A opção **Referências** faz com que o compilador importe informações de tipo público no arquivo especificado para o projeto atual, permitindo que você faça referência aos metadados dos arquivos de assembly especificados.

```
<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**:

```
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

No exemplo anterior, é o identificador C# válido que representa um namespace raiz que conterá todos os `LS` namespaces no assembly *filename.dll*. Os arquivos importados devem conter um manifesto. Use **AdditionalLibPaths** para especificar o diretório no qual uma ou mais de suas referências de assembly estão localizadas. O tópico **AdditionalLibPaths** também aborda 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 **Aliases** no elemento **References** para cada arquivo para distinguir 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.

NOTE

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 um módulo que foi criado com a `<TargetType>module</TargetType>` opção para a compilação atual:

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Em `file` que , são arquivos de saída que `file2` 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 executar. 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 executar, você obterá um `TypeLoadException` . `file` não pode conter um assembly. Por exemplo, se o arquivo de saída tiver sido criado **com a opção TargetType** do módulo , seus metadados poderão ser importados com **AddModules**.

Se o arquivo de saída tiver sido criado com uma opção **TargetType** diferente do módulo , seus metadados não poderão ser importados com **AddModules**, mas poderão ser **importados** com a opção Referências.

EmbedInteropTypes

Faz com que o compilador disponibilize as informações de tipo COM nos assemblies especificados para o projeto sendo compilado no momento.

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Em que é uma lista delimitada por ponto e vírgula de `file1;file2;file3` nomes de arquivo de assembly. 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 inseridas. 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](#).

Usar a opção **EmbedInteropTypes** é especialmente útil quando você está trabalhando com a interop 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 interop 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.

NOTE

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 do compilador [References](#), a opção do 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.

```
// 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.

```
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.

```
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 C# para relatar erros e avisos

21/01/2022 • 4 minutes to read

As opções a seguir controlam como o compilador relata erros e avisos. A nova sintaxe de MSBuild é mostrada em **negrito**. A sintaxe de *csc.exe* mais antiga é mostrada em `code style`.

- **WarningLevel** / `-warn` : definir nível de aviso.
- **TreatWarningsAsErrors** / `-warnaserror` : tratar todos os avisos como erros
- **WarningsAsErrors** / `-warnaserror` : trata um ou mais avisos como erros
- **WarningsNotAsErrors** / `-warnnotaserror` : trata um ou mais avisos não como erros
- **DisabledWarnings** / `-nowarn` : definir uma lista de avisos desabilitados.
- **CodeAnalysisRuleSet** / `-ruleset` : especifique um arquivo RuleSet que desabilita o diagnóstico específico.
- Log de erros / `-errorlog` : especifique um arquivo para registrar todos os diagnósticos do compilador e do analisador.
- **ReportAnalyzer** / `-reportanalyzer` : relatar informações adicionais do analisador, como tempo de execução.

WarningLevel

A opção **WarningLevel** especifica o nível de aviso para o compilador Exibir.

```
<WarningLevel>3</WarningLevel>
```

O valor do elemento é o nível de aviso que você deseja exibir para a compilação: os números inferiores mostram apenas avisos de alta severidade. Números mais altos mostram mais avisos. O valor deve ser zero ou um inteiro positivo:

NÍVEL DE AVISO	SIGNIFICADO
0	Desativa a emissão de todas as mensagens de aviso.
1	Exibe mensagens de aviso graves.
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.
5	Exibe avisos de nível 4 mais avisos adicionais do compilador fornecidos com o C# 9.0.

NÍVEL DE AVISO	SIGNIFICADO
Maior que 5	Qualquer valor maior que 5 será tratado como 5. Para ter certeza de que sempre terá todos os avisos se o compilador for atualizado com novos níveis de aviso, coloque um valor grande arbitrário (por exemplo, 9999).

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.

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 o [TreatWarningsAsErrors](#), poderá usar [TreatWarningsAsErrors](#) para listar avisos que não devem ser tratados como erros.

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

Em vez disso, todas as mensagens de aviso são relatadas como erros. O processo de compilação é interrompido (nenhum arquivo de saída é compilado). Por padrão, [TreatWarningsAsErrors](#) não está em vigor, 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 [DisabledWarnings](#) para desabilitar determinados avisos.

WarningsAsErrors e WarningsNotAsErrors

As opções [WarningsAsErrors](#) e [WarningsNotAsErrors](#) substituem a opção [TreatWarningsAsErrors](#) para uma lista de avisos.

Habilite os avisos 0219 e 0168 como erros:

```
<WarningsAsErrors>0219,0168</WarningsAsErrors>
```

Desabilitar os mesmos avisos como erros:

```
<WarningsNotAsErrors>0219,0168</WarningsNotAsErrors>
```

Use [WarningsAsErrors](#) para configurar um conjunto de avisos como erros. Use [WarningsNotAsErrors](#) para configurar um conjunto de avisos que não devem ser erros quando você tiver definido todos os avisos como erros.

DisabledWarnings

A opção [DisabledWarnings](#) permite suprimir o compilador de exibir um ou mais avisos. Separe vários números de aviso com uma vírgula.

```
<DisabledWarnings>number1, number2</DisabledWarnings>
```

`number1 , number2` Número (s) de aviso que você deseja que o compilador omita. Você especifica a parte numérica do identificador de aviso. Por exemplo, se você quiser suprimir *CS0028*, poderá especificar `<DisabledWarnings>28</DisabledWarnings>`. O compilador ignora silenciosamente os números de aviso passados para **DisabledWarnings** que eram válidos em versões anteriores, mas que foram removidos. por exemplo, *CS0679* era válido no compilador em Visual Studio .net 2002, mas foi removido posteriormente.

Os seguintes avisos não podem ser suprimidos pela opção **DisabledWarnings** :

- Aviso do compilador (nível 1) CS2002
- Aviso do compilador (nível 1) CS2023
- Aviso do compilador (nível 1) CS2029

CodeAnalysisRuleSet

Especifique um arquivo RuleSet que configure diagnósticos específicos.

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

Em que `MyConfiguration.ruleset` é o caminho para o arquivo RuleSet. para obter mais informações sobre como usar conjuntos de regras, consulte 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.

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

A opção de log de **erros** faz com que o compilador gere um [log de SARIF \(formato de intercâmbio de resultados de análise\) estático](#). Os logs do SARIF são normalmente lidos por ferramentas que analisam os resultados do compilador e do diagnóstico do Analyzer.

ReportAnalyzer

Reporte informações adicionais do analisador, como o tempo de execução.

```
<ReportAnalyzer>true</ReportAnalyzer>
```

a opção **ReportAnalyzer** faz com que o compilador emita informações adicionais de log de MSBuild que detalham as características de desempenho dos analisadores na compilação. Normalmente, ele é usado por autores do analisador como parte da validação do analisador.

Opções do compilador C# que controlam a geração de código

21/01/2022 • 5 minutes to read

As opções a seguir controlam a geração de código pelo compilador. A nova sintaxe de MSBuild é mostrada em **negrito**. A sintaxe de `csc.exe` mais antiga é mostrada em `code style`.

- **DebugType** / `-debug` : emitir (ou não emitir) informações de depuração.
- **Otimizar** / `-optimize` : habilitar otimizações.
- **Determinístico** / `-deterministic` : produz a saída equivalente byte a 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 com que o compilador gere informações de depuração e coloque-as no arquivo ou arquivos de saída. As informações de depuração são adicionadas por padrão para a configuração de compilação de *depuração*. Ele está desativado por padrão para a configuração de Build de *versão*.

```
<DebugType>pdbonly</DebugType>
```

Para todas as versões do compilador a partir do C# 6.0, não há nenhuma 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>	Emitir informações de depuração para o arquivo <code>.pdb</code> usando o formato padrão para a plataforma atual: Windows : um arquivo pdb Windows. Linux/MacOS : um arquivo PDB portátil .
<code>pdbonly</code>	Mesmo que <code>full</code> . Consulte a observação abaixo para obter mais informações.
<code>portable</code>	Emitir informações de depuração para o arquivo <code>.pdb</code> usando o formato PDB portátil de plataforma cruzada.
<code>embedded</code>	Emitir informações de depuração no <code>.dll/.exe</code> em si (o arquivo <code>.pdb</code> não é produzido) usando o formato PDB portátil .

IMPORTANT

As informações a seguir se aplicam somente a compiladores anteriores ao C# 6.0. O valor desse elemento pode ser `full` ou `pdbonly`. O argumento `completo`, que estará em vigor se você não especificar `pdbonly`, permitirá anexar um depurador ao programa em execução. A especificação de `pdbonly` permite a depuração do código-fonte quando o programa é iniciado no depurador, mas só exibirá o assembler quando o programa em execução estiver anexado ao depurador. Use essa opção para criar builds de depuração. Se você usar `Full`, lembre-se de que há algum impacto sobre a velocidade e o tamanho do código otimizado JIT e um pequeno impacto na qualidade do código com `Full`. Recomendamos `pdbonly` ou nenhum PDB para gerar o código de versão. Uma diferença entre `pdbonly` e `Full` é que, com `Full`, o compilador emite um `DebuggableAttribute`, que é usado para informar ao compilador JIT que as informações de depuração estão disponíveis. Portanto, você receberá um erro se o seu código contiver o `DebuggableAttribute` definido como `false` se você usar `Full`. Para obter informações sobre como configurar o desempenho de depuração de um aplicativo, consulte [Facilitando a Depuração de uma Imagem](#).

Otimizar

A opção **otimizar** habilita ou desabilita as otimizações executadas pelo compilador para tornar o arquivo de saída menor, mais rápido e mais eficiente. A opção **otimizar** é habilitada por padrão para uma configuração de compilação de `versão`. Ele está desativado por padrão para uma configuração de compilação de `depuração`.

```
<Optimize>true</Optimize>
```

Você define a opção **otimizar** na página de propriedades de **compilação** para seu projeto no Visual Studio.

O **Optimize** também diz ao Common Language Runtime para otimizar o código em tempo de execução. Por padrão, as otimizações estão desabilitadas. Especifique **otimizar +** para habilitar otimizações. Ao criar um módulo a ser usado por um assembly, use as mesmas configurações de **otimização** usadas pelo assembly. É possível combinar as opções de **otimização** e de **depuração**.

Determinística

Faz com que o compilador produza um assembly cuja saída byte a byte é idêntica entre compilações para entradas idênticas.

```
<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 que é gerado a partir de números aleatórios. Use a opção `<Deterministic>` para produzir um *assembly determinístico*, cujo conteúdo binário seja idêntico entre compilações, desde que a entrada permaneça a mesma. Nesse tipo de compilação, os campos `TIMESTAMP` 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 determinante:

- 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 fonte está publicamente disponível. Ele também pode determinar se as etapas de compilação que dependem de alterações em binários são usadas no processo de compilação.

ProduceOnlyReferenceAssembly

A opção **ProduceOnlyReferenceAssembly** indica que um assembly de referência deve ser de saída em vez de um assembly de implementação, como a saída primária. O parâmetro **ProduceOnlyReferenceAssembly** desabilita silenciosamente a saída de PDBs, pois os assemblies de referência não podem ser executados.

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Os assemblies de referência são um tipo especial de assembly. Os assemblies de referência contêm apenas a quantidade mínima de metadados necessários para representar a superfície da API pública da biblioteca. Eles incluem declarações para todos os membros que são significativos ao referenciar um assembly em ferramentas de compilação, mas excluem todas as implementações de membro e declarações de membros privados que não têm impacto observável em seu contrato de API. Para obter mais informações, consulte [assemblies de referência](#).

As opções **ProduceOnlyReferenceAssembly** e **ProduceReferenceAssembly** são mutuamente exclusivas.

Opções de compilador C# para segurança

21/01/2022 • 5 minutes to read

As opções a seguir controlam as opções de segurança do compilador. A nova sintaxe de MSBuild é mostrada em **negrito**. A sintaxe de *csc.exe* mais antiga é mostrada em `code style`.

- **PublicSign** / `-publicsign` : assinar publicamente o assembly.
- **DelaySign** / `-delaySign` : assinar 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 randomização de layout de espaço de endereço de entropia alta (ASLR)

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 tempo de execução que o arquivo está assinado.

```
<PublicSign>true</PublicSign>
```

A opção **PublicSign** requer o uso da opção **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 "logon do OSS", a assinatura pública inclui a chave pública em um assembly de saída e define o sinalizador "assinado". A assinatura pública não assina realmente o assembly com uma chave privada. Os desenvolvedores usam o Sign público para projetos de código-fonte aberto. As pessoas criam assemblies que são compatíveis com os assemblies "totalmente assinados" liberados quando não têm acesso à chave privada usada para assinar os assemblies. Como poucos consumidores realmente precisam verificar se o assembly está totalmente assinado, esses assemblies compilados publicamente são utilizáveis em quase todos os cenários em que a assinatura completa seria usada.

DelaySign

Essa opção faz com que o compilador reserve espaço no arquivo de saída para que uma assinatura digital possa ser adicionada mais tarde.

```
<DelaySign>true</DelaySign>
```

Use **delaySign** - se você quiser um assembly totalmente assinado. Use **delaySign** se você quiser apenas posicionar 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 armazena a assinatura. Em vez disso, o compilador, mas reserva espaço no arquivo para que a assinatura possa ser adicionada posteriormente.

O uso de **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.

```
<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 o **módulo-target**, o nome do arquivo de chave será mantido no módulo e incorporado ao assembly criado quando você compilar um assembly com [AddModules](#). Você também pode passar suas informações de criptografia para o compilador com **keycontainer**. Use **delaySign** se você quiser um assembly parcialmente assinado. Caso ambos os **keyfile** e **keycontainer** sejam especificados na mesma compilação, o compilador primeiro tentará o contêiner de chave. Se isso ocorrer, o assembly será assinado com as informações no contêiner de chaves. Se o compilador não encontrar o contêiner de chave, ele tentará o arquivo especificado com **keyfile**. Se isso for bem sucedido, o assembly será assinado com as informações no arquivo de chave e as informações de chave serão instaladas no contêiner de chave. Na próxima compilação, o contêiner de chave 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.

```
<KeyContainer>container</KeyContainer>
```

`container` é o nome do contêiner de chave de nome forte. Quando a opção **keycontainer** é usada, o compilador cria 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. Essa opção não tem suporte 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 ao assembly quando você compilar esse 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). Você também pode passar suas informações de criptografia para o compilador com **keyfile**. Use **delaySign** para adicionar a chave pública ao manifesto do assembly, mas assinar o assembly até que ele tenha sido testado. Para obter mais informações, consulte [Criando e usando assemblies de nomes fortes](#) e [Atraso na Assinatura de um Assembly](#).

HighEntropyVA

a opção de compilador **HighEntropyVA** informa ao kernel do Windows se um executável específico dá suporte a alta entropia (randomização de Layout de espaço de endereço).

```
<HighEntropyVA>true</HighEntropyVA>
```

Esta opção especifica que um executável de 64 bits ou um executável que é marcado pela opção de compilador [PlatformTarget](#) dá suporte a um espaço de endereço virtual de entropia alta. A opção está desabilitada por padrão. Use [HighEntropyVA](#) para habilitá-lo.

a opção [HighEntropyVA](#) permite que versões compatíveis do kernel de Windows usem graus mais altos de entropia ao aleatório do layout de espaço de endereço de um processo como parte de ASLR. Usar graus mais altos de entropia significa que um número maior de endereços pode ser alocado para regiões de memória, como pilhas e heaps. Como resultado, é mais difícil adivinhar o local de uma região de memória específica. A opção de compilador [HighEntropyVA](#) requer o executável de destino e todos os módulos dos quais ele depende podem lidar com valores de ponteiro maiores que 4 GIGABYTES (GB) quando eles estão sendo executados como um processo de 64 bits.

Opções do compilador C# que especificam recursos

21/01/2022 • 5 minutes to read

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` metadados de referência do arquivo de assembly ou arquivos especificados.
- **Win32Manifest:** / `-win32manifest` especifique um arquivo de manifesto win32 (.xml).
- **NoWin32Manifest:** / `-nowin32manifest` não inclua o manifesto win32 padrão.
- **Recursos:** / `-resource` inserir o recurso especificado (Forma curta: /res).
- **LinkResources:** / `-linkresources` vincule o recurso especificado a este assembly.

Win32Resource

A opção **Win32Resource** insere um recurso Win32 no arquivo de saída.

```
<Win32Resource>filename</Win32Resource>
```

`filename` é o arquivo de recurso que você deseja adicionar ao 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.

```
<Win32Icon>filename</Win32Icon>
```

`filename` é o *arquivo .ico* que você deseja adicionar ao arquivo de saída. Um *arquivo .ico* pode ser criado com o [Compilador de Recursos](#). O Compilador de Recursos é invocado quando você compila um Visual C++ de recursos; um *arquivo .ico* é criado a partir do *arquivo .rc*.

Win32Manifest

Use a opção **Win32Manifest** para especificar um arquivo de manifesto do aplicativo Win32 definido pelo usuário a ser inserido no arquivo PE (executável portátil) de um projeto.

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` é o nome e o local do arquivo de manifesto personalizado. Por padrão, o compilador C# incorpora um manifesto do 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.

NOTE

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 arquivo e registro no 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 do Windows (.res) usando a opção **Win32Resource**.
- Você fornece um manifesto personalizado que não especifica um nível de execução solicitado.

Visual Studio cria um arquivo **.manifest** padrão e o armazena nos diretórios de depuração e versão junto 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. Ou você pode clicar com o botão direito do mouse no ícone Project no Gerenciador de Soluções, selecionar **Adicionar Novo Item** e, em seguida, selecionar Arquivo de Manifesto do Aplicativo. Depois de adicionar o arquivo de manifesto novo ou existente, ele será exibido na lista de listadas do manifesto. Para obter mais informações, consulte [Página do Aplicativo, Project Designer \(C#\)](#).

Você pode fornecer o manifesto do aplicativo como uma etapa personalizada pós-build 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.

```
<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 do Aplicativo, Project Designer \(C#\)](#).

Recursos

Insere o recurso especificado no arquivo de saída.

```
<Resources Include="filename">
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` é o arquivo de recurso do .NET que você deseja inserir no arquivo de saída. `identifier` (opcional) é o nome lógico do recurso; o nome usado para carregar o recurso. O padrão é o nome do arquivo.

`accessibility-modifier` (opcional) é a acessibilidade do recurso: público ou privado. O padrão é público. Por padrão, os 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 criado, por exemplo, por Resgen.exe ou no ambiente de desenvolvimento, ele poderá ser acessado com membros no `System.Resources` namespace. 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 na ordem especificada no arquivo de projeto.

Linkresources

Cria um link para um recurso do .NET no arquivo de saída. O arquivo de recurso não é adicionado ao arquivo de saída. `LinkResources` difere da **opção Recurso**, que incorpora um arquivo de recurso no arquivo de saída.

```
<LinkResources Include="filename">
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` é o arquivo de recurso do .NET ao qual você deseja vincular 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` (opcional) é a acessibilidade do recurso: público ou privado. O padrão é público. Por padrão, os recursos vinculados são públicos no assembly quando são criados com 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 criado, por exemplo, por Resgen.exe ou no ambiente de desenvolvimento, ele poderá ser acessado com membros no `System.Resources` namespace. 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#

21/01/2022 • 2 minutes to read

As opções a seguir controlam o comportamento diverso do compilador. A nova MSBuild nova sintaxe é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **ResponseFiles:** / `-@` leia o arquivo de resposta para obter mais opções.
- **NoLogo** / `-nologo`: suprimir a mensagem de direitos autorais do compilador.
- **NoConfig:** / `-noconfig` não inclua automaticamente o arquivo *CSCRSP*

Responsefiles

A opção **ResponseFiles** permite especificar um arquivo que contém opções do compilador e arquivos de código-fonte a compilar.

```
<ResponseFiles>response_file</ResponseFiles>
```

O `response_file` especifica o arquivo que lista as opções do compilador ou os arquivos de código-fonte a compilar. As opções do compilador e os arquivos de código-fonte serão processados pelo compilador como se tivessem 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 aparecer em uma linha (não pode 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 conforme elas são lidas. Os argumentos de linha de comando podem substituir as opções listadas anteriormente nos 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 de 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:

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

A opção **NoLogo** suprime a exibição da faixa de logout quando o compilador é iniciado e a exibição de mensagens informativas durante a compilação.

```
<NoLogo>true</NoLogo>
```

Noconfig

A opção **NoConfig** informa ao compilador para não compilar com o arquivo `csc.rsp`.

```
<NoConfig>true</NoConfig>
```

O arquivo *csc.rsp* faz referência a todos os assemblies fornecidos com .NET Framework. As referências reais que o ambiente de desenvolvimento do Visual Studio .NET inclui dependem do tipo de projeto. Você pode modificar o arquivo *csc.rsp* e especificar opções adicionais do compilador que devem ser incluídas em cada compilação. Se você não quiser 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#

21/01/2022 • 11 minutes to read

As opções a seguir dão suporte a cenários avançados. A nova sintaxe de MSBuild é mostrada em **negrito**. A sintaxe mais antiga `csc.exe` é mostrada em `code style`.

- **MainEntryPoint**, **StartupObject** / `-main` : especifica 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 a saída de nomes de caminho de origem 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 em busca de referências.
- **GenerateFullPaths** / `-fullpath` : o compilador gera caminhos totalmente qualificados.
- **PreferredUILang** / `-preferreduilang` : especifique o nome de idioma de saída preferencial.
- **BaseAddress** / `-baseaddress` : especifique o endereço base para a biblioteca a ser criada.
- **/ChecksumAlgorithm** `-checksumalgorithm` : Especifique o algoritmo para calcular a soma de verificação do arquivo de origem armazenado em PDB.
- **Página de código** / `-codepage` : Especifique a página de código a ser usada ao abrir os arquivos de origem.
- **Utf8output** / `-utf8output` : mensagens do compilador de saída em codificação UTF-8.
- **Alinhamento** / `-filealign` de File: especifique o alinhamento usado para as seções do arquivo de saída.
- **ErrorEndLocation** / `-errorendlocation` : linha de saída e coluna do local final de cada erro.
- **NoStandardLib** / `-nostdlib` : não referencia `mscorlib.dll` de biblioteca padrão.
- **SubsystemVersion** / `-subsystemversion` : Especifique a versão do subsistema deste assembly.
- **Moduleassemblyname** / `-moduleassemblyname` : o nome do assembly do qual este módulo fará parte.

MainEntryPoint ou StartupObject

Esta opção especifica a classe que contém o ponto de entrada para o programa, se mais de uma classe contiver um `Main` método.

```
<StartupObject>MyNamespace.Program</StartupObject>
```

ou

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

Em que `Program` é o tipo que contém o `Main` método. O nome da classe informado deve ser totalmente qualificado. Ele deve incluir o namespace completo que contém a classe, seguido do nome da 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 sua compilação incluir mais de um tipo com um `Main` método, você poderá especificar qual tipo contém o `Main` método.

NOTE

Essa opção não pode ser usada para um projeto que inclui [instruções de nível superior](#), mesmo que esse projeto contenha um ou mais `Main` métodos.

PdbFile

A opção de compilador **PdbFile** especifica o nome e o local do arquivo de símbolos de depuração. O `filename` valor especifica o nome e o local do arquivo de símbolos de depuração.

```
<PdbFile>filename</PdbFile>
```

Quando você especificar **DebugType**, o compilador criará 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 base que o nome do arquivo de saída. **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 de compilador **PathMap** especifica como mapear caminhos físicos para a saída de nomes de caminhos de origem 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 por `path1` em qualquer arquivo de saída. Para especificar vários caminhos de origem mapeados, separe cada um com um ponto e vírgula.

```
<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 de compilador **ApplicationConfiguration** permite que um aplicativo C# especifique o local do arquivo de configuração do aplicativo (`app.config`) do assembly para o Common Language Runtime (CLR) no momento da associação do assembly.

```
<ApplicationConfiguration>file</ApplicationConfiguration>
```

Em que `file` é o arquivo de configuração de aplicativo que contém as configurações de associação de assembly. Um uso de **ApplicationConfiguration** é um cenário avançado no qual um assembly tem que referenciar a versão .NET Framework e o .NET Framework para a versão Silverlight de um determinado assembly de referência ao mesmo tempo. 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 de compilador **ApplicationConfiguration** permite que você especifique o local de um arquivo de app.config que desabilita o comportamento padrão usando uma `<supportPortability>` marca, conforme mostrado no exemplo a seguir.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

O compilador passa o local do arquivo para a lógica de associação de assembly do CLR.

NOTE

Para usar o arquivo de app.config que já está definido no projeto, adicione a marca de propriedade

`<UseAppConfigForCompiler>` ao arquivo `.csproj` e defina seu valor 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 de compilador **ApplicationConfiguration** especifica o local desse app.config arquivo.

```
<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](#).

```
<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 do sistema do Common Language Runtime. `dir2` é um ou mais diretórios adicionais a serem pesquisados para referências de assembly. Separe os nomes de diretório com uma vírgula e sem espaço em branco entre eles. O compilador procura 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 a referência para especificar uma referência de assembly. **AdditionalLibPaths** é aditivo. Especificá-lo mais de uma vez para os valores anteriores. Como o caminho para o assembly dependente não é especificado no manifesto do assembly, o aplicativo encontrará e usará o assembly no cache de assembly global. O compilador que faz referência ao assembly não implica na Common Language Runtime pode 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 **GenerateFullPaths** faz com que o compilador especifique o caminho completo para o arquivo ao listar erros de compilação e avisos.

```
<GenerateFullPaths>true</GenerateFullPaths>
```

Por padrão, erros e avisos oriundos da compilação especificam o nome do arquivo no qual o erro foi encontrado. A opção **GenerateFullPaths** 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 de compilador **PreferredUILang**, você pode especificar o idioma em que o compilador C# exibe a saída, como mensagens de erro.

```
<PreferredUILang>language</PreferredUILang>
```

Em que `language` é o [nome do idioma](#) do idioma a ser usado para a saída do compilador. Você pode usar a opção de compilador **PreferredUILang** para especificar o idioma que você deseja que o compilador C# use para mensagens de erro e outra saída de linha de comando. Se o pacote de idiomas do idioma não estiver instalado, a configuração de idioma do sistema operacional será usada em seu lugar.

BaseAddress

A opção **BaseAddress** permite especificar o endereço base preferencial no qual carregar uma dll. Para obter mais informações sobre quando e por que usar essa opção, consulte o [Blog do Larry Osterman](#).

```
<BaseAddress>address</BaseAddress>
```

Em que `address` é o endereço base para a dll. Esse endereço pode ser especificado como um número decimal, hexadecimal ou octal. O endereço base padrão de uma DLL é definido pelo Common Language Runtime .NET. A palavra de ordem inferior neste endereço será arredondada. Por exemplo, se você especificar `0x11110001`, ele 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 arquivos de origem no PDB.

```
<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

O `algorithm` deve ser `SHA1` (padrão) ou `SHA256`.

CodePage

Esta opção especifica qual página de código usar durante a compilação se a página necessária não for a codepage padrão atual do sistema.

```
<CodePage>id</CodePage>
```

Em que `id` é a ID da página de código a ser usada para todos os arquivos de código-fonte na compilação. O compilador tentará primeiro interpretar todos os arquivos de origem como UTF-8. Se os arquivos de código-fonte estiverem em uma codificação diferente de UTF-8 e usar caracteres que não sejam caracteres ASCII de 7 bits, use a opção **CodePage** para especificar qual página de código deve ser usada. **CodePage** aplica-se a todos os arquivos de código-fonte em 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.

```
<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 especificar o tamanho das seções no arquivo de saída. Os valores válidos são 512, 1024, 2048, 4096 e 8192. Esses valores estão em bytes.

```
<FileAlignment>number</FileAlignment>
```

Você define a opção **FileAlignment** da página **avançado** das propriedades de **compilação** para seu projeto no Visual Studio. Cada seção será alinhada em um limite que seja um múltiplo do valor **filealign**. Não há nenhum padrão fixo. Se **FileAlignment** não for especificado, o Common Language Runtime escolherá um padrão no momento da 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 a produzir a linha e a coluna de saída do local final de cada erro.

```
<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 para cada erro e aviso.

NoStandardLib

NoStandardLib impede a importação de mscorelib.dll, que define o namespace do sistema inteiro.

```
<NoStandardLib>true</NoStandardLib>
```

Use essa opção se desejar definir ou criar seus próprios objetos e namespaces System. Se você não especificar **NoStandardLib**, mscorelib.dll será importado para o programa (o mesmo que especificar `<NoStandardLib>false</NoStandardLib>`).

SubsystemVersion

Especifica a versão mínima do subsistema no qual o arquivo executável é executado. Normalmente, essa opção garante que o arquivo executável possa usar recursos de segurança que não estão disponíveis com versões mais antigas do Windows.

NOTE

Para especificar o subsistema em si, use a opção de compilador [TargetType](#).

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

O `major.minor` especifica a versão mínima necessária do subsistema, conforme expresso em uma notação de ponto para as versões principal e secundária. 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, 1, pois a tabela mais adiante neste artigo descreve. 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
Windows 8	6,02

O valor padrão da opção de compilador **SubsystemVersion** depende das condições na lista a seguir:

- O valor padrão é 6.02 se qualquer opção do compilador na lista a seguir for definida:
 - `-target:appcontainerexe`
 - `-targetwinmdobj`
 - `-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 será 4, 0 se nenhuma das condições anteriores for verdadeira.

ModuleAssemblyName

Especifica o nome de um assembly cujos tipos não públicos um `.netmodule` pode acessar.

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

`ModuleAssemblyName` deve ser usado ao compilar um `.netmodule` e onde as seguintes condições são

verdadeiras:

- 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 Friend ao assembly no qual o `.netmodule` será criado.

Para obter mais informações sobre como criar um `.netmodule`, consulte a opção **TargetType** do módulo. Para obter mais informações sobre assemblies amigos, consulte [Assemblies Amigáveis](#).

Comentários da documentação XML

21/01/2022 • 12 minutes to read

Os arquivos de origem do C# podem ter comentários estruturados que produzem 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 documentação legível na forma de páginas da Web ou arquivos PDF, por exemplo.

Esse processo fornece muitas vantagens para você adicionar documentação de API em seu 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 de API para marcas relevantes.
- As ferramentas que processam os arquivos de documentação XML podem definir elementos e atributos XML específicos para essas ferramentas.

ferramentas como Visual Studio fornecem IntelliSense para muitos elementos XML comuns usados em comentários de documentação.

Este artigo aborda estes tópicos:

- Comentários de documentação e geração de arquivo XML
- Marcas validadas pelo compilador C# e Visual Studio
- Formato do arquivo XML gerado

Criar saída de documentação XML

Você cria a documentação para seu código escrevendo 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:

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass {}
```

Você define a opção [GenerateDocumentationFile](#) ou [documentafle](#), e o compilador encontrará todos os campos de comentário com marcas XML no código-fonte e criará um arquivo de documentação XML a partir desses comentários. Quando essa opção está habilitada, o compilador gera o aviso de [CS1591](#) para qualquer membro publicamente visível declarado em seu projeto sem comentários de documentação XML.

Formatos de comentário XML

O uso de comentários de documento XML requer delimitadores que indicam onde um comentário de 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 C# usam este formulário. Se houver espaço em branco após o delimitador, ele não será incluído na saída XML.

NOTE

Visual Studio insere automaticamente as `<summary>` `</summary>` marcas e posiciona o cursor dentro dessas marcas depois de digitar o `///` delimitador no editor de códigos. Você pode ativar ou desativar esse recurso na caixa de diálogo Opções.

- `/** */` Delimitadores de várias linhas: os `/** */` delimitadores têm as seguintes regras de formatação:
 - Na linha que contém o `/**` delimitador, se o restante 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 espaço em branco até o `*/` delimitador, essa linha será ignorada. Caso contrário, o texto na linha até o `*/` delimitador será processado como parte do comentário.
 - Para as linhas após a que comece 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 comece com o `/**` delimitador ou terminar com o `*/` delimitador, ele ignorará esse padrão para cada linha.
 - A única parte do comentário a seguir que é processada é a linha que comece com `<summary>`. Os três formatos de marca produzem os mesmos comentários.

```
/** <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.

```
/**  
* <summary>  
* text </summary>*/
```

- O compilador não encontra nenhum padrão comum no comentário a seguir 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.

```
/**  
* <summary>  
text </summary>  
*/
```

- O compilador não encontra nenhum padrão no seguinte comentário por dois motivos. Primeiro, o número de espaços antes que o asterisco não seja consistente. Em segundo lugar, a quinta linha

começa com uma tabulação, que não corresponde a espaços. Todo o texto das linhas de dois a cinco é processado como parte do comentário.

```
/**  
 * <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 (`<` e `>`). Para consultar identificadores genéricos em elementos de referência de código (`cref`), você pode usar os caracteres de escape (por exemplo, `cref="List<T>"`) 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.

NOTE

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 de documentação XML

As seguintes ferramentas criam a saída de comentários XML:

- [DocFX](#): o *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 compila um site HTML estático do seu código-fonte e arquivos de redução. Além disso, o DocFX fornece a flexibilidade para personalizar o layout e o estilo do seu site por meio de modelos. Você também pode criar modelos personalizados.
- [Sandcastle](#): as *ferramentas de Sandcastle* criam arquivos de ajuda para bibliotecas de classes gerenciadas que contêm páginas de referência conceitual e de API. As ferramentas de Sandcastle são baseadas em linha de comando e não têm nenhum front-end de GUI, recursos de gerenciamento de projeto ou processo de compilação automatizado. O construtor de arquivos de ajuda do Sandcastle fornece uma GUI autônoma e ferramentas baseadas em linha de comando para criar um arquivo de ajuda de maneira automatizada. Um pacote de integração do Visual Studio também está disponível para ele, de forma que os projetos de ajuda possam ser criados e gerenciados inteiramente de dentro de Visual Studio.
- [Doxygen](#): o *Doxygen* gera um navegador de documentação online (em HTML) ou um manual de referência off-line (no LaTeX) de um conjunto de arquivos de origem documentados. Também há suporte para gerar saída em RTF (MS Word), PostScript, em PDF com hiperlink, em HTML compactado, DocBook e em páginas do manual do 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 o membro. A cadeia de caracteres de ID deve considerar operadores, parâmetros, valores de retorno, parâmetros de tipo genérico,, `ref` `in` e `out` parâmetros. 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 caracteres de ID para identificar os metadados .NET correspondentes ou o item de reflexão 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 de caracteres 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	Você não pode adicionar comentários de documentação a um namespace, mas pode fazer referências CREF a eles, quando houver suporte.
T	type	Um tipo é uma classe, interface, struct, enum ou delegado.
F	field	
P	propriedade	Inclui indexadores ou outras propriedades indexadas.
M	method	Inclui métodos especiais, como construtores e operadores.
E	event	
!	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 item em si tiver períodos, eles serão substituídos pelo sinal de hash ('#'). Supõe-se que nenhum item tenha um sinal de hash diretamente em seu nome. Por exemplo, o nome totalmente qualificado do construtor String é "System.String.#ctor".
- Para propriedades e métodos, segue a lista de parâmetros entre parênteses. Se não houver parâmetros, nenhum parêntese estará presente. Os parâmetros são separados por vírgulas. A codificação de cada parâmetro segue diretamente como ele é codificado em uma assinatura do .NET (consulte para definições de todos os elementos [Microsoft.VisualStudio.CorDebugInterop.CorElementType](#) 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, , , e) são representados como o nome totalmente `ELEMENT_TYPE_I4` qualificado do tipo completo `ELEMENT_TYPE_OBJECT` `ELEMENT_TYPE_STRING` `ELEMENT_TYPE_TYPEDBYREF` `ELEMENT_TYPE_VOID` 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 modificadora, seguindo o tipo modificado.

- `ELEMENT_TYPE_SZARRAY` é representado como "[]" após o tipo de elemento da matriz.
- `ELEMENT_TYPE_ARRAY` é representado como `[lowerbound:, lowerbound :]` em que o número de vírgulas é a classificação - 1 e os limites inferiores e o tamanho de cada dimensão, se conhecido, são representados em `size size` 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 os limites inferiores e tamanhos não especificados é `[1;1:]`.
- Para operadores de conversão somente (`e`), o valor de retorno do método é `op_Implicit` codificado como um seguido pelo tipo de `op_Explicit ~` retorno. Por exemplo:
`<member name="M:System.Decimal.op_Explicit(System.Decimal arg)~System.Int32">` é a marca para o operador cast declarado na classe `public static explicit operator int (decimal value); 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 marca para um tipo definido como
`public class SampleClass<T, U>`. Para métodos que têm tipos genéricos como parâmetros, os parâmetros de tipo genérico são especificados como números prefigurados com aticks (por exemplo, `0, `1`). Cada número representa uma notação de matriz baseada em zero para os 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 modificadora, seguindo 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: (assinatura)", em que é o tipo de retorno e a assinatura são os `type type` 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 usados para diferenciar métodos sobrecarregados:
 - convenção de chamada
 - tipo de retorno
 - `ELEMENT_TYPE_SENTINEL`

Os exemplos a seguir mostram como as cadeias de caracteres de ID de uma classe e seus membros são geradas:

```
namespace MyNamespace
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:MyNamespace.X".
    /// </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".
        /// </summary>
    }
}
```

```

/// In 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.X)~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 do C#

21/01/2022 • 12 minutes to read

Os comentários de 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 nos comentários da 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 vários motivos.

O que vem a seguir são algumas recomendações, cenários de caso de uso geral e coisas que você deve saber ao usar marcas de documentação XML em seu código C#. Embora você possa colocar qualquer marca em seus comentários de documentação, este artigo descreve as marcas recomendadas para as construções de linguagem mais comuns. Em todos os casos, você deve aderir a estas recomendações:

- Para fins de consistência, todos os tipos publicamente visíveis e seus membros públicos devem ser documentados.
- Membros particulares também podem ser documentados usando comentários XML. No entanto, ele expõe os trabalhos internos (potencialmente confidenciais) de 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.
- As classes parciais têm suporte total e as informações da documentação serão concatenadas em uma única entrada para cada tipo.

A documentação XML começa com `///`. Quando você cria um novo projeto, os modelos colocam algumas linhas iniciais `///` no 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 `<param>` marca é usada para descrever os parâmetros. Se ela é usada, o compilador verifica se o parâmetro existe e se todos os parâmetros são descritos na documentação. Se a verificação falhar, o compilador emitirá um aviso.
 - O `cref` atributo 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 falhar, o compilador emitirá um aviso. O compilador respeita qualquer instrução `using` quando procura por um tipo descrito no atributo `cref`.
 - a `<summary>` marca é usada pelo IntelliSense dentro de Visual Studio para exibir informações adicionais sobre um tipo ou membro.

NOTE

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 junto com a reflexão no tipo ou membro real.

- Os desenvolvedores são livres para criar seu próprio conjunto de marcas. O compilador irá copiá-los para o

arquivo de saída.

Algumas das marcas recomendadas podem ser usadas em qualquer elemento de linguagem. Outros 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 seu uso.

O compilador verifica a sintaxe dos elementos seguidos por um único * na lista a seguir. Visual Studio fornece 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 Visual Studio validam as ``, `<i>`, `<u>`, `
` e `<a>`. O compilador também valida `<tt>`, que é o HTML preterido.

- **Marcas gerais** usadas para vários elementos – essas marcas são o conjunto mínimo para 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 saída da documentação** -essas marcas fornecem instruções de formatação para ferramentas que geram documentação.
 - `<para>`
 - `<list>`
 - `<c>`
 - `<code>`
 - `<example>` **
- **Texto de documentação de reutilização** -essas marcas fornecem ferramentas que facilitam a reutilização de comentários XML.
 - `<inheritdoc>` **
 - `<include>` *
- **Gerar links e referências** -essas marcas geram links para outras documentações.
 - `<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>`

NOTE

Os comentários de documentação não podem ser aplicados a um namespace.

Se você quiser que os colchetes angulares apareçam no texto de um comentário de documentação, use a codificação HTML de `<` e `>`, que é `<` e, `>` respectivamente. Essa codificação é mostrada no exemplo a seguir.

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

Marcas gerais

<summary>

```
<summary>description</summary>
```

A `<summary>` marca deve ser usada para descrever um tipo ou um membro de tipo. Use `<remarks>` para adicionar informações complementares a uma descrição de tipo. Use o [atributo cref](#) para habilitar ferramentas de documentação, como [DocFX](#) e [Sandcastle](#), para criar hiperlinks internos para páginas de documentação para elementos de código. O texto da `<summary>` marca é a única fonte de informações sobre o tipo no IntelliSense e também é exibido na janela pesquisador de objetos.

<remarks>

```
<remarks>
description
</remarks>
```

A `<remarks>` marca é usada para adicionar informações sobre um tipo ou um membro de tipo, complementando as informações especificadas com `<summary>`. Essas informações são exibidas na janela do Pesquisador de Objetos. Essa marca pode incluir mais explicações demoradas. Você pode achar que usar `CDATA` seções para redução torna mais conveniente escrevê-la. Ferramentas como [docfx](#) processam o texto de redução nas `CDATA` seções.

Membros do documento

<returns>

```
<returns>description</returns>
```

A `<returns>` marca deve ser usada no comentário para uma declaração de método para descrever o valor de retorno.

<param>

```
<param name="name">description</param>
```

- `name` : O nome de um parâmetro de método. Coloque o nome entre aspas duplas (" "). Os nomes dos parâmetros devem corresponder à assinatura de API. Se um ou mais parâmetros não forem cobertos, o compilador emitirá um aviso. O compilador também emitirá um aviso se o valor de `name` não corresponder a um parâmetro formal na declaração do método.

A `<param>` marca deve ser usada no comentário para uma declaração de método para descrever um dos parâmetros para o método. Para documentar vários parâmetros, use várias `<param>` marcas. O texto da `<param>` marca é exibido no IntelliSense, no Pesquisador de objetos e no relatório da Web de comentários de código.

<paramref>

```
<paramref name="name"/>
```

- `name` : O nome do parâmetro ao qual se refere. Coloque o nome entre aspas duplas (" ").

A `<paramref>` marca fornece uma maneira de indicar que uma palavra nos comentários do código, por exemplo, em um `<summary>` `<remarks>` bloco ou se refere 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>`

```
<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 `<exception>` marca permite especificar quais exceções podem ser geradas. Essa marca pode ser aplicada às definições de métodos, propriedades, eventos e indexadores.

`<value>`

```
<value>property-description</value>
```

A `<value>` marca permite descrever o valor que uma propriedade representa. quando você adiciona uma propriedade por meio do assistente de código no ambiente de desenvolvimento Visual Studio .net, ela adiciona uma `<summary>` marca para a nova propriedade. Você adiciona manualmente uma `<value>` marca para descrever o valor que a propriedade representa.

Formatar saída da documentação

`<para>`

```
<remarks>
    <para>
        This is an introductory paragraph.
    </para>
    <para>
        This paragraph contains more details.
    </para>
</remarks>
```

A `<para>` marca é para uso dentro de uma marca, como `<summary>` , `<remarks>` ou `<returns>` , e permite que você adicione a estrutura ao texto. A `<para>` marca cria um parágrafo com espaçamento duplo. Use a `
` marca se desejar um único parágrafo com espaço.

`<list>`

```
<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 `<listheader>` bloco é usado para definir a linha de cabeçalho de uma tabela ou lista de definições. Ao definir uma tabela, você só precisa fornecer uma entrada para `term` no título. Cada item na lista é especificado com um `<item>` bloco. Ao criar uma lista de definições, você precisará 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 tantos `<item>` blocos quantos forem necessários.

`<c>`

```
<c>text</c>
```

A `<c>` marca fornece uma maneira de indicar que o texto dentro de uma descrição deve ser marcado como código. Use `<code>` para indicar várias linhas como código.

`<code>`

```
<code>
  var index = 5;
  index++;
</code>
```

A `<code>` marca é usada para indicar várias linhas de código. Use `<c>` para indicar que o texto de linha única dentro de uma descrição deve ser marcado como código.

`<example>`

```
<example>
This shows how to increment an integer.
<code>
  var index = 5;
  index++;
</code>
</example>
```

A `<example>` marca permite especificar um exemplo de como usar um método ou outro membro da biblioteca. Um exemplo geralmente envolve o uso da `<code>` marca.

Reutilizar texto da documentação

`<inheritdoc>`

```
<inheritdoc [cref=""] [path=""]/>
```

Herde comentários XML de classes base, interfaces e métodos semelhantes. O uso `inheritdoc` elimina a cópia indesejada e a colagem de comentários XML duplicados e mantém automaticamente os comentários XML sincronizados. Observe que quando você adiciona a `<inheritdoc>` marca a um tipo, todos os membros também

herdarão os comentários.

- `cref` : Especifique o membro do qual herdar a documentação. As marcas já definidas no membro atual não são substituídas pelos herdadas.
- `path` : A consulta de expressão XPath que resultará em um conjunto de nós a ser mostrado. Você pode usar esse atributo para filtrar as marcas a serem incluídas ou excluídas da documentação herdada.

Adicione seus comentários XML em classes ou interfaces base e deixe que `inheritdoc` Copie os comentários para a implementação de classes. Adicione seus comentários XML aos seus métodos síncronos e deixe que `inheritdoc` Copie os comentários para suas versões assíncronas dos mesmos métodos. Se você quiser copiar os comentários de um membro específico, use o `cref` atributo para especificar o membro.

<include>

```
<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 `filename` que leva à marca `name`. Coloque o caminho entre aspas simples (' ').
- `name` : O especificador de nome na marca que precede os comentários; terá `name` um `id` .
- `id` : A ID da marca que precede os comentários. Coloque a ID entre aspas duplas (" ").

A `<include>` marca permite que você faça referência a comentários em outro arquivo que descreva os tipos e membros em seu código-fonte. A inclusão de um arquivo externo é uma alternativa para colocar comentários de documentação diretamente no 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 check-out do arquivo de código-fonte e outra pessoa pode ter feito o check-out do arquivo de documentação. A `<include>` marca usa a sintaxe XPath XML. Consulte a documentação do XPath para obter maneiras de personalizar seu `<include>` uso.

Gerar links e referências

<see>

```
/// <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 que está disponível para ser chamado a partir 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 um 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 vinculado a `https://github.com` .
- `langword="keyword"` : Uma palavra-chave Language, como `true` ou uma das outras **palavras-chave** válidas.

A `<see>` marca permite especificar um link de dentro do texto. Use `<seealso>` para indicar que o texto deve ser colocado em uma seção ver também. Use o **atributo cref** para criar hiperlinks 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 hiperlink.

<seealso>

```
/// <seealso cref="member"/>
// or
/// <seealso href="link">Link Text</seealso>
```

- `cref="member"` : Uma referência a um membro ou campo que está disponível para ser chamado a partir 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 vinculado a `https://github.com`.

A `<seealso>` marca permite especificar o texto que você pode querer que apareça em uma seção **Ver também** . Use `<see>` para especificar um link de dentro do texto. Não é possível aninhar a `seealso` marca dentro da `summary` marca.

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 hiperlinks para a página em que o tipo ou o membro está documentado.

atributo href

O `href` atributo 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>

```
<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 `<typeparam>` marca será exibido no IntelliSense.

<typeparamref>

```
<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 Sandcastle oferecem suporte para marcas extras `<event>` , como e `<note>` , e até mesmo dão suporte a [namespaces de documentação](#). As ferramentas de geração de documentação personalizada ou interna também podem ser usadas com as marcas padrão, e é possível dar suporte a vários formatos de saída de HTML para PDF.

Exemplo de comentários de documentação XML

21/01/2022 • 18 minutes to read

Este artigo contém três exemplos para adicionar comentários de documentação XML à maioria dos elementos da linguagem C#. O primeiro exemplo mostra como documentar uma classe com membros diferentes. A segunda mostra como você reutilizaria explicações para uma hierarquia de classes ou interfaces. A terceira mostra as marcas a usar 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.

```
/// <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://docs.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 desorganizar seu código-fonte com grandes conjuntos de comentários destinados aos usuários da sua biblioteca. Você usa a `<Include>` marca para separar seus comentários XML da origem. Seu código-fonte faz referência a um arquivo XML com a `<Include>` marca :

```

/// <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 da documentação.

```

<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 `<inheritdoc>` elemento significa que um tipo ou membro *herda comentários* de documentação de uma classe base ou interface. Você também pode usar o `<inheritdoc>` elemento com o atributo para herdar comentários de um membro do mesmo `cref` tipo. O exemplo a seguir mostra maneiras de usar essa marca. Observe que, quando você adiciona o `inheritdoc` atributo 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 sobre os comentários herdados.

```

/// <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 `<typeparam>` marca 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:

```
/// <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 matemática.

```
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>
```

```
/// </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 `include` marca. Você move toda a documentação para um arquivo

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>
```

```

</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

`<include>` marca para referenciar o elemento apropriado no arquivo XML:

```

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 `id` atributo , que pode ser usado no lugar de , representa a `name` ID da marca que precede os comentários.

Erros do compilador de C#

21/01/2022 • 2 minutes to read

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 Visual Studio, escolha o número do erro (por exemplo, CS0029) na [Janela de Saída](#), em seguida, escolha a tecla F1.
- Digite o número do 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 incluam o número ou o texto do erro.

para obter informações sobre como configurar opções de erro e aviso em c#, consulte [opções do compilador c#](#) ou a [página Visual Studio Build, Project Designer \(C#\)](#).

NOTE

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 \(opções do compilador C#\)](#)
- [DisabledWarnings \(opções do compilador C#\)](#)

Correspondência de padrões para C# 7

21/01/2022 • 29 minutes to read

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 recursos básicos são: [tipos de registro](#), que são tipos cujo significado semântico é descrito pela forma dos dados; e correspondência de padrões, que é uma nova forma de expressão que permite a decomposição de vários níveis extremamente conciso desses tipos de dados. Os elementos dessa abordagem são inspirados por recursos relacionados nas linguagens de programação F# e [escala](#).

Expressão Is

O `is` operador é estendido para testar uma expressão em relação a um *padrão*.

```
relational_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 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

Padrões são usados no `is` operador e em uma *switch_statement* para expressar a forma de dados em que os dados de entrada devem ser comparados. Os padrões podem ser recursivos para que as partes dos dados possam ser correspondidas em subpadrões.

```

pattern
: declaration_pattern
| constant_pattern
| var_pattern
;

declaration_pattern
: type simple_designation
;

constant_pattern
: shift_expression
;

var_pattern
: 'var' simple_designation
;

```

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 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.

Padrão de declaração

O `declaration_pattern` testa se uma expressão é de um determinado tipo e a converte para esse tipo se o teste for bem sucedido. Se o `simple_designation` for um identificador, ele apresentará uma variável local do tipo fornecido nomeado pelo identificador fornecido. Essa variável local é *definitivamente atribuída* quando o resultado da operação de correspondência de padrões é `true`.

```

declaration_pattern
: type simple_designation
;
```

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), o resultado de `is operator` é `true`. Ele declara uma nova variável local denominada pelo *identificador* que recebe o valor do operando à esquerda quando o resultado 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 como *padrão compatível* com o tipo `T` se houver uma conversão de identidade, uma conversão de referência implícita, uma conversão boxing, uma conversão de referência explícita ou uma conversão unboxing de `E` para `T`. É um erro de tempo de compilação se uma expressão do tipo `E` não for compatível com o tipo de padrão de tipo no qual ele é correspondido.

Observação: no C# 7.1, estendemos isso para permitir uma operação de correspondência de padrões se o tipo de entrada ou o tipo `T` for um tipo aberto. Este parágrafo é substituído pelo seguinte:

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 como *padrão compatível* com o tipo `T` se houver uma conversão de identidade, uma conversão de referência implícita, uma conversão boxing, uma conversão de referência explícita ou uma conversão unboxing de `E` para `T` ou se `E` `T` for ou for um tipo aberto. É um erro de tempo de compilação se uma expressão do tipo `E` não for compatível com o tipo de padrão de tipo no qual ele é correspondido.

O padrão de declaração é útil para executar testes de tipo de tempo de execução de tipos de referência e substitui o idioma

```
var v = expr as Type;  
if (v != null) { // code using v }
```

Com um pouco mais conciso

```
if (expr is Type v) { // code using v }
```

Erro se o *tipo* for um tipo de valor anulável.

O padrão de declaração 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

```
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.

Padrão de constante

```
constant_pattern  
: shift_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 ou uma `typeof` expressão.

Se `e` e `c` forem de tipos integrais, o padrão será considerado correspondido se o resultado da expressão `e == c` for `true`.

Caso contrário, o padrão será considerado correspondente se `object.Equals(e, c)` retornar `true`. Nesse caso, é um erro de tempo de compilação se o tipo estático de `e` não for *compatível* com o tipo da constante.

Padrão de var

```
var_pattern  
: 'var' simple_designation  
;
```

Uma expressão `e` corresponde a uma `var_pattern` sempre. Em outras palavras, uma correspondência a um *padrão de var* sempre é realizada com sucesso. Se o `simple_designation` for um identificador, em tempo de execução o valor de `e` será associado a uma variável local introduzida recentemente. O tipo da variável local é o tipo estático de `e`.

Erro se o nome for `var` associado a um tipo.

Instrução switch

A `switch` instrução é estendida para selecionar para execução o primeiro bloco com um padrão associado que

corresponda à *expressão de comutador*.

```
switch_label
: 'case' complex_pattern case_guard? ':'
| 'case' constant_expression case_guard? ':'
| 'default' ':'
;

case_guard
: 'when' expression
;
```

A ordem na qual os padrões são correspondidos não está definida. Um compilador tem permissão para corresponder padrões fora de ordem e reutilizar os resultados de padrões já correspondidos para calcular o resultado da correspondência de outros padrões.

Se uma *proteção de caso* estiver presente, sua expressão será do tipo `bool`. Ele é avaliado como uma condição adicional que deve ser satisfeita para que o caso seja considerado satisfeito.

Erro se um *switch_label* não puder ter nenhum efeito no tempo de execução porque seu padrão é incorporadas por casos anteriores. [TODO: devemos ser mais precisos sobre as técnicas que o compilador precisa usar para alcançar esse julgamento.]

Uma variável de padrão declarada em um *switch_label* é definitivamente atribuída em seu bloco de caso se e somente se esse bloco de caso contiver precisamente um *switch_label*.

[TODO: devemos especificar quando um *bloco de alternância* pode ser acessado.]

Escopo de variáveis de padrão

O escopo de uma variável declarada em um padrão é o seguinte:

- Se o padrão for um rótulo de caso, o escopo da variável será o *bloco de caso*.

Caso contrário, a variável é declarada em uma expressão *is_pattern*, e seu escopo é baseado na construção que está delimitando imediatamente a expressão que contém a expressão *is_pattern* da seguinte maneira:

- Se a expressão estiver em uma lambda Expression-aptô para, seu escopo será o corpo do lambda.
- Se a expressão estiver em um método ou Propriedade apto para de expressão, seu escopo será o corpo do método ou da propriedade.
- Se a expressão estiver em uma `when` cláusula de uma `catch` cláusula, seu escopo será essa `catch` cláusula.
- Se a expressão estiver em um *iteration_statement*, seu escopo será apenas essa instrução.
- Caso contrário, se a expressão estiver em algum outro formulário de instrução, seu escopo será o escopo que contém a instrução.

Com a finalidade de determinar o escopo, um *embedded_statement* é considerado em seu próprio escopo. Por exemplo, a gramática para uma *if_statement* é

```
if_statement
: 'if' '(' boolean_expression ')' embedded_statement
| 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
;
```

Portanto, se a instrução controlada de um *if_statement* declarar uma variável de padrão, seu escopo será restrito a esse *embedded_statement*.

```
if (x) M(y is var z);
```

Nesse caso, o escopo de `z` é a instrução incorporada `M(y is var z);`.

Outros casos são erros por outros motivos (por exemplo, no valor padrão de um parâmetro ou em um atributo, ambos são um erro porque esses contextos exigem uma expressão constante).

Em C# 7.3, adicionamos os seguintes contextos nos quais uma variável de padrão pode ser declarada:

- Se a expressão estiver em um *inicializador de Construtor*, seu escopo será o *inicializador de Construtor* e o corpo do construtor.
- Se a expressão estiver em um inicializador de campo, seu escopo será o *equals_value_clause* em que ele aparece.
- Se a expressão estiver em uma cláusula de consulta que é especificada para ser convertida no corpo de um lambda, seu escopo será apenas essa expressão.

Alterações na desambiguidade sintática

Há situações que envolvem genéricos em que a gramática do C# é ambígua e a especificação da linguagem indica como resolver essas ambiguidades:

ambiguidades de gramática 7.6.5.2

As produções para *nome simples* (§ 7.6.3) e *acesso de membro* (§ 7.6.5) podem dar aumento às ambiguidades na gramática para expressões. Por exemplo, a instrução:

```
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 *nome simples* (§ 7.6.3), *acesso de membro* (§ 7.6.5) ou *acesso de membro de ponteiro* (§ 18.5.2) terminando com uma *lista de argumentos de tipo* (§ 4.4.1), o token imediatamente após o token de fechamento `>` será examinado. Se for um de

```
( ) ] } : ; , . ? == != | ^
```

em seguida, a *lista de argumentos de tipo* é mantida como parte do *nome simples*, *acesso de membro* ou *ponteiro-acesso de membro* e qualquer outra análise possível da sequência de tokens é descartada. Caso contrário, a *lista de argumentos de tipo* não é considerada como parte do *nome simples*, *acesso de membro* ou *> ponteiro- acesso de membro*, mesmo se não houver outra análise possível da sequência de tokens.

Observe que essas regras não são aplicadas ao analisar uma *lista de argumentos de tipo* em um *namespace-ou-Type-name* (§ 3.8). A instrução

```
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

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

cada um será interpretado como uma chamada para `F` com dois argumentos. A instrução

```
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 como um *nome simples* com uma *lista de argumentos de tipo* seguido por um operador Binary Plus. Na instrução

```
x = y is C<T> + z;
```

os tokens `C<T>` são interpretados como um *namespace-ou-tipo-nome* com uma *lista de argumentos de tipo*.

Há várias alterações introduzidas no C# 7 que tornam essas regras de Desambigüidade não mais suficientes para lidar com a complexidade da linguagem.

Declarações de variável out

Agora é possível declarar uma variável em um argumento out:

```
M(out Type name);
```

No entanto, o tipo pode ser genérico:

```
M(out A<B> name);
```

Como a gramática de idioma do argumento usa *expression*, esse contexto está sujeito à regra de desambiguidade. Nesse caso, o fechamento `>` é seguido por um *identificador*, que não é um dos tokens que permite que ele seja tratado como uma *lista de argumentos de tipo*. Portanto, propondo para **Adicionar o identificador ao conjunto de tokens que dispara a Desambigüide para uma lista de argumentos de tipo**.

Declarações de tuplas e de desconstrução

Um literal de tupla é executado exatamente com o mesmo problema. Considere a expressão de tupla

```
(A < B, C > D, E < F, G > H)
```

Nas regras antigas do C# 6 para analisar uma lista de argumentos, isso seria analisado como uma tupla com quatro elementos, começando com `A < B` o primeiro. No entanto, quando isso aparece à esquerda de uma desconstrução, queremos a desambiguidade disparada pelo token do *identificador*, conforme descrito acima:

```
(A<B,C> D, E<F,G> H) = e;
```

Essa é uma declaração de desconstrução que declara duas variáveis, a primeira que é do tipo `A<B,C>` e nomeada `D`. Em outras palavras, o literal de tupla contém duas expressões, cada uma delas como uma expressão de declaração.

Para simplificar a especificação e o compilador, eu propondo que esse literal de tupla seja analisado como uma tupla de dois elementos onde quer que ele apareça (seja ou não exibido no lado esquerdo de uma atribuição). Isso seria um resultado natural da Desambigüide descrita na seção anterior.

Correspondência de padrões

A correspondência de padrões introduz um novo contexto em que surge a ambiguidade do tipo de expressão.

Anteriormente, o lado direito de um `is` operador era um tipo. Agora ele pode ser um tipo ou uma expressão e, se for um tipo, pode ser seguido por um identificador. Isso pode, tecnicamente, alterar o significado do código existente:

```
var x = e is T < A > B;
```

Isso pode ser analisado sob as regras do C# 6 como

```
var x = ((e is T) < A) > B;
```

Mas, sob as regras do C# 7 (com a Inambiguidade proposta acima) seria analisado como

```
var x = e is T<A> B;
```

que declara uma variável `B` do tipo `T<A>`. Felizmente, os compiladores nativo e Roslyn têm um bug no qual eles fornecem um erro de sintaxe no código C# 6. Portanto, essa alteração significativa em particular não é uma preocupação.

Correspondência de padrões introduz tokens adicionais que devem orientar a resolução de ambiguidade para selecionar um tipo. Os seguintes exemplos de código C# 6 válido serão desfeitos sem regras de Desambiguidade adicionais:

```
var x = e is A<B> && f;           // &&
var x = e is A<B> || f;            // ||
var x = e is A<B> & f;            // &
var x = e is A<B>[];             // [
```

Alteração proposta na regra de desambiguidade

Eu proponho para revisar a especificação para alterar a lista de tokens de ambiguidade de

```
( ) ] } : ; , . ? == != | ^
```

como

```
( ) ] } : ; , . ? == != | ^ && || & [
```

E, em determinados contextos, tratamos o *identificador* como um token de ambiguidade. Esses contextos são onde a sequência de tokens que estão sendo desambiguados é imediatamente precedida por uma das palavras-chave `is`, `case` ou `ou` `out` surge durante a análise do primeiro elemento de um literal de tupla (nesse caso, os tokens são precedidos por `(` ou `:` e o identificador é seguido por um `,` `)` ou um elemento subsequente de um literal de tupla).

Regra de desambiguidade modificada

A regra de desambiguidade revisada seria algo assim

Se uma sequência de tokens puder ser analisada (no contexto) como um *nome simples* (§ 7.6.3), *acesso de membro* (§ 7.6.5) ou *acesso de membro de ponteiro* (§ 18.5.2) terminando com uma *lista de argumentos de tipo* (§ 4.4.1), o token imediatamente após o token de fechamento `>` é examinado, para ver se ele está

- Um de `()] } : ; , . ? == != | ^ && || & [`; ou

- Um dos operadores relacionais `< > <= >= is as`; ou
- Uma palavra-chave de consulta contextual que aparece dentro de uma expressão de consulta; ou
- Em determinados contextos, tratamos o *identificador* como um token de ambiguidade. Esses contextos são onde a sequência de tokens que estão sendo desambiguados é imediatamente precedida por uma das palavras-chave `is`, `case` ou `out` ou surge durante a análise do primeiro elemento de um literal de tupla (nesse caso, os tokens são precedidos por `(` ou `:` e o identificador é seguido por um `,` ou `)` ou um elemento subsequente de um literal de tupla.

Se o seguinte token estiver entre essa lista ou um identificador nesse contexto, a lista de argumentos de *tipo* será mantida como parte do *nome simples*, *acesso de membro* ou *ponteiro-acesso de membro* e qualquer outra análise possível da sequência de tokens será descartada. Caso contrário, a *lista de argumentos de tipo* não será considerada como parte do *nome simples*, *acesso de membro* ou *acesso de membro de ponteiro*, mesmo que não haja nenhuma outra análise possível da sequência de tokens. Observe que essas regras não são aplicadas ao analisar uma *lista de argumentos de tipo* em um *namespace-ou-Type-name* (§ 3,8).

Alterações recentes devido a esta proposta

Nenhuma alteração significativa é conhecida devido a essa regra de Desambigüidade proposta.

Exemplos interessantes

Aqui estão alguns resultados interessantes dessas regras de Desambigüidade:

A expressão `(A < B, C > D)` é uma tupla com dois elementos, cada uma comparando.

A expressão `(A<B,C> D, E)` é uma tupla com dois elementos, o primeiro deles é uma expressão de declaração.

A invocação `M(A < B, C > D, E)` tem três argumentos.

A invocação `M(out A<B,C> D, E)` tem dois argumentos, o primeiro é uma `out` declaração.

A expressão `e is A C` usa uma expressão de declaração.

O rótulo `case` `case A C:` usa uma expressão de declaração.

Alguns exemplos de correspondência de padrões

Is-As

Podemos substituir o idioma

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

Com um pouco mais conciso e direto

```
if (expr is Type v) {
    // code using v
}
```

Testando anulável

Podemos substituir o idioma

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

Com um pouco mais conciso e direto

```
if (x?.y?.z is Type value) {
    // code using value
}
```

Simplificação aritmética

Suponha que definimos um conjunto de tipos recursivos para representar expressões (por uma proposta separada):

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

Agora, podemos definir uma função para computar a derivada (não reduzida) de uma expressão:

```
Expr Deriv(Expr e)
{
    switch (e) {
        case X(): return Const(1);
        case Const(*): return Const(0);
        case Add(var Left, var Right):
            return Add(Deriv(Left), Deriv(Right));
        case Mult(var Left, var Right):
            return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
        case Neg(var Value):
            return Neg(Deriv(Value));
    }
}
```

Um simplificador de expressão demonstra padrões posicionais:

```
Expr Simplify(Expr e)
{
    switch (e) {
        case Mult(Const(0), *): return Const(0);
        case Mult(*, Const(0)): return Const(0);
        case Mult(Const(1), var x): return Simplify(x);
        case Mult(var x, Const(1)): return Simplify(x);
        case Mult(Const(var l), Const(var r)): return Const(l*r);
        case Add(Const(0), var x): return Simplify(x);
        case Add(var x, Const(0)): return Simplify(x);
        case Add(Const(var l), Const(var r)): return Const(l+r);
        case Neg(Const(var k)): return Const(-k);
        default: return e;
    }
}
```

Funções locais

21/01/2022 • 3 minutes to read

Estendemos o C# para dar suporte à declaração de funções no escopo de bloco. As funções locais podem usar variáveis (capturar) do escopo delimitador.

O compilador usa a análise de fluxo para detectar quais variáveis uma função local usa antes de atribuir um valor a ela. Cada chamada da função requer que essas variáveis sejam definitivamente atribuídas. Da mesma forma, o compilador determina quais variáveis são definitivamente atribuídas no retorno. Essas variáveis são consideradas definitivamente atribuídas depois que a função local é invocada.

As funções locais podem ser chamadas de um ponto léxico antes de sua definição. As instruções de declaração de função local não causam um aviso quando não estão acessíveis.

TODO: *criar especificação*

Gramática de sintaxe

Essa gramática é representada como uma comparação da gramática de especificações atual.

```
declaration-statement
  : local-variable-declaration ';'
  | local-constant-declaration ';'
+  | local-function-declaration
  ;

+local-function-declaration
+  : local-function-header local-function-body
+  ;
+local-function-header
+  : local-function-modifiers? return-type identifier type-parameter-list?
+    ( formal-parameter-list? ) type-parameter-constraints-clauses
+  ;
+local-function-modifiers
+  : (async | unsafe)
+  ;
+local-function-body
+  : block
+  | arrow-expression-body
+  ;
```

As funções locais podem usar variáveis definidas no escopo delimitador. A implementação atual requer que todas as variáveis lidas dentro de uma função local sejam definitivamente atribuídas, como se estiver executando a função local em seu ponto de definição. Além disso, a definição da função local deve ter sido "executada" em qualquer ponto de uso.

Depois de experimentar esse bit (por exemplo, não é possível definir duas funções locais recursivas mutuamente), já que revisamos como queremos que a atribuição definitiva funcione. A revisão (ainda não implementada) é que todas as variáveis locais lidas em uma função local devem ser definitivamente atribuídas em cada invocação da função local. Isso é, na verdade, mais sutil do que parece, e há um monte de trabalho restante para fazê-lo funcionar. Depois de fazer isso, você poderá mover suas funções locais para o final de seu bloco delimitador.

As novas regras de atribuição definitivas são incompatíveis com a inferência do tipo de retorno de uma função local, portanto, provavelmente, vamos remover o suporte para inferir o tipo de retorno.

A menos que você converta uma função local em um delegado, a captura é feita em quadros que são tipos de valor. Isso significa que você não obtém nenhuma pressão de GC do uso de funções locais com a captura.

Acessibilidade

Adicionamos à especificação

O corpo de uma expressão lambda apto para de instrução ou função local é considerado acessível.

Declarações de variável out

21/01/2022 • 2 minutes to read

O recurso de *declaração de variável out* permite que uma variável seja declarada no local que está sendo passado como um `out` argumento.

```
argument_value
: 'out' type identifier
| ...
;
```

Uma variável declarada dessa forma é chamada de uma *variável out*. Você pode usar a palavra-chave contextual `var` para o tipo da variável. O escopo será o mesmo para uma variável de *padrão* introduzida por meio de correspondência de padrões.

De acordo com a especificação da linguagem (seção acesso ao elemento 7.6.7), a lista de argumentos de um acesso de elemento (expressão de indexação) não contém argumentos ref ou out. No entanto, eles são permitidos pelo compilador para vários cenários, por exemplo indexadores declarados em metadados que aceitam `out`.

Dentro do escopo de uma variável local introduzida por um `argument_value`, é um erro de tempo de compilação para se referir a essa variável local em uma posição textual que precede sua declaração.

Também é um erro fazer referência a uma variável de saída de tipo implícito (§ 8.5.1) na mesma lista de argumentos que contém imediatamente sua declaração.

A resolução de sobrecarga é modificada da seguinte maneira:

Adicionamos uma nova conversão:

Há uma *conversão de expressão* de uma declaração de variável digitada implicitamente para cada tipo.

Também

O tipo de um argumento de variável explicitamente digitado é o tipo declarado.

e

Um argumento de variável de tipo implícito não tem nenhum tipo.

A *conversão de expressão* de uma declaração de variável de tipo implícito não é considerada melhor do que qualquer outra *conversão da expressão*.

O tipo de uma variável de digitação implícita é o tipo do parâmetro correspondente na assinatura do método selecionado pela resolução de sobrecarga.

O novo nó de sintaxe `DeclarationExpressionSyntax` é adicionado para representar a declaração em um argumento var de saída.

Expressão throw

21/01/2022 • 2 minutes to read

Estendemos o conjunto de formulários de expressões para incluir

```
throw_expression
: 'throw' null_coalescing_expression
;

null_coalescing_expression
: throw_expression
;
```

As regras de tipo são as seguintes:

- Um *throw_expression* não tem nenhum tipo.
- Uma *throw_expression* é conversível para cada tipo por uma conversão implícita.

Uma *expressão throw* gera o valor produzido avaliando o *null_coalescing_expression*, que 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 tem `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.

O comportamento em tempo de execução da avaliação de uma *expressão throw* é o mesmo [especificado para uma instrução Throw](#).

As regras de análise de fluxo são as seguintes:

- Para cada variável *v*, *v* é definitivamente atribuída antes da *null_coalescing_expression* de um *throw_expression* IFF ele é definitivamente atribuído antes da *throw_expression*.
- Para cada variável *v*, *v* é definitivamente atribuído após *throw_expression*.

Uma *expressão throw* é permitida somente nos seguintes contextos sintáticos:

- Como o segundo ou terceiro operando de um operador condicional Ternário `?:`
- Como o segundo operando de um operador de União nulo `??`
- Como o corpo de um lambda ou método apto para de expressão.

Literais binários

21/01/2022 • 2 minutes to read

Há uma solicitação relativamente comum para adicionar literais binários a C# e VB. Para bitmasks (por exemplo, enumerações de sinalizador), isso parece autênticamente útil, mas também seria ótimo apenas para fins educacionais.

Os literais binários teriam a seguinte aparência:

```
int nineteen = 0b10011;
```

Sintaticamente e semanticamente eles são idênticos a literais hexadecimais, exceto para usar `b` / `B` em vez de `x` / `X`, ter apenas dígitos `0` e `1` e sendo interpretados na base 2 em vez de 16.

Há pouco custo para implementá-los e pouca sobrecarga conceitual para os usuários do idioma.

Syntax

A gramática seria a seguinte:

```
integer-literal:  
  : ...  
  | binary-integer-literal  
  ;  
binary-integer-literal:  
  : `0b` binary-digits integer-type-suffix-opt  
  | `0B` binary-digits integer-type-suffix-opt  
  ;  
binary-digits:  
  : binary-digit  
  | binary-digits binary-digit  
  ;  
binary-digit:  
  : `0`  
  | `1`  
  ;
```

Separadores de dígito

21/01/2022 • 2 minutes to read

Ser capaz de agrupar dígitos em grandes literais numéricos teria um grande impacto de legibilidade e nenhuma desvantagem significativa.

A adição de literais binários (#215) aumentaria a probabilidade de literais numéricos serem longos, de modo que os dois recursos se aprimorem.

Seguimos o Java e outros e usamos um sublinhado _ como separador de dígito. Seria possível ocorrer em todos os lugares em um literal numérico (exceto o primeiro e último caractere), já que diferentes agrupamentos podem fazer sentido em cenários diferentes e, especialmente, para diferentes bases numéricas:

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2_3_4_5_6_7_8_9;
double real = 1_000.111_1e-1_000;
```

Qualquer sequência de dígitos pode ser separada por sublinhados, possivelmente mais de um sublinhado entre dois dígitos consecutivos. Eles são permitidos em decimais, bem como expoentes, mas após a regra anterior, eles podem não aparecer ao lado do decimal (`10_.0`), ao lado do caractere expoente (`1.1e_1`) ou ao lado do especificador de tipo (`10_f`). Quando usado em literais binários e hexadecimais, eles podem não aparecer imediatamente após o `0x` ou o `0b`.

A sintaxe é simples e os separadores não têm impacto semântico-eles são simplesmente ignorados.

Isso tem um valor amplo e é fácil de implementar.

Tipos de tarefas assíncronas em C

21/01/2022 • 5 minutes to read

Estenda `async` para oferecer suporte a *tipos de tarefas* que correspondam a um padrão específico, além dos tipos bem conhecidos `System.Threading.Tasks.Task` e `System.Threading.Tasks.Task<T>`.

Tipo de Tarefa

Um *tipo de tarefa* é um `class` ou `struct` com um *tipo de Construtor* associado identificado com `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`. O *tipo de tarefa* pode ser não genérico, para métodos assíncronos que não retornam um valor, ou genérico, para métodos que retornam um valor.

Para dar suporte `await` a, o *tipo de tarefa* deve ter um método acessível correspondente `GetAwaiter()` que retorna uma instância de um *tipo de aguardador* (consulte *expressões Await 7.7.7.1 do C#*).

```
[AsyncMethodBuilder(typeof(MyTaskMethodBuilder<>))]
class MyTask<T>
{
    public Awarter<T> GetAwaiter();
}

class Awarter<T> : INotifyCompletion
{
    public bool IsCompleted { get; }
    public T GetResult();
    public void OnCompleted(Action completion);
}
```

Tipo de Construtor

O *tipo de Construtor* é um `class` ou `struct` que corresponde ao tipo de *tarefa* específico. O *tipo de Construtor* pode ter no máximo 1 parâmetro de tipo e não deve ser aninhado em um tipo genérico. O *tipo de Construtor* tem os `public` métodos a seguir. Para tipos não genéricos de *Construtor*, `SetResult()` não tem parâmetros.

```

class MyTaskMethodBuilder<T>
{
    public static MyTaskMethodBuilder<T> Create();

    public void Start<TStateMachine>(ref TStateMachine stateMachine)
        where TStateMachine : IAsyncStateMachine;

    public void SetStateMachine(IAsyncStateMachine stateMachine);
    public void SetException(Exception exception);
    public void SetResult(T result);

    public void AwaitOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : INotifyCompletion
        where TStateMachine : IAsyncStateMachine;
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : ICriticalNotifyCompletion
        where TStateMachine : IAsyncStateMachine;

    public MyTask<T> Task { get; }
}

```

Execução

Os tipos acima são usados pelo compilador para gerar o código para o computador de estado de um `async` método. (O código gerado é equivalente ao código gerado para métodos assíncronos que retornam `Task`, `Task<T>` ou `void`. A diferença é, para esses tipos bem conhecidos, os *tipos de Construtor* também são conhecidos pelo compilador.)

`Builder.Create()` é invocado para criar uma instância do *tipo de Construtor*.

Se o computador de estado for implementado como um `struct`, o `builder.SetStateMachine(stateMachine)` será chamado com uma instância em caixa do computador de estado que o construtor pode armazenar em cache, se necessário.

`builder.Start(ref stateMachine)` é invocado para associar o Construtor à instância de máquina de estado gerada pelo compilador. O construtor deve chamar `stateMachine.MoveNext()` em `Start()` ou depois de `Start()` foi retornado para avançar o computador de estado. Depois de `Start()` retorna, o `async` método chama `builder.Task` a tarefa para retornar do método `Async`.

Cada chamada para `stateMachine.MoveNext()` irá avançar a máquina de estado. Se a máquina de estado for concluída com êxito, `builder.SetResult()` será chamada, com o valor de retorno do método, se houver. Se uma exceção for lançada no computador de estado, `builder.SetException(exception)` será chamado.

Se a máquina de estado atingir uma `await expr` expressão, `expr.GetAwaiter()` será invocada. Se o aguardador implementar `ICriticalNotifyCompletion` e `IsCompleted` for false, o computador de estado será invocado `builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine)`. `AwaitUnsafeOnCompleted()` deve chamar `awaiter.OnCompleted(action)` com uma ação que chama `stateMachine.MoveNext()` quando o aguardador é concluído. Da mesma forma para `INotifyCompletion` e `builder.AwaitOnCompleted()`.

Resolução de sobrecarga

A resolução de sobrecarga é estendida para reconhecer *tipos de tarefa* além de `Task` e `Task<T>`.

Um `async` lambda sem valor de retorno é uma correspondência exata para um parâmetro candidato de sobrecarga de tipo de *tarefa* não genérica e um `async` lambda com tipo de retorno `T` é uma correspondência exata para um parâmetro candidato de sobrecarga de *tipo de tarefa* genérico.

Caso contrário, se um `async` lambda não for uma correspondência exata para um dos dois parâmetros candidatos de *tipos de tarefa*, ou uma correspondência exata para ambos, e houver uma conversão implícita de um tipo candidato para o outro, o de candidato vence. Caso contrário, avalie recursivamente os tipos `A` e `B` em `Task1<A>` e `Task2` para obter uma correspondência melhor.

Caso contrário, se um `async` lambda não for uma correspondência exata para um dos dois parâmetros candidatos de *tipos de tarefa*, mas um candidato for um tipo mais especializado do que o outro, o candidato mais especializado será o vencedor.

Assíncrono principal

21/01/2022 • 4 minutes to read

- [x] proposta
- [] Protótipo
- [] Implementação
- [] Especificação

Resumo

Permitir que `await` seja usado no método Main/EntryPoint de um aplicativo, permitindo que o ponto de entrada retorne `Task` / `Task<int>` e seja marcado `async`.

Motivação

É muito comum ao aprender sobre o C#, ao escrever utilitários baseados em console e ao escrever pequenos aplicativos de teste que desejam chamar e `await` `async` métodos do principal. Hoje, adicionamos um nível de complexidade, forçando tal `await` a ser feito em um método assíncrono separado, o que faz com que os desenvolvedores precisem escrever texto clichê como o seguinte apenas para começar:

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

private static async Task MainAsync()
{
    ... // Main body here
}
```

Podemos remover a necessidade desse texto clichê e torná-lo mais fácil de começar simplesmente permitindo que o próprio principal seja `async` tal que `await` s possa ser usado.

Design detalhado

No momento, as seguintes assinaturas são de entryPoints permitidas:

```
static void Main()
static void Main(string[])
static int Main()
static int Main(string[])
```

Estendemos a lista de entryPoints permitidos para incluir:

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

Para evitar riscos de compatibilidade, essas novas assinaturas só serão consideradas como entryPoints válidos se não houver sobrecargas do conjunto anterior. O idioma/compilador não exigirá que o ponto de entrada seja

marcado como `async`, embora espere que a grande maioria dos usos será marcada como tal.

Quando um deles é identificado como EntryPoint, o compilador sintetiza um método EntryPoint real que chama um desses métodos codificados:

- `static Task Main()` resultará no compilador que emite o equivalente de
`private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[])` resultará no compilador que emite o equivalente de
`private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` resultará no compilador que emite o equivalente de
`private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` resultará no compilador que emite o equivalente de
`private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

Exemplo de uso:

```
using System;
using System.Net.Http;

class Test
{
    static async Task Main(string[] args) =>
        Console.WriteLine(await new HttpClient().GetStringAsync(args[0]));
}
```

Desvantagens

A principal desvantagem é simplesmente a complexidade adicional de dar suporte a assinaturas de EntryPoint adicionais.

Alternativas

Outras variantes consideradas:

Permitindo `async void`. Precisamos manter a semântica o mesmo para o código chamando diretamente, o que dificultaria um ponto de entrada gerado para chamá-lo (nenhuma tarefa foi retornada). Poderíamos resolver isso gerando dois outros métodos, por exemplo,

```
public static async void Main()
{
    ... // await code
}
```

se torna

```
public static async void Main() => await $MainTask();

private static void $EntryptionMain() => Main().GetAwaiter().GetResult();

private static async Task $MainTask()
{
    ... // await code
}
```

Também há preocupações em relação à incentivação do uso do `async void`.

Usando "MainAsync" em vez de "Main" como o nome. Embora o sufixo assíncrono seja recomendado para métodos de retorno de tarefas, trata-se principalmente da funcionalidade da biblioteca, que principal não é, e o suporte a nomes de EntryPoint adicionais além de "Main" não vale a pena.

Perguntas não resolvidas

N/D

Criar reuniões

N/D

Literal "padrão" de tipo de destino

21/01/2022 • 3 minutes to read

- [x] proposta
- [x] protótipo
- [x] implementação
- [] Especificação

Resumo

O recurso de tipo de destino `default` é uma variação de forma mais curta do `default(T)` operador, que permite que o tipo seja omitido. Seu tipo é inferido por digitação de destino em vez disso. Além disso, ele se comporta como `default(T)`.

Motivação

A principal motivação é evitar digitar informações redundantes.

Por exemplo, ao invocar `void Method(ImmutableArray<SomeType> array)`, o literal *padrão* permite `M(default)` no lugar de `M(default(ImmutableArray<SomeType>))`.

Isso é aplicável em vários cenários, como:

- declarando locais (`ImmutableArray<SomeType> x = default;`)
- operações ternário (`var x = flag ? default : ImmutableArray<SomeType>.Empty;`)
- retornando em métodos e lambdas (`return default;`)
- declarando valores padrão para parâmetros opcionais (
`void Method(ImmutableArray<SomeType> arrayOpt = default)`)
- incluindo valores padrão em expressões de criação de matriz (
`var x = new[] { default, ImmutableArray.Create(y) };`)

Design detalhado

Uma nova expressão é introduzida, a literal *padrão*. Uma expressão com essa classificação pode ser convertida implicitamente em qualquer tipo, por uma *conversão literal padrão*.

A inferência do tipo para o literal *padrão* funciona da mesma forma que para o literal *nulo*, exceto que qualquer tipo é permitido (não apenas tipos de referência).

Essa conversão produz o valor padrão do tipo inferido.

O literal *padrão* pode ter um valor constante, dependendo do tipo inferido. Portanto `const int x = default;`, é legal, mas `const int? y = default;` não é.

O literal *padrão* pode ser o operando de operadores de igualdade, desde que o outro operando tenha um tipo. Portanto `default == x` `x == default`, são expressões válidas, mas são `default == default` ilegais.

Desvantagens

Uma desvantagem secundária é que o literal *padrão* pode ser usado no lugar do literal *nulo* na maioria dos contextos. Duas das exceções são `throw null;` e `null == null`, que são permitidas para o literal *nulo*, mas não

para o literal *padrão*.

Alternativas

Há algumas alternativas a serem consideradas:

- O status quo: o recurso não é justificado em seus próprios méritos e os desenvolvedores continuam a usar o operador padrão com um tipo explícito.
- Estendendo o literal nulo: essa é a abordagem do VB com `Nothing`. Poderíamos permitir `int x = null;`.

Perguntas não resolvidas

- [x] o *padrão* deve ser permitido como o operando dos operadores *is* ou *as*? Resposta: não permitir `default is T`, permitir `x is default`, permitir `default as RefType` (com aviso sempre nulo)

Criar reuniões

- [LDM 3/7/2017](#)
- [LDM 3/28/2017](#)
- [LDM 5/31/2017](#)

Inferir nomes de tupla (também conhecido como inicializadores de projeção de tupla)

21/01/2022 • 4 minutes to read

Resumo

Em vários casos comuns, esse recurso permite que os nomes de elementos de tupla sejam omitidos e, em vez disso, sejam inferidos. Por exemplo, em vez de digitar `(f1: x.f1, f2: x?.f2)`, os nomes de elemento "F1" e "F2" podem ser inferidos de `(x.f1, x?.f2)`.

Isso paraleliza o comportamento de tipos anônimos, que permitem inferir nomes de membros durante a criação. Por exemplo, `new { x.f1, y?.f2 }` declara os membros "F1" e "F2".

Isso é particularmente útil ao usar tuplas no LINQ:

```
// "c" and "result" have element names "f1" and "f2"
var result = list.Select(c => (c.f1, c.f2)).Where(t => t.f2 == 1);
```

Design detalhado

Há duas partes para a alteração:

1. Tente inferir um nome de candidato para cada elemento de tupla que não tem um nome explícito:
 - Usando as mesmas regras que a inferência de nomes para tipos anônimos.
 - No C#, isso permite três casos: `y` (identificador), `x.y` (acesso de membro simples) e `x?.y` (acesso condicional).
 - No VB, isso permite casos adicionais, como `x.y()`.
 - Rejeitar nomes de tupla reservada (diferencia maiúsculas de minúsculas em C#, não diferencia maiúsculas de minúsculas no VB), pois elas são proibidas ou já implícitas. Por exemplo, como `ItemN`, `Rest` e `ToString`.
 - Se qualquer nome de candidato for duplicado (diferencia maiúsculas de minúsculas no C#, não diferencia maiúsculas de minúsculas no VB) em toda a tupla, descartamos esses candidatos,
2. Durante as conversões (que verificam e avisam sobre descartar nomes de literais de tupla), nomes deduzidos não produzirão nenhum aviso. Isso evita a interrupção do código de tupla existente.

Observe que a regra para lidar com duplicatas é diferente daquela para tipos anônimos. Por exemplo, `new { x.f1, x.f1 }` produz um erro, mas `(x.f1, x.f1)` ainda seria permitido (apenas sem nenhum nome deduzido). Isso evita a interrupção do código de tupla existente.

Para consistência, o mesmo se aplicaria a tuplas produzidas por degroup-assignments (em C#):

```
// tuple has element names "f1" and "f2"
var tuple = ((x.f1, x?.f2) = (1, 2));
```

O mesmo também se aplicaria a tuplas VB, usando as regras específicas do VB para inferir o nome da expressão e comparações de nome que não diferenciam maiúsculas de minúsculas.

Ao usar o compilador C# 7,1 (ou posterior) com a versão de idioma "7,0", os nomes de elemento serão inferidos (apesar de o recurso não estar disponível), mas haverá um erro de site de uso para tentar acessá-los. Isso

limitará adições de novo código que posteriormente enfrentaria o problema de compatibilidade (descrito abaixo).

Desvantagens

A principal desvantagem é que isso apresenta uma interrupção de compatibilidade do C# 7,0:

```
Action y = () => M();  
var t = (x: x, y);  
t.y(); // this might have previously picked up an extension method called "y", but would now call the  
lambda.
```

O Conselho de compatibilidade descobriu essa interrupção aceitável. Considerando que ela é limitada e a janela de tempo desde que as tuplas enviadas (em C# 7,0) é curta.

Referências

- [LDM 4º 2017 de abril](#)
- [Discussão sobre o GitHub](#) (Obrigado @alrz por trazer esse problema)
- [Design de tuplas](#)

correspondência de padrões com genéricos

21/01/2022 • 3 minutes to read

- [x] proposta
- [] Protótipo:
- [] Implementação:
- [] Especificação:

Resumo

A especificação para o [C# como operador existente](#) permite que não haja nenhuma conversão entre o tipo do operando e o tipo especificado quando um é um tipo aberto. No entanto, no C# 7, o `Type identifier` padrão exige que haja uma conversão entre o tipo de entrada e o tipo fornecido.

Sugerimos que você relaxe e altere `expression is Type identifier`, além de ser permitido nas condições, quando permitido no C# 7, também seja permitido quando for `expression as Type` permitido. Especificamente, os novos casos são casos em que o tipo da expressão ou o tipo especificado é um tipo aberto.

Motivação

Casos em que a correspondência de padrões deve ser permitida no momento com falha na compilação. Consulte, por exemplo, <https://github.com/dotnet/roslyn/issues/16195>.

Design detalhado

Alteramos o parágrafo na especificação de correspondência de padrões (a adição proposta é mostrada em negrito):

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 como *padrão compatível* com o tipo `T` se houver uma conversão de identidade, uma conversão de referência implícita, uma conversão boxing, uma conversão de referência explícita ou uma conversão unboxing de `E` para `T` se `E` `T` for ou for um tipo aberto. É um erro de tempo de compilação se uma expressão do tipo `E` não for compatível com o tipo de padrão de tipo no qual ele é correspondido.

Desvantagens

Nenhum.

Alternativas

Nenhum.

Perguntas não resolvidas

Nenhum.

Criar reuniões

O LDM considerou essa pergunta e sentiu que era uma alteração no nível de correção de bug. Estamos

tratando-o como um recurso de idioma separado porque apenas fazer a alteração depois que o idioma foi liberado introduzirá uma incompatibilidade posterior. O uso da alteração proposta requer que o programador especifique a versão de idioma 7,1.

Referências somente leitura

21/01/2022 • 47 minutes to read

- [x] proposta
- [x] protótipo
- [x] implementação: iniciada
- [] Especificação: não iniciada

Resumo

O recurso "referências somente leitura" é, na verdade, um grupo de recursos que aproveitam a eficiência da passagem de variáveis por referência, mas sem expor os dados às modificações:

- `in` parâmetro
- Retornos de `ref readonly`
- `readonly` structs
- `ref / in` métodos de extensão
- `ref readonly` locais
- `ref` expressões condicionais

Passando argumentos como referências somente leitura.

Há uma proposta existente que toca este tópico <https://github.com/dotnet/roslyn/issues/115> como um caso especial de parâmetros `ReadOnly` sem entrar em muitos detalhes. Aqui, quero apenas reconhecer que a ideia propriamente dita não é muito nova.

Motivação

Antes desse recurso, C# não tinha uma maneira eficiente de expressar um desejo de passar variáveis de `struct` em chamadas de método para fins somente leitura sem nenhuma intenção de modificar. O argumento regular por valor passando implica em copiar, o que adiciona custos desnecessários. Isso orienta os usuários a usar o argumento-`ref` passando e a contar com comentários/documentação para indicar que os dados não devem ser modificados pelo receptor. Não é uma boa solução por muitos motivos.

Os exemplos são muitos operadores matemáticos/matrizes matemáticas em bibliotecas gráficas, como o `XNA` é conhecido por ter operandos de referência puramente devido a considerações de desempenho. Há código no próprio compilador Roslyn que usa `structs` para evitar alocações e as passa por referência para evitar a cópia de custos.

Solução (`in` parâmetros)

Da mesma forma que os `out` parâmetros, os `in` parâmetros são passados como referências gerenciadas com garantias adicionais do receptor.

Ao contrário dos `out` parâmetros que *devem* ser atribuídos pelo receptor antes de qualquer outro uso, os `in` parâmetros não podem ser atribuídos pelo receptor.

Como um resultado, `in` os parâmetros permitem a eficácia da passagem de argumentos indiretos sem expor argumentos a mutações pelo receptor.

Declarando parâmetros `in`

`in` os parâmetros são declarados usando `in` a palavra-chave como um modificador na assinatura de parâmetro.

Para todas as finalidades `in`, o parâmetro é tratado como uma `readonly` variável. A maioria das restrições sobre o uso de `in` parâmetros dentro do método são as mesmas dos `readonly` campos.

Na verdade, um `in` parâmetro pode representar um `readonly` campo. A similaridade de restrições não é uma coincidência.

Por exemplo, os campos de um `in` parâmetro que tem um tipo struct são todos classificados recursivamente como `readonly` variáveis.

```
static Vector3 Add (in Vector3 v1, in Vector3 v2)
{
    // not OK!!
    v1 = default(Vector3);

    // not OK!!
    v1.X = 0;

    // not OK!!
    foo(ref v1.X);

    // OK
    return new Vector3(v1.X + v2.X, v1.Y + v2.Y, v1.Z + v2.Z);
}
```

- `in` os parâmetros são permitidos em qualquer lugar em que parâmetros de ByVal comuns são permitidos. Isso inclui indexadores, operadores (incluindo conversões), delegados, lambdas, funções locais.

```
(in int x) => x                                // lambda expression
TValue this[in TKey index];                      // indexer
public static Vector3 operator +(in Vector3 x, in Vector3 y) => ... // operator
```

- `in` Não é permitido em combinação com `out` ou com qualquer coisa que `out` não combine com.
- Não é permitido sobrecarregar as `ref` / `out` / `in` diferenças.
- Ele tem permissão para se sobrecarregar em ByVal e `in` diferenças comuns.
- Com a finalidade de OHI (sobrecarregar, ocultar, implementar), `in` o comporta de forma semelhante a um `out` parâmetro. Todas as mesmas regras se aplicam. Por exemplo, o método de substituição terá que corresponder `in` parâmetros com `in` parâmetros de um tipo de identidade conversível.
- Para fins de conversões de grupo delegate/lambda/Method, `in` comporta-se da mesma forma que um `out` parâmetro. Os lambdas e os candidatos à conversão do grupo de métodos aplicáveis precisarão corresponder aos `in` parâmetros do delegado de destino com `in` parâmetros de um tipo de identidade conversível.
- Para fins de variância genérica, os `in` parâmetros são nonvariant.

Observação: não há avisos sobre `in` parâmetros que têm tipos de referência ou primitivos. Pode ser inútil em geral, mas, em alguns casos, o usuário deve/deseja passar primitivos como `in`. Exemplos – substituindo um método genérico como `Method(in T param)` quando `T` foi substituído para ser `int`, ou ao ter métodos como `Volatile.Read(in int location)`

É concebível ter um analisador que avisa em casos de uso ineficiente de `in` parâmetros, mas as regras para essa análise seriam muito difusas para fazer parte de uma especificação de linguagem.

Uso do `in` em sites de chamada. (`in` argumentos)

Há duas maneiras de passar argumentos para `in` parâmetros.

`in` os argumentos podem corresponder a `in` parâmetros:

Um argumento com um `in` modificador no site de chamada pode corresponder a `in` parâmetros.

```
int x = 1;

void M1<T>(in T x)
{
    // ...
}

var x = M1(in x); // in argument to a method

class D
{
    public string this[in Guid index];
}

D dictionary = . . . ;
var y = dictionary[in Guid.Empty]; // in argument to an indexer
```

- `in` o argumento deve ser um LValue /legível/(*). Exemplo: `M1(in 42)` é inválido

(*) A noção de [LValue/rvalue](#) varia entre os idiomas.

Aqui, por LValue, quero dizer uma expressão que representa um local que pode ser referenciado diretamente. E RValue significa uma expressão que produz um resultado temporário que não persiste por conta própria.

- Em particular, é válido passar `readonly` campos, `in` parâmetros ou outras variáveis formais `readonly` como `in` argumentos. Exemplo: `dictionary[in Guid.Empty]` é legal. `Guid.Empty` é um campo somente leitura estático.
- `in` o argumento deve ter o tipo *Identity-conversível* para o tipo do parâmetro. Exemplo:
`M1<object>(in Guid.Empty)` é inválido. `Guid.Empty` Não é *conversível de identidade* para `object`

A motivação das regras acima é que os `in` argumentos garantem a *alias* da variável de argumento. O receptor sempre recebe uma referência direta para o mesmo local, conforme representado pelo argumento.

- em raras situações em que `in` os argumentos devem ser despejados em pilha devido a `await` expressões usadas como operandos da mesma chamada, o comportamento é o mesmo que with `out` e `ref` arguments – se a variável não puder ser despejada de maneira referencialmente transparente, um erro será relatado.

Exemplos:

1. `M1(in staticField, await SomethingAsync())` é válido. `staticField` é um campo estático que pode ser acessado mais de uma vez sem efeitos colaterais observáveis. Portanto, a ordem dos efeitos colaterais e os requisitos de alias podem ser fornecidos.
2. `M1(in RefReturningMethod(), await SomethingAsync())` produzirá um erro. `RefReturningMethod()` é um `ref` método de retorno. Uma chamada de método pode ter efeitos colaterais observáveis, portanto, ela deve ser avaliada antes do `SomethingAsync()` operando. No entanto, o resultado da invocação é uma referência que não pode ser preservada no ponto de suspensão, o `await` que torna impossível o requisito de referência direta.

Observação: os erros de despejo de pilha são considerados limitações específicas da implementação.

Portanto, eles não têm efeito sobre a resolução de sobrecarga ou a inferência de lambda.

Argumentos de ByVal comuns podem corresponder a `in` parâmetros:

Argumentos regulares sem modificadores podem corresponder a `in` parâmetros. Nesse caso, os argumentos têm as mesmas restrições relaxadas que os argumentos comuns de ByVal teriam.

A motivação para esse cenário é que os `in` parâmetros nas APIs podem resultar em inconveniências para o usuário quando argumentos não podem ser passados como referência direta-ex: literais, resultados computados ou `await`-Ed ou argumentos que ocorrem para ter tipos mais específicos.

Todos esses casos têm uma solução trivial de armazenar o valor do argumento em um local temporário do tipo apropriado e passar esse local como um `in` argumento.

Para reduzir a necessidade de tal compilador de código clichê pode executar a mesma transformação, se necessário, quando o `in` modificador não está presente no site de chamada.

Além disso, em alguns casos, como invocação de operadores ou `in` métodos de extensão, não há nenhuma maneira sintática de especificar `in`. Isso sozinho requer a especificação do comportamento de argumentos de ByVal comuns quando eles correspondem aos `in` parâmetros.

Especialmente:

- é válido passar RValues. Uma referência a um temporário é passada nesse caso. Exemplo:

```
Print("hello");      // not an error.

void Print<T>(in T x)
{
    //...
}
```

- conversões implícitas são permitidas.

Isso é, na verdade, um caso especial para passar um RValue

Uma referência a um valor temporário em retenção é transmitida nesse caso. Exemplo:

```
Print<int>(Short.MaxValue)      // not an error.
```

- em um caso de um receptor de um `in` método de extensão (em oposição aos `ref` métodos de extensão), os rvalues ou as *conversões implícitas desse argumento* são permitidas. Uma referência a um valor temporário em retenção é transmitida nesse caso. Exemplo:

```
public static IEnumerable<T> Concat<T>(in this (IEnumerable<T>, IEnumerable<T>) arg) => ...;

("aa", "bb").Concat<char>()      // not an error.
```

Mais informações sobre `ref` / `in` métodos de extensão são fornecidas mais adiante neste documento.

- o despejo do argumento devido a `await` operandos pode despejar "por valor", se necessário. Em cenários em que não é possível fornecer uma referência direta ao argumento porque a interrupção `await` de uma cópia do valor do argumento é despejada.

Exemplo:

```
M1(RefReturningMethod(), await SomethingAsync()) // not an error.
```

Como o resultado de uma invocação de efeito colateral é uma referência que não pode ser preservada na `await` suspensão, um temporário contendo o valor real será preservado (como faria em um caso de parâmetro `ByVal` comum).

Argumentos opcionais omitidos

É permitido que um `in` parâmetro especifique um valor padrão. Isso torna o argumento correspondente opcional.

Omitir o argumento opcional no site de chamada resulta na passagem do valor padrão por meio de um temporário.

```
Print("hello");      // not an error, same as
Print("hello", c: Color.Black);

void Print(string s, in Color c = Color.Black)
{
    // ...
}
```

Comportamento de alias em geral

Assim como `ref` as `out` variáveis, as `in` variáveis são referências/aliases para os locais existentes.

Embora o receptor não tenha permissão para gravar neles, a leitura de um `in` parâmetro pode observar valores diferentes como um efeito colateral de outras avaliações.

Exemplo:

```
static Vector3 v = Vector3.UnitY;

static void Main()
{
    Test(v);
}

static void Test(in Vector3 v1)
{
    Debug.Assert(v1 == Vector3.UnitY);
    // changes v1 deterministically (no races required)
    ChangeV();
    Debug.Assert(v1 == Vector3.UnitX);
}

static void ChangeV()
{
    v = Vector3.UnitX;
}
```

`in` parâmetros e captura de variáveis locais.

A finalidade dos parâmetros de captura lambda/Async `in` se comporta da mesma forma que os `out` `ref` parâmetros e.

- `in` os parâmetros não podem ser capturados em um fechamento
- `in` parâmetros não são permitidos em métodos iteradores
- `in` parâmetros não são permitidos em métodos Async

Variáveis temporárias.

Alguns usos da `in` passagem de parâmetros podem exigir o uso indireto de uma variável local temporária:

- `in` os argumentos são sempre passados como aliases diretos quando o site de chamada usa `in`. O temporário nunca é usado nesse caso.
- `in` Não é necessário que os argumentos sejam aliases diretos quando o site de chamadas não usa `in`. Quando o argumento não é um LValue, um temporário pode ser usado.
- `in` o parâmetro pode ter um valor padrão. Quando o argumento correspondente é omitido no site de chamada, o valor padrão é passado por meio de um temporário.
- `in` os argumentos podem ter conversões implícitas, incluindo aqueles que não preservam a identidade. Um temporário é usado nesses casos.
- receptores de chamadas struct comuns não podem ser LValue graváveis (**caso existente!**). Um temporário é usado nesses casos.

O tempo de vida do argumento temporaries corresponde ao escopo de abrangência mais próximo do site de chamada.

O tempo de vida formal das variáveis temporárias é semanticamente significativo em cenários que envolvem a análise de escape de variáveis retornadas por referência.

Representação de metadados de `in` parâmetros.

Quando `System.Runtime.CompilerServices.IsReadOnlyAttribute` é aplicado a um parâmetro ByRef, isso significa que o parâmetro é um `in` parâmetro.

Além disso, se o método for *abstract* ou *virtual*, a assinatura desses parâmetros (e apenas esses parâmetros) deverá ter `[modreq[System.Runtime.InteropServices.InAttribute]]`.

Motivação: isso é feito para garantir que, em um caso de método, substituir/implementar os `in` parâmetros correspondam.

Os mesmos requisitos se aplicam a `Invoke` métodos em delegados.

Motivação: isso é para garantir que os compiladores existentes não possam simplesmente ignorar `readonly` ao criar ou atribuir delegados.

Retornando por referência `ReadOnly`.

Motivação

A motivação para esse subrecurso é aproximadamente simétrica para os motivos para os `in` parâmetros, evitando a cópia, mas no lado de retorno. Antes desse recurso, um método ou um indexador tinha duas opções: 1) retornar por referência e ser exposto a possíveis mutações ou 2) retornar por valor, o que resulta em cópia.

Solução (`ref readonly` retorna)

O recurso permite que um membro retorne variáveis por referência sem expô-las a mutações.

Declarando `ref readonly` Membros retornados

Uma combinação de modificadores `ref readonly` na assinatura de retorno é usada para indicar que o membro retorna uma referência somente leitura.

Para todas as finalidades `ref readonly`, um membro é tratado como uma `readonly` variável-semelhante a `readonly` campos e `in` parâmetros.

Por exemplo `ref readonly`, os campos de membro que têm um tipo de struct são todos classificados recursivamente como `readonly` variáveis. Ele tem permissão para passá-los como `in` argumentos, mas não `ref` como `out` argumentos ou.

```

ref readonly Guid Method1()
{
}

Method2(in Method1()); // valid. Can pass as `in` argument.

Method3(ref Method1()); // not valid. Cannot pass as `ref` argument

```

- `ref readonly` os retornos são permitidos nos mesmos locais que `ref` são retornados. Isso inclui indexadores, delegados, lambdas, funções locais.
- Não é permitido sobrecarregar `ref` / `ref readonly` /diferenças.
- É permitido sobrecarregar as diferenças comuns de ByVal e `ref readonly` Return.
- Com a finalidade de OHI (sobrecarregar, ocultar, implementar), `ref readonly` é semelhante, mas diferente de `ref`. Por exemplo, o método que substitui `ref readonly` um, ele mesmo deve ser `ref readonly` e ter tipo de identidade conversível.
- Para fins de conversões de grupo delegate/lambda/Method, `ref readonly` é semelhante, mas diferente de `ref`. As lambdas e os candidatos à conversão do grupo de métodos aplicáveis precisam corresponder ao `ref readonly` retorno do delegado de destino com `ref readonly` o retorno do tipo que tem a identidade conversível.
- Para fins de variância genérica, os `ref readonly` retornos são nonvariant.

Observação: não há avisos em `ref readonly` retornos que tenham tipos de referência ou primitivos. Pode ser inútil em geral, mas, em alguns casos, o usuário deve/deseja passar primitivos como `in`. Exemplos – substituindo um método genérico como `ref readonly T Method()` quando o `T` foi substituído para ser `int`.

É concebível ter um analisador que avisa em casos de uso ineficiente de `ref readonly` Devoluções, mas as regras para essa análise seriam muito difusas para fazer parte de uma especificação de linguagem.

Retornando de `ref readonly` Membros

Dentro do corpo do método, a sintaxe é a mesma que com retornos de referência regulares. O `readonly` será inferido do método que o contém.

A motivação é que o `return ref readonly <expression>` tempo é desnecessário e só permite incompatibilidades na `readonly` parte que sempre resultaria em erros. O `ref` é, no entanto, necessário para a consistência com outros cenários em que algo é passado por meio de alias estrito vs. por valor.

Ao contrário do caso com `in` parâmetros, os `ref readonly` retornos nunca retornam por meio de uma cópia local. Considerar que a cópia deixaria de existir imediatamente ao retornar tal prática seria inútil e perigosa. Portanto, os `ref readonly` retornos são sempre referências diretas.

Exemplo:

```

struct ImmutableArray<T>
{
    private readonly T[] array;

    public ref readonly T ItemRef(int i)
    {
        // returning a readonly reference to an array element
        return ref this.array[i];
    }
}

```

- Um argumento de `return ref` deve ser um LValue (**regra existente**)
- Um argumento de `return ref` deve ser "Safe para retornar" (**regra existente**)
- Em um `ref readonly` membro, um argumento `return ref` de *não precisa ser gravável*. Por exemplo, esse membro pode ser ref-retornar um campo `ReadOnly` ou um de seus `in` parâmetros.

Seguro para retornar regras.

A segurança normal para retornar regras para referências também será aplicada a referências `ReadOnly`.

Observe que um `ref readonly` pode ser obtido de um `ref` local/parâmetro/retorno regular, mas não o contrário. Caso contrário, a segurança de `ref readonly` Devoluções será inferida da mesma maneira que para `ref` retornos regulares.

Considerando que os RValues podem ser passados como `in` parâmetro e retornados, pois precisamos de `ref readonly` mais uma regra- **rvalues não são seguros para retornar por referência**.

Considere a situação em que um RValue é passado para um `in` parâmetro por meio de uma cópia e retornado de volta em uma forma de um `ref readonly`. No contexto do chamador, o resultado dessa invocação é uma referência a dados locais e, como tal, não é seguro retornar. Depois que os RValues não forem seguros para retornar, a regra existente #6 já tratará esse caso.

Exemplo:

```

ref readonly Vector3 Test1()
{
    // can pass an RValue as "in" (via a temp copy)
    // but the result is not safe to return
    // because the RValue argument was not safe to return by reference
    return ref Test2(default(Vector3));
}

ref readonly Vector3 Test2(in Vector3 r)
{
    // this is ok, r is returnable
    return ref r;
}

```

Regras atualizadas `safe to return` :

1. **refs para variáveis no heap é seguro para retornar**
2. os **parâmetros REF/in são seguros para retornar** `in` os parâmetros naturalmente só podem ser retornados como `ReadOnly`.
3. os **parâmetros de saída são seguros para retornar** (mas devem ser definitivamente atribuídos, como já é o caso hoje)
4. os **campos de struct da instância são seguros para retornar, contanto que o receptor seja**

- seguro para retornar
- 5. 'this' não é seguro para retornar de membros de struct
- 6. uma ref, retornada de outro método, é segura para retornar se todos os refs/esgotamentos passados para esse método como parâmetros formais eram seguros para retornar.
Especificamente, é irrelevante se o destinatário é seguro de retornar, independentemente de o destinatário ser uma struct, uma classe ou um parâmetro de tipo genérico.
- 7. Os rvalues não são seguros para retornar por referência. *Especificamente, os rvalues são seguros como em parâmetros.*

Observação: há regras adicionais sobre a segurança de retornos que entram em jogo quando tipos de referência e reatribuições de referência estão envolvidos. As regras se aplicam igualmente aos `ref` `ref readonly` Membros e, portanto, não são mencionadas aqui.

Comportamento de alias.

`ref readonly` os membros fornecem o mesmo comportamento de alias que `ref` os membros comuns (exceto para serem `ReadOnly`). Portanto, para a finalidade de capturar em lambdas, `Async`, iteradores, despejo de pilha, etc... as mesmas restrições se aplicam., devido à incapacidade de capturar as referências reais e devido à natureza do efeito colateral da avaliação do membro, esses cenários não são permitidos.

É permitido e necessário fazer uma cópia quando `ref readonly` Return é um destinatário de métodos struct regulares, que assumem `this` como uma referência gravável comum. Historicamente, em todos os casos em que essas invocações são aplicadas à variável `ReadOnly`, é feita uma cópia local.

Representação de metadados.

Quando `System.Runtime.CompilerServices.IsReadOnlyAttribute` é aplicado ao retorno de um método de retorno `ByRef`, isso significa que o método retorna uma referência `ReadOnly`.

Além disso, a assinatura de resultado de tais métodos (e apenas esses métodos) deve ter
`[modreq[System.Runtime.CompilerServices.IsReadOnlyAttribute]]`.

Motivação: isso é para garantir que os compiladores existentes não possam simplesmente ignorar `readonly` ao invocar métodos com `ref readonly` Devoluções

Structs `ReadOnly`

Em um recurso curto que torna o `this` parâmetro de todos os membros de instância de uma struct, exceto para construtores, um `in` parâmetro.

Motivação

O compilador deve assumir que qualquer chamada de método em uma instância de struct pode modificar a instância. Na verdade, uma referência gravável é passada para o método como `this` parâmetro e habilita totalmente esse comportamento. Para permitir tais invocações em `readonly` variáveis, as invocações são aplicadas a cópias temporárias. Isso pode ser não intuitivo e, às vezes, força as pessoas a abandonar `readonly` por motivos de desempenho.

Exemplo: <https://codeblog.jonskeet.uk/2014/07/16/micro-optimization-the-surprising-inefficiency-of-readonly-fields/>

Depois de adicionar suporte para `in` parâmetros e `ref readonly` retornar o problema de cópia defensiva será pior, pois as variáveis `ReadOnly` se tornarão mais comuns.

Solução

Permitir `readonly` modificador em declarações de struct que resultaria em `this` ser tratado como `in` parâmetro em todos os métodos de instância de struct, exceto para construtores.

```

static void Test(in Vector3 v1)
{
    // no need to make a copy of v1 since Vector3 is a readonly struct
    System.Console.WriteLine(v1.ToString());
}

readonly struct Vector3
{
    ...
    public override string ToString()
    {
        // not OK!! `this` is an `in` parameter
        foo(ref this.X);

        // OK
        return $"X: {X}, Y: {Y}, Z: {Z}";
    }
}

```

Restrições em membros de struct ReadOnly

- Os campos de instância de uma struct ReadOnly devem ser ReadOnly.
Motivação: só pode ser gravada externamente, mas não por meio de membros.
- As propriedades autopropriedade de uma struct ReadOnly devem ser somente obtenção.
Motivação: consequência de restrição em campos de instância.
- Struct ReadOnly não pode declarar eventos do tipo campo.
Motivação: consequência de restrição em campos de instância.

Representação de metadados.

Quando `System.Runtime.CompilerServices.IsReadOnlyAttribute` é aplicado a um tipo de valor, isso significa que o tipo é um `readonly struct`.

Especialmente:

- A identidade do `IsReadOnlyAttribute` tipo não é importante. Na verdade, ele pode ser inserido pelo compilador no assembly que o contém, se necessário.

`ref / in` métodos de extensão

Na verdade, há uma proposta existente (<https://github.com/dotnet/roslyn/issues/165>) e o protótipo correspondente de RP (<https://github.com/dotnet/roslyn/pull/15650>). Quero apenas reconhecer que essa ideia não é totalmente nova. No entanto, é relevante aqui, já que `ref readonly` Remove elegantemente o problema mais contenciosos sobre tais métodos, o que fazer com os receptores de rvalue.

A ideia geral é permitir que os métodos de extensão adotem o `this` parâmetro por referência, desde que o tipo seja conhecido como um tipo struct.

```

public static void Extension(ref this Guid self)
{
    // do something
}

```

Os motivos para escrever esses métodos de extensão são principalmente:

1. Evite copiar quando o destinatário é um struct grande
2. Permitir a mutação de métodos de extensão em structs

Os motivos pelos quais não queremos permitir isso em classes

1. Seria uma finalidade muito limitada.
2. Ela iria parar muito em constante distância que uma chamada de método não pode transformar o `null` receptor em ficar `null` após a invocação.

Na verdade, atualmente uma não `null` variável não pode se `null` tornar *explicitamente* atribuída ou passada por `ref` ou `out`. Isso ajuda muito a legibilidade ou outras formas da análise "pode ser uma nula aqui". 3. Seria difícil reconciliar com a semântica "avaliar uma vez" de acessos condicionais nulos. Exemplo: `obj.stringField?.RefExtension(...)` -é necessário capturar uma cópia do `stringField` para tornar a verificação nula significativa, mas as atribuições para `this` dentro de RefExtension não seriam refletidas de volta para o campo.

Uma capacidade de declarar métodos de extensão em **structs** que usam o primeiro argumento por referência era uma solicitação de longa duração. Uma das considerações de bloqueio foi "o que acontece se o destinatário não for um LValue?".

- Há um precedente de que qualquer método de extensão também poderia ser chamado como um método estático (às vezes, é a única maneira de resolver a ambiguidade). Isso ditaria que os receptores de RValue não devem ser permitidos.
- Por outro lado, há uma prática de fazer a invocação em uma cópia em situações semelhantes quando os métodos de instância de struct estão envolvidos.

O motivo pelo qual a "cópia implícita" existe é porque a maioria dos métodos struct não modifica realmente a estrutura, embora não seja capaz de indicar isso. Portanto, a solução mais prática era simplesmente fazer a invocação em uma cópia, mas essa prática é conhecida por prejudicar o desempenho e causar bugs.

Agora, com a disponibilidade de `in` parâmetros, é possível que uma extensão sinalize a intenção. Portanto, o enigma pode ser resolvido exigindo que `ref` as extensões sejam chamadas com receptores graváveis, enquanto `in` as extensões permitem a cópia implícita, se necessário.

```
// this can be called on either RValue or an LValue
public static void Reader(in this Guid self)
{
    // do something nonmutating.
    WriteLine(self == default(Guid));
}

// this can be called only on an LValue
public static void Mutator(ref this Guid self)
{
    // can mutate self
    self = new Guid();
}
```

`in` extensões e genéricos.

A finalidade dos `ref` métodos de extensão é mutar o receptor diretamente ou invocar os membros mutantes. Portanto, `ref this T` as extensões são permitidas desde que `T` seja restrito a ser uma struct.

Por outro lado, os `in` métodos de extensão existem especificamente para reduzir a cópia implícita. No entanto, qualquer uso de um parâmetro precisará `in T` ser feito por meio de um membro de interface. Como todos os membros da interface são considerados mutação, qualquer uso exigiria uma cópia. -Em vez de reduzir a cópia, o efeito seria o oposto. Portanto, `in this T` não é permitido quando `T` é um parâmetro de tipo genérico, independentemente de restrições.

Tipos válidos de métodos de extensão (recapitulação):

As seguintes formas de `this` declaração em um método de extensão agora são permitidas:

1. `this T arg` - extensão de ByVal regular. (caso existente)

- T pode ser qualquer tipo, incluindo tipos de referência ou parâmetros de tipo. A instância será a mesma variável após a chamada. Permite conversões implícitas desse tipo de *conversão de argumento*. Pode ser chamado em RValues.
- `in this T self` - `in` extensão. T deve ser um tipo struct real. A instância será a mesma variável após a chamada. Permite conversões implícitas desse tipo de *conversão de argumento*. Pode ser chamado em RValues (pode ser invocado em um temp, se necessário).
- `ref this T self` - `ref` extensão. T deve ser um tipo struct ou um parâmetro de tipo genérico restrito a ser um struct. A instância pode ser gravada pela invocação. Permite apenas conversões de identidade. Deve ser chamado em LValue gravável. (nunca é invocado por meio de uma Temp).

Locais de referência ReadOnly.

Motivação.

Depois que `ref readonly` os membros foram introduzidos, estavam claros do uso de que precisam ser emparelhados com o tipo apropriado de local. A avaliação de um membro pode produzir ou observar efeitos colaterais, portanto, se o resultado precisar ser usado mais de uma vez, ele precisará ser armazenado. Os `ref` locais comuns não ajudam aqui, pois não é possível atribuir uma `readonly` referência.

Soluções.

Permitir declaração de `ref readonly` locais. Esse é um novo tipo de `ref` localidades que não é gravável. Como resultado, os `ref readonly` locais podem aceitar referências a variáveis ReadOnly sem expor essas variáveis a gravações.

Declarando e usando `ref readonly` locais.

A sintaxe de tais locais usa `ref readonly` modificadores no site da declaração (nessa ordem específica). Da mesma forma que os `ref` locais comuns, os `ref readonly` locais devem ser inicializados como REF na declaração. Ao contrário `ref` de locais regulares, os `ref readonly` locais podem se referir a `readonly` lvalues como `in` parâmetros, `readonly` campos, `ref readonly` métodos.

Para todas as finalidades `ref readonly`, um local é tratado como uma `readonly` variável. A maioria das restrições no uso são as mesmas de `readonly` campos ou `in` parâmetros.

Por exemplo, os campos de um `in` parâmetro que tem um tipo struct são todos classificados recursivamente como `readonly` variáveis.

```

static readonly ref Vector3 M1() => . . .

static readonly ref Vector3 M1_Trace()
{
    // OK
    ref readonly var r1 = ref M1();

    // Not valid. Need an LValue
    ref readonly Vector3 r2 = ref default(Vector3);

    // Not valid. r1 is readonly.
    Mutate(ref r1);

    // OK.
    Print(in r1);

    // OK.
    return ref r1;
}

```

Restrições no uso de `ref readonly` locais

Exceto por sua `readonly` natureza, os `ref readonly` locais se comportam como `ref` locais comuns e estão sujeitos exatamente às mesmas restrições.

Por exemplo, as restrições relacionadas à captura em fechamentos, `async` a declaração em métodos ou à `safe-to-return` análise se aplicam igualmente a `ref readonly` locais.

Expressões ternários `ref`. (também conhecido como "LValues condicionais")

Motivação

O uso de `ref` e de `ref readonly` locais expôs uma necessidade de ref-Initialize desses locais com uma ou outra variável de destino com base em uma condição.

Uma solução alternativa típica é introduzir um método como:

```

ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
        return ref consequence;
    }
    else
    {
        return ref alternative;
    }
}

```

Observe que `choice` não é uma substituição exata de um ternário, já que *todos os* argumentos devem ser avaliados no local de chamada, o que estava levando a um comportamento e a bugs não intuitivos.

O seguinte não funcionará conforme o esperado:

```

// will crash with NRE because 'arr[0]' will be executed unconditionally
ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);

```

Solução

Permitir tipo especial de expressão condicional que é avaliada como uma referência a um dos argumentos

LValue com base em uma condição.

Usando a `ref` expressão Ternário.

A sintaxe para o `ref` tipo de uma expressão condicional é

```
<condition> ? ref <consequence> : ref <alternative>;
```

Assim como somente com a expressão condicional comum `<consequence>` ou `<alternative>` é avaliado dependendo do resultado da expressão de condição booliana.

Diferentemente da expressão condicional comum, `ref` expressão condicional:

- requer que `<consequence>` e `<alternative>` sejam lvalues.
- `ref` a própria expressão condicional é um LValue e
- `ref` a expressão condicional é gravável se os dois `<consequence>` e `<alternative>` são lvalue graváveis

Exemplos:

`ref` ternário é um LValue e, como tal, pode ser passado/atribuído/retornado por referência;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Sendo um LValue, ele também pode ser atribuído a.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;

// error. readOnlyField is readonly and thus conditional expression is readonly
(arr != null ? ref arr[0]: ref obj.readOnlyField) = 1;
```

Pode ser usado como um receptor de uma chamada de método e ignorar a cópia, se necessário.

```
// no copies
(arr != null ? ref arr[0]: ref otherArr[0]).StructMethod();

// invoked on a copy.
// The receiver is `readonly` because readOnlyField is readonly.
(arr != null ? ref arr[0]: ref obj.readOnlyField).StructMethod();

// no copies. `ReadonlyStructMethod` is a method on a `readonly` struct
// and can be invoked directly on a readonly receiver
(arr != null ? ref arr[0]: ref obj.readOnlyField).ReadonlyStructMethod();
```

`ref` ternário também pode ser usado em um contexto regular (não ref).

```
// only an example
// a regular ternary could work here just the same
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

Desvantagens

Posso ver dois argumentos principais em relação ao suporte aprimorado para referências e referências `ReadOnly`:

1. Os problemas que são resolvidos aqui são muito antigos. Por que repentinamente solucioná-los agora,

especialmente porque não ajudariam a usar o código existente?

À medida que encontramos C# e .net usados em novos domínios, alguns problemas se tornam mais proeminentes.

Como exemplos de ambientes que são mais críticos do que a média de sobrecargas de computação, posso listar

- cenários de nuvem/datacenter em que a computação é cobrada e a capacidade de resposta é uma vantagem competitiva.
- Jogos/VR/AR com requisitos de tempo real em latências

Esse recurso não sacrifica nenhum dos pontos fortes existentes, como a segurança de tipo, ao mesmo tempo que permite reduzir sobrecargas em alguns cenários comuns.

2. Podemos garantir razoavelmente que o receptor será tocado pelas regras quando ele se aprofundar nos `readonly` contratos?

Temos confiança semelhante ao usar `out`. A implementação incorreta de `out` pode causar comportamento não especificado, mas, na realidade, raramente acontece.

Fazer com que as regras de verificação formais familiarizadas `ref readonly` reduza ainda mais o problema de confiança.

Alternativas

O design principal da concorrência é realmente "não fazer nada".

Perguntas não resolvidas

Criar reuniões

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-02-22.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-03-01.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-08-28.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-25.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-27.md>

Tempo de compilação imposição de segurança para tipos semelhantes a ref

21/01/2022 • 31 minutes to read

Introdução

O principal motivo para as regras de segurança adicionais ao lidar com tipos como `Span<T>` e `ReadOnlySpan<T>` é que esses tipos devem ser confinados para a pilha de execução.

Há dois motivos pelos quais `Span<T>` e tipos semelhantes devem ser tipos somente de pilha.

1. `Span<T>` é semanticamente uma estrutura que contém uma referência e um intervalo (`ref T data, int length`). Independentemente da implementação real, as gravações em tal struct não seriam atômicas. A "divisão" simultânea dessa estrutura levaria à possibilidade de `length` não corresponder ao `data`, causando acessos fora do intervalo e violações de segurança de tipo, o que, por fim, pode resultar em uma corrupção de heap de GC no código "seguro" aparentemente.
2. Algumas implementações de `Span<T>` literalmente contêm um ponteiro gerenciado em um de seus campos. Não há suporte para ponteiros gerenciados, pois campos de objetos de heap e código que gerencia para colocar um ponteiro gerenciado no heap de GC normalmente falham no momento do JIT.

Todos os problemas acima seriam aliviados se as instâncias do `Span<T>` estiverem restritas apenas na pilha de execução.

Um problema adicional surge devido à composição. Geralmente, seria desejável criar tipos de dados mais complexos que seriam inseridos `Span<T>` e `ReadOnlySpan<T>` instâncias. Esses tipos compostos teriam de ser structos e compartilhariam todos os riscos e requisitos de `Span<T>`. Como resultado, as regras de segurança descritas aqui devem ser exibidas como aplicáveis a todo o intervalo de *tipos de referência*.

A [especificação de idioma de rascunho](#) destina-se a garantir que os valores de um tipo de referência ocorram somente na pilha.

Tipos generalizados `ref-like` no código-fonte

`ref-like` as structs são explicitamente marcadas no código-fonte usando o `ref` modificador:

```
ref struct TwoSpans<T>
{
    // can have ref-like instance fields
    public Span<T> first;
    public Span<T> second;
}

// error: arrays of ref-like types are not allowed.
TwoSpans<T>[] arr = null;
```

A designação de uma struct como `ref` permitirá que a estrutura tenha campos de instância semelhantes a `ref` e também fará com que todos os requisitos de tipos de referência sejam aplicáveis à estrutura.

Representação de metadados ou estruturas semelhantes a ref

As estruturas semelhantes a ref serão marcadas com o atributo `System.Runtime.CompilerServices.IsRefLikeAttribute`.

O atributo será adicionado às bibliotecas base comuns, como `mscorlib`. Em um caso, se o atributo não estiver disponível, o compilador irá gerar um interno de forma semelhante a outros atributos incorporados sob demanda, como `IsReadOnlyAttribute`.

Uma medida adicional será tomada para evitar o uso de structs de referência em compiladores que não estejam familiarizados com as regras de segurança (isso inclui compiladores C# antes daquele em que esse recurso é implementado).

Não tendo outras boas alternativas que funcionem em compiladores antigos sem manutenção, um `Obsolete` atributo com uma cadeia de caracteres conhecida será adicionado a todas as estruturas semelhantes a ref. Os compiladores que sabem como usar tipos de referência irão ignorar essa forma específica de `Obsolete`.

Uma representação de metadados típica:

```
[IsRefLike]
[Obsolete("Types with embedded references are not supported in this version of your compiler.")]
public struct TwoSpans<T>
{
    // . . .
}
```

Observação: não é o objetivo de fazê-lo para que qualquer uso de tipos de referência em compiladores antigos falhe em 100%. Isso é difícil de atingir e não é estritamente necessário. Por exemplo, sempre há uma maneira de contornar o `Obsolete` uso do código dinâmico ou, por exemplo, criar uma matriz de tipos de referência por meio de reflexão.

Em particular, se o usuário quiser colocar um `Obsolete` atributo ou `Deprecated` em um tipo de referência, não haverá outra opção além de não emitir o predefinido, uma vez que o `Obsolete` atributo não pode ser aplicado mais de uma vez.

Exemplos:

```

SpanLikeType M1(ref SpanLikeType x, Span<byte> y)
{
    // this is all valid, unconcerned with stack-referring stuff
    var local = new SpanLikeType(y);
    x = local;
    return x;
}

void Test1(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    // this is allowed
    stackReferring2 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    stackReferring2 = M1(ref param1, stackReferring1);

    // this is NOT allowed
    param1 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    param2 = stackReferring1.Slice(10);

    // this is allowed
    param1 = new SpanLikeType(param2);

    // this is allowed
    stackReferring2 = param1;
}

ref SpanLikeType M2(ref SpanLikeType x)
{
    return ref x;
}

ref SpanLikeType Test2(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    ref var stackReferring3 = M2(ref stackReferring2);

    // this is allowed
    stackReferring3 = M1(ref stackReferring2, stackReferring1);

    // this is allowed
    M2(ref stackReferring3) = stackReferring2;

    // this is NOT allowed
    M1(ref param1) = stackReferring2;

    // this is NOT allowed
    param1 = stackReferring3;

    // this is NOT allowed
    return ref stackReferring3;

    // this is allowed
    return ref param1;
}

```

Especificação da linguagem de rascunho

Abaixo, descrevemos um conjunto de regras de segurança para tipos de referência, `ref struct` a fim de garantir que os valores desses tipos ocorram apenas na pilha. Um conjunto de regras de segurança diferente e mais simples seria possível se os locais não puderem ser passados por referência. Essa especificação também permitiria a reatribuição segura de locais de referência.

Visão geral

Nós associamos a cada expressão em tempo de compilação o conceito de qual escopo a expressão tem permissão para escapar, "seguro para escapar". Da mesma forma, para cada lvalue, mantemos um conceito de qual escopo uma referência a ele tem permissão para escapar, "ref-safe-to-escape". Para uma determinada expressão lvalue, elas podem ser diferentes.

Eles são análogos ao "seguro para retornar" do recurso de locais de referência, mas é mais refinado. Onde o "seguro-para-retornar" de uma expressão registra apenas se (ou não) ele pode escapar do método delimitador como um todo, os registros seguros para o escape para os quais o escopo pode escapar (em qual escopo ele pode não escapar). O mecanismo de segurança básico é imposto da seguinte maneira. Devido a uma atribuição de uma expressão E1 com um escopo de segurança para escape S1, para uma expressão (lvalue) E2 com escopo seguro para escape S2, será um erro se S2 for um escopo maior do que S1. Por construção, os dois escopos S1 e S2 estão em uma relação de aninhamento, porque uma expressão legal sempre é segura para retornar de algum escopo delimitando a expressão.

Por enquanto, é o suficiente para a finalidade da análise, dar suporte a apenas dois escopos – externos ao método e ao escopo de nível superior do método. Isso ocorre porque os valores like ref com escopos internos não podem ser criados e os locais de referência não dão suporte à reatribuição. As regras, no entanto, podem dar suporte a mais de dois níveis de escopo.

As regras precisas para a computação do status de *segurança para retorno* de uma expressão e as regras que regem a legalidade de expressões, seguem.

ref-safe-to-escape

O *ref-safe-to-escape* é um escopo, colocando uma expressão lvalue, à qual é seguro para uma ref ao lvalue para escapar. Se esse escopo for o método inteiro, dizemos que uma referência ao lvalue é *segura para retornar* do método.

seguro para escapar

O *seguro para escapar* é um escopo, delimitando uma expressão, à qual é seguro para o valor escapar. Se esse escopo for o método inteiro, dizemos que o valor é *seguro para retornar* do método.

Uma expressão cujo tipo não é um `ref struct` tipo é *seguro para retornar* de todo o método delimitador. Caso contrário, nos referimos às regras abaixo.

Parâmetros

Um lvalue que designa um parâmetro formal é *ref-safe-to-escape* (por referência) da seguinte maneira:

- Se o parâmetro for um `ref` `out` parâmetro, ou `in`, ele será de *ref-safe-to-escape* de todo o método (por exemplo, por uma `return ref` instrução); caso contrário,
- Se o parâmetro for o `this` parâmetro de um tipo struct, ele será de *ref-safe-to-escape* para o escopo de nível superior do método (mas não de todo o próprio método); [Exemplo](#) de
- Caso contrário, o parâmetro é um parâmetro de valor e ele é de *referência segura para escape* para o escopo de nível superior do método (mas não do próprio método).

Uma expressão que é um Rvalue que designa o uso de um parâmetro formal é *segura para escapar* (por valor) de todo o método (por exemplo, por uma `return` instrução). Isso também se aplica ao `this` parâmetro.

Locais

Um lvalue que designa uma variável local é *ref-safe-to-escape* (por referência) da seguinte maneira:

- Se a variável for uma `ref` variável, seu *ref-safe-to-escape* será obtido do *ref-safe-to-escape* de sua expressão de inicialização; caso contrário,
- A variável é *ref-safe-to-escape* no escopo no qual ele foi declarado.

Uma expressão que é um Rvalue que designa o uso de uma variável local é *segura para escapar* (por valor) da seguinte maneira:

- Mas a regra geral acima, um local cujo tipo não é um `ref struct` tipo é *seguro para retornar* de todo o método delimitador.
- Se a variável for uma variável de iteração de um `foreach` loop, o escopo de *seguro para escape* da variável será o mesmo que o *seguro para escapar* da `foreach` expressão do loop.
- Um local do `ref struct` tipo e não inicializado no ponto de declaração é *seguro para retornar* de todo o método delimitador.
- Caso contrário, o tipo da variável é um `ref struct` tipo, e a declaração da variável requer um inicializador. O escopo de *segurança para escape* da variável é o mesmo que o de *segurança para escapar* de seu inicializador.

Referência de campo

Um lvalue que designa uma referência a um campo, `e.F`, é *ref-safe-to-escape* (por referência) da seguinte maneira:

- Se `e` for de um tipo de referência, ele será de *ref-safe-to-escape* de todo o método; caso contrário,
- Se `e` for de um tipo de valor, seu *ref-safe-to-escape* será obtido do *ref-safe-to-escape* de `e`.

Um Rvalue que designa uma referência a um campo, `e.F`, tem um escopo *seguro para escapar*, que é o mesmo que o *seguro para escapar* do `e`.

Operadores, incluindo `?:`

O aplicativo de um operador definido pelo usuário é tratado como uma invocação de método.

Para um operador que produz um Rvalue, como `e1 + e2` ou `c ? e1 : e2`, o seguro- *para-escape* do resultado é o escopo mais estreito entre os operandos do operador de *segurança para escape*. Em virtude disso, para um operador unário que produz um Rvalue, como `+e`, o *seguro para escapar* do resultado é o *seguro para escapar* do operando.

Para um operador que produz um lvalue, como `c ? ref e1 : ref e2`

- a *ref-safe-to-escape* do resultado é o escopo mais estreito entre o *ref-safe-to-escape* dos operandos do operador.
- o *seguro para escapar* dos operandos deve concordar, e isso é o *seguro para escapar* do lvalue resultante.

Invocação de método

Um lvalue resultante de uma invocação de método ref-retorne `e1.M(e2, ...)` é *ref-safe-to-escape* do menor dos seguintes escopos:

- O método de circunscrição inteiro
- o *ref-safe-to-escape* de todas `ref` as `out` expressões de argumento e (excluindo o receptor)
- Para cada `in` parâmetro do método, se houver uma expressão correspondente que seja um lvalue, sua *ref-safe-to-escape*, caso contrário, o escopo do delimitador mais próximo
- o *seguro para escapar* de todas as expressões de argumento (incluindo o receptor)

Observação: o último marcador é necessário para manipular código como

```
var sp = new Span(...)  
return ref sp[0];
```

ou

```
return ref M(sp, 0);
```

Um Rvalue resultante de uma invocação de método `e1.M(e2, ...)` é *seguro para escapar* do menor dos seguintes escopos:

- O método de circunscrição inteiro
- o *seguro para escapar* de todas as expressões de argumento (incluindo o receptor)

Um Rvalue

Um Rvalue é *ref-safe-to-escape* do escopo de delimitação mais próximo. Isso ocorre, por exemplo, em uma invocação, como `M(ref d.Length)` Where `d` é do tipo `dynamic`. Ele também é consistente com (e talvez incorporou) nossa manipulação de argumentos correspondentes aos `in` parâmetros.

Invocações de propriedade

Uma invocação de propriedade (`get` ou `set`) tratada como uma invocação de método do método subjacente pelas regras acima.

`stackalloc`

Uma expressão `stackalloc` é um Rvalue que é *seguro para escapar* do escopo de nível superior do método (mas não de todo o próprio método).

Invocações de Construtor

Uma `new` expressão que invoca um Construtor obedece às mesmas regras que uma invocação de método que é considerada para retornar o tipo que está sendo construído.

Além disso, o *safe-to-escape* não é mais largo do que o menor de *todos os argumentos* /operandos das expressões de inicializador de objeto, recursivamente, se o inicializador estiver presente.

Construtor de span

O idioma depende `Span<T>` não ter um construtor da seguinte forma:

```
void Example(ref int x)  
{  
    // Create a span of length one  
    var span = new Span<int>(ref x);  
}
```

Esse construtor faz com `Span<T>` que sejam usados como campos indistinguíveis de um `ref` campo. As regras de segurança descritas neste documento dependem de os `ref` campos não serem uma construção válida em C# ou .net.

Expressões `default`

Uma `default` expressão é *segura para escapar* de todo o método delimitador.

Restrições de idioma

Gostaríamos de garantir que nenhuma `ref` variável local, e nenhuma variável do `ref struct` tipo, se refere à memória de pilha ou a variáveis que não estão mais ativas. Portanto, temos as seguintes restrições de idioma:

- Nem um parâmetro `ref`, nem um local `ref`, nem um parâmetro ou local de um `ref struct` tipo podem ser

levantados em uma função lambda ou local.

- Nem um parâmetro ref nem um parâmetro de um `ref struct` tipo pode ser um argumento em um método Iterator ou um `async` método.
- Nem um local de referência, nem um local de um `ref struct` tipo pode estar no escopo no ponto de uma `yield return` instrução ou uma `await` expressão.
- Um `ref struct` tipo não pode ser usado como um argumento de tipo ou como um tipo de elemento em um tipo de tupla.
- Um `ref struct` tipo não pode ser o tipo declarado de um campo, exceto que ele pode ser o tipo declarado de um campo de instância de outro `ref struct`.
- Um `ref struct` tipo não pode ser o tipo de elemento de uma matriz.
- Um valor de um `ref struct` tipo não pode ser Boxed:
 - Não há nenhuma conversão de um `ref struct` tipo para o tipo `object` ou o tipo `System.ValueType`.
 - Um `ref struct` tipo não pode ser declarado para implementar qualquer interface
 - Nenhum método de instância declarado em `object` ou em `System.ValueType`, mas não substituído em um `ref struct` tipo, pode ser chamado com um receptor desse `ref struct` tipo.
 - Nenhum método de instância de um `ref struct` tipo pode ser capturado pela conversão de método para um tipo delegado.
- Para uma reatribuição de referência `ref e1 = ref e2`, o *ref-safe-to-escape* de `e2` deve ser pelo menos tão amplo quanto um escopo como *ref-safe-to-escape* `e1`.
- Para uma instrução ref Return `return ref e1`, o *ref-safe-to-escape* de `e1` deve ser *ref-safe-to-escape* do método inteiro. (TODO: também precisamos de uma regra que `e1` deve ser *segura para escapar* do método inteiro ou que seja redundante?)
- Para uma instrução return `return e1`, a *segurança para escapar* do `e1` deve ser *segura para escapar* do método inteiro.
- Para uma atribuição `e1 = e2`, se o tipo de `e1` for um `ref struct` tipo, o *safe-to-escape* `e2` deve ser pelo menos o maior de um escopo como o *seguro para escapar* de `e1`.
- Para uma invocação de método se houver um `ref` `out` argumento ou de um `ref struct` tipo (incluindo o receptor), com o E1 *de segurança para escape*, nenhum argumento (incluindo o receptor) poderá ter um limite *mais* estreito do que o E1. [Amostra](#)
- Uma função local ou anônima pode não se referir a um local ou parâmetro do `ref struct` tipo declarado em um escopo delimitador.

Abrir problema: Precisamos de alguma regra que nos permita produzir um erro quando precisar despejar um valor de pilha de um `ref struct` tipo em uma expressão Await, por exemplo no código

```
Foo(new Span<int>(...), await e2);
```

Explicações

Essas explicações e exemplos ajudam a explicar por que muitas das regras de segurança acima existem

Argumentos de método devem corresponder

Ao invocar um método em que há um `out` `ref` parâmetro, que é um que `ref struct` inclui o receptor, todas

as `ref struct` necessidades precisam ter o mesmo tempo de vida. Isso é necessário porque o C# deve tomar todas as suas decisões sobre a segurança do tempo de vida com base nas informações disponíveis na assinatura do método e no tempo de vida dos valores no site de chamada.

Quando há `ref` parâmetros que `ref struct`, em seguida, há a possibilidade que podem alternar em todo o seu conteúdo. Portanto, no local de chamada, devemos garantir que todas essas trocas **potenciais** sejam compatíveis. Se o idioma não tiver aplicado isso, ele permitirá um código inadequado como o seguinte.

```
void M1(ref Span<int> s1)
{
    Span<int> s2 = stackalloc int[1];
    Swap(ref s1, ref s2);
}

void Swap(ref Span<int> x, ref Span<int> y)
{
    // This will effectively assign the stackalloc to the s1 parameter and allow it
    // to escape to the caller of M1
    ref x = ref y;
}
```

A restrição no receptor é necessária porque, embora nenhum dos seus conteúdos seja `ref-safe-to-escape`, ele pode armazenar os valores fornecidos. Isso significa que, com tempos de vida incompatíveis, você poderia criar um buraco de segurança de tipo da seguinte maneira:

```
ref struct S
{
    public Span<int> Span;

    public void Set(Span<int> span)
    {
        Span = span;
    }
}

void Broken(ref S s)
{
    Span<int> span = stackalloc int[1];

    // The result of a stackalloc is now stored in s.Span and escaped to the caller
    // of Broken
    s.Set(span);
}
```

Estruturar este escape

Quando se trata de estender as regras de segurança, o `this` valor em um membro de instância é modelado como um parâmetro para o membro. Agora, para um `struct` tipo de `this` é `ref S` na verdade, onde é `class` simplesmente `s` (para membros de um `class / struct` nome de um).

Ainda `this` tem diferentes regras de escape que outros `ref` parâmetros. Especificamente, não é válido para `ref-safe-to-escape` enquanto outros parâmetros são:

```

ref struct S
{
    int Field;

    // Illegal because `this` isn't safe to escape as ref
    ref int Get() => ref Field;

    // Legal
    ref int GetParam(ref int p) => ref p;
}

```

A razão para essa restrição realmente tem pouco a ver com a `struct` invocação de membro. Há algumas regras que precisam ser configuradas em relação à invocação de membro em `struct`. Membros em que o receptor é um Rvalue. Mas isso é muito acessível.

A razão para essa restrição é, na verdade, sobre a invocação de interface. Especificamente, isso se resume se o exemplo a seguir deve ou não ser compilado:

```

interface I1
{
    ref int Get();
}

ref int Use<T>(T p)
    where T : I1
{
    return ref p.Get();
}

```

Considere o caso em que o `T` é instanciado como um `struct`. Se o `this` parâmetro for ref-safe-to-escape, o retorno de `p.Get` poderá apontar para a pilha (especificamente, poderia ser um campo dentro do tipo instanciado de `T`). Isso significa que o idioma não pôde permitir a compilação deste exemplo, pois ele poderia retornar um `ref` para um local de pilha. Por outro lado, se `this` não for ref-safe-to-escape, `p.Get` não poderá se referir à pilha e, portanto, será seguro retornar.

É por isso que a saída de `this` em um `struct` é realmente tudo sobre as interfaces. Ele pode ser absolutamente feito para funcionar, mas tem uma compensação. O design eventualmente surgiu em favor de tornar as interfaces mais flexíveis.

No entanto, é possível relaxar isso no futuro.

Considerações futuras

Comprimento um `Span<T>` sobre valores de referência

Embora não seja legal hoje, há casos em que a criação de um comprimento uma `Span<T>` instância em um valor seria benéfica:

```
void RefExample()
{
    int x = ...;

    // Today creating a length one Span<int> requires a stackalloc and a new
    // local
    Span<int> span1 = stackalloc [] { x };
    Use(span1);
    x = span1[0];

    // Simpler to just allow length one span
    var span2 = new Span<int>(ref x);
    Use(span2);
}
```

Esse recurso será mais atraente se compararmos as restrições em [buffers de tamanho fixo](#), pois isso permitiria `Span<T>` instâncias de comprimento ainda maior.

Se houver alguma necessidade de reduzir esse caminho, o idioma poderá acomodar isso, garantindo que essas `Span<T>` instâncias estivessem apenas para frente. Isso é que, no momento, eles eram seguros para o escopo no qual foram criados. Isso garante que o idioma nunca tenha que considerar um `ref` valor que escape um método por meio `ref struct` de um retorno ou campo de `ref struct`. Isso provavelmente também exigiria outras alterações para reconhecer esses construtores como capturar um `ref` parâmetro dessa maneira.

Argumentos nomeados que não estejam à direita

21/01/2022 • 5 minutes to read

Resumo

Permitir que os argumentos nomeados sejam usados em posição não à direita, desde que eles sejam usados na posição correta. Por exemplo: `DoSomething(isEmployed:true, name, age);`.

Motivação

A principal motivação é evitar digitar informações redundantes. É comum nomear um argumento que é um literal (como `null`, `true`) com a finalidade de esclarecer o código, em vez de passar argumentos fora de ordem. Atualmente, isso não é permitido (`CS1738`), a menos que todos os argumentos a seguir também sejam nomeados.

```
DoSomething(isEmployed:true, name, age); // currently disallowed, even though all arguments are in position  
// CS1738 "Named argument specifications must appear after all fixed arguments have been specified"
```

Alguns exemplos adicionais:

```
public void DoSomething(bool isEmployed, string personName, int personAge) { ... }  
  
DoSomething(isEmployed:true, name, age); // currently CS1738, but would become legal  
DoSomething(true, personName:name, age); // currently CS1738, but would become legal  
DoSomething(name, isEmployed:true, age); // remains illegal  
DoSomething(name, age, isEmployed:true); // remains illegal  
DoSomething(true, personAge:age, personName:name); // already legal
```

Isso também funcionaria com params:

```
public class Task  
{  
    public static Task When(TaskStatus all, TaskStatus any, params Task[] tasks);  
}  
Task.When(all: TaskStatus.RanToCompletion, any: TaskStatus.Faulted, task1, task2)
```

Design detalhado

Em § 7.5.1 (listas de argumentos), a especificação atualmente diz:

Um argumento com um *nome de argumento* é conhecido como um **argumento nomeado**, enquanto que um argumento sem um *nome de argumento* é um **argumento posicional**. É um erro para que um argumento posicional apareça após um argumento nomeado em uma *lista de argumentos*.

A proposta é remover esse erro e atualizar as regras para localizar o parâmetro correspondente para um argumento (§ 7.5.1.1):

Argumentos no argumento-lista de construtores de instância, métodos, indexadores e delegados:

- [regras existentes]

- Um argumento sem nome corresponde a nenhum parâmetro quando é depois de um argumento nomeado fora de posição ou um argumento chamado params.

Em particular, isso impede a invocação `void M(bool a = true, bool b = true, bool c = true,); com M(c: false, valueB);`. O primeiro argumento é usado fora de posição (o argumento é usado na primeira posição, mas o parâmetro chamado "c" está na terceira posição), portanto, os argumentos a seguir devem ser nomeados.

Em outras palavras, argumentos nomeados não à direita são permitidos somente quando o nome e a posição resultam na localização do mesmo parâmetro correspondente.

Desvantagens

Essa proposta exacerba as sutilezas existentes com argumentos nomeados na resolução de sobrecarga. Por exemplo:

```
void M(int x, int y) { }
void M<T>(T y, int x) { }

void M2()
{
    M(3, 4);
    M(y: 3, x: 4); // Invokes M(int, int)
    M(y: 3, 4); // Invokes M<T>(T, int)
}
```

Você pode obter essa situação hoje alternando os parâmetros:

```
void M(int y, int x) { }
void M<T>(int x, T y) { }

void M2()
{
    M(3, 4);
    M(x: 3, y: 4); // Invokes M(int, int)
    M(3, y: 4); // Invokes M<T>(int, T)
}
```

Da mesma forma, se você tiver dois métodos `void M(int a, int b)` e `void M(int x, string y)` a invocação errada `M(x: 1, 2)` produzir um diagnóstico baseado na segunda sobrecarga ("não é possível converter de 'int' para 'String'"). Esse problema já existe quando o argumento nomeado é usado em uma posição à direita.

Alternativas

Há algumas alternativas a serem consideradas:

- O status quo
- Fornecer assistência IDE para preencher todos os nomes de argumentos à direita quando você digita um nome específico no meio.

Ambos sofrem com mais detalhes, pois eles introduzem vários argumentos nomeados, mesmo que você precise apenas de um nome de um literal no início da lista de argumentos.

Perguntas não resolvidas

Criar reuniões

O recurso foi brevemente discutido no LDM em 16 de maio de 2017, com aprovação no princípio (OK para mudar para a proposta/protótipo). Ele também foi brevemente discutido em 28 de junho de 2017.

Relacionado à discussão inicial <https://github.com/dotnet/csharplang/issues/518> relacionada ao problema proespecialista <https://github.com/dotnet/csharplang/issues/570>

privado protegido

21/01/2022 • 19 minutes to read

- [x] proposta
- [x] protótipo: concluído
- [x] implementação: concluída
- [x] especificação: concluída

Resumo

Exportar o nível de acessibilidade do CLR `protectedAndInternal` em C# como `private protected`.

Motivação

Há muitas circunstâncias em que uma API contém membros que se destinam apenas a serem implementados e usados por subclasses contidas no assembly que fornece o tipo. Embora o CLR forneça um nível de acessibilidade para essa finalidade, ele não está disponível em C#. Consequentemente, os proprietários da API são forçados a usar a `internal` proteção e a autodisciplina ou um analisador personalizado, ou para usar `protected` com documentação adicional explicando que, embora o membro apareça na documentação pública do tipo, ele não se destina a fazer parte da API pública. Para obter exemplos do último, consulte membros de Roslyn `CSharpCompilationOptions` cujos nomes começam com `Common`.

Fornecer suporte direto para esse nível de acesso em C# permite que essas circunstâncias sejam expressas naturalmente na linguagem.

Design detalhado

`private protected` modificador de acesso

Sugerimos que você adicione uma nova combinação de modificador de acesso `private protected` (que pode aparecer em qualquer ordem entre os modificadores). Isso é mapeado para a noção de `protectedAndInternal` do CLR e empresta a mesma sintaxe atualmente usada em [C++/CLI](#).

Um membro declarado `private protected` pode ser acessado dentro de uma subclass de seu contêiner se essa subclass estiver no mesmo assembly que o membro.

Modificamos a especificação da linguagem da seguinte maneira (adições em negrito). Os números de seção não são mostrados abaixo, pois podem variar dependendo de qual versão da especificação está integrada.

A acessibilidade declarada de um membro pode ser uma das seguintes:

- Público, que é selecionado incluindo um modificador público na declaração de membro. O significado intuitivo do público é "acesso não limitado".
- Protegido, que é selecionado com a inclusão de um modificador protegido na declaração de membro. O significado intuitivo de protegido é "acesso limitado à classe que a contém ou tipos derivados da classe que a contém".
- Interno, que é selecionado incluindo um modificador interno na declaração de membro. O significado intuitivo de interno é "acesso limitado a este assembly".
- Interno protegido, que é selecionado incluindo um modificador protegido e um interno na declaração de

membro. O significado intuitivo de interno protegido é "acessível dentro desse assembly, bem como os tipos derivados da classe que a contém".

- **Privada protegida**, que é selecionada com a inclusão de um modificador privado e protegido na declaração de membro. O significado intuitivo de particular protegido é "acessível nesse assembly por tipos derivados da classe que a contém".

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 têm acessibilidade declarada pública. Nenhum modificador de acesso é permitido em declarações de namespace.
- Tipos declarados diretamente em unidades de compilação ou namespaces (ao contrário de outros tipos) podem ter acessibilidade pública ou interna declarada e o padrão para a acessibilidade declarada interna.
- Membros de classe podem ter qualquer um dos cinco tipos de acessibilidade declarada e por padrão para a acessibilidade declarada privada. [Observação: um tipo declarado como membro de uma classe pode ter qualquer um dos cinco tipos de acessibilidade declarada, enquanto um tipo declarado como um membro de um namespace pode ter apenas acessibilidade pública ou interna declarada. Nota de término]
- Os membros de struct podem ter acessibilidade definida pública, interna ou privada e, por padrão, a acessibilidade declarada privada, pois as estruturas são seladas implicitamente. Membros de struct introduzidos em uma struct (ou seja, não herdados por essa struct) não podem ter acessibilidade protegida, ou interna protegida , ou protegida privada . [Observação: um tipo declarado como membro de uma struct pode ter acessibilidade pública, interna ou privada declarada, enquanto um tipo declarado como membro de um namespace pode ter apenas acessibilidade pública ou interna declarada. Nota de término]
- Os membros da interface implicitamente têm acessibilidade declarada pública. Nenhum modificador de acesso é permitido em declarações de membro de interface.
- Os membros de enumeração têm implicitamente a acessibilidade declarada. Nenhum modificador de acesso é permitido em declarações de membro de enumeração.

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 a própria M pode ser um tipo):

- Se a acessibilidade declarada de M for pública, o domínio de acessibilidade de M é o domínio de acessibilidade de T.
- Se a acessibilidade declarada de M for protegida internamente, deixe D ser a União do texto do programa de P e o texto do programa de qualquer tipo derivado de T, que é declarado fora do P. O domínio de acessibilidade de M é a interseção do domínio de acessibilidade de T com D.
- **Se a acessibilidade declarada de M for privada protegida, deixe D ser a interseção do texto do programa de P e o texto do programa de qualquer tipo derivado de T. O domínio de acessibilidade de M é a interseção do domínio de acessibilidade de T com D.**
- Se a acessibilidade declarada de M estiver protegida, deixe D ser a União do texto do programa T e o texto do programa de qualquer tipo derivado de T. O domínio de acessibilidade de M é a interseção do domínio de acessibilidade de T com D.
- Se a acessibilidade declarada de M for interna, o domínio de acessibilidade de M é a interseção do domínio de acessibilidade de T com o texto do programa de P.
- Se a acessibilidade declarada de M for privada, o domínio de acessibilidade de M é o texto do programa de T.

Quando um membro de instância protegida protegida ou privada é acessado fora do texto do programa da classe na qual ele é declarado e quando um membro da instância interna protegida é acessado fora do

texto do programa do programa no qual ele é declarado, o acesso deve ocorrer dentro de uma declaração de classe que deriva da classe na qual ela é declarada. 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.

Os modificadores de acesso permitidos e o acesso padrão para uma declaração de tipo dependem do contexto no qual a declaração ocorre (§ 9.5.2):

- Tipos declarados em unidades de compilação ou namespaces podem ter acesso público ou interno. O padrão é acesso interno.
- Os tipos declarados em classes podem ter acesso público, interno **protegido**, **privada** protegida, interno ou privado. O padrão é acesso privado.
- Os tipos declarados em structs podem ter acesso público, interno ou privado. O padrão é acesso privado.

Uma declaração de classe estática está sujeita às seguintes restrições:

- Uma classe estática não deve incluir um modificador lacrado ou abstrato. (No entanto, como uma classe estática não pode ser instanciada ou derivada de, ela se comporta como se fosse selada e abstrata.)
- Uma classe estática não deve incluir uma especificação de base de classe (§ 16.2.5) 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 deve conter apenas membros estáticos (§ 16.4.8). [Observação: todas as constantes e tipos aninhados são classificados como membros estáticos. Nota de término]
- Uma classe estática não deve ter membros com acessibilidade **privada** protegida, **privada** protegida ou protegida interna declarada.

É um erro de tempo de compilação para violar qualquer uma dessas restrições.

Uma declaração de membro de classe pode ter um dos **cinco** tipos possíveis de acessibilidade declarada (§ 9.5.2): pública, **privada** protegida, protegida interna, protegida, interna ou privada. Exceto para as combinações protegidas interna e **privada** protegida, trata-se de um erro de tempo de compilação para especificar mais de um modificador de acesso. Quando uma declaração de membro de classe não inclui nenhum modificador de acesso, a particular é assumida.

Os tipos não aninhados podem ter acessibilidade declarada pública ou interna e ter uma acessibilidade declarada interna 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 uma das **cinco** formas de acessibilidade declaradas (pública, **privada** protegida, protegida interna, protegida, interna ou privada) e, como outros membros da classe, usa como padrão a acessibilidade declarada privada.
- Um tipo aninhado declarado em um struct pode ter qualquer uma das três formas de acessibilidade declarada (pública, interna ou privada) e, como outros membros de struct, usa como padrão a acessibilidade declarada privada.

O método substituído por uma declaração de substituição é conhecido como o método base substituído para um método de substituição 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 de 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 de argumentos de tipo. Para a finalidade de localizar o método base substituído, um método é considerado acessível se for público, se estiver protegido, se estiver protegido, se for **um** banco de dados interno ou **interno ou privado**, e declarado no mesmo programa que C.

O uso de acessadores-modificadores é regido pelas seguintes restrições:

- Um modificador de acessador não deve ser usado em uma interface ou em uma implementação de membro de interface explícita.
- Para uma propriedade ou um indexador que não tem um modificador de substituição, um modificador de acessador é permitido somente se a propriedade ou o indexador tiver um acessador get e set e, em seguida, for permitido somente em um desses acessadores.
- Para uma propriedade ou um indexador que inclui um modificador de substituição, um acessador deve corresponder ao modificador de acessador, se houver, do acessador que está sendo substituído.
- O modificador de acessador 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 de Public, o modificador de acessador poderá ser **particular protegido**, interno, protegido ou privado.
 - Se a propriedade ou o indexador tiver uma acessibilidade declarada de interna protegida, o modificador de assessor poderá ser **particular protegido**, interno, protegido ou privado.
 - Se a propriedade ou o indexador tiver uma acessibilidade declarada interna ou protegida, o modificador de acessador deverá ser particular ou **protegido**.
 - Se a propriedade ou o indexador tiver uma acessibilidade declarada de protegida privada, o modificador de acessador deverá ser **privado**.
 - Se a propriedade ou o indexador tiver uma acessibilidade declarada de particular, nenhum assessor-modificador poderá ser usado.

Como a herança não tem suporte para structs, a acessibilidade declarada de um Membro struct não pode ser protegida, privada protegida ou interna protegida.

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

Uma alternativa seria o provisionamento de uma API que combina um atributo e um analisador. O atributo é colocado pelo programador em um `internal` membro para indicar que o membro deve ser usado somente em subclasses, e o analisador verifica se essas restrições estão em obedecer.

Perguntas não resolvidas

A implementação está totalmente completa. O único item de trabalho aberto é o rascunho de uma especificação correspondente para o VB.

Criar reuniões

TBD

Expressões de referência condicional

21/01/2022 • 3 minutes to read

O padrão de associação de uma variável ref a uma ou outra expressão condicionalmente não é expresso no momento em C#.

A solução alternativa típica é introduzir um método como:

```
ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
        return ref consequence;
    }
    else
    {
        return ref alternative;
    }
}
```

Observe que isso não é uma substituição exata de um ternário, já que todos os argumentos devem ser avaliados no site de chamada.

O seguinte não funcionará conforme o esperado:

```
// will crash with NRE because 'arr[0]' will be executed unconditionally
ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);
```

A sintaxe proposta teria a seguinte aparência:

```
<condition> ? ref <consequence> : ref <alternative>;
```

A tentativa acima com "Choice" pode ser gravada *corretamente* usando ref ternário como:

```
ref var r = ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

A diferença da escolha é que a consequência e as expressões alternativas são acessadas de maneira *realmente* condicional, portanto, não vemos uma falha se `arr == null`

A referência ternário é apenas um ternário em que tanto a alternativa quanto a consequência são refs. Naturalmente, isso exigirá que os operandos/conseqüências alternativos sejam LValue. Ele também exigirá que a consequência e a alternativa tenham tipos que são conversíveis de identidade entre si.

O tipo da expressão será computado de forma semelhante à do ternário regular., caso a consequência e a alternativa tenham identidade conversível, mas tipos diferentes, as regras de mesclagem de tipos existentes serão aplicadas.

A segurança a ser retornada será presumida de forma conservadora dos operandos condicionais. Se não for seguro retornar, não será seguro retornar o item inteiro.

Ref ternário é um LValue e, como tal, pode ser passado/atribuído/retornado por referência;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Sendo um LValue, ele também pode ser atribuído a.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;
```

Ref ternário também pode ser usado em um contexto regular (não ref). Embora isso não seja comum, já que você poderia usar também um ternário regular.

```
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

Notas de implementação:

A complexidade da implementação parece ser o tamanho de uma correção de bug moderada para grande. -l. E não é muito caro. Não acho que precisamos de nenhuma alteração na sintaxe ou análise. Não há nenhum efeito nos metadados ou na interoperabilidade. O recurso é baseado em expressão completamente. Nenhum efeito na depuração/PDB pode ser

Permitir separador de dígitos após 0B ou 0x

21/01/2022 • 2 minutes to read

No C# 7,2, estendemos o conjunto de lugares que os separadores de dígitos (o caractere de sublinhado) podem aparecer em literais inteiros. [A partir do C# 7,0, os separadores são permitidos entre os dígitos de um literal.](#)

Agora, no C# 7,2, também permitimos separadores de dígitos antes do primeiro dígito significativo de um literal binário ou hexadecimal, após o prefixo.

```
123      // permitted in C# 1.0 and later
1_2_3    // permitted in C# 7.0 and later
0x1_2_3  // permitted in C# 7.0 and later
0b101    // binary literals added in C# 7.0
0b1_0_1  // permitted in C# 7.0 and later

// in C# 7.2, _ is permitted after the `0x` or `0b`
0x_1_2   // permitted in C# 7.2 and later
0b_1_0_1 // permitted in C# 7.2 and later
```

Não permitimos que um literal inteiro decimal tenha um sublinhado à esquerda. Um token como `_123` é um identificador.

Restrição de tipo não gerenciado

21/01/2022 • 8 minutes to read

Resumo

O recurso de restrição não gerenciado fornecerá a imposição de linguagem para a classe de tipos conhecida como "tipos não gerenciados" na especificação da linguagem C#. Isso é definido na seção 18.2 como um tipo que não é um tipo de referência e não contém campos de tipo de referência em nenhum nível de aninhamento.

Motivação

A principal motivação é facilitar o autor do código de interoperabilidade de nível baixo no C#. Os tipos não gerenciados são um dos principais blocos de construção do código de interoperabilidade, mas a falta de suporte em genéricos torna impossível criar rotinas reutilizáveis em todos os tipos não gerenciados. Em vez disso, os desenvolvedores são forçados a criar o mesmo código de placa para cada tipo não gerenciado em sua biblioteca:

```
int Hash(Point point) { ... }
int Hash(TimeSpan timeSpan) { ... }
```

Para habilitar esse tipo de cenário, o idioma apresentará uma nova restrição: não gerenciado:

```
void Hash<T>(T value) where T : unmanaged
{
    ...
}
```

Essa restrição só pode ser atendida por tipos que se encaixam na definição de tipo não gerenciado na especificação da linguagem C#. Outra maneira de observar isso é que um tipo satisfaz a restrição não gerenciada se ela também pode ser usada como um ponteiro.

```
Hash(new Point()); // Okay
Hash(42); // Okay
Hash("hello") // Error: Type string does not satisfy the unmanaged constraint
```

Os parâmetros de tipo com a restrição não gerenciada podem usar todos os recursos disponíveis para tipos não gerenciados: ponteiros, fixos, etc...

```
void Hash<T>(T value) where T : unmanaged
{
    // Okay
    fixed (T* p = &value)
    {
        ...
    }
}
```

Essa restrição também possibilitará ter conversões eficientes entre dados estruturados e fluxos de bytes. Essa é uma operação que é comum em pilhas de rede e camadas de serialização:

```

Span<byte> Convert<T>(ref T value) where T : unmanaged
{
    ...
}

```

Essas rotinas são vantajosas porque são provavelmente seguras no tempo de compilação e na alocação gratuita. Os autores de interoperabilidade atualmente não podem fazer isso (embora seja em uma camada em que o desempenho é crítico). Em vez disso, eles precisam contar com a alocação de rotinas que têm verificações de tempo de execução caras para verificar se os valores estão corretamente gerenciados.

Design detalhado

O idioma apresentará uma nova restrição chamada `unmanaged`. Para atender a essa restrição, um tipo deve ser uma struct e todos os campos do tipo devem se enquadrar em uma das seguintes categorias:

- Ter o tipo `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `IntPtr` OU `UIntPtr`.
- Ser qualquer `enum` tipo.
- Ser um tipo de ponteiro.
- Ser um struct definido pelo usuário que satisfaça a `unmanaged` restrição.

Os campos de instância gerados pelo compilador, como os que fazem backup das propriedades implementadas automaticamente, também devem atender a essas restrições.

Por exemplo:

```

// Unmanaged type
struct Point
{
    int X;
    int Y {get; set;}
}

// Not an unmanaged type
struct Student
{
    string FirstName;
    string LastName;
}

```

A `unmanaged` restrição não pode ser combinada com `struct`, `class` ou `new()`. Essa restrição deriva do fato que `unmanaged` implica `struct`, portanto, as outras restrições não fazem sentido.

A `unmanaged` restrição não é imposta pelo CLR, somente pelo idioma. Para evitar o uso incorretamente por outras linguagens, os métodos que têm essa restrição serão protegidos por um mod-req. Isso impedirá que outras linguagens usem argumentos de tipo que não sejam tipos não gerenciados.

O token `unmanaged` na restrição não é uma palavra-chave nem uma palavra-chave contextual. Em vez disso, é como se `var` ele fosse avaliado nesse local e também:

- Associar ao tipo de usuário definido ou referenciado chamado `unmanaged`: isso será tratado assim que qualquer outra restrição de tipo nomeado for tratada.
- Associar a nenhum tipo: isso será interpretado como a `unmanaged` restrição.

No caso de haver um tipo chamado `unmanaged` e está disponível sem qualificação no contexto atual, não haverá nenhuma maneira de usar a `unmanaged` restrição. Isso paraleliza as regras que envolvem o recurso `var` e os

tipos definidos pelo usuário com o mesmo nome.

Desvantagens

A principal desvantagem desse recurso é que ele atende a um pequeno número de desenvolvedores: normalmente, autores ou estruturas de biblioteca de nível baixo. Portanto, está gastando um tempo de linguagem precioso para um pequeno número de desenvolvedores.

Ainda assim, essas estruturas são a base para a maioria dos aplicativos .NET. Portanto, o desempenho/exatidão WINS nesse nível pode ter um efeito de ondulação no ecossistema do .NET. Isso faz com que o recurso valha a pena considerar mesmo com o público limitado.

Alternativas

Há algumas alternativas a serem consideradas:

- O status quo: o recurso não é justificado em seus próprios méritos e os desenvolvedores continuam a usar o comportamento de consentimento implícito.

Perguntas

Representação de metadados

A linguagem F# codifica a restrição no arquivo de assinatura, o que significa que o C# não pode usar novamente sua representação. Será necessário escolher um novo atributo para essa restrição. Além disso, um método que tem essa restrição deve ser protegido por um mod-req.

Blittable versus não gerenciado

A linguagem F# tem um [recuso muito semelhante](#) que usa a palavra-chave unmanaged. O nome blittable é proveniente do uso em Midori. Talvez você queira verificar a precedência aqui e usar não gerenciado.

Resolução A linguagem decide usar não gerenciado

Verificação

O verificador/tempo de execução precisa ser atualizado para entender o uso de ponteiros para parâmetros de tipo genérico? Ou pode simplesmente funcionar como está sem alterações?

Resolução Nenhuma alteração é necessária. Todos os tipos de ponteiro simplesmente não são verificáveis.

Criar reuniões

N/D

Os campos de indexação `fixed` não devem exigir fixação, independentemente do contexto móvel/não móvel.

21/01/2022 • 2 minutes to read

A alteração tem o tamanho de uma correção de bug. Ele pode estar em 7,3 e não entrar em conflito com a direção que demoramos. Essa alteração é apenas para permitir que o cenário a seguir funcione, embora `s` seja móvel. Ele já é válido quando `s` não é móvel.

Observação: em ambos os casos, ele ainda requer `unsafe` contexto. É possível ler dados não inicializados ou até mesmo fora do intervalo. Que não está mudando.

```
unsafe struct S
{
    public fixed int myFixedField[10];
}

class Program
{
    static S s;

    unsafe static void Main()
    {
        int p = s.myFixedField[5]; // indexing fixed-size array fields would be ok
    }
}
```

O principal "desafio" que vejo aqui é como explicar o relaxamento na especificação. Em particular, como o seguinte ainda precisaria de fixação. (porque `s` é móvel e usamos explicitamente o campo como um ponteiro)

```
unsafe struct S
{
    public fixed int myFixedField[10];
}

class Program
{
    static S s;

    unsafe static void Main()
    {
        int* ptr = s.myFixedField; // taking a pointer explicitly still requires pinning.
        int p = ptr[5];
    }
}
```

Um motivo pelo qual exigimos a fixação do destino quando ele é móvel é o artefato de nossa estratégia de geração de código, sempre convertemos em um ponteiro não gerenciado e, assim, forçamos o usuário a fixar por meio de `fixed` instrução. No entanto, a conversão para não gerenciado é desnecessária ao fazer a indexação. O mesmo matemática de ponteiro não seguro é igualmente aplicável quando temos o receptor na forma de um ponteiro gerenciado. Se fizermos isso, a referência intermediária será gerenciada (GC-acompanhado) e a fixação será desnecessária.

A alteração <https://github.com/dotnet/roslyn/pull/24966> é um protótipo de RP que relaxará esse requisito.

Declaração `fixed` baseada em padrão

21/01/2022 • 5 minutes to read

Resumo

Introduza um padrão que permitiria que os tipos participem de `fixed` instruções.

Motivação

O idioma fornece um mecanismo para fixar dados gerenciados e obter um ponteiro nativo para o buffer subjacente.

```
fixed(byte* ptr = byteArray)
{
    // ptr is a native pointer to the first element of the array
    // byteArray is protected from being moved/collected by the GC for the duration of this block
}
```

O conjunto de tipos que podem participar `fixed` é codificado e limitado a matrizes e `System.String`. O código de tipos "especiais" não é dimensionado quando novos primitivos, como `ImmutableArray<T>`, `Span<T>`, `Utf8String` são introduzidos.

Além disso, a solução atual para se `System.String` baseia em uma API razoavelmente rígida. A forma da API implica que `System.String` é um objeto contíguo que incorpora dados codificados UTF16 em um deslocamento fixo do cabeçalho do objeto. Essa abordagem foi encontrada com problemas em várias propostas que poderiam exigir alterações no layout subjacente. Seria desejável poder mudar para algo mais flexível que dissocia `System.String` o objeto de sua representação interna para fins de interoperabilidade não gerenciada.

Design detalhado

Padrão

Uma "fixa" com base em padrões viável precisa:

- Forneça as referências gerenciadas para fixar a instância e inicializar o ponteiro (preferivelmente esta é a mesma referência)
- Transmite sem ambigüidade o tipo do elemento não gerenciado (ou seja, "char" para "String")
- Prescrever o comportamento no caso "Empty" quando não houver nada para fazer referência a ele.
- Não deve enviar por push autores de API para decisões de design que prejudicam o uso do tipo fora do `fixed`.

Acho que a acima poderia ser satisfeita reconhecendo um membro de retorno de referência especialmente nomeado: `ref [readonly] T GetPinnableReference()`.

Para ser usado pela `fixed` instrução, as seguintes condições devem ser atendidas:

1. Há apenas um desses membros fornecido para um tipo.
2. Retorna por `ref` ou `ref readonly`. (`readonly` é permitido para que os autores de tipos imutável/ReadOnly pudessem implementar o padrão sem adicionar uma API gravável que pudesse ser usada em código seguro)

3. T é um tipo não gerenciado. (desde que `T*` se torne o tipo de ponteiro. Naturalmente, a restrição expandirá se/quando a noção de "não gerenciado" for expandida)
4. Retorna gerenciado `nullptr` quando não há dados para fixar – provavelmente a maneira mais barata de transmitir a esvaziação. (Observe que "" a cadeia de caracteres retorna uma referência a '\0', uma vez que as cadeias são terminadas em nulo)

Como alternativa para o `#3`, podemos permitir que o resultado em casos vazios seja indefinido ou específico da implementação. No entanto, isso pode tornar a API mais perigosa e propenso a abusos e incômodos de compatibilidade indesejados.

Tradução

```
fixed(byte* ptr = thing)
{
    // <BODY>
}
```

torna-se o pseudocódigo a seguir (nem todos expressos no C#)

```
byte* ptr;
// specially decorated "pinned" IL local slot, not visible to user code.
pinned ref byte _pinned;

try
{
    // NOTE: null check is omitted for value types
    // NOTE: `thing` is evaluated only once (temporary is introduced if necessary)
    if (thing != null)
    {
        // obtain and "pin" the reference
        _pinned = ref thing.GetPinnableReference();

        // unsafe cast in IL
        ptr = (byte*)_pinned;
    }
    else
    {
        ptr = default(byte*);
    }

    // <BODY>
}
finally    // finally can be omitted when not observable
{
    // "unpin" the object
    _pinned = nullptr;
}
```

Desvantagens

- O `GetPinnableReference` destina-se a ser usado apenas no `fixed`, mas nada impede seu uso em código seguro, portanto, o implementador deve ter isso em mente.

Alternativas

Os usuários podem introduzir `GetPinnableReference` ou membro semelhante e usá-lo como

```
fixed(byte* ptr = thing.GetPinnableReference())
{
    // <BODY>
}
```

Não há nenhuma solução para `System.String` se a solução alternativa for desejada.

Perguntas não resolvidas

- [] Comportamento no estado "Empty". - `nullptr` ou `undefined` ?
- [] Os métodos de extensão devem ser considerados?
- [] Se um padrão for detectado em `System.String` , ele deverá vencer?

Criar reuniões

Nenhum ainda.

Reatribuição local de referência

21/01/2022 • 2 minutes to read

No C# 7.3, adicionamos suporte para reassociar o Referent de uma variável local ref ou um parâmetro ref.

Adicionamos o seguinte ao conjunto de `assignment_operator`s.

```
assignment_operator
: '=' 'ref'
;
```

O `=ref` operador é chamado de ***ref Assignment Operator**. Não é um operador de atribuição _compound *. O operando esquerdo deve ser uma expressão que seja associada a uma variável local ref, um parâmetro ref (diferente de `this`) ou um parâmetro out. O operando à direita deve ser uma expressão que produz um lvalue que designa um valor do mesmo tipo que o operando esquerdo.

O operando à direita deve ser definitivamente atribuído no ponto da atribuição de referência.

Quando o operando esquerdo for associado a um `out` parâmetro, será um erro se esse `out` parâmetro não tiver sido atribuído definitivamente no início do operador de atribuição de referência.

Se o operando esquerdo for uma referência gravável (ou seja, ele designa algo diferente de um `ref readonly` local ou `in` parâmetro), o operando à direita deverá ser um lvalue gravável.

O operador ref Assignment produz um lvalue do tipo atribuído. É gravável se o operando esquerdo é gravável (ou seja, não `ref readonly` ou `in`).

As regras de segurança para esse operador são:

- Para uma reatribuição de referência `e1 = ref e2`, o *ref-safe-to-escape* de `e2` deve ser pelo menos tão amplo quanto um escopo como *ref-safe-to-escape* `e1`.

Onde *ref-safe-to-escape* é definido em [segurança para tipos do tipo ref](#)

Inicializadores de matriz stackalloc

21/01/2022 • 2 minutes to read

Resumo

Permitir que a sintaxe do inicializador de matriz seja usada com `stackalloc`.

Motivação

Matrizes comuns podem ter seus elementos inicializados no momento da criação. Parece razoável permitir isso no `stackalloc` caso.

A pergunta sobre por que essa sintaxe não é permitida com `stackalloc` surge com muita frequência.

Consulte, por exemplo, [#1112](#)

Design detalhado

Matrizes comuns podem ser criadas por meio da seguinte sintaxe:

```
new int[3]
new int[3] { 1, 2, 3 }
new int[] { 1, 2, 3 }
new[] { 1, 2, 3 }
```

Devemos permitir que matrizes alocadas da pilha sejam criadas por meio de:

```
stackalloc int[3]           // currently allowed
stackalloc int[3] { 1, 2, 3 }
stackalloc int[] { 1, 2, 3 }
stackalloc[] { 1, 2, 3 }
```

A semântica de todos os casos é praticamente a mesma que com matrizes.

Por exemplo: no último caso, o tipo de elemento é inferido do inicializador e deve ser um tipo "não gerenciado".

Observação: o recurso não depende do destino ser um `Span<T>`. Ele é tão aplicável no `T*` caso, portanto, não parece razoável para preditar isso no `Span<T>` caso.

Tradução

A implementação ingênuo poderia simplesmente inicializar a matriz logo após a criação por meio de uma série de atribuições de elemento.

De maneira semelhante ao caso com matrizes, pode ser possível e desejável detectar casos em que todos ou a maioria dos elementos são tipos blittable e usar técnicas mais eficientes copiando o estado pré-criado de todos os elementos constantes.

Desvantagens

Alternativas

Esse é um recurso de conveniência. É possível simplesmente não fazer nada.

Perguntas não resolvidas

Criar reuniões

Nenhum ainda.

Atributos de Field-Targeted de propriedade implementados automaticamente

21/01/2022 • 3 minutes to read

Resumo

Esse recurso pretende permitir que os desenvolvedores apliquem atributos diretamente aos campos de backup das propriedades implementadas automaticamente.

Motivação

Atualmente, não é possível aplicar atributos aos campos de backup das propriedades implementadas automaticamente. Nesses casos em que o desenvolvedor deve usar um atributo de direcionamento de campo, ele é forçado a declarar o campo manualmente e usar a sintaxe de propriedade mais detalhada. Considerando que o C# sempre tem suporte para atributos direcionados a campo no campo de apoio gerado para eventos, faz sentido estender a mesma funcionalidade para sua propriedade parentes.

Design detalhado

Em suma, o seguinte seria um C# legal e não produz um aviso:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public string MySecret { get; set; }
}
```

Isso resultaria na aplicação dos atributos direcionados por campo ao campo de backup gerado pelo compilador:

```
[Serializable]
public class Foo
{
    [NonSerialized]
    private string _mySecretBackingField;

    public string MySecret
    {
        get { return _mySecretBackingField; }
        set { _mySecretBackingField = value; }
    }
}
```

Como mencionado, isso traz paridade com a sintaxe de evento do C# 1.0, pois o seguinte já é legal e se comporta conforme o esperado:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public event EventHandler MyEvent;
}
```

Desvantagens

Há duas desvantagens em potencial para implementar essa alteração:

1. A tentativa de aplicar um atributo ao campo de uma propriedade implementada automaticamente produz um aviso do compilador de que os atributos nesse bloco serão ignorados. Se o compilador foi alterado para dar suporte a esses atributos, eles seriam aplicados ao campo de backup em uma recompilação subsequente, o que poderia alterar o comportamento do programa em tempo de execução.
2. Atualmente, o compilador não valida os destinos AttributeUsage dos atributos ao tentar aplicá-los ao campo da propriedade implementada automaticamente. Se o compilador tiver sido alterado para dar suporte a atributos direcionados a campo e o atributo em questão não puder ser aplicado a um campo, o compilador emitiria um erro em vez de um aviso, dividindo a compilação.

Alternativas

Perguntas não resolvidas

Criar reuniões

Variáveis de expressão em inicializadores

21/01/2022 • 2 minutes to read

Resumo

Estendemos os recursos introduzidos no C# 7 para permitir expressões que contêm variáveis de expressão (saída de declarações de variáveis e padrões de declaração) em inicializadores de campo, inicializadores de propriedade, Construtor-inicializadores e cláusulas de consulta.

Motivação

Isso conclui algumas das bordas aproximadas deixadas na linguagem C# devido à falta de tempo.

Design detalhado

Removemos a restrição que impede a declaração de variáveis de expressão (saída de declarações de variável e padrões de declaração) em um inicializador de construtor. Tal variável declarada está no escopo ao longo do corpo do construtor.

Removemos a restrição que impede a declaração de variáveis de expressão (saída de declarações de variável e padrões de declaração) em um inicializador de campo ou propriedade. Essa variável declarada está no escopo durante a expressão de inicialização.

Removemos a restrição que impede a declaração de variáveis de expressão (saída de declarações de variável e padrões de declaração) em uma cláusula de expressão de consulta que é convertida no corpo de um lambda. Essa variável declarada está no escopo por toda a expressão da cláusula de consulta.

Desvantagens

Nenhum.

Alternativas

O escopo apropriado para variáveis de expressão declaradas nesses contextos não é óbvio e merece mais discussão LDM.

Perguntas não resolvidas

- [] Qual é o escopo apropriado para essas variáveis?

Criar reuniões

Nenhum.

Supporte para == e != em tipos de tupla

21/01/2022 • 9 minutes to read

Permita expressões `t1 == t2` onde `t1` e `t2` sejam tupla ou tipos de tupla anuláveis da mesma cardinalidade e avalie-os aproximadamente como `temp1.Item1 == temp2.Item1 && temp1.Item2 == temp2.Item2` (supondo que `var temp1 = t1; var temp2 = t2;`).

Por outro lado, ele poderia permitir `t1 != t2` lo e avaliá-lo como

```
temp1.Item1 != temp2.Item1 || temp1.Item2 != temp2.Item2 .
```

No caso anulável, verificações adicionais para `temp1.HasValue` e `temp2.HasValue` são usadas. Por exemplo,

```
nullableT1 == nullableT2 é avaliada como
```

```
temp1.HasValue == temp2.HasValue ? (temp1.HasValue ? ... : true) : false .
```

Quando uma comparação por elemento retorna um resultado não bool (por exemplo, quando uma definição de usuário não booleana `operator ==` ou `operator !=` é usada, ou em uma comparação dinâmica), esse resultado será convertido em `bool` ou executado `operator true` ou `operator false` para obter um `bool`. A comparação de tupla sempre acaba retornando um `bool`.

A partir do C# 7.2, esse código produz um erro (

```
error CS0019: Operator '==' cannot be applied to operands of type '(...)' and '(...)' ), a menos que haja um definido pelo usuário operator== .
```

Detalhes

Ao ligar o `==` operador (ou `!=`), as regras existentes são: (1) caso dinâmico, (2) resolução de sobrecarga e (3) falham. Essa proposta adiciona um caso de tupla entre (1) e (2): se ambos os operandos de um operador de comparação são tuplas (têm tipos de tupla ou literais de tupla) e têm cardinalidade correspondente, a comparação é executada com um elemento-Wise. Essa igualdade de tupla também é levantada em tuplas anuláveis.

Ambos os operandos (e, no caso de literais de tupla, seus elementos) são avaliados na ordem da esquerda para a direita. Cada par de elementos é usado como operandos para associar o operador `==` (ou `!=`), recursivamente. Todos os elementos com o tipo de tempo de compilação `dynamic` causam um erro. Os resultados dessas comparações por elemento são usados como operandos em uma cadeia de operadores condicionais e (or ou).

Por exemplo, no contexto de `(int, (int, int)) t1, t2; , t1 == (1, (2, 3))` o seria avaliado como

```
temp1.Item1 == temp2.Item1 && temp1.Item2.Item1 == temp2.Item2.Item1 && temp1.Item2.Item2 == temp2.Item2.Item2
```

Quando um literal de tupla é usado como operando (em qualquer lado), ele recebe um tipo de tupla convertido formado pelas conversões de elemento-Wise que são introduzidas ao ligar o operador `==` (ou `!=`) elemento-Wise.

Por exemplo, no `(1L, 2, "hello") == (1, 2L, null)` , o tipo convertido para ambos os literais de tupla é `(long, long, string)` e o segundo literal não tem nenhum tipo natural.

Desconstrução e conversões para tupla

No `(a, b) == x` , o fato de que `x` pode desconstruir em dois elementos não desempenha uma função. Isso pode estar em uma proposta futura, embora isso gere dúvidas `x == y` (é uma comparação simples ou uma

comparação de elemento, e se estiver usando qual cardinalidade?). Da mesma forma, as conversões para a tupla não desempenham nenhuma função.

Nomes de elementos de tupla

Ao converter um literal de tupla, avisamos quando um nome de elemento de tupla explícito era fornecido no literal, mas ele não corresponde ao nome do elemento de tupla de destino. Usamos a mesma regra na comparação de tupla, de modo que supomos que `(int a, int b) t` avisamos sobre `d` no `t == (c, d: 0)`.

Resultados de comparação de elemento não bool

Se uma comparação por elemento for dinâmica em uma igualdade de tupla, usamos uma invocação dinâmica do operador `false` e a desnegarei para obter um `bool` e continuar com outras comparações com elemento.

Se uma comparação por elemento retornar algum outro tipo não bool em uma igualdade de tupla, haverá dois casos:

- Se o tipo não bool converter em `bool`, aplicamos essa conversão,
- Se não houver tal conversão, mas o tipo tiver um operador `false`, usaremos isso e negarei o resultado.

Em uma desigualdade de tupla, as mesmas regras se aplicam, exceto pelo fato de usarmos o operador `true` (sem negação) em vez do operador `false`.

Essas regras são semelhantes às regras envolvidas para usar um tipo não bool em uma `if` instrução e alguns outros contextos existentes.

Ordem de avaliação e casos especiais

O valor do lado esquerdo é avaliado primeiro, depois o valor do lado direito e, em seguida, as comparações por elemento da esquerda para a direita (incluindo conversões e com a saída inicial com base em regras existentes para operadores condicionais e/ou).

Por exemplo, se houver uma conversão de tipo `A` para tipo `B` e um método `(A, A) GetTuple()`, avaliar `(new A(1), (new B(2), new B(3))) == (new B(4), GetTuple())` significa:

- `new A(1)`
- `new B(2)`
- `new B(3)`
- `new B(4)`
- `GetTuple()`
- em seguida, as conversões e comparações e a lógica condicional do elemento são avaliadas (Converta `new A(1)` para o tipo `B`, compare com o `new B(4)` assim por diante).

Comparando `null` com `null`

Esse é um caso especial de comparações regulares, que são transferidas para comparações de tupla. A `null == null` comparação é permitida e os `null` literais não obtêm nenhum tipo. Na igualdade de tupla, isso `(0, null) == (0, null)` também é permitido e os `null` literais e a tupla não obtêm um tipo.

Comparando uma struct anulável com `null` sem `operator==`

Esse é outro caso especial de comparações regulares, que são transferidas para comparações de tupla. Se você tiver um `struct S` sem `operator==`, a `(S?)x == null` comparação será permitida e será interpretada como `((S?)x.HasValue`. Na igualdade de tupla, a mesma regra é aplicada; portanto, `(0, (S?)x) == (0, null)` é permitido.

Compatibilidade

Se alguém escreveu seus próprios `ValueTuple` tipos com uma implementação do operador de comparação, ele teria sido previamente selecionado pela resolução de sobrecarga. Mas como o novo caso de tupla vem antes da resolução de sobrecarga, tratamos desse caso com a comparação de tupla em vez de depender da comparação definida pelo usuário.

Relaciona-se aos [operadores de teste relacional e de tipo](#) relacionados a [#190](#)

Candidatos de sobrecarga aprimorados

21/01/2022 • 2 minutes to read

Resumo

As regras de resolução de sobrecarga foram atualizadas em quase todas as atualizações de linguagem C# para melhorar a experiência de programadores, fazendo chamadas ambíguas selecionando a opção "óbvia". Isso deve ser feito cuidadosamente para preservar a compatibilidade com versões anteriores, mas como geralmente estamos resolvendo o que seria casos de erro, esses aprimoramentos geralmente funcionam bem.

1. Quando um grupo de métodos contém membros de instância e estáticos, descartamos os membros da instância se forem invocados sem um receptor de instância ou contexto e descartaremos os membros estáticos se forem invocados com um receptor de instância. Quando não há receptor, incluímos somente membros estáticos em um contexto estático, caso contrário, os membros estáticos e de instância. Quando o receptor é ambigamente uma instância ou tipo devido a uma situação de cor de cor, incluímos ambos. Um contexto estático, no qual um receptor de instância implícito não pode ser usado, inclui o corpo de membros onde não é definido, como membros estáticos, bem como locais onde isso não pode ser usado, como inicializadores de campo e inicializadores de construtor.
2. Quando um grupo de métodos contém alguns métodos genéricos cujos argumentos de tipo não satisfazem suas restrições, esses membros são removidos do conjunto de candidatos.
3. Para uma conversão de grupo de métodos, os métodos candidatos cujo tipo de retorno não corresponda ao tipo de retorno do delegado são removidos do conjunto.

Tipos de referência anuláveis em C

21/01/2022 • 13 minutes to read

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.

```
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

```
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.

```
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

21/01/2022 • 23 minutes to read

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 Scala.

Design detalhado

Expressão is

O `is` operador é estendido para testar uma expressão em relação a um *padrão*.

```
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.

```

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

```

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

```
var v = expr as Type;  
if (v != null) { // code using v
```

Com um pouco mais conciso

```
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

```
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

```
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

```

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

```

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`.

```

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 extraír 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

```

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 qualquer um dos seguintes formulários

```
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

```
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:

```

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_arms* 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 opcionais 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

```

switch ((a, b))
{

```

Para permitir

```

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

21/01/2022 • 50 minutes to read

- [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.

```
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.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

Uma classe que implementa essa interface não precisa implementar seu método concreto.

```
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:

```
new C().M(); // error: class 'C' does not contain a member 'M'
```

Dentro de um membro de instância de uma interface, `this` o 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`.

Todo: 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.*

***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>

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.

```
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

```
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:

```

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

```

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.

```

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.

```
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.

```
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:

```
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:

```
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:

```
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

```
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> 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:

```
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:

```
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

```

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

```

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:

```

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:

```

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):

```

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` em 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 M_1 é considerada *mais específica* do que outra substituição M_2 se M_1 é declarada no tipo T_1 , M_2 é declarada no tipo T_2 e qualquer uma

1. T_1 contém T_2 entre suas interfaces diretas ou indiretas, ou
2. T_2 é um tipo de interface, mas T_1 não é um tipo de interface.

O cenário é este

```
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>.

Métodos de interface vs structs (fechados)

Há algumas interações de infeliz entre as estruturas e os métodos de interface padrão.

```
interface IA
{
    public void M() { }
}
struct S : IA
{
}
```

Observe que os membros da interface não são herdados:

```
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

```
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:

```
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?

```
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

```
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

```
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>.

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?

```
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>.

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.

```
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>.

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?

```
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?

```
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?

```

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>

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 propondo a sintaxe a seguir para chamadas base em interfaces. Não estou de amor, mas ilustra o que qualquer sintaxe deve ser capaz de expressar:

```

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

```

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

```

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

```

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>

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 subminim 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>

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>

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

21/01/2022 • 39 minutes to read

- [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/114>

Design detalhado

Interfaces

`IAsyncDisposable`

Houve muita discussão sobre `IAsyncDisposable` (por exemplo, <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`):

```
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 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:

```
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:

```

IAsyncEnumerator<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>` 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:

```

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:

```
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 dobradas, 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 operações 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 `IAsyncEnumerable` 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:

```
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:

```
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` syntaxe, para que eles permaneçam consistentes entre si e `using (await ...)` já seja uma syntaxe 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:

```
var enumerable = ...;
await foreach (T item in enumerable)
{
    ...
}
```

é convertido para o equivalente de:

```
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:

```
await foreach (T item in enumerable.ConfigureAwait(false))
{
    ...
}
```

Isso se baseará em tipos que também adicionaremos ao .NET, provavelmente System.Threading.Tasks.Extensions.dll:

```

// 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 s e `IAsyncEnumerator<T>` s, além de consumi-los. Atualmente, a linguagem dá suporte à gravação de um iterador como:

```

static IEnumerable<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:

```

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 maquinaria 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 estivéssemos 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 compreensão das consultas permitiria "simplesmente funcionar" com alguns operadores, por exemplo, se o IX fornecer os seguintes métodos:

```
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":

```
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:

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this IAsyncEnumerable<TSource> source,
Func<TSource, ValueTask<TResult>> func);
```

Então, isso "simplesmente funciona":

```

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:

```

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

21/01/2022 • 23 minutes to read

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

```
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:

```
int System.Index.GetOffset(int length);
```

A `..` sintaxe do `System.Range` exigirá o `System.Range` tipo, bem como um ou mais dos seguintes membros:

```
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:

```

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:

```
System.Index operator ^(int fromEnd);
```

O comportamento desse operador é definido apenas para valores de entrada maiores ou iguais a zero.

Exemplos:

```

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):

```

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:

```
System.Range operator ..(Index start = 0, Index end = ^0);
```

Exemplos:

```

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

Supor te 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:

```
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:

```
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:

```

class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.WriteLine("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.WriteLine("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:

```
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"

21/01/2022 • 7 minutes to read

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.

```
{
    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:

```
{
    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:

```
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`.

```

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`:

```

{
    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

21/01/2022 • 2 minutes to read

Resumo

Supor 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`.
As regras de acessibilidade para `private` Membros no escopo delimitador são as mesmas para `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 sobreposição 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>

Atribuição de coalescência nula

21/01/2022 • 4 minutes to read

- [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

```
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

```
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:

```
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

21/01/2022 • 8 minutes to read

Problema defensor: <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:

```

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.

```
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`.

```
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.

```
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.

```
// 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).

```

// 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

21/01/2022 • 2 minutes to read

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

```
local_variable_initializer_unsafe
: stackalloc_initializer
;

stackalloc_initializer
: 'stackalloc' unmanaged_type '[' expression ']'
;
```

e substitua-os por

```
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

21/01/2022 • 26 minutes to read

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:

```
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:

```
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:

```
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.

```
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:

```
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:

```
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:

```
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:

```
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(EqualityComparer)`.

Por exemplo, considere os seguintes tipos de registro:

```
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:

```

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:

```
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:

```
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:

```
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:

```
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:

```

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(", ");

        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.

```
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

21/01/2022 • 7 minutes to read

- [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:

```
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:

```
compilation_unit
: extern_alias_directive* using_directive* global_attributes? statement* namespace_member_declaration*
;
```

Somente um *compilation_unit* pode ter a *instrução*s.

Exemplo:

```

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:

```

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<int> 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:

```
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:

```
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
```

produziria:

```
static class $Program
{
    static async Task $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
    }
}
```

Um exemplo como este:

```
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
return 0;
```

produziria:

```
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:

```
System.Console.WriteLine("Hi!");
return 2;
```

produziria:

```
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

21/01/2022 • 36 minutes to read

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:

```
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?`:

```
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).

```
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*:

```

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:

```

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.

```

pp_directive
: ...
| pp_nullable
;

pp_nullable
: whitespace? '#' whitespace? 'nullable' whitespace nullable_action (whitespace 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:

```

pragma_warning_body
: ...
| 'warning' whitespace warning_action whitespace 'nullable'
;

```

Por exemplo:

```
#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*, *nonnullable* 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

Se `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 `?` 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:

```
// 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:

```
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.

```
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.

```

// 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.

```

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 $\text{await } E$ é o estado nulo padrão de seu tipo.

O operador `as`

O estado nulo de uma $E \text{ as } T$ 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 $E1 \text{ op=} E2$ 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 $v@?$, $v@$ 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 $u@?$, $u@$ 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 -):

- $\text{Merge}(T, T, d) = T$
- $\text{Merge}(S, T?, +) = \text{mesclar}(S?, T, +) = \text{Merge}(S, T, +) ?$
- $\text{Merge}(S, T?, -) = \text{mesclar}(S?, T, -) = \text{Merge}(S, T, -)$
- $\text{Mesclar}(C<S_1, \dots, S_n>, C<T_1, \dots, T_n>, +) = \text{mesclar } C<(S_1, T_1, D_1), \dots, \text{mesclagem}(S_n, T_n, D_n)>, \text{em que}$
 - $d_i = +$ Se o i parâmetro 'th Type de $C<\dots>$ for Covariance
 - $d_i = -$ Se o i parâmetro 'th Type de $C<\dots>$ for contrato ou constante
- $\text{Mesclar}(C<S_1, \dots, S_n>, C<T_1, \dots, T_n>, -) = \text{mesclar } C<(S_1, T_1, D_1), \dots, \text{mesclagem}(S_n, T_n, D_n)$

- > , em que
 - o `di` = - Se o `i` parâmetro 'th Type de `c<...>` for Covariance
 - o `di` = + Se o `i` parâmetro 'th Type de `c<...>` for contrato ou constante
- $\text{Mesclar}(\langle s_1 \ s_1, \dots, \ s_n \ s_n \rangle, \langle t_1 \ t_1, \dots, \ t_n \ t_n \rangle, d) = \text{mesclar}(\langle \langle s_1, \ t_1, d \rangle \ n_1, \dots, \ \text{mesclar}(\langle s_n, \ t_n, d \rangle \ n_n) \rangle, \ em \ que$
 - o `ni` está ausente se `si` e `ti` diferir ou se ambos estiverem ausentes
 - o `ni` é `si` If `si` e é `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

21/01/2022 • 21 minutes to read

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> and Padrões conjuntiva que exigem dois de dois padrões diferentes para corresponder;
- <https://github.com/dotnet/csharplang/issues/1350> or 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.

```
primary_pattern
  : parenthesized_pattern
  | // all of the existing forms
  ;
parenthesized_pattern
  : '(' pattern ')'
  ;
```

Padrões de tipo

Permitimos um tipo como um padrão:

```
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

```
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:

```
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`.

```
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

```
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 serão úteis para testar intervalos de valores

```
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:

```
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`.

```
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

```
bool IsValidPercentage(int x) => x is >= 0 and <= 100;
```

Mas quando a entrada não é um tipo primitivo, para qual tipo tentar convertê-la?

```
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:

```
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,

```
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 e 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

```
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*.

```
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?

```
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:

```
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:

```
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.

```
case 1 and 2:
```

Da mesma forma, isso seria um erro porque um valor não pode ser `1` e `2`.

```
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.

```
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.

```
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

21/01/2022 • 28 minutes to read

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:

```
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:

```
struct Point
{
    public int X { get; init; }
    public int Y { get; init; }
}
```

O consumidor pode usar inicializadores de objeto para criar o objeto

```
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:

```
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:

```
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:

```

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`

```

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:

```

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`.

```

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:

```

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.

```

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:

```

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 clientes do `init` vão ler o membro. Os compiladores `init` que não sabem do irão ignorar o `set` acessador e, portanto, não tratarão acidentalmente 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 emitim 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 straight-forward.

init vs. InitOnly

Havia três formas de sintaxe que tiveram uma consideração significativa durante nossa reunião LDM:

```
// 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.

```
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:

```
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 *gracioso* 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:

```
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:

```
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.

```
var student = new Student()
{
    Name = await SomeMethod()
};
```

Aqui, o resultado de `new Student()` será hoisted 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 participar no processo de construção para inicializar `init` campos e propriedades.

Esses membros teriam todos os restrições 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

21/01/2022 • 6 minutes to read

- [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.

```
Dictionary<string, List<int>> field = new() {
    { "item1", new() { 1, 2, 3 } }
};
```

Permitir a omissão do tipo quando ele puder ser inferido do uso.

```
XmlReader.Create(reader, new() { IgnoreWhitespace = true });
```

Crie uma instância de um objeto sem soletrar o tipo.

```
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.

```
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.

```
(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 declaração `using`, 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

21/01/2022 • 3 minutes to read

- [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.

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
    public sealed class ModuleInitializerAttribute : Attribute { }
}
```

O atributo pode ser usado da seguinte maneira:

```
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

21/01/2022 • 10 minutes to read

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.

```
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.

```
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:

```
[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.

```
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`:

```

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#.

```

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:

```

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

21/01/2022 • 2 minutes to read

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

21/01/2022 • 6 minutes to read

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:

```
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:

```
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:

```
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:

```
_ = (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:

```
_ = (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 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

21/01/2022 • 21 minutes to read

Resumo

Suporte 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

```
class Compilation ...
{
    public virtual Compilation WithOptions(Options options)...
}
```

```
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 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.

```

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:

```

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

```

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,

```
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.

21/01/2022 • 11 minutes to read

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 de 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

21/01/2022 • 2 minutes to read

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

21/01/2022 • 2 minutes to read

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

21/01/2022 • 16 minutes to read

Resumo

Supor 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.

```
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.)

OPERANDO	DESTINO	CONVERSÃO	IL
object	nint	Conversão unboxing	unbox
void*	nint	PointerToVoid	conv.i
sbyte	nint	ImplicitNumeric	conv.i
byte	nint	ImplicitNumeric	conv.u
short	nint	ImplicitNumeric	conv.i
ushort	nint	ImplicitNumeric	conv.u
int	nint	ImplicitNumeric	conv.i
uint	nint	ExplicitNumeric	conv.u / conv.ovf.u
long	nint	ExplicitNumeric	conv.i / conv.ovf.i
ulong	nint	ExplicitNumeric	conv.i / conv.ovf.i
char	nint	ImplicitNumeric	conv.i
float	nint	ExplicitNumeric	conv.i / conv.ovf.i
double	nint	ExplicitNumeric	conv.i / conv.ovf.i
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

OPERANDO	DESTINO	CONVERSÃO	IL
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	
OPERANDO	DESTINO	CONVERSÃO	IL
nint	object	Conversão boxing	box
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

OPERANDO	DESTINO	CONVERSÃO	IL
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(ulong)
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
nuint	uint	ExplicitNumeric	conv.u4 / conv.ovf.u4
nuint	long	ExplicitNumeric	conv.i8 / conv.ovf.i8
nuint	ulong	ImplicitNumeric	conv.u8 / conv.ovf.u8
nuint	char	ExplicitNumeric	conv.u2 / conv.ovf.u2.un
nuint	float	ImplicitNumeric	conv.r.un conv.r4
nuint	double	ImplicitNumeric	conv.r.un conv.r8
nuint	decimal	ImplicitNumeric	conv.u8 decimal decimal.op_Implicit(ulong)
nuint	IntPtr	Nenhum	

OPERANDO	DESTINO	CONVERSÃO	IL
<code>nuint</code>	<code>UIntPtr</code>	Identidade	
<code>nuint</code>	Enumeração	ExplicitEnumeration	

A conversão de `A` para `Nullable` 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` 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><</code>	<code>bool operator <(nint left, nint right)</code>	<code>blt / clt</code>
<code><</code>	<code>bool operator <(nuint left, nuint right)</code>	<code>blt.un / clt.un</code>
<code><=</code>	<code>bool operator <=(nint left, nint right)</code>	<code>ble</code>
<code><=</code>	<code>bool operator <=(nuint left, nuint right)</code>	<code>ble.un</code>
<code>></code>	<code>bool operator >(nint left, nint right)</code>	<code>bgt / cgtr</code>
<code>></code>	<code>bool operator >(nuint left, nuint right)</code>	<code>bgt.un / cgtr.un</code>
<code>>=</code>	<code>bool operator >=(nint left, nint right)</code>	<code>bge</code>

BINÁRIO	ASSINATURA DE OPERADOR	IL
<code>>=</code>	<code>bool operator >=(nuint left, nuint right)</code>	<code>bge.un</code>
<code>&</code>	<code>nint operator &(nint left, nint right)</code>	<code>and</code>
<code>&</code>	<code>nuint operator &(nuint left, nuint right)</code>	<code>and</code>
<code> </code>	<code>nint operator (nint left, nint right)</code>	<code>or</code>
<code> </code>	<code>nuint operator (nuint left, nuint right)</code>	<code>or</code>
<code>^</code>	<code>nint operator ^(nint left, nint right)</code>	<code>xor</code>
<code>^</code>	<code>nuint operator ^(nuint left, nuint right)</code>	<code>xor</code>
<code><<</code>	<code>nint operator <<(nint left, int right)</code>	<code>shl</code>
<code><<</code>	<code>nuint operator <<(nuint left, int right)</code>	<code>shl</code>
<code>>></code>	<code>nint operator >>(nint left, int right)</code>	<code>shr</code>
<code>>></code>	<code>nuint operator >>(nuint left, int right)</code>	<code>shr.un</code>

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
```

```
public class A
{
    public static nint F;
}
```

```
csc -langversion:8 -r:A.dll B.cs
```

```
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`.

```
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`:

```
// 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:

```
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.

```
static object GetItem(object[] array, nint index)
{
    return array[index]; // ok
}
```

`nint` e `nuint` pode ser usado como um `enum` tipo base.

```
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.

```

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`.

```

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-2017-06-13.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/2019/LDM-2019-10-23.md>
- <https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-03-25.md>

Ponteiros de função

21/01/2022 • 44 minutes to read

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>

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.

```
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.

```

//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.

```

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:

```

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_convention_specifier` for fornecido, o padrão será `managed`. A codificação de metadados precisa dos `calling_convention_specifier` e quais `identifier`s são válidos no `unmanaged_calling_convention` é abordada na [representação de metadados das convenções de chamada](#).

```

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:

```
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:

```
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>

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>

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>

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 `T1...Tk` e tipo de retorno `Tb`, e a resolução de sobreposição de `E` com os tipos `T1...Tk` gera

um único método com tipo de retorno `u`, uma *inferência de limite inferior* é feita de `u` para `Tb`.

Inferências exatas

<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 `delegate*<V2..vk, v1>` e `u` é um tipo de ponteiro de função `delegate*<U2..uk, u1>`, e a Convenção de chamada de `v` é idêntica a `u` e o refness de `vi` é idêntico a `ui`.

Inferências de limite inferior

<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>

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:

```
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_conventionSpecifier` para um `CallKind`

Um `calling_conventionSpecifier` 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 os `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.

```
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.

```
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:

```
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:

```
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`.

```
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 ao 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.

```
(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.

21/01/2022 • 7 minutes to read

- [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 local 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.
- Omite `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

21/01/2022 • 4 minutes to read

Resumo

Permitir anotações anuláveis para parâmetros de tipo que não estão restritos a tipos de valor ou tipos de referência: `T?`.

```
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.

```
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`.

```
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??`.

```
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.

```
#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.

```

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.

```

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.

```

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>
- [#t](https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-06-17.md)