Course Overview

MCSD – Microsoft Certification

Microsoft currently offers four MCSD certifications based on various technologies

* MCSD: Windows Store Apps
* MCSD: Web Applications
* MCSD: SharePoint Applications
* MCSD: Application Lifecycle Management

The MCSD: Windows Store Apps can be achieved by passing three exams:

* Programming in C# (Exam 70-483)
* Essentials of Developing Windows Store Apps Using C# (Exam 70-484)

**Esse exame foi desativado**

* Advanced Windows Store App Development Using C# (Exam 70-485)

**Esse exame foi desativado**

**Exame 70-483/Curso 20483B**

**Programação em C#**

Este exame é destinado a desenvolvedores com um ano de experiência ou mais em programação de lógica de aplicativo/negócio essencial para diversos tipos de aplicativos e plataformas de hardware/software usando C#.

**Exame 70-357: Desenvolvendo Aplicativos Móveis**

**Desenvolvendo Aplicativos Móveis**

Candidatos para o exame são os desenvolvedores experientes que projetam e desenvolvem aplicativos para Windows 10 de implantação para Windows Store ou Windows Store for Business.

Os candidatos têm, normalmente, mais de dois anos de experiência no desenvolvimento de aplicativos do Windows usando c# e XAML, além de experiência com WPF ou Silverlight para Windows Phone. Os candidatos também devem ter experiência com o padrão de projeto Model-View-ViewModel (MVVM) e Entity Framework, além das tecnologias de autenticação. Experiência com Visual Studio 2015 é altamente recomendada.

**Exame 70-486/Curso 20486B**

**Desenvolvendo Aplicativos Web ASP .NET MVC**

Este exame é destinado a desenvolvedores com no mínimo três a cinco anos de experiência no desenvolvimento de aplicativos Web ASP .NET, incluindo dois anos ou mais desenvolvendo soluções baseadas em MVC.

## Habilidades medidas

## 60 QUESTÕES

## 1-4 HORAS PARA COMPLETAR O EXAME

## PODE REVER QUESTÕES (Mas Não pode mover entre Caso de Estudos)

## 700 passou! (Mas 700 não é 70%)

Este exame mede sua habilidade em executar as tarefas técnicas listadas abaixo. As porcentagens indicam o peso relativo de cada área de tópico principal no exame. Quanto mais alta a porcentagem, mais perguntas você verá nessa área de conteúdo no exame.

**Observe que as perguntas não estarão limitadas aos tópicos descritos no texto com marcadores.**

Observação: Existem atualizações de exame efetivas a partir de quinta-feira, 14 de dezembro de 2017. Para aprender mais sobre estas mudanças e como elas afetam as habilidades medidas, por favor baixe e reveja o [Exam 483 change document](http://download.microsoft.com/download/A/D/F/ADF027D4-0541-4CFB-9202-3C48571ABB54/483_OD_Changes.pdf).

[**Criar e usar tipos (25–30%)**](https://www.microsoft.com/pt-br/learning/exam-70-483.aspx#syllabus-2)

* Criar tipos
  + Criar tipos de valor, incluindo structs e num; criar tipos de referência, tipos genéricos, variáveis estáticas
* Consumir tipos
  + Box ou unbox para converter entre tipos de valor; casting tipos; converter tipos; lidar com tipos dinâmicos; garantir a interoperabilidade com código que acesse APIs COM
* Aplicar encapsulamento
  + Aplicar encapsulamento usando propriedades; forçar encapsulamento usando assessores/modificadores, incluindo público, privado, protegido e interno; forçar encapsulamento usando implementação explícita de interface, construtores, métodos, classes, métodos de extensão, parâmetros opcionais e nomeados e propriedades indexadas; criar métodos sobrecarregados(overloading) e substituídos(overriding)
* Criar e implementar uma hierarquia de classe
  + Projetar e implementar uma interface; herdar de uma classe base; criar e implementar classes com base nas interfaces IComparable, IEnumerable, IDisposable e IUnknown
* Localizar, executar e criar tipos em tempo de execução usando reflexão
  + Criar e aplicar atributos; ler atributos; gerar código em tempo de execução usando expressões CodeDom e lambda; usar tipos do namespace System.Reflection, incluindo Assembly, PropertyInfo, MethodInfo, Type
* Gerenciar o ciclo de vida do objeto
  + Gerenciar recursos não gerenciados; implementar IDisposable, incluindo interação com finalização; gerenciar IDisposable usando a instrução Using; gerenciar finalização e coleta de lixo
* Manipular cadeias de caracteres
  + Manipular cadeias de caracteres usando as classes StringBuilder, StringWriter e StringReader; pesquisar cadeias de caracteres; enumerar métodos de cadeias de caracteres; formatar cadeias de caracteres; usar interpolação de cadeia

[**Gerenciar fluxo de programa (25–30%)**](https://www.microsoft.com/pt-br/learning/exam-70-483.aspx#syllabus-1)

* Implementar processamento multithreading e assíncrono
  + Usar a biblioteca Task Parallel, incluindo o Parallel.For method, PLINQ, Tasks; gerar threads usando ThreadPool; desbloquear a UI; usar palavras-chave async e await; gerenciar dados usando coleções simultâneas
* Gerenciar multithreading
  + Sincronizar recursos; implementar bloqueio; cancelar uma tarefa de execução longa; implementar métodos thread-safe para manipular condições de corrida
* Implementar fluxo de programa
  + Iterar em itens de coleção e de matriz; programar decisões usando instruções switch, se/então e operadores; avaliar expressões
* Criar e implementar eventos e retornos de chamada
  + Criar manipuladores de eventos; assinar e cancelar assinatura de eventos; usar tipos delegados integrados para criar eventos; criar delegados; expressões lambda; métodos anônimos
* Implementar manipulação de exceções
  + Manipular tipos de exceções, incluindo exceções SQL, exceções de rede (network exceptions), exceções de comunicação (communication exceptions), exceções de tempo limite de rede (network timeout exceptions); usar declarações de captura; usar uma classe base de uma exceção; implementar blocos try-catch-finally; lançar exceções; relançar uma exceção; criar exceções personalizadas; manipular exceções internas (inner exceptions); manipular exceções agregadas(aggregate exception)

[**Implementar acesso aos dados (25–30%)**](https://www.microsoft.com/pt-br/learning/exam-70-483.aspx#syllabus-4)

* Realizar operações de E/S
  + Ler e escrever arquivos e fluxos; ler e escrever a partir da rede usando classes no namespace System.Net; implementar operações de E/S assíncronas
* Dados de consumo
  + Recuperar dados de um banco de dados; atualizar dados em um banco de dados; consumir dados JSON e XML; recuperar dados usando serviços Web
* Consultar e manipular dados e objetos usando LINQ
  + Consultar dados usando operadores, incluindo projetar, juntar, agrupar, obter, ignorar, agregar; criar consultas LINQ baseadas em método; consultar dados usando sintaxe de compreensão de consulta; selecionar dados usando tipos anônimos; forçar a execução de uma consulta; ler, filtrar, criar e modificar as estruturas de dados de usando LINQ to XML
* Serializar e de-serializar dados
  + Serializar e de-serializar dados usando serialização binária, serialização personalizada, Serializador XML, Serializador JSON e Serializador de Contrato de Dados
* Armazenar dados e recuperar dados de coleções
  + Armazenar e recuperar dados usando dicionários, matrizes, listas, conjuntos e filas; escolher um tipo de coleção; inicializar uma coleção; adicionar e remover itens de uma coleção; usar coleções tipificadas vs. coleções não tipificadas; implementar coleções personalizadas; implementar interfaces de coleções

[**Depurar aplicativos e implementar segurança (25–30%)**](https://www.microsoft.com/pt-br/learning/exam-70-483.aspx#syllabus-3)

* Validar entrada de aplicativo
  + Validar dados JSON; escolher o tipo de coleta de dados adequado; gerenciar integridade de dados; avaliar uma expressão regular para validar o formato de entrada; usar funções internas para validar tipos e conteúdos
* Realizar criptografia simétrica e assimétrica
  + Escolher um algoritmo de criptografia apropriado; gerenciar e criar certificados; implementar gerenciamento de chave; implementar o namespace System.Security; fazendo hash de dados; criptografar fluxos
* Gerenciar montagens
  + Controlar versão de montagens; assinar montagens usando nomes fortes; implementar hospedagem lado a lado; colocar uma montagem no cache de montagens global; criar uma montagem WinMD
* Depurar um aplicativo
  + Criar e gerenciar diretrizes de pré-processador; escolher um tipo de compilação apropriado; gerenciar arquivos e símbolos de banco de dados do programa (símbolos de depuração)
* Implementar diagnóstico em um aplicativo
  + Implementar log e rastreamento; criação de perfil de aplicativos; criar e monitor contadores de desempenho; escrever para o log de eventos



<https://global3.mindhub.com/70-483-programming-in-c-microsoft-official/p/MU-70-483?utm_source=microsoft&utm_medium=certpage&utm_campaign=msofficialpractice>

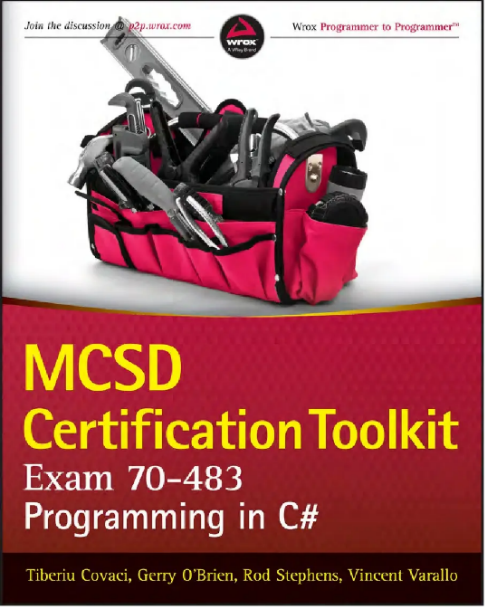
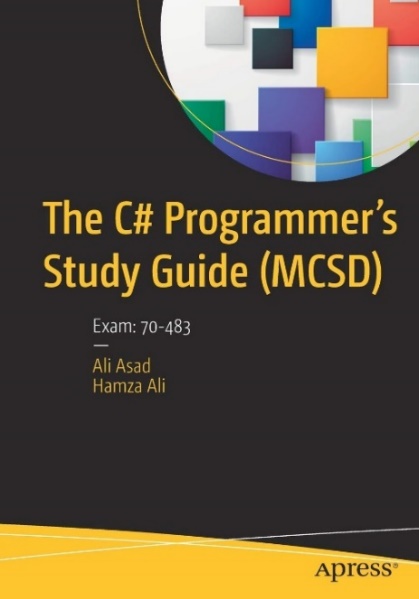
C# or VB programming skills are required to pass a number of other MCSD exams

* DevelopingASP.NET MVC WebApplications (70-486)
* Developing Microsoft Azure and Web Services (70-487)
* More... See www.microsoft.com/learning for more information

VCE FILE

Um arquivo Visual CertExam exame podem ser criados e modificados usando um aplicativo incluído no software Visual CertExam Suite, enquanto outro aplicativo também vem com o programa Visual CertExam Suíte é usado para tirar esses testes ou exames armazenados em um arquivo exame CertExam Visual. O programa Visual CertExam Designer é implementado com um conjunto de funcionalidades utilizadas para a inserção de perguntas e respostas de múltipla escolha em um teste ou exame que pode ser salva como um arquivo de exame CertExam Visual. O aplicativo Visual CertExam Manager é por outro lado um programa que é usado para fazer os testes e exames armazenados em um arquivo exame Visual CertExam, e este software também oferece recursos utilizados na gestão e facilitar esses testes ou exames. O conteúdo de um arquivo de exame CertExam Visual é integrado com as especificações de codificação de dados utilizados para armazenar os dados inseridos pelo usuário Visual CertExam Designer no formato VCE. Afixada com o [.vce](https://www.reviversoft.com/pt/file-extensions/vce) extensão, esses testes e exames são geralmente utilizados para as avaliações de certificação ou avaliações.

**PS: Neste link (**[**http://www.examcollection.com/70-536.html**](http://www.examcollection.com/70-536.html)**) tem mais exames no formato vce pra testar os conhecimentos pra prova do 70-536.**



# The C# Programmer's Study Guide (MCSD): Exam: 70-483 (Inglês)– 23 jun 2017

por [Ali Asad](https://www.amazon.com.br/s/ref=dp_byline_sr_book_1?ie=UTF8&field-author=Ali+Asad&search-alias=books) (Autor), [Hamza Ali](https://www.amazon.com.br/s/ref=dp_byline_sr_book_2?ie=UTF8&field-author=Hamza+Ali&search-alias=books) (Autor)

# Exam Ref 70-483 Programming in C# (MCSD)

## de [Wouter de Kort](https://www.kobo.com/br/pt/author/wouter-de-kort)

## MCSD Certification Toolkit (Exam 70-483): Programming in C# (Inglês)– 12 mai 2013

## por [Tiberiu Covaci](https://www.amazon.com.br/s/ref=dp_byline_sr_book_1?ie=UTF8&field-author=Tiberiu+Covaci&search-alias=books) (Autor), [Gerry O'Brien](https://www.amazon.com.br/s/ref=dp_byline_sr_book_2?ie=UTF8&field-author=Gerry+O%27Brien&search-alias=books) (Autor), [& 1 mais](https://www.amazon.com.br/MCSD-Certification-Toolkit-Exam-70-483/dp/1118612094)

.NET Framework

O .NET Framework é uma iniciativa da empresa [Microsoft](https://pt.wikipedia.org/wiki/Microsoft), que visa uma plataforma única para desenvolvimento e execução de sistemas e aplicações. Todo e qualquer código gerado para .NET pode ser executado em qualquer dispositivo que possua um [framework](https://pt.wikipedia.org/wiki/Framework) de tal plataforma. Com ideia semelhante à [plataforma Java](https://pt.wikipedia.org/wiki/Plataforma_Java), o programador deixa de escrever código para um sistema ou dispositivo específico, e passa a escrever para a plataforma .NET. Aplicações escritas para ele funcionam em um ambiente de software controlado, em oposição a um ambiente de hardware, através de uma máquina virtual de aplicação.

O .NET Framework consiste basicamente em dois componentes principais que permite executar diversas linguagens permitindo grande interoperabilidade entre elas, ou seja, ela é executada no CLR e interagindo com o FCL:

* [Common Language Runtime](https://pt.wikipedia.org/wiki/Common_Language_Runtime) - CLR (Ambiente de Execução Independente de Linguagem). O CLR fornece gerenciamento de memória, controle de exceção, interoperabilidade, manipulação de processamento paralelo e concorrente, reflexão, segurança, serviços de compilação para a arquitetura específica, entre outros.
* Framework Class Library - FCL (Conjunto de Bibliotecas Unificadas). A FCL oferece [APIs](https://pt.wikipedia.org/wiki/Interface_de_programa%C3%A7%C3%A3o_de_aplica%C3%A7%C3%B5es) para UI de console, acesso a dados, conectividade com banco de dados, redes, web, criptografia, acesso aos serviços do sistema operacional, estruturas de dados e algoritmos diversos, facilidades para a linguagem e muito mais.

Originalmente só funcionava no [Windows](https://pt.wikipedia.org/wiki/Microsoft_Windows), agora também funciona no [Linux](https://pt.wikipedia.org/wiki/Linux) e no [macOS](https://pt.wikipedia.org/wiki/MacOS) através do [.NET Core](https://pt.wikipedia.org/wiki/.NET_Core). Inicialmente era proprietário, mas seus fontes foram liberados. O .NET Core já nasceu como um projeto 100% aberto, contando com contribuições da comunidade através da .NET Foundation. Existem diversas variações, tais como [.NET Compact Framework](https://pt.wikipedia.org/wiki/.NET_Compact_Framework), .NET Micro Framework e [Silverlight](https://pt.wikipedia.org/wiki/Silverlight).

Esta CLR é capaz de executar através da [Common Language Infrastructure](https://pt.wikipedia.org/wiki/Infraestrutura_de_linguagem_comum), uma grande quantidade de linguagens de programação, interagindo entre si como se fossem uma única linguagem.

Algumas linguagens são:



Esta plataforma permite a execução, construção e desenvolvimento de [Web Services](https://pt.wikipedia.org/wiki/Web_Services) ([Aplicações Web](https://pt.wikipedia.org/wiki/Aplica%C3%A7%C3%B5es_Web)) de forma integrada e unificada.

Originalmente a plataforma .NET baseia-se em um dos princípios utilizados na tecnologia [Java](https://pt.wikipedia.org/wiki/Java_(linguagem_de_programa%C3%A7%C3%A3o)) (Just In Time Compiler - JIT), os programas desenvolvidos para ela são compilados duas vezes, uma na distribuição (gerando um código que é conhecido como "[bytecodes](https://pt.wikipedia.org/wiki/Bytecode)") e outra na execução.

Um programa é escrito em qualquer das linguagens de programação disponíveis para a plataforma, o código fonte gerado pelo programador é então compilado pela linguagem escolhida gerando um código intermediário em uma linguagem chamada CIL ([Common Intermediate Language](https://pt.wikipedia.org/wiki/Common_Intermediate_Language)).

Este novo código fonte gera um arquivo assembly, de acordo com o tipo de projeto:

* [EXE](https://pt.wikipedia.org/wiki/EXE) - Arquivos Executáveis, Programas
* [DLL](https://pt.wikipedia.org/wiki/DLL) - Biblioteca de Funções
* [ASPX](https://pt.wikipedia.org/wiki/ASPX) - Página Web
* [ASMX](https://pt.wikipedia.org/w/index.php?title=ASMX&action=edit&redlink=1) - [Web Service](https://pt.wikipedia.org/wiki/Web_Service)

No momento da execução do programa ele é novamente compilado, desta vez pelo compilador [JIT](https://pt.wikipedia.org/wiki/JIT)(Just-In-Time), de acordo com a utilização do programa, por exemplo: Temos um Web Site desenvolvido em [ASP.NET](https://pt.wikipedia.org/wiki/ASP.NET), ao entrar pela primeira vez em uma página o JIT irá compila-la, nas outras vezes que algum outro usuário acessar esta página, ele usará esta compilação.

Também é possível, através de ferramentas específicas, "pré-compilar" o código para que não se tenha o custo da compilação JIT durante a execução.

O fato desta arquitetura utilizar a [CIL](https://pt.wikipedia.org/wiki/Common_Intermediate_Language) gera uma possibilidade pouco desejada entre os criadores de software que é a de fazer a "[engenharia reversa](https://pt.wikipedia.org/wiki/Engenharia_reversa)", ou seja, a partir de um código compilado, recuperar o código original. Isto não é uma ideia agradável para as empresas que sobrevivem da venda de softwares produzidos nesta plataforma.

Por causa disso, existem ferramentas que "ofuscam" o código CIL, trocando nomes de variáveis, métodos, interfaces e etc para dificultar o trabalho de quem tentar uma engenharia reversa no mesmo.

Para melhorar a performance de execução é possível gerar um código nativo após instalado com o NGEN (Gerador de Imagem Nativa). Este NGEN é uma ferramenta que melhora o desempenho de aplicativos gerenciados. Ngen.exe cria imagens nativas, que são arquivos que contém o código de máquina específico do processamento compilado e as instala no cache de imagem nativa do computador local. O tempo de execução pode usar imagens nativas do cache em vez de usar o compilador JIT para compilar o assembly original.

### .

O .NET Framework 4 veio para melhorar, alguns pontos do Framework anterior, como por exemplo:

* Aplicações legadas podem continuar rodando no release anterior do Framework, para não haver problemas de compatibilidade
* Possui Background Garbage Collection
* Tem suporte para aplicações Multitouch
* Consegue fazer uso das novas funcionalidades do [Windows 7](https://pt.wikipedia.org/wiki/Windows_7)

Se você é um desenvolvedor Web, algumas das melhorias que são encontradas na nova versão do Framework:

* Pré-carregamento da sua aplicação
* A utilização de Routing no ASP.NET para Web Forms
* Controle/Redução de ViewState
* A utilização do padrão MVC

Visual Studio IDE

Integrated Development Environment (IDE) Available Editions:

* Ultimate
* Premium
* Professional
* Express
* Community

Ultimate, Premium, Professional, Test Professional editions

* Fornecer vários recursos para diferentes equipes de desenvolvimento
* Varia em custo e licenciamento

Express edition

* Livre
* Direcionado para plataforma específica (Web, C #, VB, etc.)
* Não é possível utilizar extensões

Community edition

* Introduzido no final de 2014
* Livre
* Versão completa e extensível com poucas restrições
* Não pode ser usado em ambientes corporativos ou equipes com mais de cinco pessoas
* Pode ser usado para qualquer outro tipo de projeto comercial ou não comercial

Getting Visual Studio Community Edition

* www.visualstudio.com
* Free developer tools
* www.microsoft.com
* Search for Visual Studio Community'
* Download is 6.9 GB

Hardware requirements

* 1.6 GHz or faster processor
* 1 GB of RAM (1.5 GB if running on a virtual machine)
* 20 GB of available hard disk space
* 5400 RPM hard disk drive
* DirectX 9-capable video card that runs at 1024 x 768 orhigher display resolution

C# Basics

CARACTERÍSTICAS DE C #

O C# é uma linguagem de programação visual dirigida por eventos e totalmente orientada a objetos. Permite um novo grau de intercâmbio entre linguagens (componentes de software de diferentes linguagens podem interagir). Os desenvolvedores podem empacotar até software antigo, para trabalhar com novos programas C#. Além disso, os aplicativos C# podem interagir pela Internet usando padrões do setor, como SOAP (protocolo de acesso a objetos simples) e XML (linguagem de marcação extensível).

O C# tem raízes em C, C++ e Java, adaptando os melhores recursos de cada linguagem e acrescentando novas capacidades próprias. Combina os melhores recursos das línguagens existentes anteriormente

* C ++
* Pascal
* Java
* Visual Basic

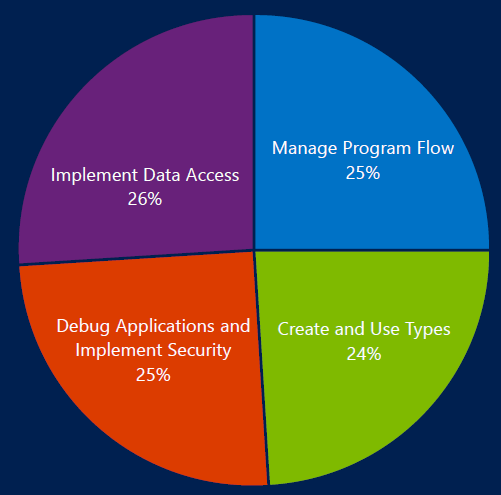
Ele fornece os recursos que são mais importantes para os programadores, como programação orientada a objetos, strings, elementos gráficos, componentes de interface com o usuário gráfica (GUI), tratamento de exceções, múltiplas linhas de execução, multimídia (áudio, imagens, animação e vídeo), processamento de arquivos, estruturas de dados pré-empacotadas, processamento de banco de dados, redes cliente/servidor com base na Internet e na World Wide Web e computação distribuída.

A versão final do **C# 8.0**esta prestes a sair do forno, provavelmente junto com a .NET Core 3.0.

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **.NET Framework** | **Visual Studio** | **Important Features** |
| C# 1.0 | .NET Framework 1.0/1.1 | Visual Studio .NET 2002 | * Basic features |
| C# 2.0 | .NET Framework 2.0 | Visual Studio 2005 | * Generics * Partial types * Anonymous methods * Iterators * Nullable types * Private setters (properties) * Method group conversions (delegates) * Covariance and Contra-variance * Static classes |
| C# 3.0 | .NET Framework 3.0\3.5 | Visual Studio 2008 | * Implicitly typed local variables * Object and collection initializers * Auto-Implemented properties * Anonymous types * Extension methods * Query expressions * Lambda expressions * Expression trees * Partial Methods |
| C# 4.0 | .NET Framework 4.0 | Visual Studio 2010 | * Dynamic binding (late binding) * Named and optional arguments * Generic co- and contravariance * Embedded interop types |
| C# 5.0 | .NET Framework 4.5 | Visual Studio 2012/2013 | * Async features * Caller information |
| C# 6.0 | .NET Framework 4.6 | Visual Studio 2013/2015 | * Expression Bodied Methods * Auto-property initializer * nameof Expression * Primary constructor * Await in catch block * Exception Filter * String Interpolation |
| C# 7.0 | .NET Core 2.0 | Visual Studio 2017 | * out variables * Tuples * Discards * Pattern Matching * Local functions * Generalized async return types * [Numeric literal syntax improvements](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7#numeric-literal-syntax-improvements) |
| C# 8.0 | .NET Core 3.0 | Visual Studio 2019 | * Readonly members * Default interface methods * Using declarations * Static local functions * Disposable ref structs * Nullable reference types * [Asynchronous streams](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#asynchronous-streams) * [Indices and ranges](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#indices-and-ranges) * [Null-coalescing assignment](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#null-coalescing-assignment) * [Unmanaged constructed types](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#unmanaged-constructed-types) * [Stackalloc in nested expressions](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#stackalloc-in-nested-expressions) * [Enhancement of interpolated verbatim strings](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#enhancement-of-interpolated-verbatim-strings) |

C # foi projetado com base em sete pontos básicos

|  |  |
| --- | --- |
| Simples | * Menos operadores, sem ponteiros, etc. |
| Moderno | * Coleta de lixo, segurança aprimorada |
| Orientado a objetos | * Encapsulamento, herança, polimorfismo * Nenhuma função global, variáveis ​​ou constantes * Tudo é encapsulado dentro de uma classe |
| Tipo seguro | * Não é possível usar variáveis ​​não inicializadas * Casts(Conv ersões de tipos) inseguros não são permitidos |
| Versão | * O C # suporta o de controle de versão fornecida pelo Common Language Specification (CLS) |
| Compatível | * Permite acessar diferentes APIs * Compilador garante conformidade com CLS |
| Flexível | * Classes e/ou métodos podem ser declarados como inseguros ' * Habilitando o uso de ponteiros e outros itens necessários para usar outras APIs |



**CRIAR E USAR TIPOS (25–30%)**

Criar tipos

* Criar tipos de valor, incluindo structs e num; criar tipos de referência, tipos genéricos, variáveis ​​estáticas.

POO - PROGRAMAÇÃO ORIENTADA A OBJETOS

Essa colocação nos traz a um detalhe importante: a programação estruturada, quando bem feita, possui um desempenho superior ao que vemos na programação orientada a objetos. Isso ocorre pelo fato de ser um paradigma sequencial, em que cada linha de código é executada após a outra, sem muitos desvios, como vemos na **POO**. Além disso, o paradigma estruturado costuma permitir mais liberdades com o hardware, o que acaba auxiliando na questão desempenho. As vantagens do OOP são:

* O código é dividido em grupos independentes
* O código pode ser facilmente reutilizado
* Complexidade reduzida
* Desacoplamento pode ser alcançado
* Extensibilidade

Classes

Uma **classe** é uma estrutura que abstrai um conjunto de **objetos** com características similares. Uma **classe** define o comportamento de seus **objetos** - através de métodos - e os estados possíveis destes **objetos** - através de atributos. Uma classe é um modelo usado para criar objetos

* Fornece detalhes necessários para criar uma versão de software de um objeto real.
* Uma classe é semelhante a um planta usada para construir edifícios. Um projeto pode ser usado para construir vários edifícios, uma classe pode ser usada para criar vários objetos

Objetos

Um objeto, em programação orientada a objetos, é uma instância (ou seja, um exemplar) de uma classe. Um objeto é capaz de armazenar estados através de seus atributos e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos.

Atributos são características de um objeto. Basicamente a estrutura de dados que vai representar a classe.

Um objeto é uma variável que contém duas coisas

* As propriedades que foram definidas na classe
* As habilidades definidas como métodos na classe

Vários objetos podem ser criados com base em uma única classe. Apesar de cada objeto tem seu próprio 'estado' ou coleção exclusiva de valores de dados, como todos os objetos são baseados em classes, ao alterar uma classe afetará todos os objetos criados a partir dessa classe

Exemplos: um objeto da classe "Funcionário" teria como atributos "nome", "endereço", "telefone", "CPF", etc

O conjunto de valores dos atributos de um determinado objeto é chamado de estado.

Herança (Veja mais detalhes em Aplicar Hierarquia, mais abaixo)

**Herança** é um princípio de orientação a objetos, que permite que classes compartilhem atributos e métodos, através de "**heranças**". Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.

* Permite que uma classe reutilize o código de outra classe
* Um único objeto pode possuir a funcionalidade de várias classes
* Evita duplicação de código

Oferece a capacidade de estender e modificar o comportamento em um ambiente de fácil manutenção. A classe cujos membros são herdados é chamada classe base e a classe que herdou esses membros é chamada classe derivada.

classParent1

{

publicstring SurName { get; set; }

}

classChild1 : Parent1

{

privatestring \_name;

publicstring Name

{

get

{

return (\_name + " " + base.SurName);

}

set

{

Name = value;

}

}

}

Em C#, uma classe não pode herdar de várias classes, mas pode ser herdada em vários níveis, o que é conhecido como **herança** múltipla. Outra observação é que uma classe não pode herdar várias classes, mas pode implementar várias interfaces ao mesmo tempo.

Lembre-se, na herança ao criar um objeto de uma classe derivada, o compilador sempre executa suao construtor da classe base/pai primeiro e depois executa o construtor da classe filho. Se a mesma classe pai herdou de outra classe, o construtor dessa classe será executado primeiro e, em seguida, descerá ao nível de filho e neto para executar o construtor um após o outro.

Encapsulamento (Veja mais detalhes em Encapsulamento, mais abaixo)

A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados *getters* e *setters*, que irão retornar e setar o valor da propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos. Por exemplo, quando clicamos no botão ligar da televisão, não sabemos o que está acontecendo internamente. Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

* Agrupamento de dados e funções em um único componente
* Permite 'ocultação' seletiva de dados e funcionalidades
* Fornece segurança e facilidade de uso para objetos
* Exemplo: fazer uma chamada usando um smartphone

Polimorfismo (Veja mais detalhes em Encapsulamento, mais abaixo)

Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus “ancestrais”. Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o *polimorfismo* consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

Como um exemplo, temos um objeto genérico “Eletrodoméstico”. Esse objeto possui um método, ou ação, “Ligar()”. Temos dois objetos, “Televisão” e “Geladeira”, que não irão ser ligados da mesma forma. Assim, precisamos, para cada uma das classes filhas, reescrever o método “Ligar()”.

Com relação ao polimorfismo, valem algumas observações. Como se trata de um assunto que está intimamente conectado à herança, entender os dois juntamente é uma boa ideia. Outro ponto é o fato de que as linguagens de programação implementam o polimorfismo de maneiras diferentes. O C#, por exemplo, faz uso de método virtuais (com a palavra-chave *virtual*) que podem ser reimplementados (com a palavra-chave *override*) nas classes filhas. Já em Java, apenas o atributo “@Override” é necessário.

* O resultado da herança e encapsulamento
* A capacidade de fornecer funcionalidades diferentes para o mesmo nome de método
* Com base no contexto de como o método é chamado
* Uma hierarquia de herança de classes e objetos pode fornecer várias funções com o mesmo nome: mas funcionalidades diferentes

TIPOS DE C #

C # é uma linguagem fortemente tipada

* Toda variável e constante tem um tipo específico
* Expressões que avaliam um valor têm um tipo

Um tipo especifica até quatro características principais dos dados que podem ser armazenados

* O espaço de armazenamento que uma variável do tipo requer
* Os valores máximo e mínimo que ele pode representar
* Os membros (métodos, campos, eventos etc.) que ele contém
* O tipo de base herdado

O que significa 'tipo seguro'? O compilador garante que todas as operações executadas sejam válidas para o tipo de dados que está sendo manipulado

Exemplos:

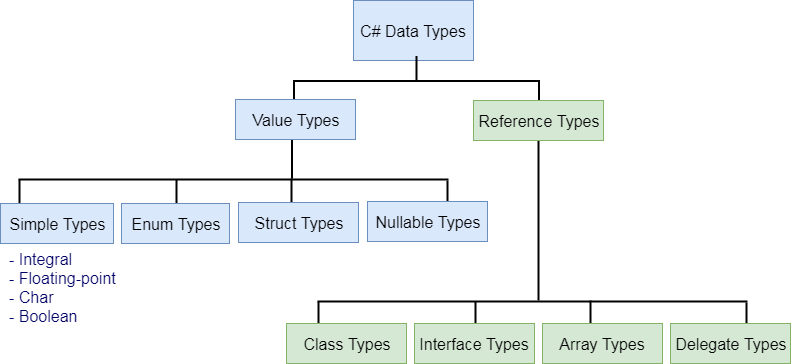
* Variáveis ​​que são do tipo inteiro (int) podem ser usadas em operações de adição e subtração
* Variáveis ​​do tipo bool não podem ser usadas em operações de adição e subtração

C # fornece um conjunto padrão de tipos internos

* Numérico
* String
* Objeto

VALUE TYPES (Tipos de Valor) E REFERENCE TYPES (Tipos de Referência)

O C # categorizou principalmente os tipos de dados em dois tipos: tipos de valor e tipos de referência. Os tipos de valor incluem tipos simples (por exemplo, int, float, bool e char), tipos de enumeração, tipos de estrutura e tipos de valor Nullable. Os tipos de referência incluem tipos de classe, tipos de interface, tipos de delegados e tipos de matriz.



Os tipos são divididos em duas categorias segundo:

* Regras em tempo de compilação
* Comportamento em tempo de execução (uso e gerenciamento de memória)

As categorias são:

* **Tipos de valor**(**Value-Type)**:derivados de System.ValueType, são tipos leves como os tipos primitivos e structs, que ficam armazenados diretamente na memória stack. Os valores das variáveis ficam armazenados juntamente com as próprias variáveis, sendo o acesso ao seu conteúdo feito de maneira direta;
* **Tipos de referência(Reference-Type)**: derivados de System.Object, são tipos pesados como objetos criados a partir de classes, que ficam armazenados na heap. Para não sacrificar a performance, é criada uma referência (ponteiro) na stack que aponta para qual posição de memória o objeto está armazenado na heap. O acesso é feito via essa referência na stack. Sendo assim, o acesso ao conteúdo é indireto, dependendo dessa referência;

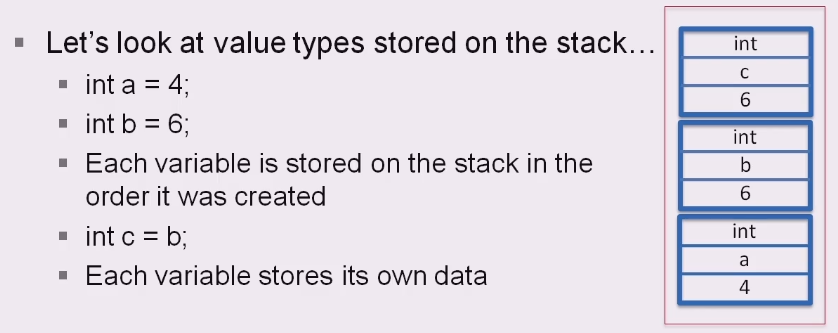
O .NET Framework armazena itens na memória em um dos dois locais

* **Stack (Pilha):** Os tipos de valor são armazenados. Porção de memória pequena onde os value-types e os ponteiros ficam;
* **Heap(montes, porções)**: Os tipos de referência são armazenados. Porção maior de memória onde os reference-types ficam de fato alocados. Para se fazer o acesso a eles, precisamos de um ponteiro na stack que indique a posição de memória na heap onde o objeto está de fato alocado.

Na configuração padrão do .NET Framework, para que você tenha uma idéia melhor de como a *stack* é muito menor que a *heap*, o tamanho padrão para a memória *stack* é de apenas 1MB!

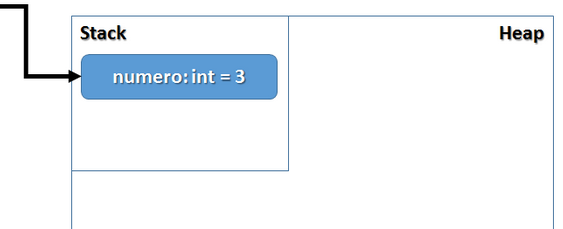
Ambas trabalham como pilhas, porém, a maneira como cada uma provê acesso a seu conteúdo é diferente. A *stack* é bem mais eficiente para localizar as coisas em seu interior com relação a *heap*, mesmo porque ela é bem menor.

As variáveis de alguns tipos de dados leves (tipos primitivos – int, double, bool etc. – e structs) são armazenadas diretamente na stack, a área menor e mais eficiente para localização dos conteúdos. Elas ficam diretamente nessa área justamente por serem tipos de dados que não ocupam tanto espaço na memória. O mais interessante é que o valor que elas contêm também fica junto com elas na *stack*. Ou seja, quando você faz a declaração abaixo:

Vejamos os tipos de valor armazenados na **Stack - pilha**...

* A quantidade de memória necessária para armazenar um tipo de valor é uma quantidade definida
* Quando a variável não é mais necessária, ela é excluída da pilha. Exemplo:
  + int a = 4;
  + int b = 6;
* Cada variável é armazenada na pilha na ordem em que foi criada
  + int c = b;
* Cada variável armazena seus próprios dados

Perceba que o valor da variável fica junto com a própria variável. Variáveis onde isso acontece são chamadas de Value-Types, justamente porque o valor delas fica junto com a própria variável na memória stack. Assim, quando você tem o seguinte código



Agora, outros tipos de dados ocupam muito mais espaço de memória do que estes tipos leves que são *value-types*. Por isso, eles não podem ser armazenados diretamente na *stack* (caso fossem, rapidamente a memória *stack* seria “estourada”, causando o famoso erro *StackOverflowException*). Sendo assim, estes dados são armazenados na memória *heap*. Vamos imaginar que você tenha o seguinte código:

class Pessoa

{

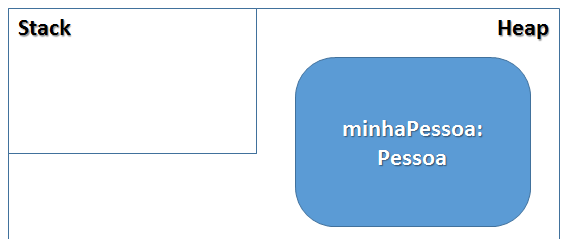
public int Id {get; set;}

public string Nome {get; set;}

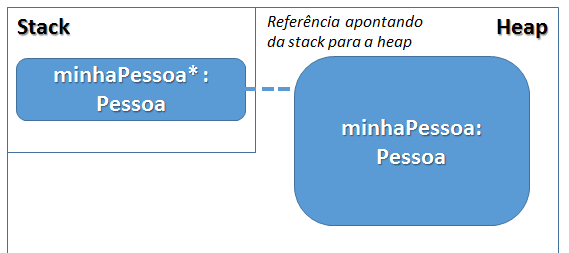
}

Quando você cria um objeto dessa classe, este objeto será armazenado na memória *heap*:

Pessoa minhaPessoa = new Pessoa();

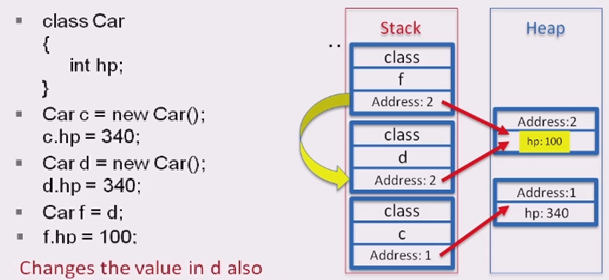


Porém, o compilador não acessa a heap diretamente. Por que ele não acessa? Justamente porque ela é muito grande… Se ele fosse procurar o objeto minhaPessoa dentro da heap, ele iria demorar um tantinho bom de tempo. O compilador precisaria ter um jeito de acessar pela stack (que é rápida pra encontrar as coisas até mesmo por ser bem menor) o que está alocado na heap (que é bem maior). Como o compilador contorna isso? Criando uma referência dentro da stack para o objeto minhaPessoa, apontando onde na memória heap que este objeto está de fato guardado!



Essa porção de memória que é alocada na *stack* para apontar para uma posição de memória da *heap* é chamada de **ponteiro**. Por isso ele tem esse asterisco (\*) na frente do seu nome.

Repare então que é criada uma **referência** da *stack* para uma determinada posição de memória da *heap*, referência essa guardada por um **ponteiro** na stack. Esse tipo de variável (como no caso da variável *minhaPessoa*, do tipo *Pessoa*) é chamada de *Reference-Type*, já que é necessário uma referência da *stack* para a *heap* para que esta variável seja acessível. Variáveis *reference-type* geralmente precisam que seja chamado o respectivo construtor através da palavra-chave **new**, pois ele é que define que uma porção de memória da *heap* deverá ser utilizada para guardar aquele objeto.



Agora, veja outro exemplo de **valores de referência na Heap** ...

class Car  
{  
 int hp;  
}  
Car c = new Car ();

c.hp = 340;  
Car d = new CarO;

d.hp = 340;  
Car f = d;  
f.hp = 100; <- altera o valor em d também

As variáveis baseadas em tipos de valor (value types) contêm diretamente o valor. Quando copiamos uma variável para outra, uma cópia do valor é passada; em tipos de referência (reference types), é passada somente uma referência do objeto. Os values types dividem-se em duas categorias principais: estruturas (structs) e enumerações (enum).

TIPOS DE REFERÊNCIA – reference types

Nas variáveis de tipos de referência (reference types), somente uma referência doobjeto é passada. A seguir, temos os tipos de referências: class, interface, delegate, object, string e Array.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Descrição** | **Alcance** | **Sufixo** |
| object | Tipo base de todos os outros tipos. | ----- | ---- |
| string | Uma sequência de caracteres Unicode | ----- | ---- |

Tipo object

Todos os tipos são herdados direta ou indiretamente da classe Object. Assim, é possívelconverter qualquer tipo para object. O ato de converter um variável value type para object é chamado de Boxing. E quando um tipo object é convertido para um value type, é chamado de Unboxing. O tipo object é representado pela classe Object no .NET Framework.

object x = 1;

Tipo string

O tipo string é o mais utilizado, uma vez que todas as variáveis que não contêm númerosquase sempre são declaradas como string. O tipo string é representado por uma sequência de caracteres Unicode. Uma string deve estar cercada por aspas duplas (""):

string b = "texto aqui";

char extrai = b[6];

Console.WriteLine(extrai); //a

Console.WriteLine(extrai + b); //atexto aqui

Console.WriteLine(extrai + "2"); //a2

Console.WriteLine(b + '2'); //texto aqui2

Console.WriteLine(extrai + '2'); //147

Console.WriteLine((int)extrai); //97 , valor ASCII

Console.WriteLine(extrai + 2); //99

Um recurso importante é a facilidade de extrair um caractere definido na variável, como ocorre em char extrai = b[6]. O sinal de mais (+) é usado para concatenar uma string mas, devemos ter cuidado ao utilizar este recurso com outros tipos pois podem trazer resultados indesejáveis. Você pode extrair uma letra de uma palavra:

char extrai = "Alfredo"[3]; // r

O arroba (@) evita que sequências de escape sejam processadas:

@"C:\Livro\Capitulo1\Construtores.cs"

É o mesmo que:

"C:\\Livro\\Capitulo1\\Construtores.cs"

Evita que aspas sejam interpretadas dentro de uma string cercada por aspas duplas.

string aspas = @"""abc"" teste";

A linha anterior retorna: "abc" teste.

TIPOS DE VALOR – value types

As estruturas – struct são divididas em tipos numéricos (tipos integrais, ponto flutuante e decimal), bool e estruturas personalizadas criadas pelos programadores.O C # inclui alguns tipos de valor predefinidos. A tabela a seguir lista os tipos de dados predefinidos:

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Descrição** | **Alcance** | **Sufixo** |
| byte | Número inteiro não assinado de 8 bits | 0 a 255 |  |
| sbyte | Número inteiro assinado de 8 bits | -128 a 127 |  |
| short | Número inteiro assinado de 16 bits | -32.768 a 32.767 |  |
| ushort | Inteiro não assinado de 16 bits | 0 a 65.535 |  |
| int | Número inteiro assinado de 32 bits | -2.147.483.648a2.147.483.647 |  |
| uint | Número inteiro não assinado de 32 bits | 0 a 4.294.967.295 | u |
| long | Número inteiro assinado de 64 bits | -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 | I |
| ulong | Número inteiro não assinado de 64 bits | 0 a 18.446.744.073.709.551.615 | ul |
| char | Caractere Unicode único de 16 bits | Qualquer caractere válido, por exemplo, a, \*, \ x0058 (hex) ou \ u0058 (Unicode) |  |
| bool | Valor verdadeiro / falso lógico de 8 bits | Verdadeiro ou falso |  |
| Datetime | Representa data e hora | 0:00:00 1/1/01 para 11:59:59 pm 31/12/9999 |  |

Os tipos de ponto flutuante são: float, double, os quais diferem entre si na faixa e na precisão

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Descrição** | **Alcance** | **Sufixo** |
| float | Tipo de ponto flutuante de precisão única de 32 bits | -3.402823e38 a 3.402823e38 | f |
| double | Tipo de ponto flutuante de precisão dupla de 64 bits | -1.79769313486232e308 a 1.79769313486232e308 | d |
| decimal | Tipo decimal de 128 bits para cálculos financeiros e monetários | (+ ou -) 1,0 x 10e-28para7.9 x 10e28 | m |

Se não atribuirmos um valor ao declarar uma variável, gerará um erro sobre o uso de uma variável não atribuída. Isso evita o uso de um valor que foi armazenado no local da memória antes da criação dessa variável. A atribuição é feita com um valor ou através da palavra-chave new e o alias do tipo, neste caso, essa instrução funcionará bem e imprimirá o valor padrão para esse tipo, que no exemplo abaixo caso é 0 para integer.

int myInt; // emite erro na compilação

int myNewInt = newint(); // 0

System.Int32 myInt32 = new System.Int32();// 0

Além do intervalo padrão suportado por cada tipo, podemos acrescentar um valor nulo. Para isso, basta declarar o nome do tipo com um ponto de interrogação.

int? x = 10;

Use o operador ?? para definir um valor padrão, caso o tipo seja nulo. Ou use o método GetValueOrDefault:

int d = x ?? 0;

int? x = 10;

x.GetValueOrDefault(0);

A propriedade HasValue verifica se um tipo contém ou não um valor nulo.

int? x=5;

if (x.HasValue)Console.WriteLine(x.Value);

else Console.WriteLine("Valor nulo.");

Tipo decimal

O tipo decimal é de alta precisão. Ideal para cálculos financeiros e monetários, é umtipo de dados de 128 bits que pode representar valores de aproximadamente:

±1.0 × 10−28 até ±7.9 × 1028

Com 28 ou 29 dígitos significantes. A precisão é dada em dígitos e não em casas decimais.

O sufixo m ou M deve ser utilizado para declarar variáveis do tipo decimal.

decimal x = 102.89m;

Sem o sufixo m ou M, a variável será tratada como se fosse do tipo double. Um erro égerado em tempo de compilação.

BigInteger e Complex

O .NET Framework 4.0 inclui dois novos tipos: BigInteger e Complex, sendo que ambos fazem parte do namespace System.Numerics e não contêm intervalos fixos, ou seja, não têm limites predefinidos como um inteiro tradicional.Para utilizar um tipo BigInteger ou Complex, é preciso adicionar uma referência a namespace System.Numerics.

using System;

using System.Numerics;

namespace ComplexType

{

classProgram

{

staticvoid Main(string[] args)

{

long varLong = 931548458140;

BigInteger fromLong = varLong;

Console.WriteLine("Número convertido para BigInteger.");

Console.WriteLine(fromLong);

Console.Read();

}

}

}

O tipo Complex suporta operações aritméticas e trigonométricas, bastante úteis para aplicações na área de engenharia elétrica e eletrônica.

staticvoid Main(string[] args)

{

var c1 = new Complex(2, 5);

var c2 = new Complex(4, 7);

var add = c1 + c2;

Console.WriteLine("Adição: " + add);

var sub = c1 - c2;

Console.WriteLine("Subtração: " + sub);

var mult = c1 \* c2;

Console.WriteLine("Multiplicação: " + mult);

var div = c1 / c2;

Console.WriteLine("Divisão: " + div);

Console.Read();

}

Tipo char

Representa um único caractere Unicode de 16 bits. É utilizado para representar a maioria das linguagens no mundo. Assim, podemos criar variáveis do tipo char e adicionar caracteres:

char letra = 'A';

char letra1 = 'H';

Uma variável char pode conter sequências de escape hexadecimal (prefixo \x) ou uma representação Unicode (prefixo \u):

char letra2 = '\x0072'; // Hexadecimal

char letra3 = '\u0072'; // Unicode

Podemos transformar, de forma explícita, um integral em um char ou vice-versa.

char letra4 = (char)72; // corresponde a letra H

int numero = (int)'B'; // inteiro 66

Combinações de caracteres que consistem de uma barra invertida (\) seguida de uma letra ou combinação de dígitos são chamadas de sequência de escape. Sequências de escape são usadas em situações específicas, como: salto de linha, retorno de carro, avanço de página, tabulação horizontal e vertical. Veja a seguir as sequências de escape usadas no C#:

|  |  |
| --- | --- |
| **Sequência** | **Significado** |
| \' | Apóstrofo |
| \" | Aspas |
| \\ | Barra invertida |
| \a | Alerta |
| \b | Retrocesso |
| \f | Avanço de página |
| \n | Salto de linha |
| \r | Retorno de carro |
| \t | Tabulação horizontal |
| \v | Tabulação vertical |

Tipo bool

Representa um valor verdadeiro ou falso. É usado com variáveis ou métodos que retornam o valor true ou false. O valor-padrão do tipo bool é false:

bool x = true;

bool b = false;

ENUM - Enumerações

As enumerações permitem criar um tipo distinto, formado por um conjunto de cons-tantes nomeadas. A seguir, temos sua forma mais simples:

enumDias { Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado };

Os elementos da enumeração são por padrão do tipo int. No entanto, os elementos podem ser de qualquer tipo integral, exceto char. O primeiro elemento é zero, o se-gundo elemento é igual a 1, e assim sucessivamente.

Na enumeração anterior, Domingo é igual a 0 (zero), Segunda é igual a 1, Terça é igual a 2 etc.Podemos também atribuir valores arbitrários a cada elemento:

enumDias { Domingo = 10, Segunda = 25, Terça = 48, Quarta = 8, Quinta, Sexta, Sábado };

Os elementos aos quais não atribuímos valores arbitrários são incrementados a partir do último elemento com valor. Onde Domingo é igual a 10, Segunda é 25, Terça é 48, Quarta é 8, Quinta é 9, Sexta é 10, Sábado é 11.

Um tipo integral diferente de int pode ser definido:

enumDias : short { Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado };

Para acessar o valor de cada elemento, é necessário converter, de forma explícita, cada elemento da enumeração para int:

int x = (int)Dias.Domingo;

int y = (int)Dias.Sexta;

Console.WriteLine(x); // 0

Console.WriteLine(y); // 5

Console.ReadKey();

Para acessar o valor de vários elementos ao mesmo tempo, é preciso antes preparar a enumeração. O atributo System.FlagsAttribute deve ser usado na declaração da enumeração. O exemplo completo que segue retorna Segunda, Sexta, Sábado:

[Flags] publicenumDias { Segunda = 0x01, Quarta = 0x02, Sexta = 0x04, Sábado = 0x08, }

staticvoid Main(string[] args)

{

Dias d = Dias.Sábado | Dias.Segunda | Dias.Sexta;

Console.WriteLine(d); // Segunda, Sexta, Sábado

Console.ReadKey();

}

Você aprendeu anteriormente que cada tipo de valor tinha seu próprio tipo de sistema equivalente, como System.Int32 ou System.Byte. O tipo de enumeração não é diferente porque é uma instância do tipo System.Enum, que contém vários métodos que você pode usar com suas próprias enumerações. A seguir está um código de exemplo que mostra alguns dos métodos disponíveis quando você trabalha com suas enumerações

string name = Enum.GetName(typeof(Dias), 2);

Console.WriteLine("The 2th day in the enum is " + name); //The 2th day in the enum is Quarta

foreach (int values in Enum.GetValues(typeof(Dias)))

{

Console.WriteLine(values); //1 2 4 8

}

STRUCTS

Uma estrutura é muito semelhante a uma classe. Parece e age como uma classe e compartilham muitas funcionalidades de classes como campos, propriedades, métodos e eventos

Como as estruturas são diferentes das classes

* Não é possível declarar um construtor padrão (construtor sem parâmetros). Mas é possível ter um construtor com parâmetros.
* Não é possível herdar de outra estrutura ou classe
* Pode ser instanciado sem a palavra-chave 'new', mas não há chamada do construtor e todos os campos devem ser inicializados antes que o objeto possa ser usado
* **É um tipo de valor, não um tipo de referência (classe)**

classProgram

{

staticvoid Main(string[] args)

{

//Possível ter construtor com parametros...

Mark m = new Mark(6,7);

m.Dothis();

//Se não colocar new, tem que inicializar variaveis...

Mark m2;

m2.x = 3;

m2.y = 12;

m2.Dothis();

}

}

publicstructMark

{

publicint x;

publicint y;

public Mark(int x, int y)

{

this.x = x;

this.y = y;

}

publicvoid Dothis()

{

Console.WriteLine(x + y);

Console.ReadLine();

}

}

GENERIC - Tipos genéricos

Tipos Genéricos fornecem a solução para uma limitação nas versões anteriores do Common Language Runtime e da linguagem C #. Os Tipos Genéricos foram adicionados à versão 2.0 da linguagem C #, tendo como ideia básica, permitir que os tipos (Número inteiro, String e tipos definidos pelo usuário) seja um parâmetro para métodos, classes e interfaces.

* Introduzido no C # versão 2.0
* Uma maneira mais rápida, segura e eficiente de lidar com coleções tipo em tempo de compilação
* Tornar possível projetar classes e métodos que adiam a especificação de um ou mais tipos até que a classe ou método seja declarado e instanciado pelo código do cliente

Antes do C # versão 2.0, coleções como ArrayList não eram genéricas e tinham uma limitação quanto a ausência de verificação de tipo eficaz. Você pode colocar qualquer objeto em uma ArrayList, porque todas as classes na linguagem de programação C # se estendem da classe base do objeto, mas isso compromete a segurança de tipo e contradiz a definição básica de C # como uma linguagem de segurança de tipo. Além disso, o uso de ArrayList envolve uma sobrecarga significativa de desempenho na forma de conversão de tipo implícita e explícita, necessária para adicionar ou recuperar objetos.

Tipos Genéricos introduzem o conceito de parâmetros de tipo, pelo qual é possível criar classes, estruturas, interfaces e métodos que possuem espaços reservados para os tipos usados e adiam o enquadramento do tipo de dados até que a classe ou método seja declarado e instanciado pelo código do cliente. Isso permite definir as estruturas de dados seguras para tipos, além de resultar em um notável aumento de desempenho e código de alto nível, porque ajuda a reutilizar algoritmos de processamento de dados sem replicar código específico de tipo e reduzem a necessidade de boxe, unboxing e conversão de tipos de variáveis ​​ou objetos.

Vantagens dos genéricos

* **Reutilização**: você pode usar uma única definição de tipo genérico para várias finalidades no mesmo código sem nenhuma alteração. Por exemplo, você pode criar um método genérico para adicionar dois números. Este método pode ser usado para adicionar dois números inteiros e dois flutuadores sem nenhuma modificação no código.
* **Segurança de tipo**: tipos de dados genéricos fornecem melhor segurança de tipo, especialmente no caso de coleções. Ao usar genéricos, você precisa definir o tipo de objetos a serem passados ​​para uma coleção. Isso ajuda o compilador a garantir que apenas os tipos de objetos definidos na definição possam ser passados ​​para a coleção.
* **Desempenho**: os tipos genéricos oferecem melhor desempenho em comparação aos tipos de sistema normais, porque reduzem a necessidade de boxe, unboxing e conversão de tipos de variáveis ​​ou objetos.
* Podemos usar o Generic com classes, interfaces, eventos e delegados.
* Podemos colocar restrições na classe genérica, para que o cliente possa usar apenas os tipos selecionados.

Generic Nullables

Uma área do .NET Framework na qual você pode ver o uso de genéricos está no suporte para Nullables. Um tipo de referência pode ter um valor real nulo, o que significa que não tem valor. Um tipo de valor não pode ter um valor nulo, no entanto. Por exemplo, como você expressaria que algum valor booleano é verdadeiro, falso ou desconhecido? Um booleano comum pode ser verdadeiro ou falso.

É por isso que os Nullables foram adicionados ao C# 2.0. Um Nullable é um wrapper em torno de um tipo de valor com um sinalizador booleano que ele armazena se o Nullable tiver um valor definido. Pense nisso como algo como a seguinte estrutura.

[Serializable]

publicstructNullable<T>where T : struct

{

publicbool HasValue { get; }

public T Value { get; }

// other implementation

}

Você pode usar o '?' operador para abreviar a sintaxe p. int ?, por muito tempo? em vez de usar Nullable <T>.

Em vez de criar um tipo Nullable para cada tipo de valor possível, agora existe apenas uma implementação que usa um parâmetro de tipo genérico para torná-lo mais flexível. Dessa forma, os genéricos podem ser usados para promover a reutilização de código. Um nulo do tipo int é o mesmo que um int comum mais um sinalizador que diz se o int tem um valor ou não (é nulo ou não). Todo o resto é trabalho do compilador que trata "nulo" como um valor válido.

O .NET Framework possui várias implementações genéricas de classes de coleção no namespace System.Collections.Generic. Sempre que possível, essas coleções genéricas devem ser usado em favor de suas contrapartes não genéricas.

C# tem muitas possibilidades ao usar genéricos. Eles podem ser usados em estruturas, classes, interfaces, métodos, propriedades e delegados. Você pode até especificar vários parâmetros tipos genéricos quando necessário.

Como você pode ver no exemplo de Nullable <T>, um parâmetro de tipo genérico também pode ser restringido.

No caso de Nullable<T>, não faria sentido se T pudesse ser um tipo de referência. Os tipos de referência por natureza já têm a opção de serem nulos.

Ao trabalhar com um tipo de referência, o valor padrão é nulo; com um tipo de valor como int, é 0. Mas qual é o valor padrão ao trabalhar com um parâmetro de tipo genérico? Nesse caso, você pode usar a palavra-chave padrão especial (T). Isso fornece o valor padrão para o tipo específico de T.

publicvoid MyGenericMethod<T>()

{

T defaultValue = default(T);

}

Compreendendo genérico versus não genérico

A maioria dos tipos de coleção possui uma versão genérica e uma não genérica. Ao trabalhar com objetos de um tipo específico (ou tipo base), use a coleção genérica. Isso melhorará a segurança e o desempenho do tipo, porque não há necessidade de casting.

As coleções não genéricas podem ser encontradas em System.Collections e coleções genéricas podem ser encontradas em System.Collections.Generic. Se você usar um tipo de valor como parâmetro de tipo para uma coleção genérica, precisará eliminar todos os cenários nos quais o boxe pode ocorrer. Por exemplo, se seu tipo de valor não implementar IEquatable <T>, seu objeto precisará de um box para chamar Object.Equals (Object) para verificar a igualdade. O mesmo vale para a interface IComparable <T>. Ao usar tipos de referência, você não terá esses problemas.

classProgram

{

staticvoid Main(string[] args)

{

// AO ADICIONAR A STRING "HELLO"

// EM UM ARRAYLIST DE NUMEROS NÃO DÁ ERRO NA COMPILAÇÃO

// POIS ESTÁ SENDO ADICIONADO COMO OBJETO

// MAS DA ERRO NA EXECUÇÃO POIS NÃO CONSEGUE REALIZAR O CAST DO TIPO(UNBOXING)

System.Collections.ArrayList list1 = new System.Collections.ArrayList();

list1.Add(3);

list1.Add(5);

list1.Add("Hello");

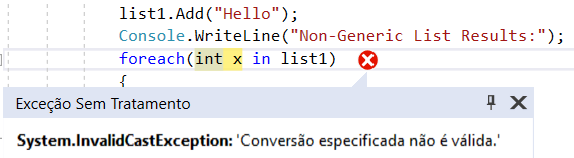
Console.WriteLine("Non-Generic List Results:");

foreach(int x in list1)

{

Console.WriteLine(x);

}



// COM UMA LISTA GENERICS, COMO O TIPO É DEFINIDO PREVIAMENTE(int)

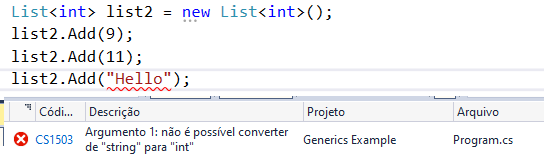
// JÁ NA COMPILAÇÃO DO CÓDIGO, APARECEM OS ERROS DE CONVERSÃO

List<int> list2 = new List<int>();

list2.Add(9);

list2.Add(11);

list2.Add("Hello");

}

}

O .NET fornece várias classes e interfaces genéricas. Isso existe no namespace System.Collections.Generic. Alguns estão listados abaixo:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| HashSet<T> | List<T> | Stack<T> | IComparer<T> | IEnumerator<T> |
| LinkedList<T> | Queue<T> | ICollection<T> | IEnumerable<T> | ILIst<T> |

publicclassNodeList<T>

{

private List<T> nodes;

public NodeList()

{

this.nodes = new List<T>();

}

publicvoid AddNode(T newNode)

{

nodes.Add(newNode);

}

publicvoid DeleteNode(T nodeToRemove)

{

nodes.Remove(nodeToRemove);

}

publicvoid ProcessAllNodes()

{

foreach (var node in nodes)

{

Console.WriteLine(node.ToString());

}

}

}

classProgram

{

staticvoid Main(string[] args)

{

NodeList<int> nodesOfInt = new NodeList<int>();

nodesOfInt.AddNode(2);

nodesOfInt.AddNode(4);

nodesOfInt.AddNode(6);

nodesOfInt.AddNode(7);

nodesOfInt.DeleteNode(7);

nodesOfInt.ProcessAllNodes(); //2 4 6

NodeList<string> nodesOfString = new NodeList<string>();

nodesOfString.AddNode("Lory");

nodesOfString.AddNode("Julia");

nodesOfString.AddNode("Lerman");

nodesOfString.AddNode("James");

nodesOfString.ProcessAllNodes(); //Lory Julia Julia James

Console.ReadKey();

}

}

Generic Type/Class

A definição de tipos genéricos é feita através do uso de um parâmetro de tipo genérico entre colchetes angulares, <T>.T é apenas a representação padrão para tipos genéricos usada na maioria da documentação relativa aos genéricos. Você pode usar a letra de sua própria escolha. Um exemplo de uma classe genérica a seguir:

// We use <> to specify Parameter type

publicclassGFG<T>

{

// private data members

private T data;

// using properties

public T value

{

// using accessors

get

{

returnthis.data;

}

set

{

this.data = value;

}

}

// Generics method

publicvoid Display<TypeOfValue>(string msg, TypeOfValue value)

{

Console.WriteLine("{0}:{1}", msg, value);

}

}

classProgram

{

staticvoid Main(string[] args)

{

// instance of string type

GFG<string> name = new GFG<string>();

name.value = "GeeksforGeeks";

// instance of float type

GFG<float> version = new GFG<float>();

version.value = 5.0F;

Console.WriteLine(name.value); //GeeksforGeeks

Console.WriteLine(version.value); //5

// calling Generics method

name.Display<int>("Integer", 122); //Integer: 122

name.Display<char>("Character", 'H'); //Character: H

version.Display<double>("Decimal", 255.67); // Decimal:255,67

Console.ReadKey();

}

}

Embora uma classe seja demonstrada aqui chamada GFG<T>, você deve examinar o namespace System.Collection.Generic para determinar se o .NET Framework já contém uma classe genérica. Sempre reutilize o código existente sempre que possível. Seus tipos genéricos agem exatamente como outros tipos de referência e podem incluir construtores, variáveis de membro e métodos. Os métodos, incluindo o construtor, também podem incluir parâmetros de tipo.

Métodos Generic

Métodos genéricos também serão declarados com parâmetros de tipo. Isso significa que, como a assinatura da classe, a assinatura do método usará um espaço reservado para o tipo que será passado para o método. Da mesma maneira que as classes genéricas são seguras para o tipo e não exigem boxe/unboxing ou conversão explícita, os métodos genéricos também compartilham essa mesma característica. Um dos exemplos mais simples que existem no MSDN e em vários outros exemplos de documentação está usando um método de troca (Swap). A troca é uma função comumente usada em algoritmos de ordenação simples. Um exemplo de método de troca genérico a seguir:

// example of generic method with type parameters

public staticvoid Swap<T>(ref T valueOne, ref T valueTwo)

{

T temp = valueOne;

valueOne = valueTwo;

valueTwo = temp;

}

int[] arrInts = newint[] { 2, 5, 4, 7, 6, 7, 1, 3, 9, 8 };

// Sorting: integer Sort

for (int i = 0; i < arrInts.Length; i++)

{

for (int j = i + 1; j < arrInts.Length; j++)

{

if (arrInts[i] > arrInts[j])

{

Swap<int>(ref arrInts[i], ref arrInts[j]);

}

}

}

Na assinatura do método, você ainda usa modificadores de acesso e tipos de retorno. No exemplo acima, o método é público e o tipo de retorno é nulo. Em seguida, vem o nome do método Swap que, semelhante à classe genérica, usa colchetes angulares e um espaço reservado de tipo<T>. Os parâmetros usam a palavra-chave ref e o espaço reservado T significa que os argumentos serão do tipo T, com base no tipo usado no momento em que o método é chamado. Os argumentos podem ser tipos de valor ou tipos de referência. A palavra-chave ref indica que os argumentos estão sendo passados ​​por referência. O método atua sobre os valores reais passados ​​através da referência ao endereço da memória para os argumentos. A seguir, alguns exemplos de métodos que utilizam Multiparâmetros onde os argumentos são passados como valor e não referência:

publicvoid MultipleGenericMethodArgs<T, U>(T first, U second)

{

Console.WriteLine("{0}: {1}", first, second);

}

public U ReturnFromMultipleGenericMethodArgs<T, U>(T first)

{

U temp = default(U);

return temp;

}

methods.MultipleGenericMethodArgs<string, int>("Exam", 70483); //Exam: 70483

int FromMultiple = methods.ReturnFromMultipleGenericMethodArgs<string, int>("Exam: ");

Console.WriteLine(FromMultiple); //0

Restrições Generic

O C# permite adicionar uma cláusula where simples. 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 básicas ou exigir que um tipo genérico seja um tipo de referência, valor ou não gerenciado. Eles declaram capacidades que o argumento do tipo deve ter.

publicvoid MyGenericMethod<T>(T parameter) { /\*....\*/ }

publicvoid MyGenericMethod2<T>(T parameter) where T : class { /\*....\*/ }

No primeiro, você pode chamá-lo com um tipo não ref, por exemplo. MyGenericMethod<int> (10). No segundo método, o T só pode aceitar tipo de referência(classe) e não pode ser um tipo de estrutura. No momento da instanciação da classe genérica, se o cliente fornecer parâmetro de tipo inválido, a compilação causará um erro.

Na Tabela abaixo, você pode ver as diferentes restrições que pode usar. Eles diferem apenas no tempo de compilação, pois o compilador verifica se T é do tipo ref ou não.

|  |  |
| --- | --- |
| **Restrição** | **Descrição** |
| where T: struct | O argumento type deve ser um tipo de valor (apenas Nullable não é permitido). |
| where T : class | O argumento type deve ser um tipo de referência: por exemplo, uma classe, interface, delegado ou matriz. |
| where T : new() | O tipo deve ter um construtor público padrão, sem parâmetros acessível.A restrição new() não pode ser combinada às restrições struct ou unmanaged. |
| where T : <base class name> | O argumento type deve ser ou derivar da classe base especificada. |
| where T : <interface name> | O argumento type deve ser ou implementar a interface especificada. Múltiplas restrições de interface podem ser especificadas. A interface de restrição também pode ser genérica. |
| where T : U | O argumento de tipo fornecido para T deve ser ou derivar do argumento fornecido para U. |
| where T : notnull | A notnull restrição limita o parâmetro de tipo a tipos não anulados. Esse tipo pode ser um tipo de valor ou um tipo de referência não anulado. A notnull restrição está disponível a partir de C# 8.0 para código compilado em um contexto. Ao contrário de outras restrições, se um argumento de tipo violar a notnull restrição, o compilador gera um aviso em vez de um erro. Os avisos só nullable enable são gerados em um contexto. |
| where T : 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: |

É possível aplicar restrições a vários parâmetros e várias restrições a um único parâmetro, conforme mostrado no exemplo a seguir:

classBase { }

classTest<T, U>

where U : struct

where T : Base, new()

{ }

Várias restrições podem ser aplicadas ao mesmo parâmetro de tipo e as restrições em si podem ser tipos genéricos, da seguinte maneira:

classEmployeeList<T>where T : Employee, IEmployee, System.IComparable<T>, new()

{

// ...

}

As restrição podem ser aplicadas também em métodos genéricos:

publicstaticvoid OpEqualsTest<T>(T s, T t) where T : class

{

System.Console.WriteLine(s == t);

}

string s1 = "target";

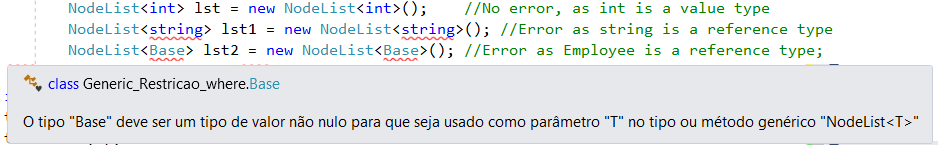
System.Text.StringBuilder sb = new System.Text.StringBuilder("target");

string s2 = sb.ToString();

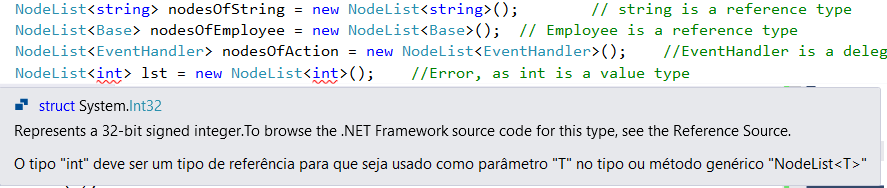
OpEqualsTest<string>(s1, s2);

Abaixo, alguns exemplos de uso de restrições de classes genéricas:

publicclassNodeList<T>where T : struct



publicclassNodeList<T>where T : class



publicclassEmployee

{

public Employee() {}

}

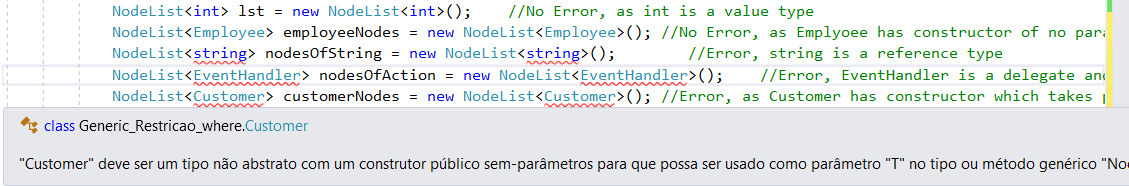
publicclassCustomer

{

public Customer(string customerName) {}

}

publicclassNodeList<T>where T : new()



publicclassBase { }

publicclassEmployee : Base

{

public Employee() {}

}

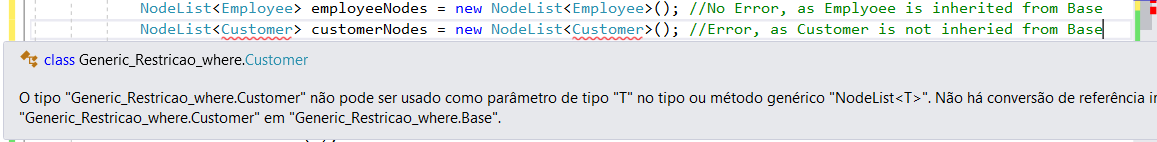
publicclassCustomer

{

public Customer(string customerName) {}

}

publicclassNodeList<T>where T : Base



publicinterfaceIEmployee{}

publicclassEmployee : IEmployee

{

public Employee() {}

}

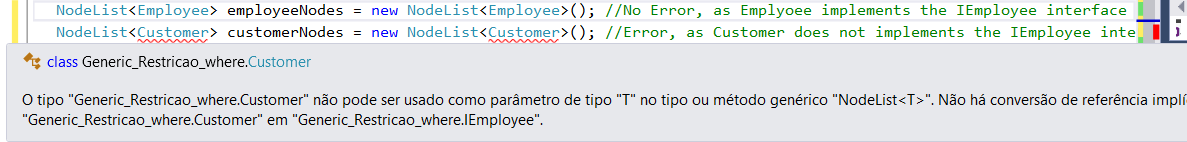
publicclassCustomer

{

public Customer(string customerName) {}

}

publicclassNodeList<T>where T : IEmployee



publicclassEmployee : IEmployee

{

public Employee() { }

}

publicclassCustomer : Base

{

public Customer(string customerName) { }

}

publicclassNodeListMult<T, U>where T : U

{

publicvoid DoWork(T subClass, U baseClass)

{

}

}

NodeListMult<Employee, IEmployee> employeeNodesmult = new NodeListMult<Employee, IEmployee>();

NodeListMult<Customer, Base> employeeNodesbase = new NodeListMult<Customer, Base>();

NodeListMult<Customer, Customer> employeeNodescust = new NodeListMult<Customer, Customer>();

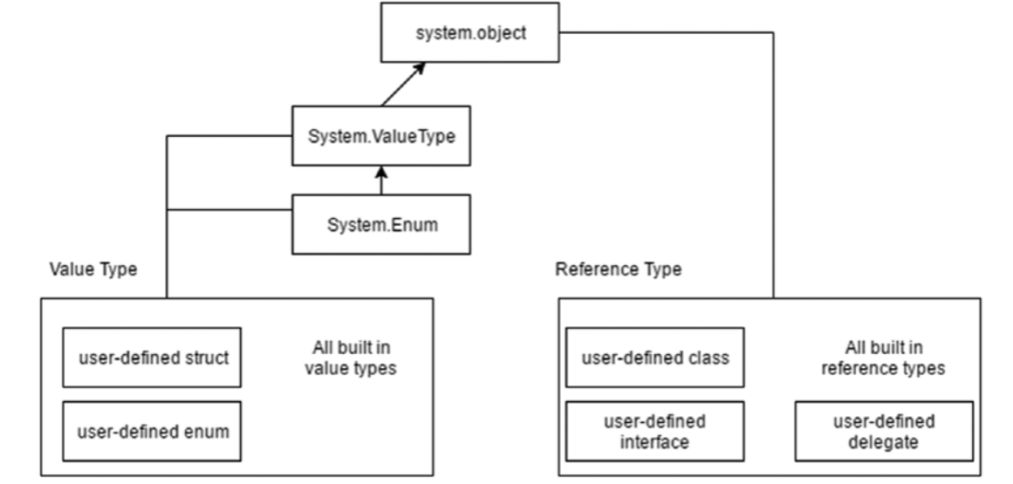
Tipos especiais em c #

O C # fornece tipos especiais que são açúcar sintático para os usuários. Esses tipos ajudam os usuários a maximizar suaprodutividade escrevendo código auxiliar dentro deles. Esses tipos especiais estão listados abaixo.

* System.Object type
* Anonymous type
* Dynamic type
* Nullable type
* Static type

Tipo System.Object

Todos os tipos de valor e referência são derivados do tipo system.object. No .NET, o objeto é a base de toda a hierarquia de tipos. A figura abaixo mostra a hierarquia de tipos de system.object.



Como todos os tipos no .NET são derivados de system.object, ele pode manipular valores de todos os tipos. Por exemplo, no seguinte snippet de código, o objeto armazenou os valores de string e int.

object name = "Ali";

object age = 22;

Todo tipo no .NET herda métodos do tipo system.object, que eles podem substituir. Esses métodos são:

* Equals: use para comparar dois objetos.
* Finalize: use para executar operações de limpeza antes que o objeto seja destruído.
* GetHashCode: use para obter o valor do hash do objeto no HashTable.
* ToString: use para obter as informações do objeto em forma de texto.

Tipo anônimo

Tipos que não têm nomes são chamados tipos anônimos. Eles são usados para agrupar dados em propriedade read-only. Diferentemente da classe, os tipos anônimos não têm um blueprint para definir os tipos de propriedade. Portanto, cada propriedade deve ter um dado para determinar seu tipo de propriedade.

Tipos anônimos são criados usando um novo operador com inicializador de objetos. A variável de tipo implícito var é usado para manter a referência de tipos anônimos.

Sintaxe: var variableName = new {/ \* inicializador de objetos \* /};

var person = new { Name = "Ali", Age = 22 };

Console.WriteLine("Name = {0}", person.Name);// Ali

Console.WriteLine("Age = {0}", person.Age);// 22

Observações

* No tipo anônimo, o valor da propriedade não pode ser nulo.
* Tipo anônimo não tem método definição.
* Tipos anônimos são um tipo de classe.
* Tipo anônimo não pode ser usado como parâmetro de método que retona um tipo
* O tipo anônimo é útil para armazenar um resultado de consulta em uma coleção.

Tipo Dinâmico

O tipo dinâmico é usado para armazenar e manipular quaisquer dados cuja definição de tipos e erros de operação sejam determinados em tempo de execução. Ignora verificações em tempo de compilação. Portanto, é fácil acessar APIs COM e DOM com tipo dinâmico. É definido usando uma palavra-chave dinâmica.

dynamic price = 20;

Console.WriteLine(price.GetType()); // System.Int32

dynamic name = "Ali";

Console.WriteLine(name.GetType()); // System.String

Diferentemente do tipo implícito (palavra-chave var), o tipo dinâmico pode armazenar valores de tipos diferentes com a mesma variável dinâmica. Portanto, uma variável dinâmica pode alterar seu tipo em tempo de execução.

dynamic tipo\_dinamico = "Ali";

Console.WriteLine(tipo\_dinamico.GetType()); // System.String

tipo\_dinamico = 3.4m;

Console.WriteLine(tipo\_dinamico.GetType()); // System.Decimal

tipo\_dinamico = newdouble[10];

Console.WriteLine(tipo\_dinamico.GetType()); // System.Double[]

Observações

* O tipo dinâmico pode ser usado para campo, propriedade, indexador, parâmetro de método e tipo de retorno.
* Exceção é lançado no tempo de execução se o tipo de dados ou o nome do membro não for compatível

Tipo Nullable

Normalmente, o tipo de valor não pode ser atribuído com valor nulo. Mas com o tipo Nullable, o tipo de valor pode ser atribuído com valor nulo. O tipo de valor pode se tornar anulável usando "?".

bool? isMarried = null;

isMarried = true;

Já o operador null-coalaescing “??” possibilita definir um valor padrão caso a variável seja nula, permitindo assim a atribuição de valor a uma variável de mesmo tipo não-anulável.

Nullable<int> idade = null;

int Age = idade ?? 3;

O HasValue retornará true se o objeto tiver sido atribuído a um valor; se não tiver sido atribuído nenhum valor ou tiver sido atribuído um valor nulo, ele retornará false.

if (i.HasValue) Console.WriteLine(i.Value); // or Console.WriteLine(i)

else Console.WriteLine("Null");

Nulo é considerado menor que qualquer valor. Portanto, os operadores de comparação não funcionarão com nulo. Considere o seguinte exemplo em que i não é menor que j, maior que j nem igual a j:

int? i = null;

int j = 10;

if (i < j) Console.WriteLine("i < j");

else Console.WriteLine("Could not compare");

A classe estática Nullable fornece um método de comparação para comparar tipos anuláveis.

if (Nullable.Compare<int>(i, j) < 0) Console.WriteLine("i < j");

elseif (Nullable.Compare<int>(i, j) > 0) Console.WriteLine("i > j");

else Console.WriteLine("i = j");

Ele também possui um método GetUnderlyingType que retorna o argumento de tipo subjacente de tipos anuláveis.

publicclassExample

{

publicint? MyMethod()

{

return 0;

}

}

Type t = typeof(Example);

MethodInfo mi = t.GetMethod("MyMethod");

Type retval = mi.ReturnType;

//Return value type...System.Nullable`1[System.Int32]

Console.WriteLine("Return value type ... {0}", retval);

Type answer = Nullable.GetUnderlyingType(retval);

//Underlying type ..... System.Int32

Console.WriteLine("Underlying type ..... {0}", answer);

Pontos importantes

* O tipo <T> anulável permite a atribuição de nulo a tipos de valor.
* ? operador é uma sintaxe abreviada para tipos Nullable.
* Use a propriedade value para obter o valor do tipo anulável.
* Use a propriedade HasValue para verificar se o valor está atribuído ao tipo anulável ou não.
* A classe Nullable estática é uma classe auxiliar para comparar tipos anuláveis.
* O tipo de valor é boxed sempre que se tornar anulável.

Tipo Estático

Diferentemente das classes normais, as classes estáticas não pode ser instanciado. Elas são úteis para definir métodos estáticos auxiliares. A classe estática não possui nenhum membro da instância. Todos membros devem ser estáticos na classe estática. Portanto, membros de classes estáticas podem acesso usando o próprio nome da classe. A classe estática é definida escrevendo uma palavra-chave estática antes da definição da classe.

Diferente da classe normal, a classe estática não contém um construtor de instância pública. Ele contém um construtor estático privado para inicializar membros estáticos. É chamado automaticamente antes da criação da primeira instância ou de qualquer membros estáticos são referenciados.

staticclassHelper

{

publicstaticint age;

static Helper()

{

age = 22;

}

}

Console.WriteLine(Helper.age); // 22

Sumário

* Os valores da constante enum podem ser sobreescritos (override)com valor inteiro.
* Construtor Struct deve inicializar todos membros de dados.
* O valor padrão do tipo de referência é sempre Null.
* Tipos definidos com struct e enum são exemplos de tipos de valor.
* Tipos definidos com classe, interface e delegados são exemplos de tipo de referência.
* O tipo System.Object é classe base de todos os tipos na hierarquia C #.
* Tipos anônimos devem ter um ou mais propriedades read only.
* Os tipos dinâmicos são úteis para interagir com COM, DOM e APIs dinâmicas.
* Tipo de valor pode armazenar nulo quando é declarado como nullable "?".
* Tipos estáticos não podem ser instanciados.
* Tipos estáticos têm apenas membros estáticos.

Consumir tipos

* Box ou unbox para converter entre tipos de valor; tipos cast; converter tipos; lidar com tipos dinâmicos; garantir a interoperabilidade com código que acesse APIs COM

BOXING E UNBOXING

O .NET Framework possui duas grandes divisões em seus tipos de dados, os Value Types (Tipos por Valor) e os Reference Types (Tipos por Referência). Os Value Types são armazenados na stack e os Reference Types são armazenados na heap.

C # introduz dois métodos para Boxing e Unboxing, que vincula o tipo de valor ao tipo de referência. A diferença básica entre Boxing e Unboxing é que Boxing é a conversão do tipo de valor para um tipo de objeto enquanto, por outro lado, o termo Unboxing se refere à conversão do tipo de objeto para o tipo de valor. Vamos estudar as outras diferenças entre o Boxing e o Unboxing.

Por exemplo, a biblioteca .NET contém um método string.Concat que adiciona o argumentos que recebe juntos e os retorna como uma sequência. Suponha que você chame string.Concat com os seguintes argumentos:

string.Concat (“Para encaixar ou não encaixar”, 42, verdadeiro);

Se você colocar esse código no Visual Studio e passar o mouse sobre a função Concat, poderá veja que são necessários três argumentos, todos do tipo objeto. Você pode perceber que está lidando com uma string, um int e um valor booleano. Isto é onde o boxe entra. O exemplo abaixo mostra um boxing, coloca um int dentro de um objeto, e o recupera novamente.

int i = 42;

object o = i;

int x = (int)o;

Boxing

Boxing é um procedimento de conversão de um tipo de valor para um tipo de objeto. Aqui, o tipo de valor é armazenado na pilha e o tipo de objeto é armazenado na memória heap. Essa conversão do tipo de valor para o tipo de objeto é uma conversão implícita. Você pode atribuir diretamente um valor a um objeto e o C # manipulará a conversão restante.

Boxing consome memória extra junto com o tempo extra. A razão é que um novo objeto, que se referirá ao tipo de valor, deve alocar o espaço de memória no heap. Em seguida, o valor do tipo de valor armazenado na pilha será transferido para o tipo de objeto, no local da memória de heap.

Unboxing

Unboxing é uma conversão do tipo de objeto para o tipo de valor. Em Unboxing, o valor do tipo de objeto em caixa armazenado no heap é transferido para o tipo de valor armazenado na pilha. Ao contrário do boxe, o Unboxing tem que ser feito explicitamente. O tipo de objeto é explicitamente convertido para o tipo de valor e o tipo de valor deve ser o mesmo valor ao qual o tipo de objeto está se referindo.

Unboxing também consome mais memória e mais tempo. Desde que, quando um tipo de objeto precisa ser descompactado, o valor do tipo de objeto armazenado no heap deve ser transferido para o novo tipo de valor armazenado na pilha. O tipo de objeto cujo valor foi recuperado estará agora disponível para coleta de lixo.

staticvoid Main(string[] args)

{

object o;

int x = 123;

//BOXING: CLASSE É CONVERTIDA EM OBJETO

//coloca um int dentro de um objeto

o = x; //Boxing Implicito

o = (object)x; //Boxing Explicito

Console.WriteLine("O objeto o = ", o); // O objeto o =

Console.WriteLine("x= " + x); // x = 123

//UNBOXING: OBJETO É CONVERTIDO EM CLASSE

x = (int)o;

Console.WriteLine("x= " + x); // x = 123

long y = 0; // Declarando Int64 double

//y = (Int64)o; // Unboxing Explicito para double, da erro de Cast

y = (Int64)(Int32)o; // Unboxing e depois convertendo pra double

Console.WriteLine("o= " + o); // o = 123

Console.WriteLine("y= " + x); // y = 123

Console.ReadKey();

}

|  |  |  |
| --- | --- | --- |
| **Base para Comparação** | **Boxe** | **Unboxing** |
| Basic | Tipo de objeto refere-se ao tipo de valor. | processo de recuperação de valor do objeto em caixa. |
| Armazenamento | O valor armazenado na pilha é copiado para o objeto armazenado na memória heap. | O valor do objeto armazenado na memória heap é copiado para o tipo de valor armazenado na pilha. |
| Conversão | Conversão implícita. | Conversão explícita. |
| Exemplo | int n = 24;  objeto ob = n; | int m = (int) ob; |

Se você executar uma operação unbox inválida, o tempo de execução lançará uma InvalidCastException, como ocorreu ao tentar um unboxing de int para long em y = (Int64)o . Você não verá o erro no momento da compilação porque o compilador confia em você ao fazer a chamada certa. No tempo de execução, no entanto, a conversão falha e uma exceção é lançada.

A única outra coisa importante a saber é que, quando o boxe e o unboxing acontecem (como mostrado no exemplo, o unboxing é claro), você precisa converter explicitamente seu objeto de uma referência para um tipo de valor. O boxe, por outro lado, não é tão óbvio. Por exemplo, chamar GetType sempre coloca seu tipo de valor em caixa, porque GetType é definido apenas em um objeto e não pode ser substituído. O boxe também ocorre em outras situações. Uma coisa que pode surpreender é que um tipo de valor está em caixa quando você o usa como interface. Este trecho de caixa contém o valor 3 para que você possa usá-lo como uma interface.

IFormattable x = 3;

Existem algumas implicações de desempenho em cada operação de caixa e unbox. Ao usar as coleções não genéricas para armazenar um tipo de valor, você tem muitas dessas operações. As operações de boxe e unboxing podem prejudicar o desempenho; no entanto, agora que você possui suporte genérico no .NET Framework, isso é um problema menor, pois você pode armazenar tipos de valor em uma coleção sem encaixotá-los.

CASTING E CONVERT

As variáveis ​​C # tem tipagem estática, o que significa que o seu tipo não pode ser mudado, garantindo que sempre está sendo usado um tipo que é esperado em todas as situações.

A verificação do tipo é feita no código fonte pelo processo de compilação em tempo de execução. Esta análise ajuda na segurança de tipos do código ajudando a evitar erros e permite que o programador se preocupe menos, já que os dados já têm seu tipo definido.

O compilador fornece garantias que alguns problemas não poderão ocorrer após o programa ser compilado, exemplo disso é se for declarado uma variável inteira ela não poderá receber um número fracionário, desta forma, erros são detectados antes do programa ser executado.

Casting

Porém, existem muitos casos em que uma variável de um tipo específico precisa armazenar dados de um tipo diferente. Casting e Converting são os processos fornecidos para alterar o tipo de dados. Existem dois tipos principais de conversões de tipo em C#:

* **Implícito**: Essas conversões ocorrem automaticamente porque não há risco de perda de dados (do tipo maior para o menor). Por exemplo, uma variável maior do tipo long (8 bytes inteiros) pode armazenar qualquer valor que um int (4 bytes) possa armazenar

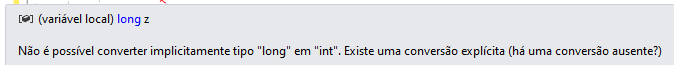
long z = 7;

int x = 7;

z = x;

//MAS SE...

x = z;



Outra conversão implícita é a de um tipo de referência para um de seus tipos básicos. Por exemplo, cada tipo de referência pode ser armazenado dentro de um objeto porque, em última análise, cada tipo de referência herda de um objeto. Se um objeto implementa uma interface, ele também pode ser convertido implicitamente na interface. A Listagem 2-21 mostra a conversão implícita de um objeto em um de seus tipos básicos.

HttpClient client = new HttpClient();

object o = client;

IDisposable dc = client;

* **Explícito**: Se a conversão não puder ser feita sem o risco de perda de informações, é necessária uma conversão explícita.

int x=7;

double y=12.6;

x = (int)y;

**Operador “as”** é uma palavra-chave usada para explicitamente converter um tipo para outro. Se um tipo for convertido com êxito, ele retornará valor nesse tipo. Se um tipo não se converter adequadamente, ele retornará valor nulo.

publicclassPerson { }

publicclassEmployee : Person { }

publicclassManager : Person { }

Person person = new Employee();

//convert explicitly from type A to B

Employee employe = person as Employee;

As mesmas regras para converter dados em um tipo para outro tipo compatível também são válidas para Arrays. Até os operadores "is" e "as" funcionam para Arrays.

// Implicit cast to an array of Persons

Employee[] employes = new Employee[10];

Person[] persons = employes;

// Explicit cast back to an array of Employees

employes = (Employee[])persons;

employes = persons as Employee[];

// After this as statement, managers is null.

Manager[] managers = persons as Manager[];

**Operador “is”** é uma palavra-chave usada para corresponder a um tipo. Se um tipo corresponde, ele retorna true; caso contrário, ele retornará false. Um uso comum deste operador vem com a conversão de tipo explícita, evitar essa exceção de convesrão de tipo.

if (persons is Employee[])

{

// Treat them as Employees. ...

}

Convertendo valores

A conversão permite que um programa converta um valor de um tipo para outro tipo compatível, mas às vezes você pode converter um valor de um tipo para um tipo incompatível. Por exemplo, convém converter o valor da sequência 10 para o valor int 10 ou converter o valor da sequência True no valor booleano true. Em casos como esses, a conversão não funcionará. Para converter um valor de um tipo para um tipo incompatível, você deve usar algum tipo de classe auxiliar. O .NET Framework fornece três métodos principais para esses tipos de conversões:

* **Métodos Parsing**
* **System.Convert**
* **System.BitConverter**

Para conversão entre tipos compatíveis, você pode usar o System.Convert e os métodos Parse ou TryParse em vários tipos.Para converterentre tipos não compatíveis, você pode usar System.BitConverter.

Métodos Parsing (veja mais detalhes no arquivo 4-Depurar)

Cada um dos tipos de dados C # primitivos (int, bool, doubl, etc) possui um método Parse que converte uma representação de seqüência de caracteres de um valor nesse tipo de dados. Por exemplo, bool.Parse usa como argumento uma string que representa um valor booleano como true e retorna o valor correspondentevalor bool true.Estes métodos Parse lançam exceções se suas entradas estiverem em um formato não reconhecido. Por exemplo, a instrução bool.Parse("yes") lança uma FormatException porque esse método entende apenas os valores true e false.

Você pode usar um bloco try-catch para proteger o programa dessas exceções, mas para tornar a verificação de valor ainda mais fácil, cada uma dessas classes também fornece um método TryParse. Este método tenta analisar uma string e retorna true se for bem-sucedido oufalse se falhar. Se for bem-sucedido, o método também salvará o valor analisado em uma variável de saída que você passa para o método.

int value = Convert.ToInt32(“42”);

value = int.Parse(“42”);

bool success = int.TryParse(“42”, out value);

A lista abaixo contém os tipos de dados mais comuns que fornecem os métodos Parse e TryParse.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bool | Float | Uint | Datetime | Sbyte |
| Char | Long | Ushort | Double | Timespan |
| decimal | short | byte | Int | ulong |

Geralmente, os métodos de análise funcionam razoavelmente bem se a entrada deles faz sentido. Por exemplo, a declaraçãoint.Parse ("645") retorna o valor 645 sem confusão. Mesmo o método Parse do tipo de dados DateTime pode fazer sentido com as entradas mais razoáveis. Por exemplo, no inglês dos EUA, as seguintes declarações são todas convertidas para 15h45 de 1º de abril de 2014.

DateTime.Parse("3:45 PM April 1, 2014").ToString()

DateTime.Parse("1 apr 2014 15:45").ToString()

DateTime.Parse("15:45 4/1/14").ToString()

DateTime.Parse("3:45pm 4.1.14").ToString()

Por padrão, no entanto, os métodos de análise não tratam bem os valores da moeda. Por exemplo, o código a seguir lança uma FormatException (na localidade em inglês dos EUA):

valor decimal = decimal.Parse ("$123.456,78");

A razão pela qual esse código falha é que, por padrão, o método decimal.Parse permite milhares e separadores decimais, mas não símbolos de moeda. Você podeativar símbolos de moeda no decimal. Parse, adicionando parâmetros que seja uma combinação de valores definidos pelo método Enumeração System.Globalization.NumberStyles.

Se você fornecer quaisquer valores NumberStyles, quaisquer valores padrão serão removidos. Por exemplo, por padrão decimal.Parse permite milhares e separadores decimais. Se você passar o valor NumberStyles.AllowCurrencySymbol para o método, ele não permitirá mais milhares e separadores decimais. Para permitir todos os três, você precisa passar o método para todos os três valores, como no código a seguir:

decimal amount = decimal.Parse("$123,456.78", NumberStyles.AllowCurrencySymbol | NumberStyles.AllowThousands | NumberStyles.AllowDecimalPoint);

Como alternativa, você pode transmitir ao método o estilo composto Currency, conforme mostrado no código a seguir:

decimal amount = decimal.Parse("$123,456.78", NumberStyles.AllowCurrencySymbol);

System.Convert(veja mais detalhes no arquivo 4-Depurar)

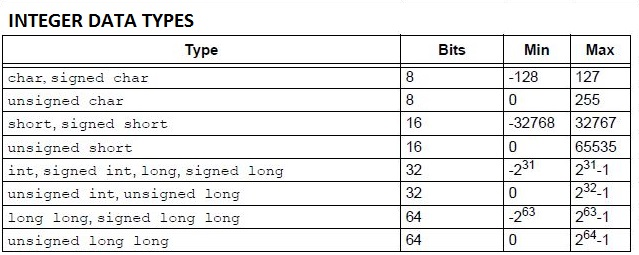
A classe System.Convert fornece mais de 300 métodos (incluindo versões sobrecarregadas) para converter um tipo de dados em outro. Por exemplo, o método ToInt32 converte um valor em um número inteiro de 32 bits (um int). Diferentes versões sobrecarregadas dos métodos usam parâmetros de diferentes tipos, comobools, bytes, DateTimes, duplas, seqüências de caracteres e assim por diante

* Convert fornece funcionalidade semelhante, mas fica um pouco mais complexo
* Convert éótimo ao trabalhar com strings
* Se Casting falhar, tente Converter

A Tabela abaixo lista os métodos de conversão mais úteis fornecidos pela classe System.Convert.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ToBoolean | ToInt16 | ToUInt16 | ToDateTime | ToSByte |
| ToChar | ToInt64 | ToUInt64 | ToDouble | ToString |
| ToDecimal | ToSingle | ToByte | ToInt32 | ToUInt32 |

Básicamente não só no char como em int, float entre outros, temos os tipos signed e o unsigned (há outros além desses), a diferença é que o bit mais significativo que faz a alteração de valores positivos para negativos é ignorado e utilizado como espaço, sendo assim é liberado mas espaço, já em contrapartida os números negativos não são possíveis. Veja a tabela a seguir, todos os unsigneds começam com 0. Vale ressaltar que por padrão é utilizado o signed portanto você declarar



long z = 7;

int x = 7;

double y = 12.6;

double yy = 12.6;

//CAST: IGNORA AS CASAS DECIMAIS(.6)

//RESULTADO: 12

x = (int)y;

//CONVERT: ARREDONDA PARA CIMA/BAIXO O VALOR

//RESULTADO: 13

xx = Convert.ToInt32(yy);

Conversões definidas pelo usuário

O C # permite que os usuários forneçam a definição de conversão para seu tipo personalizado. Sua definição é semelhante à definição de sobrecarga do operador. A conversão definida pelo usuário é de dois tipos:

* **Implícita**: A conversão implícita ocorre automaticamente, um método estático especial é definido com uma palavra-chave implícita e de operador dentro da definição de tipo.

classByte

{

publicint

bits = 8;

publicstaticimplicitoperatorint(Byte b)

{

return b.bits;

}

}

// Conversão Inplicita do usuario

Byte b = new Byte();

int totalBits = b;

Console.WriteLine(totalBits); //8

Onde:

* + O operator int informa que tipo de dados seria retornado na conversão implícita.
  + O tipo Byte b informa qual tipo seria convertido implicitamente.
  + O byte "b" obtem uma conversão implícita em "int" retornando o número total de bits em um byte
* Explícita: Casting é necessário para converter um tipo em outro, um método estático especial é definido com uma palavra-chave explícita e operator dentro da definição de tipo. Pode haver perda de dados na conversão explícita.

classPerson

{

publicint Age { get; set; }

publicstring Name { get; set; }

publicstaticexplicitoperatorstring(Person per)

{

return per.Name;

}

}

// Conversão Explicita do usuario

Person per = new Person { Age = 22, Name = "Ali" };

string name = (string)per;

Console.WriteLine(name); //Ali

Onde:

* + O operator string informa que tipo de dados seria retornado na conversão implícita.
  + O Person per informa qual tipo seria convertido implicitamente
  + Com o Casting (string)per; lança os dados de uma pessoa em "string" explicitamente, retornando o nome da pessoa ou mais dados.

Suponha que você esteja trabalhando em uma classe Money que encapsule todos os tipos de algoritmos de arredondamento para trabalhar com moedas diferentes.

classMoney

{

public Money(decimal amount)

{

Amount = amount;

}

publicdecimal Amount { get; set; }

publicstaticimplicitoperatordecimal(Money money)

{

return money.Amount;

}

publicstaticexplicitoperatorint(Money money)

{

return (int)money.Amount;

}

}

Money m = new Money(42.42M);

decimal amount = m;

int truncatedAmount = (int)m;

Console.WriteLine(amount); //42,42

Console.WriteLine(truncatedAmount); //42

Adicionar esses tipos de conversão pode realmente melhorar a usabilidade do seu tipo. Como você puder veja, o operador implícito e explícito deve ser declarado como um método estático público em seu classe. Você precisa especificar o tipo de retorno (o tipo para o qual está fazendo a conversão) e o tipo para o qual estátransmissão de (uma instância da sua classe).

System.BitConverter

A classe System.BitConverter fornece métodos para converter valores de e para matrizes de bytes. O método GetBytes retorna uma matriz de bytes que representa o valor que você passa para ele. Por exemplo, se você passar um int (que ocupa 4 bytes de memória) no método, ele retornará uma matriz de 4 bytes que representa o valor.A classe System.BitConverter também fornece métodos como ToInt32 e ToSingle para converter byte os valores armazenados nas matrizes retornam a tipos de dados específicos. Por exemplo, suponha que uma função API retorne dois valores de 16 bits compactados nas metades esquerda e direita de um número inteiro de 32 bits. Um inteiro assinado de 16 bits, que pode conter um valor entre -32.768 e 32.767. Você pode usar o seguinte código para descompactar os dois valores:

int packedValue;

// The API function call sets packedValue here....

packedValue = 47483648;

// Convert the packed value into an array of bytes.

byte[] valueBytes = BitConverter.GetBytes(packedValue);

Console.WriteLine(BitConverter.ToString(valueBytes));

if (BitConverter.IsLittleEndian)

Array.Reverse(valueBytes);

// Unpack the two values.

short firstHalf = BitConverter.ToInt16(valueBytes, 0); // 20

short secondHalf = BitConverter.ToInt16(valueBytes, 2); // 0

Console.WriteLine(firstHalf); // -29952

Console.WriteLine(secondHalf); // 724

byte[] recbytes = newbyte[4];

recbytes[0] = BitConverter.GetBytes(firstHalf)[0];

recbytes[1] = BitConverter.GetBytes(firstHalf)[1];

recbytes[2] = BitConverter.GetBytes(secondHalf)[0];

recbytes[3] = BitConverter.GetBytes(secondHalf)[1];

int reconstituted = BitConverter.ToInt32(recbytes, 0);

Console.WriteLine(reconstituted); // 47483648

Depois que a função API define o valor de packedValue, o código usa o BitConverter.GetBytes para converter o valor em uma matriz de 4 bytes. A ordem dos bytes depende se a arquitetura do computador é big-endian ou little-endian. (Você pode usar o campo IsLittleEndian do BitConverter para determinar se o valor é big endian ou little endian.)

Em sistemas pequenos endian, os bytes de ordem inferior precedem bytes de ordem superior. Em um sistema big endian, os bytes de ordem superior precedem os bytes de ordem inferior. A tabela a seguir ilustra a diferença nas matrizes de bytes que resultam da passagem do inteiro 1.234.567.890 (0x499602D2) para o método GetBytes(Int32). Os bytes são listados na ordem do byte no índice 0 para o byte no índice 3.

|  |  |
| --- | --- |
| Little-endian | D2-02-96-49 |
| Big endian | 49-96-02-D2 |

A classe BitConverter inclui métodos estáticos para converter cada um dos tipos primitivos de e para uma matriz de bytes, como ilustra a tabela a seguir.

|  |  |  |
| --- | --- | --- |
| **Tipo** | **Para conversão de bytes** | **Conversão de byte** |
| [Boolean](https://docs.microsoft.com/pt-br/dotnet/api/system.boolean?view=netcore-3.1) | [GetBytes(Boolean)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_Boolean_) | [ToBoolean](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.toboolean?view=netcore-3.1) |
| [Char](https://docs.microsoft.com/pt-br/dotnet/api/system.char?view=netcore-3.1) | [GetBytes(Char)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_Char_) | [ToChar](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.tochar?view=netcore-3.1) |
| [Double](https://docs.microsoft.com/pt-br/dotnet/api/system.double?view=netcore-3.1) | [GetBytes(Double)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_Double_)- ou - [DoubleToInt64Bits(Double)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.doubletoint64bits?view=netcore-3.1#System_BitConverter_DoubleToInt64Bits_System_Double_) | [ToDouble](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.todouble?view=netcore-3.1)- ou - [Int64BitsToDouble](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.int64bitstodouble?view=netcore-3.1) |
| [Int16](https://docs.microsoft.com/pt-br/dotnet/api/system.int16?view=netcore-3.1) | [GetBytes(Int16)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_Int16_) | [ToInt16](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.toint16?view=netcore-3.1) |
| [Int32](https://docs.microsoft.com/pt-br/dotnet/api/system.int32?view=netcore-3.1) | [GetBytes(Int32)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_Int32_) | [ToInt32](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.toint32?view=netcore-3.1) |
| [Int64](https://docs.microsoft.com/pt-br/dotnet/api/system.int64?view=netcore-3.1) | [GetBytes(Int64)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_Int64_) | [ToInt64](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.toint64?view=netcore-3.1) |
| [Single](https://docs.microsoft.com/pt-br/dotnet/api/system.single?view=netcore-3.1) | [GetBytes(Single)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_Single_) | [ToSingle](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.tosingle?view=netcore-3.1) |
| [UInt16](https://docs.microsoft.com/pt-br/dotnet/api/system.uint16?view=netcore-3.1) | [GetBytes(UInt16)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_UInt16_) | [ToUInt16](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.touint16?view=netcore-3.1) |
| [UInt32](https://docs.microsoft.com/pt-br/dotnet/api/system.uint32?view=netcore-3.1) | [GetBytes(UInt32)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_UInt32_) | [ToUInt32](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.touint32?view=netcore-3.1) |
| [UInt64](https://docs.microsoft.com/pt-br/dotnet/api/system.uint64?view=netcore-3.1) | [GetBytes(UInt64)](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.getbytes?view=netcore-3.1#System_BitConverter_GetBytes_System_UInt64_) | [ToUInt64](https://docs.microsoft.com/pt-br/dotnet/api/system.bitconverter.touint64?view=netcore-3.1) |

Garantindo a interoperabilidade com código não gerenciado

A interoperabilidade permite que um programa C # use classes fornecidas por código não gerenciado que não foi gravado sob o controle do Common Language Runtime (CLR), o ambiente de tempo de execução que executa os programas C#. Os componentes ActiveX e a API do Win32 são exemplos de código não gerenciado que você pode chamar de um programa C#. As duas técnicas mais comuns para permitir que programas gerenciados usem código não gerenciado são:

* Chamada de plataforma (P/invoke)
* Tipo dinâmicos e Interoperabilidade COM

Plataforma P/invoke

Para usar P/invoke para acessar um recurso não gerenciado, como uma chamada de API, um programa inclui primeiro um atributo DllImport para definir os métodos não gerenciados que serão usados ​​pelo programa gerenciado. O atributo DllImport faz parte do namespace System.Runtime.InteropServices, e utiliza parâmetros que informam o programa gerenciado sobre um método não gerenciado. Os parâmetros indicam coisas como a DLL que contém o método, o conjunto de caracteres usado pelo método (Unicode ou ANSI) e o ponto de entrada na DLL usada pelo método. (Se você omitir isso, o padrão é o nome do método.)

O programa aplica o atributo a uma declaração de método externo estático. A declaração inclui quaisquer parâmetros que o método requeira e define o tipo de retorno do método. Essa declaração deve estar dentro de uma classe como a classe que contém o código que usa o método.Por exemplo, o seguinte fragmento de código mostra onde a instrução using e o atributo DllImport são colocados no programa.

using System;

using System.Runtime.InteropServices;

namespace Interoperabilidade

{

classProgram

{

staticvoid Main(string[] args)

{

// Get the long file name.

string longName = @"C:\Users\RICARDO\Documents";

// Allocate a buffer to hold the result.

char[] buffer = newchar[1024];

long length = GetShortPathName( longName, buffer, buffer.Length);

// Get the short name.

string shortName = newstring(buffer);

Console.WriteLine(shortName); // C:\Users\RICARDO\DOCUME~1(com espaços em branco)

shortName = shortName.Substring(0, (int)length);

Console.WriteLine(shortName);// C:\Users\RICARDO\DOCUME~1

Console.ReadKey();

}

[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]

staticexternuint GetShortPathName(string lpszLongPath, char[] lpszShortPath, int cchBuffer);

}

}

A instrução DllImport indica que o método está na biblioteca kernel32.dll, que o programa deve determinar automaticamente se deve usar o conjunto de caracteres Unicode ou ANSI e que o método deve chamar SetLastError se houver algum problema. Se houver um erro, o programa pode usar GetLastWin32Error para ver o que deu errado.

A declaração do método indica que o programa usará o método GetShortPathName, que converte um caminho completo em um arquivo em um caminho curto que pode ser reconhecido pelo Windows. (Se o método usa o conjunto de caracteres Unicode, o nome do método geralmente termina com um "W" para "caracteres largos", como em GetShortPathNameW.) Esse método retorna uma uint e usa como parâmetros uma string, matriz de caracteres e int.

O primeiro parâmetro é o caminho do arquivo que você deseja converter para um caminho curto. Quando você chama o método, o P/Invoke o converte automaticamente em uma sequência terminada em nulo. O segundo parâmetro deve ser um buffer pré-alocado, onde GetShortPathName pode armazenar seus resultados. O terceiro parâmetro fornece o comprimento do buffer que você alocou, para que GetShortPathName saiba quanto espaço ele tem para trabalhar. O método retorna um uint indicando o comprimento da string que o método depositou no buffer lpszLongPath.

Geralmente, o tipo de instrução DllImport mostrado anteriormente é bom o suficiente para fazer o trabalho. Se precisar de mais controle sobre como os valores são convertidos entre código gerenciado e não gerenciado, você pode adicionar o atributo MarshalAs aos parâmetros do método ou valor de retorno. O código a seguir mostra uma nova versão da instrução DllImport para o método GetShortPathName que usa os atributos MarshalAs:

[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]

staticexternuint GetShortPathName([MarshalAs(UnmanagedType.LPTStr)] string lpszLongPath,

[MarshalAs(UnmanagedType.LPTStr)] StringBuilder lpszShortPath, uint cchBuffer);

O primeiro atributo MarshalAs indica que o primeiro e segundo parâmetro é um tipo de dados LPTStr no código não gerenciado e deve ser tratado como uma string e StringBuilder, respectivamente, no código gerenciado.Claro, se você usar esta declaração, precisará alterar o código para usar um StringBuilder para um buffer em vez de uma matriz de caracteres.

var shortName = new StringBuilder();

long length = GetShortPathName(longName, shortName, 1024);

Console.WriteLine(shortName); // C:\Users\RICARDO\DOCUME~1

Tipos DinâmicoseInteroperabilidade COM

Os atributos DllImport e MarshalAs descritos na seção anterior permitem que você diga ao programa onde encontrar um método não gerenciado e que tipos de dados ele usa para parâmetros e um tipo de retorno. Isso permite que o programa chame métodos não gerenciados por meio da P/invoke.

Como mencionado anteriormente, o C# é uma linguagem digitada parcialmente estática. A palavra-chave dinâmica, adicionada no C# 4.0, é onde você entra no mundo das linguagens de tipo fraco. Trabalhar em um sistema de tipo fraco é útil ao se comunicar com recursos externos (como Interoperabilidade COM, Iron-Python, conjuntos de resultados JSON), ou o HTML Document Object Model (DOM) ou ao trabalhar com reflexão dentro de C #.

Quando o compilador C # encontra a palavra-chave dinâmica, ele pára com a verificação de tipo estaticamente (por exemplo, verificando se existe um método em um tipo ou se possui certos argumentos).Em vez disso, o compilador salva a intenção do código para que possa ser executado posteriormente em tempo de execução. É por isso que o uso de tipos dinâmicos não gera erros em tempo de compilação, embora certamente possa gerar erros de tempo de execução.

Ao integrar com aplicativos COM (Component Object Model), você usa um PIA (Primary Interop Assembly). Os PIAs são assemblies .NET que preenchem a lacuna entre .NET e COM. COM Interop fornece uma outra maneira pela qual um programa gerenciado pode interagir com código não gerenciado. Para usar a interoperabilidade COM, você precisa fornecer ao programa uma referência a uma biblioteca apropriada instalada no sistema ou através do Gerenciador pacotes do NuGet.

Adicionar a referência da biblioteca informa ao seu programa (e Visual Studio) muito sobre o aplicativo COM não gerenciado. Se você abrir o menu Exibir e selecionar Pesquisador de objetos, poderá usar o Pesquisador de objetos para pesquisar os objetos e tipos definidos pela biblioteca. A biblioteca fornece ao Visual Studio informações suficientes para fornecer ao IntelliSense sobre alguns membros da biblioteca, mas o Visual Studio ainda pode não entender tudo. os tipos usados ​​pela biblioteca.

O tipo de dados dinâmico é usado para resolver essa situação. Este é um tipo de dados estático, mas seu tipo verdadeiro não é avaliado até o tempo de execução, portanto não corrige erros de sintaxe para problemas como incompatibilidade de tipos, porque ainda não avaliou o tipo dinâmico. Isso pode ser útil se você não puder fornecer informações completas sobre o tipo de um item para o compilador. O C# considera os objetos definidos pelo código de interoperabilidade COM não gerenciado como tendo o tipo dinâmico, portanto, não importa em tempo de compilação quais são seus tipos reais . Ele ignora a verificação da sintaxe dos objetos e aguarda o tempo de execução para ver se o código faz sentido.

Por exemplo, o Microsoft Office possui os tipos de montagens que permitem a integração com Word, Excel e outros aplicativos do Office do seu aplicativo .NET. Antes da adição da palavra-chave dinâmica, fazer alguma automação do Office não era algo simples, mas com a nova palavra-chave dinâmica, acessar o Office é muito mais fácil. Essas montagens estão disponíveis como pacotes do NuGet e para instalar, basta clicar com o botão direito do mouse em References no seu projeto e selecionando Gerenciando pacotes do NuGet e procure por um dos pacotes listados abaixo:

* Microsoft.Office.Interop.Excel
* Microsoft.Office.Interop.Word
* Microsoft.Office.Interop.Outlook
* Microsoft.Office.Interop.PowerPoint
* Microsoft.Office.Interop.Graph

Ou instale usando o Console do Gerenciador de Pacotes:

PM> Install-Package Microsoft.Office.Interop.Excel

O código abaixo mostra como a palavra-chave dinâmica permite exportar alguns dados para o Excel.

classProgram

{

staticvoid Main(string[] args)

{

var entities = new List<dynamic> {

new{ ColumnA = 1, ColumnB = "Foo"},

new { ColumnA= 2, ColumnB= "Bar"}};

DisplayInExcel(entities);

}

staticvoid DisplayInExcel(IEnumerable<dynamic> entities)

{

var excelApp = new Microsoft.Office.Interop.Excel.Application();

excelApp.Visible = true;

excelApp.Workbooks.Add();

dynamic workSheet = excelApp.ActiveSheet;

workSheet.Cells[1, "A"] = "Header A";

workSheet.Cells[1, "B"] = "Header B";

var row = 1;

foreach (var entity in entities)

{

row++;

workSheet.Cells[row, "A"] = entity.ColumnA;

workSheet.Cells[row, "B"] = entity.ColumnB;

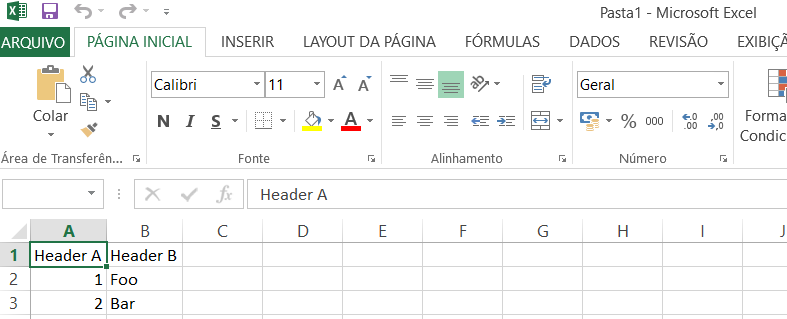
}

workSheet.Columns[1].AutoFit();

workSheet.Columns[2].AutoFit();

}

}



Neste exemplo, o tipo de planilha é dinâmico. As instruções que usam a variável de planilha são avaliadas em tempo de execução e despachadas para as interfaces de programação de aplicativos do Office (APIs). Toda a verificação de tipo e conversões necessárias ocorrem em tempo de execução.O Visual Studio não entende realmente o tipo de dados da sheet.Cells[1, 1], portanto, adia a verificação de tipo para esse valor. Isso permite que o programa se refira à propriedade Value dessa entidade, mesmo que o programa não saiba se a célula possui essa propriedade. Na verdade, você pode tentar definir sheet.Cells[1, 1].Whatever = ie o Visual Studio não se queixará até o tempo de execução, quando tentar acessar a propriedade Whatever e descobrir que não existe. Da mesma forma, o Visual Studio trata sheet.Columns[1] como tendo tipo dinâmico, portanto, não sabe que o método AutoFit existe até o tempo de execução.

Para um exemplo mais específico de C#, considere o código a seguir, demonstrado pelo programa de exemplo CloneArray:

// Make an array of numbers.

int[] array1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

//Error compilação: This doesn't work because array1.Clone is an object.

//int[] array2 = array1.Clone();

// This works.

int[] array3 = (int[])array1.Clone();

array3[5] = 55;

// This also works.

dynamic array4 = array1.Clone();

array4[6] = 66;

//Error execução: 'Não é possível converter implicitamente tipo 'string' em 'int''

array4[7] = "This won't work";

Este código inicializa uma matriz de números inteiros. O código comentado tenta usar o método Clone da matriz para fazer uma cópia da matriz. Infelizmente, o método Clone retorna um objeto não específico, portanto o código não pode salvá-lo em uma variável que se refere a uma Array de int.

A próxima instrução lança corretamente o objeto em um int[] para que ele funcione. Em seguida, o código armazena um novo valor inteiro na matriz. A declaração final tenta salvar uma string no array4 [7]. No tempo de design e compilação, o Visual Studio não tenta validar essa instrução porque o array4 foi declarado dinâmico. No entanto, em tempo de execução, isso falha porque o array4 é realmente um int[] e não pode conter uma string.

DynamicObject e ExpandoObject

O .NET Framework oferece duas classes especiais ao trabalhar com tipos dinâmicos:

1. **DynamicObject**: é o mais flexível. Ao herdar do DynamicObject, você pode substituir membros que permitem substituir operações como obter ou definir um membro, chamar um método ou realizar conversões. Usando DynamicObject, você pode criar objetos verdadeiramente dinâmicos e ter controle total sobre como eles operam em tempo de execução.

publicclassSampleObject : DynamicObject

{

publicoverridebool TryGetMember(GetMemberBinder binder, outobject result)

{

result = binder.Name;

returntrue;

}

}

dynamic obj = new SampleObject();

Console.WriteLine(obj.SomeProperty); // Displays ‘SomeProperty’

1. **ExpandoObject**: é uma implementação selada que permite obter e definir propriedades em um tipo. No ASP.NET Model-View-Controller (MVC), por exemplo, existe um ViewBag que pode ser usado para passar dados do Controller para o View. ViewBag é um ExpandoObject. Em vez de criar uma nova propriedade de tipo estaticamente para cada elemento de dados que você deseja transmitir, você pode usar o ViewBag, como mostra abaixo.

public ActionResult Index()

{

ViewBag.MyDynamicValue = “This property is not statically typed”;

return View();

}

A palavra-chave dinâmica deve ser usada com cuidado. Ele oferece uma grande flexibilidade, mas como você perde a digitação estática, também pode facilmente levar a erros que só podem ser encontrados em tempo de execução. Mas, ao integrar-se a outros idiomas ou para substituir a reflexão, o suporte dinâmico é uma boa adição ao .NET Framework.

Sumário

* O Boxing ocorre quando um tipo de valor é tratado como um tipo de referência.
* Ao converter entre tipos, você pode ter uma conversão implícita ou explícita.
* Uma conversão explícita é chamada de conversão e requer sintaxe especial.
* A conversão especial é necessária para a conversão explícita de tipos.
* Operador “as” é usado para converter um tipo em outro tipo.
* O usuário pode escrever sua definição para conversão de tipo usando palavras-chave implícitas e explícitas com métodos estáticos especiais
* O .NET Framework oferece vários métodos auxiliares para converter tipos.
* A palavra-chave dinâmica pode ser usada para facilitar a digitação estática de C# e melhorar a interoperabilidade com outras linguagens.

Aplicar encapsulamento

* Aplicar encapsulamento usando propriedades; forçar encapsulamento usando assessores, incluindo público, privado, protegido e interno; forçar encapsulamento usando implementação explícita de interface, construtores, métodos, classes, métodos de extensão, parâmetros opcionais e nomeados e propriedades indexadas; criar métodos sobrecarregados (overload) e substituídos (override)

O encapsulamento é um dos pilares do desenvolvimento orientado a objetos, que diz que, quando dados (atributos) e métodos (comportamentos) são definidos em uma única entidade. Também se refere a um princípio de design orientado a objetos chamado Data Hiding, que que restringe os elementos privados de outros objetos dentro de um sistema orientado a objetos que não são necessários para o usuário. Quando você quiser algo, peça a outro objeto para executar esta ação. A única coisa que você precisa saber é a interface externa. A implementação está oculta e você não precisa entender como ela funciona, desde que esteja em conformidade com a interface.

O C# ajuda a encapsular o funcionamento interno de um objeto, fornecendo propriedades, modificadores de acesso e interfaces.O princípio do encapsulamento é um dos aspectos fundamentais do design orientado a objetos

* O princípio afirma que os dados armazenados dentro do objeto devem ser acessíveis apenas a esse objeto
* Não é diretamente acessível ao público de fora do objeto
* Explica a diferença entre propriedades e campos

O encapsulamento é implementado de duas maneiras:

1. Proteção de Dados
2. Especificadores de acesso

PROTEÇÃO DE DADOS

Propriedades e Campos

Os campos servem para armazenar os dados que descrevem as características de suas classes. Os campos são declarados como variáveis dentro da classe e podem ser de qualquer tipo, incluindo tipos de valor e referência.

Os campos vêm em dois tipos básicos, instância e estática, e uma classe pode conter um ou ambos. Um campo de instância é aquele que você usará com mais frequência em suas classes. Campos da instância são aqueles que estão contidos em cada objeto que você cria a partir da definição de classe. Cada campo de instância contém dados específicos para o objeto ao qual está atribuído. Como exemplo, crie um arquivo de classe simples no código e, em seguida, crie duas instâncias do arquivo de classe, definindo valores diferentes para os campos da classe.Tanto as propriedades e campos fornecem aos objetos a capacidade de armazenar dados.

|  |  |
| --- | --- |
| Campo | * Campo é uma variável privada que só pode ser acessada por código dentro do objeto, só está disponível de dentro do objeto |
| Propriedade | * Se parecem com membros de dados públicos (public data members), mas, são métodos especiais chamados accessors * Uma propriedade é usada para fornecer acesso público, capacidade de ler e gravar dados, ao campo privado * Especificamente, eles fornecem encapsulamento para os campos. Ele cria uma caixa de areia sobre os campos, o que a protege de salvar dados falsos. |

publicclassPerson

{

//CAMPO: PRIVATE

privateint age;

//PROPRIEDADE: PUBLIC

publicint Age

{

get

{

return age;

}

set

{

if ((value > 0) && (value < 15))

{

age = value;

}

else

{

thrownew Exception("Age cannot be over 15...");

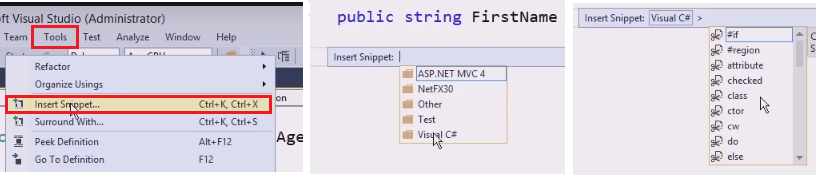
}

}

}

publicint Weight { get; set; }

}



Em C #, os dados são armazenados em uma única variável ou em uma matriz. Para proteger esses dados contra danos acidentais, temos:

* **Propriedades**
* **Indexadores**

As propriedades são de dois tipos:

1. **Propriedade Full:** nesta propriedade, declaramos campos particulares e os encapsulamos dentro da definição de uma propriedade.

classStudent\_Full

{

privateint age;

publicint Age

{

get { returnthis.age; }

set { this.age = value; }

}

}

1. **Propriedade Auto:** é muito parecida com a propriedade completa. A única diferença importante é que ele não requer nenhum campo ou lógica extra em seu get e set para manipular valores, porque um compilador cria seu próprio campo privado automaticamente. É apenas um versão sintática que o C # fornece aos seus desenvolvedores.

classStudent\_Auto

{

publicint Age { get; set; }

}

Podemos transformar uma propriedade Full quanto a Auto em somente leitura de duas maneiras. Remova o bloco set{} da definição da propriedade completa ou marque o bloco set com um especificador de acesso privado. Uma propriedade somente leitura é usada para retornar o valor de um campo e um usuário não pode definir seu valor de fora da classe.

Indexadores

Os indexadores são usados para encapsular o valor de uma matriz. Ele se comporta e funciona como propriedade. Ele também usa especificadores de acesso, que oferecem melhor controle para ler, gravar ou manipular o valor de uma matriz. Ele cria uma sandbox sobre uma matriz, que a protege de:

1. Salvar dados falsos em uma matriz;
2. Usar o valor do índice errado em uma matriz;
3. Alterar a referência de uma matriz do mundo exterior.

classTemperature

{

//declare private array of float type.

privatefloat[] weekTemp = { 47.5F, 40.0F, 52.5F, 45.5F, 48.0F, 38.0F, 35.7F };

//use float indexer, to encapsulate weekTemp

publicfloatthis[int index]

{

get

{

if (index >= 0 && index < weekTemp.Length)

{

return weekTemp[index];

}

elsereturn 0;

}

set

{

if (value > 0) weekTemp[index] = value;

else Console.WriteLine("Please set value greater than 0");

}

}

}

Temperature temp = new Temperature();

float todayTemp = temp[1]; //read

Console.WriteLine(todayTemp); //40

temp[1] = -5.0F; //Please set value greater than 0

temp[1] = 10.0F;

Console.WriteLine(temp[1]); //10

publicclassCar

{

string[] car = newstring[40];

publicstringthis[int carNum]

{

get

{

if (carNum >= 0 && carNum < car.Length)

return car[carNum];

return"Out of index range";

}

set

{

if (carNum >= 0 && carNum < car.Length)

car[carNum] = value;

}

}

}

Car c = new Car();

c[1] = "Mercedes";

c[2] = "Dodge";

c[3] = "Chevrolet";

Console.WriteLine("Car 1 is: " + c[1]); //Car 1 is: Mercedes

Console.WriteLine("Car 2 is: " + c[2]); //Car 2 is: Dodge

Console.WriteLine("Car 3 is: " + c[3]); //Car 3 is: Chevrolet

Aqui estão alguns pontos-chave a serem lembrados para propriedades indexadas:

* Eles aceitam um valor de índice no lugar do parâmetro de valor da propriedade padrão.
* Eles são identificados pelo uso da palavra-chave this
* Propriedades indexadas podem ser criadas em classes ou estruturas.
* Somente uma propriedade indexada pode existir em uma única classe ou estrutura. No entanto, você pode sobrecarregar um único indexador várias vezes.
* Eles podem conter métodos get e set, assim como outras propriedades

Os dados podem ser validados quando configurados ou obtidos usando instruções if-else. Por exemplo, podemos verificar se um valor é maior que 0; então pode ser definido na memória. Da mesma forma, podemos verificar se o valor do índice deve ser menor que o comprimento de sua matriz e maior ou igual a 0.

ESPECIFICADORES/MODIFICADORES

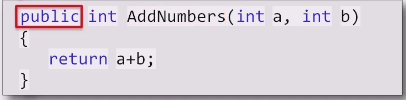
Antes de criar classes, você deve entender o uso de modificadores em C #. Modificadores são usados na declaração de tipos e nos membros de dados dos seus tipos de referência. A Tabela abaixo lista os modificadores disponíveis em C # junto com uma descrição do que os modificadores fazem.

|  |  |
| --- | --- |
| **Modificador** | **Descrição** |
| abstract | Usado para classes para indicar que essa classe não pode ser instanciada, mas que serve como uma classe base para outras classes em uma hierarquia de herança. |
| virtual | Se você cria uma classe e deseja permitir que o método seja substituído em uma classe derivada, você pode usar o modificador virtual. É usada para modificar um método, propriedade, indexador ou evento declarado na classe base e permitir que eles sejam sobrescritos na classe derivada; |
| override | Usado ao herdar a funcionalidade de uma classe base que você deseja alterar. É usada para estender ou modificar um método virtual ou abstrato, propriedade, indexador ou evento da classe base na classe derivada; |
| parcial | Arquivos de classe podem existir em vários arquivos na mesma montagem. Esse modificador informa ao compilador que a classe existe em outro arquivo ou arquivos no assembly. |
| sealed | Aplicado às classes. Define que uma classe não pode ser uma classe base, ou não é possível herdar propriedade, indexador ou evento específico de uma classe. Se a classe for **selada não é possível a herança**, mas se apenas algum método for selado, então, é possível herdar da classe e esconder/sobrescrever o método selado na classe derivada com a palavra-chave **new**. |
| async | Configura o método ou expressão lambda à qual é aplicado como um método assíncrono. Isso permite que os métodos chamem processos de longa execução sem bloquear o código de chamada. |
| const | A aplicação disso a um campo indica que o campo não pode ser modificado. As constantes devem ser inicializadas no momento em que são criadas. |
| event | Usado para declarar eventos no seu código. |
| extern | Usado para indicar que o método foi declarado e implementado externamente. Você pode usar isso com DLLs importadas ou assemblies externos. |
| new | Quando usado com os membros da classe, esse modificador oculta os membros herdados dos membros (método, propriedade, indexador ou evento) da classe base na classe derivada. Você faria isso se herdou um membro de uma classe base, mas sua classe derivada precisa usar sua própria versão desse membro. |
| readonly | Membros somente leitura podem ser atribuídos apenas durante a declaração ou em um construtor de classe. Nenhum outro meio de alterar ou atribuir um valor a esse membro é permitido. |
| static | Quando aplicado a um membro da classe, significa que o membro pertence apenas à classe e não a objetos específicos criados a partir da classe. O .NET Framework tem muitos exemplos disso, como a classe Math ou a classe String. |
| unsafe | C # é um código gerenciado, o que significa que as operações de memória são tratadas de maneira protegida. Usando a palavra-chave não segura declara um contexto que não é seguro em termos de gerenciamento de memória. Ponteiros C ++ são exemplos de operações de memória não segura. Para usar ponteiros em C #, você precisa declarar um contexto não seguro. |
| volatile | Quando esse modificador é aplicado a um campo, ele pode ser modificado por outros componentes que não o seu código. Exemplos podem ser o sistema operacional. |
| base | chamada de dentro de uma classe derivada de uma propriedade, indexador ou evento da classe base. |

Modificador de acesso

Uma das principais funções modificadores é definir o nível de acessibilidade de um tipo (classe, estrutura, enum) e todos os membros e métodos de dados definidos em um assembly ou em outros assemblies. Em C #, temos cinco tipos de modificadores de acesso. Cada modificador de acesso define um nível de acessibilidade exclusivo. Esses especificadores de acesso são:

|  |  |
| --- | --- |
| **Modificador** | **Descrição** |
| public | Um modificador de acesso que declara a acessibilidade do tipo ao qual está atribuído. Este é o nível mais permissivo. O acesso fora do corpo ou estrutura da classe é permitido. Os tipos de referência e valor, assim como os métodos, podem ser declarados públicos. |
| private | Um modificador de acesso que declara a acessibilidade do tipo ao qual está atribuído. O menos permissivo, permite o acesso apenas ao corpo da classe ou estrutura. Os tipos de referência e valor podem ser declarados privados. Os métodos também podem ser declarados privados. |
| internal | Um modificador de acesso que declara a acessibilidade do tipo ao qual está atribuído. Permite acessar apenas dentro de arquivos no mesmo assembly .NET. , mas não em outros assemblies |
| protected | Um modificador de acesso de membro. Membros declarados protegidos são acessíveis apenas de dentro da classe e em classes derivadas. |
| protectedinternal | O tipo ou membro pode ser acessado por qualquer código no assembly em que é declarado ... Ou de dentro de uma classe derivada em outro assembly |



Public

Membros definidos com especificadores de acesso public podem ser acessados dentro e fora da classe. Os dados públicos também podem ser acessíveis de fora do projeto.

classAccess\_Publico

{

publicvoid Method()

{

Console.WriteLine("Método Público");

}

}

Access\_Publico publico = new Access\_Publico();

publico.Method(); // Método Público

Private

Membros definidos com especificadores de acesso privado são acessíveis apenas dentro da classe e não podem ser acessados de fora da classe. Use métodos públicos para acessar um membro privado no mundo exterior.

classAccess\_Privado

{

privateint age = 10;

publicint GetAge()

{

return age;

}

publicvoid SetAge(int a)

{

age = a;

}

}

Access\_Privado privado = new Access\_Privado();

int age = privado.GetAge();

Console.WriteLine("Private age = {0}", age); // Private age = 10

Protected

Membros definidos com especificadores de acesso protegido podem ser acessados dentro da classe e também dentro de suas classes filho. Eles não podem ser acessíveis fora da classe. Base é uma palavra-chave usada para acessar membros definidos como especificadores de acesso público/protegido em umparent/classe base.

classParent

{

protectedint age;

}

classChild : Parent

{

publicvoid Display()

{

Console.WriteLine("Protect Age is = {0}", age);

//Console.WriteLine("Age is = {0}", base.age);

}

}

Child filho = new Child();

filho.Display(); // Protect Age is = 0

Internal

Dentro da montagem do projeto, os membros definidos com especificadores de acesso interno podem ser acessados dentro e fora da classe. Mas eles não são acessíveis a nenhuma classe definida fora da montagem do projeto.Em C #, as classes por padrão são internas, o que significa que nenhum assembly externo pode acessar as classes padrão, eles só poderão ser acessíveis a outros assemblies se as classes estiverem marcadas com especificadores de acesso público.

classAccess\_Interno

{

internalint age = 20;

}

Access\_Interno access = new Access\_Interno();

int age\_int = access.age;

Console.WriteLine("Internal age = {0}", age\_int); // Internal age = 20

Protected internal

Protegido interno é uma união de comportamento interno e protegido de especificadores de acesso, que diz que, dentro da montagem do projeto, os membros definidos com especificadores de acesso protegido interno podem ser acessados como bem como fora da classe e também para as classes filho. Mas eles não são acessíveis a qualquer classe definida fora do escopo de montagem do projeto.Membros protegidos internos não são acessíveis apenas às classes filho, mas também são acessíveis a outras classes da montagem do mesmo projeto.

classPerson

{

internalprotectedint idade = 30;

}

classEmployee : Person

{

publicvoid Display()

{

Console.WriteLine("Protect Internar Idade = {0}", base.idade);

}

}

Person pessoa = new Person();

int idade = pessoa.idade;

Console.WriteLine("Protect Internar Idade = {0}", idade); // Protect Internar Idade = 30

MÉTODOS

Métodos são os componentes de um aplicativo que permitem dividir os requisitos de computação do seu aplicativo em partes menores de funcionalidade. As boas práticas de programação determinam que você crie métodos para executar partes discretas da funcionalidade em seu código e que o método execute apenas o necessário para alcançar o resultado desejado. Alguns argumentam que a codificação dessa maneira resulta em um código que consome mais recursos devido à necessidade do sistema operacional manter indicadores e instruções de instruções para todas as chamadas de função, mas facilita muito a leitura e a manutenção do código. Se seu programa está gerando erros, é muito mais fácil rastrear o método que fornece a funcionalidade incorreta e depurar esse pequeno pedaço de código.Em essência, ummétodo é uma construção de código que contém um nome, uma assinatura, um bloco de instrução que inclui uma instrução ou conjunto de instruções e uma declaração de retorno opcional.

* Um programa faz com que as instruções sejam executadas chamando o método e especificando quaisquer argumentos de método necessários
* Semelhante a sub-rotinas em outros idiomas
* Em C #: todas as instruções executadas são executadas no contexto de um método

Os métodos têm algumas características que você deve entender

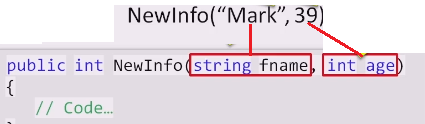
Tipo de retorno

* Métodos podem retornar um valor para o código de chamada
* Às vezes, chamado de função quando retorna um valor
* Se o tipo de retorno não for void; o método retornará um valor
* Uma instrução 'return' dentro do método interrompe a execução e retorna um valor para o chamador se o valor corresponder ao tipo de retorno

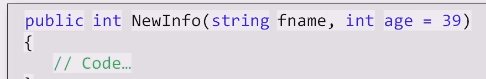


Parâmetros

* Especifica o nome e o tipo de dados dos valores que devem ser fornecidos quando o método é chamado. Por padrão, todos os parâmetros listados são necessários ao chamar o método
* Os valores passados ​​para os parâmetros do método são chamados de argumentos. Assim, as palavras 'parâmetros' e 'argumentos' são frequentemente usadas de forma intercambiável
* Os argumentos devem ser compatíveis com o tipo de dados do parâmetro
* Os parâmetros de um método são geralmente utilizados posicionalmente

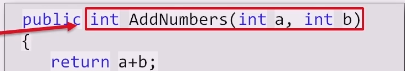


* Parâmetros opcionais são definidos no final da lista de parâmetros, após todos os parâmetros necessários. Se nenhum argumento for enviado para esse parâmetro, o valor padrão será usado.



Assinatura

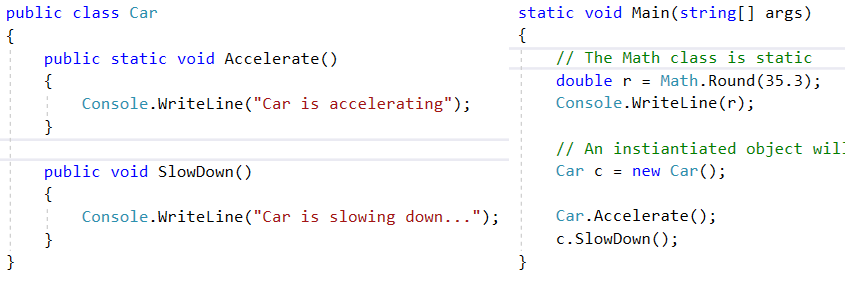
* Refere-se à estrutura do método
* Para sobrecarga(overloading), a assinatura inclui o nome do método e os tipos de parâmetro
* Para delegados(delegates), a assinatura inclui o tipo de retorno e os tipos de parâmetro



Static

O modificador estático altera o comportamento dos tipos e membros de C#. Uma classe não estática é usada para criar vários objetos usando a palavra-chave 'new' com base no tipo não estático.

* Classe ou membro estático não pode ser acessado por meio de uma instância
* Classes e membros podem ser marcados como estáticos
* Em uma classe estática, todos os métodos devem ser marcados como estáticos
* Apenas uma instância do objeto existe



CONSTRUTORES

Um método que é executado automaticamente toda vez que um objeto é instanciado da classe. Se você não fornecer seu próprio construtor, o C # criará um construtor padrão. O construtor padrão define os valores de cada variável de membro como seu valor padrão.

O construtor leva o mesmo nome que a classe. Dentro dos colchetes, as declarações de inicialização são opcionais. Um construtor é um método, mas não inclui nenhum tipo de retorno, nem mesmo nulo. Incluir um tipo de retorno em um construtor é uma sintaxe incorreta e gerará um aviso do compilador.

* Toda classe ou estrutura C # tem um construtor que tem o mesmo nome da classe.
* Se um construtor não for fornecido pelo programador, o C # criará um construtor padrão
* Permite que ações sejam executadas quando um objeto é criado com base na classe
* Definir valores padrão
* Limita instanciação
* Executa uma ação específica
* Eles podem ser sobrecarregados (overloaded)
* Existem três tipos principais de construtores
  + **Instância (Instance)**>> Usado para criar e inicializar qualquer variável de membro da instância quando você usa a nova expressão para criar um objeto de uma classe
  + **Particular (Private)**>> Impede que outras classes criem instâncias dessa classe. Somente classes aninhadas podem criar uma instância dessa classe
  + **Estático (Static)**>> Usado para inicializar qualquer dado estático. O construtor estáticoé chamado apenas uma única vez antes da criação da primeira instância da classe ou da referência a qualquer membro estático

POLIMORFISMO

Polimorfismo é tudo sobre mudança de comportamentos; em outras palavras, são formas diferentes de um objeto. Em C #, o polimorfismo é de dois tipos:

1. **Polimorfismo estático (early binding); em tempo de compilação (Overloading/Sobrecarga);**

Utilizar a sobrecarga nos permite criar mais de um método ou função que tem o mesmo nome, mas assinaturas ou parâmetros diferentes na mesma classe. Isto é chamado de sobrecarga de método, pontos importantes:

* O compilador chama automaticamente o método ou a função necessária, verificando o número de parâmetros e seu tipo, que são passados ​​para esse método.
* Se o número de parâmetros e tipo não corresponder a nenhuma assinatura de método, ocorrerá o erro de tempo de compilação.
* Podemos obter sobrecarga de método alterando o número de parâmetros usados ​​ou usando diferentes tipos de parâmetros ou alterando a ordem dos parâmetros.
* Com isto o compilador se encarrega de verificar a existência e coerência de diferentes métodos com o mesmo nome, alterando apenas a quantidade ou tipos de parâmetros necessários.

1. **Polimorfismo dinâmico (late binding); em tempo de execução (Overriding/Sobrescrita);**

Com o override, podemos criar um método em uma classe derivada com a mesma assinatura e mesmo nome que um método na classe base. Isto é chamado de substituição de método ou ainda ter dois métodos com o mesmo nome e a mesma assinatura, um método na classe base e outro na classe derivada, pontos importantes:

* Podemos substituir um método na classe base, criando função semelhante na classe derivada. Isso pode ser alcançado usando herança e usando virtual & override.
* Falar que o método possui a mesma assinatura que o outro significa que os métodos devem ter o mesmo nome, o mesmo número de argumentos e o mesmo tipo de argumento.
* A substituição de método é possível apenas nas classes derivadas, mas não dentro da mesma classe por isto temos Overload ou Shadowing (que iremos ver em outro post).
* Quando a classe derivada precisa de um método com a mesma assinatura da classe base, mas deseja executar um código diferente daquele fornecido pela classe base, a substituição do método será usada.

|  |  |
| --- | --- |
| **Overload** | **Override** |
| Criação de mais de um método ou função com o mesmo nome, mas com assinaturas ou parâmetros diferentes na mesma classe | Criação de um método na classe derivada com a mesma assinatura que um método na classe base. |
| É chamado de polimorfismo em tempo de compilação | É chamado polimorfismo de tempo de execução |
| Ele tem o mesmo nome de método, mas com assinaturas ou parâmetros diferentes | Ele deve ter o mesmo nome do método, bem como as assinaturas ou os parâmetros. |
| Sobrecarga de método não precisa de herança | A substituição de método precisa de herança |
| Sobrecarga de método é possível apenas em classe única | A substituição do método precisa do nível hierárquico das classes, ou seja, uma classe pai e outra classe filho. |
| O modificador de acesso pode ser qualquer um. | O modificador de acesso deve ser público. |
| A sobrecarga de método também é chamada de ligação antecipada. | A substituição de método também é chamada de ligação tardia. |

POLIMORFISMO ESTÁTICO - Overloading/Sobrecarga

Em C #, o polimorfismo estático pode ser implementado de duas maneiras:

1. Sobrecarga de método
2. Sobrecarga do Operador

Overloading (Sobrecarga)

Os métodos são definidos pelo modificador, tipo de retorno, nome e número e tipo de argumentos. Mas um método também tem uma assinatura. A assinatura é o que identifica exclusivamente o método de qualquer outro método com o mesmo nome e consiste no nome do método mais o tipo de dados e tipo dos parâmetros no método. Podemos 'sobrecarregar' métodos com base em suas assinaturas exclusivas. Em C #, podemos sobrecarregar um método de duas maneiras:

1. **Por tipos de parâmetros:** sobrecarregado definindo diferentes tipos de parâmetros para cada método que compartilham o mesmo nome do método.

classCalculator

{

publicvoid Add(int a, int b)

{

int result = a + b;

Console.WriteLine("Sum of ints = {0}", result);

}

publicvoid Add(string a, string b)

{

string result = a + b;

Console.WriteLine("Sum of strings = {0}", result);

}

}

Calculator cal = new Calculator();

cal.Add(1, 2); // Sum of ints = 3

cal.Add("C", "Sharp"); // Sum of strings = CSharp

1. **Pelo comprimento dos parâmetros:**sobrecarregado definindo um comprimento de parâmetro diferente para cada método que compartilha o mesmo nome do método.

classPerson

{

publicdecimal salario;

publicstring endereco;

public Person()

{

}

public Person(decimal salario)

{

this.salario = salario;

}

public Person(decimal sal, string end)

{

this.salario = sal;

this.endereco = end;

}

publicvoid Show(string name)

{

Console.WriteLine("Name = {0}", name);

}

publicvoid Show(string name, int age)

{

Console.WriteLine("Name = {0} - Age = {1}", name, age);

}

}

Person per = new Person();

per.Show("Ali"); // Name = Ali

per.Show("Ali", 22); // Name = Ali - Age = 22

Person per2 = new Person(150);

Person per3 = new Person(50, "Rua Jacu");

Outro uso comum de métodos sobrecarregados está nos construtores de classes como podemos veracima na classe Person. A sobrecarga fornece a oportunidade de inicializar variáveis de membros seletivamente. Lembre-se, um construtor é apenas um método sem um tipo de retorno.

Sobrecarga do operador

Em C#, podemos sobrecarregar a definição de um operador para tipos personalizados (classe, estrutura). Para sobrecarregar a definição de um operador, definimos métodos especiais dentro de um tipo personalizado. Esses métodos ajudam o compilador a distinguir entre diferentes significados de um operador que produz resultados diferentes para um tipo diferente. Geralmente, em C #, podemos sobrecarregar três tipos de operadores:

1. **Operadores Unários:** opera em um único operando (+, -, !, ++, -, true, false). Esses são operadores unários que podem ser sobrecarregados em C#.

classDistance\_Unitary

{

publicint meter { get; set; }

publicstatic Distance\_Unitary operator ++(Distance\_Unitary dis)

{

dis.meter += 1;

return dis;

}

}

Distance\_Unitary distance = new Distance\_Unitary();

distance.meter = 5;

distance++;

Console.WriteLine(distance.meter); // 6

Explicação

distance++ é chamado método operator ++; passou seu próprio objeto para o parâmetro operator ++.

1. **Operadores binários:** opera em dois operandos (+, -, \*, /, %, &, |, ^, <<, >>). Esses são operadores binários que podem ser sobrecarregados em c #

classStudent\_Binary

{

publicint Marks { get; set; }

// + Operator Overloading Method

publicstatic Student\_Binary operator +(Student\_Binary s1, Student\_Binary s2)

{

Student\_Binary std = new Student\_Binary();

std.Marks = s1.Marks + s2.Marks;

return std;

}

}

Student\_Binary s1 = new Student\_Binary { Marks = 10 };

Student\_Binary s2 = new Student\_Binary { Marks = 20 };

Student\_Binary s3 = s1 + s2;

Console.WriteLine(s3.Marks); // 30

Explicação

Estudante s3 = s1 + s2; quando é chamado, o operador + método será executado, o que leva s1 e s2 para seus valores de parâmetro.

1. **Operadores de comparação:** opera em dois operandos e retorna o valor booleano quando compara o valor do operando do lado esquerdo com o valor do operando do lado direito (==, !=, <, >, <=, >=). Esses são operadores de comparação que podem ser sobrecarregados em C#.

classDistance\_Compare

{

publicint meter { get; set; }

publicstaticbooloperator<(Distance\_Compare d1, Distance\_Compare d2)

{

return (d1.meter < d2.meter);

}

publicstaticbooloperator>(Distance\_Compare d1, Distance\_Compare d2)

{

return (d1.meter > d2.meter);

}

}

Distance\_Compare d1 = new Distance\_Compare { meter = 10 };

Distance\_Compare d2 = new Distance\_Compare { meter = 20 };

if (d1 < d2) Console.WriteLine("d1 is less than d2");

elseif (d2 < d1) Console.WriteLine("d2 is less than d1");

Explicação

if (d1 <d2): quando esse código é executado, o operador <método será executado, o que leva d1 e d2 como seu parâmetro. Retorna true se o valor do medidor de d1 for menor que o valor do medidor de d2.

POLIMORFISMO DINÂMICO - Overriding/Sobrescrita

O polimorfismo na programação é sobre mudar o comportamento, enquanto o polimorfismo dinâmico significa mudar o comportamento de um objeto no tempo de execução, substituindo a definição de um método. Também é conhecido como **late binding**.Uma classe derivada poderá substituir um membro de classe base somente se o membro da classe base tiver sido declarado como [virtual](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/virtual) ou [abstrato](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/abstract). O membro derivado deve usar a palavra-chave [override](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/override) para indicar explicitamente que o método destina-se a participar da invocação virtual.

Em C #, o método pode ser overridden por duas maneiras:

1. **Método Abstrato**
2. **Método Virtual**

Método Abstrato

Classes abstratas não podem ser instanciadas. É usado como classe base, onde fornece membros comuns a todas as suas classes derivadas. Ou ela é parcialmente sobrescrita (overridden) ou não é. Também é usado para declarar métodos abstratos (método sem definição) que, quando herdados, devem ser substituídos por suas classes derivadas.Um **método abstrato** é aquele que declara uma assinatura de método, não é considerado um método real porque não possui implementação.

Ao contrário do método virtual, o método abstrato não tem sua definição de método. Assim, sua classe derivada deve implementar a definição de método abstrato, caso contrário, o erro de tempo de compilação será gerado. Os métodos abstratos sempre declaram dentro de uma classe abstrata.

abstractclassVehicle\_Abs

{

publicabstractvoid Run();

}

classBike\_Abs : Vehicle\_Abs

{

publicoverridevoid Run()

{

Console.WriteLine("Run Bike\_Abs");

}

}

//Não podeser instanciada, classe abstrata

//Vehicle\_Abs vei\_inst = new Vehicle\_Abs();

Vehicle\_Abs vc\_abs = new Bike\_Abs();

vc\_abs.Run(); // Run Bike\_Abs

Console.WriteLine(vc\_abs.Wheels); //2

Explicação

* Resumo público vazio Run(); declarar método abstrato sem implementar sua definição.
* substituição pública void Run(); substitui a definição do método Run na classe derivada "Bike". Assim, quando o método Run() é chamado, o método sempre derivado será chamado.
* A classe de veículo não pode ser instanciada, mas pode armazenar referência à sua bicicleta de objeto filho.

Método Virtual

Virtual é uma palavra-chave usada com método na classe base para definir um método virtual. Quando uma classe derivada herda de uma classe base, ela ganha todos os métodos, campos, propriedades e eventos da classe base. Método virtual tem uma definição de seu método; sua classe derivada pode:

* **Substituir** os membros virtuais na classe base,
* **Herdar**o método da classe base mais próxima, sem ignorá-lo
* **Definir** nova implementação não virtual desses membros que ocultam as implementações da classe base

Portanto, ao chamar o nome do método, o tempo de execução determinará qual método chamar.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. Uma interface fornece uma outra maneira de definir um método ou conjunto de métodos cuja implementação é deixada para classes derivadas.

publicclassBaseClass

{

publicint WorkField;

publicvirtualvoid DoWork()

{

WorkField++;

Console.WriteLine("BaseClass valor virtual int WorkProperty(1) = " + WorkProperty);

Console.WriteLine("BaseClass valor int WorkField++ é: " + WorkField);

}

publicvirtualint WorkProperty

{

get

{

return 1;

}

}

}

publicclassDerivedClass : BaseClass

{

publicoverridevoid DoWork()

{

WorkField++;

Console.WriteLine("DerivedClass valor override WorkProperty(2) é: " + WorkProperty);

Console.WriteLine("DerivedClass valor int (BaseClass)WorkField++ é: " + WorkField);

}

publicoverrideint WorkProperty

{

get

{

return 2;

}

}

}

BaseClass bc = new BaseClass();

bc.DoWork();

//Calls the new method BaseClass.

//BaseClass valor virtual int WorkProperty(1) = 1

//BaseClass valor int WorkField++ é: 1

BaseClass dc = new DerivedClass();

dc.DoWork();

//Calls the new method DerivedClass.

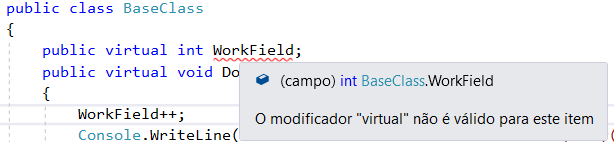
//DerivedClass valor override WorkProperty(2) é: 2

//DerivedClass valor int (BaseClass) WorkField++ é: 1

O método public virtual void DoWork() {..} define um método virtual na classe base. E o método public override void DoWork() {..} substitui o método DoWorkna classe derivada, definindosua própria implementação do método DoWork().

Ao instanciarmos BaseClass dc = new DerivedClass(); dc mantém objeto de sua classe filho DerivedClass. Assim, quando executamos dc.DoWork(); o controle passará primeiro para o método "DoWork" na classe base. Como estamos em tempo de execução, o método DoWork é um método virtual; ele passará para a definiçãoda classe derivadaDerivedClasspara encontrar a implementação do método DoWork. Se encontrar o método, ele o invocará; caso contrário, retornará à classe base para executar o método DoWork virtual.

Os campos não podem ser virtuais, apenas os 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:

BaseClass dc = new DerivedClass();

dc.DoWork();

// Calls the new method DerivedClass.

//DerivedClass valor override WorkProperty(2) é: 2

//DerivedClass valor int (BaseClass) WorkField++ é: 1

BaseClass bc\_dc = (BaseClass)dc;

bc\_dc.DoWork();

// Also calls the method DerivedClass.

//BaseClass valor virtual int WorkProperty(1) = 2

//BaseClass valor int WorkField++ é: 2

Mas observe aqui que, como não foi utilizado o operador new, o campo WorkField foi incrementado em mais uma unidade.

### Ocultando membros de classe base com novos membros

Se você quiser que o seu membro derivado tenha o mesmo nome de um membro de uma classe base, mas não quiser que ele participe da invocação virtual, será possível usar a palavra-chave [**new**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/new-modifier). 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:

publicclassBaseClass

{

publicint WorkField;

publicvirtualvoid DoWork()

{

WorkField++;

Console.WriteLine("BaseClass valor virtual int WorkProperty(1) = " + WorkProperty);

Console.WriteLine("BaseClass valor int WorkField++ é: " + WorkField);

}

publicvirtualint WorkProperty

{

get

{

return 1;

}

}

}

publicclassNewDerivedClass : BaseClass

{

publicnewint WorkField;

publicnewvoid DoWork()

{

WorkField++;

Console.WriteLine("NewDerivedClass valor override WorkProperty(3) é: " + WorkProperty);

Console.WriteLine("NewDerivedClass valor int new WorkField++ é: " + WorkField);

}

publicnewint WorkProperty

{

get { return 3; }

}

}

NewDerivedClass ndc\_first = new NewDerivedClass();

ndc\_first.DoWork();

// Calls the new method NewDerivedClass.

//NewDerivedClass valor override WorkProperty(3) é: 3

//NewDerivedClass valor int new WorkField++ é: 1

BaseClass ndc\_sec = new NewDerivedClass();

ndc\_sec.DoWork();

//BaseClass valor virtual int WorkProperty(1) = 1

//BaseClass valor int WorkField++ é: 1

Você ainda pode acessar os membros da classe base ocultos a partir do código do cliente, convertendo a instância da classe derivada a uma instância da classe base. Por exemplo:

BaseClass ndc\_third = (BaseClass)ndc\_first;

ndc\_third.DoWork();

//Calls the old method BaseClass.

//BaseClass valor virtual int WorkProperty(1) = 1

//BaseClass valor int WorkField++ é: 1

NewDerivedClass ndc\_four = (NewDerivedClass)ndc\_third;

ndc\_four.DoWork();

//NewDerivedClass valor override WorkProperty(3) é: 3

//NewDerivedClass valor int new WorkField++ é: 2

### Impedindo que classes derivadas substituam membros virtuais

Os membros virtuais permanecem virtuais por tempo indeterminado, independentemente de quantas classes foram declaradas entre o membro virtual e a classe que originalmente o declarou. Se a classe BaseClass declara um membro virtual, a classe DerivedClass deriva de BaseClass e a classe ThirdClass eFourthClassderiva de DerivedClass, a classe ThirdClass/FourthClass herda o membro virtual e tem a opção de substituí-lo, independentemente de a classe DerivedClass ter declarado uma substituição para esse membro. O código a seguir mostra um exemplo, de como a herança multi-níveis pode ficar confusa e com resultados imprevisíveis se nãp possuírem um controle de heranças:

BaseClass multi\_bc = new BaseClass();

multi\_bc.DoWork();

// Calls the new method BaseClass.

//BaseClass valor virtual int WorkProperty(1) = 1

//BaseClass valor int WorkField++ é: 1

BaseClass multi\_dc = new DerivedClass();

multi\_dc.DoWork();

// Calls the new method DerivedClass.

//DerivedClass valor override WorkProperty(2) é: 2

//DerivedClass valor int (BaseClass) WorkField++ é: 1

BaseClass multi\_th = new ThirdClass();

multi\_th.DoWork();

//ThirdClass valor override WorkProperty(3) é: 3

//ThirdClass valor int (BaseClass) WorkField++ é: 1

BaseClass multi\_de\_th = (DerivedClass)multi\_th;

multi\_de\_th.DoWork();

//ThirdClass valor override WorkProperty(3) é: 3

//ThirdClass valor int (BaseClass)WorkField++ é: 2

//FourthClass NÃO implementa o método DoWork

//Então utiliza o método virtual da classe herdada DerivedClass

BaseClass multi\_bc\_fo = new FourthClass();

multi\_bc\_fo.DoWork();

//DerivedClass valor override WorkProperty(2) é: 4

//DerivedClass valor int (BaseClass) WorkField++ é: 1

O modificador sealed (também conhecido como selado) define que uma classe não pode ser uma classe base, ou seja, não pode ter herdeiras. Uma classe sealed está pronta para uso e deve ser empregada no programa. Devido especialmente às suas características, as classes sealed podem ser consideradas o oposto das classes abstratas.

Além das explicações mencionadas referentes às classes sealed, devemos considerar alguns outros, conforme podemos ver abaixo:

* Os membros sealed não podem ser sobrescritos, mas devem sobrescrever um membro virtual ou virtual deduzido (por exemplo, um membro abstrato);
* O modificador sealed deve ser combinado com a palavra-chave override. Apesar de não poder ser aplicado a membros estáticos, ele pode ser aplicado a propriedades, métodos, indexadores e eventos;

Uma classe derivada pode interromper a herança virtual, declarando uma substituição como [sealed](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/sealed). Isso exige a colocação da palavra-chave sealed antes da palavra-chave override na declaração de membro de classe. O código a seguir mostra um exemplo:

publicclassSealedThirdClass : DerivedClass

{

publicsealedoverridevoid DoWork()

{

WorkField++;

Console.WriteLine("SealedThirdClass valor override WorkProperty(3) é: " + WorkProperty);

Console.WriteLine("SealedThirdClass valor int (BaseClass)WorkField++ é: " + WorkField);

}

publicoverrideint WorkProperty

{

get

{

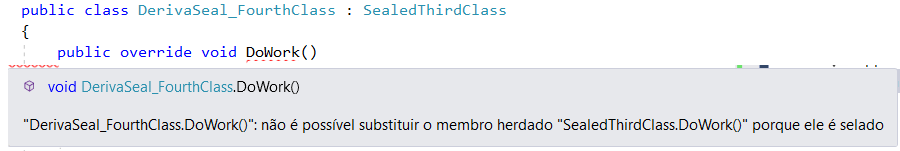
return 3;

}

}

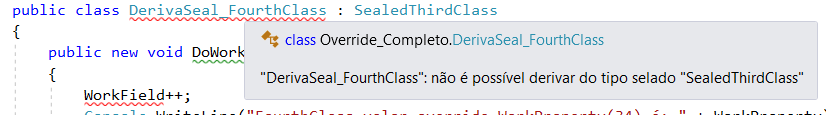
}

No exemplo anterior, o método DoWork não será mais virtual para qualquer classe derivada de SealedThirdClass. Ele ainda será virtual para instâncias de SealedThirdClass, mesmo se elas forem convertidas em métodos tipo DerivedClass ou tipo BaseClass.



Se invés do método o selead estiver na classe como abaixo, não será possível nem herdar da classe.

publicsealedclassSealedThirdClass : DerivedClass



Métodos lacrados podem ser substituídos por classes derivadas usando a palavra-chave new, como mostra o exemplo a seguir:

publicclassDerivaSeal\_FourthClass : SealedThirdClass

{

publicnewvoid DoWork()

{

WorkField++;

Console.WriteLine("FourthClass valor override WorkProperty(34) é: " + WorkProperty);

Console.WriteLine("FourthClass valor int (BaseClass)WorkField++ é: " + WorkField);

}

}

Neste caso, se DoWork é chamado em DerivaSeal\_FourthClass usando uma variável do tipo DerivaSeal\_FourthClass, o novo DoWork é chamado. Se uma variável do tipo SealedThirdClass, DerivedClass ou BaseClass é usada para acessar uma instância de DerivaSeal\_FourthClass, uma chamada de DoWork seguirá as regras de herança virtual, encaminhando as chamadas para a implementação de DoWork na classe SealedThirdClass.

### Acessando membros virtuais da classe base das 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:

publicclassBaseClass

{

publicint WorkField;

publicvirtualvoid DoWork()

{

WorkField++;

Console.WriteLine("BaseClass valor virtual int WorkProperty(1) = " + WorkProperty);

Console.WriteLine("BaseClass valor int WorkField++ é: " + WorkField);

}

publicvirtualint WorkProperty

{

get

{

return 1;

}

}

}

publicclassDerivedClass : BaseClass

{

publicoverridevoid DoWork()

{

//Metodo alternativa de chamada direta da classe base

base.DoWork();

}

publicoverrideint WorkProperty

{

get

{

return 2;

}

}

}

BaseClass bc = new BaseClass();

bc.DoWork();

//BaseClass valor virtual int WorkProperty(1) = 1

//BaseClass valor int WorkField++ é: 1

BaseClass dc = new DerivedClass();

dc.DoWork();

//BaseClass valor virtual int WorkProperty(1) = 2

//BaseClass valor int WorkField++ é: 1

Métodos de Extensão (Extension Methods)

No .NET 4.0, foi adicionado um novo recurso chamado métodos de extensão. Os métodos de extensão precisam ser declarados em uma classe estática não genérica, não aninhada e permite adicionar novos recursos para um tipo existente. Métodos de extensão são métodos estáticos especiais, pois injetam métodos de adição:

* Sem criar um novo tipo derivado
* Sem recompilar
* Sem modificar o tipo de original
* Eles são sempre chamados como se fossem um método de instância.
* Os métodos de extensão são sempre definidos dentro da classe estática.
* O primeiro parâmetro do método de extensão deve ter o operador "this", que informa em qual instância esse método de extensão deve fornecer acesso.
* O método de extensão deve ser definido no mesmo namespace em que é usado ou importar o namespace no qual o método de extensão foi definido

Lembre-se de que a diferença entre um método estático regular e um método de extensão é a palavra-chave especial this para o primeiro argumento. Abaixo um exemplo de método estático regular:

staticclassHelper

{

publicstaticint age;

static Helper()

{

age = 22;

}

publicstaticbool isLessThan\_Extension(int compareValue)

{

//return true if origin value is less

if (age < compareValue)

returntrue;

else

returnfalse;

}

}

Console.WriteLine(Helper.isLessThan\_Extension(16)); //True

Agora, um método de extensão em classe não estática utilizando o parâmetro this:

publicclassProduct

{

publicdecimal Price { get; set; }

}

publicstaticclassMyExtensions

{

publicstaticdecimal Discount(this Product product)

{

return product.Price \* .9M;

}

}

publicclassCalculator

{

publicdecimal CalculateDiscount(Product p)

{

return p.Discount();

}

}

classProgram

{

staticvoid Main(string[] args)

{

var oprod = new Product();

var ocalc = new Calculator();

oprod.Price = 1.99m;

var desconto = ocalc.CalculateDiscount(oprod);

Console.WriteLine(desconto); // 1,791

Console.ReadKey();

}

}

O método de extensão não pode ser declarado apenas em uma classe ou estrutura.Também pode ser declarado em uma interface (como IEnumerable <T>). Normalmente, uma interface não teria nenhuma implementação. Com os métodos de extensão, no entanto, você pode adicionar métodos que estará disponível em todas as implementações concretas da interface.

A Consulta Integrada à Linguagem (LINQ) é um dos melhores exemplos de como você pode usar estetécnica para aprimorar o código existente. Em vez de precisar adicionar todos os operadores LINQ a cada e todas as classes, eles são criados como métodos de extensão nas interfaces básicas de cada coleção tipo. Dessa forma, todas as coleções podem usar repentinamente o LINQ.

Interface Explicitamente(Ver também, mais a frente, Implementando Interfaces)

As interfaces são úteis ao usar o encapsulamento. Mais abaixo, na implementação de hierarquia de classe, há um explicação mais detalhada de como projetar e usar interfaces. Mas com relação ao tópico do encapsulamento, é necessário entender sobre a implementação explícita da interface.

Como exemplo de implementação explícita da interface, consulte o Entity Framework (um mapeador de objeto-relacional que faz parte do .NET Framework). Ao trabalhar com o Entity Framework, você trabalha com uma classe DbContext, que é um invólucro do ObjectContext e expõe uma interface mais fácil de usar. O DbContext implementa a seguinte interface:

publicinterfaceIObjectContextAdapter

{

ObjectContext ObjectContext { get; }

}

Embora a interface mostre uma propriedade ObjectContext, o código a seguir não será compilado:

DbContext ctx = …; // create a new context

var context = ctx.ObjectContext;

O seguinte será compilado:

var adaptedContext = ((IObjectContextAdapter)ctx).ObjectContext;

É possível porque o DbContext implementa a interface IobjectContextAdapterexplicitamente. A implementação explícita da interface significa que um elemento do tipo de interface pode ser acessado apenas ao usar a interface diretamente. Você pode criar uma implementação explícita da interface adicionando o nome da interface e um ponto à implementação.

interfaceIInterfaceA

{

void MyMethod();

}

classImplementation : IInterfaceA

{

void IInterfaceA.MyMethod() { }

}

A classe Implementation implementa a interface IInterfaceA explicitamente. Quando você tem uma instância de Implementação, não pode acessar o MyMethod. Mas quando você lança Implementation para IInterfaceA, você tem acesso ao MyMethod. Dessa forma, a implementação explícita da interface pode ser usada para ocultar membros de uma classe para usuários externos.

Há outra situação em que a implementação explícita da interface é necessária: quando uma classe implementa duas interfaces que contêm assinaturas de método duplicadas, mas deseja uma implementação diferente para ambas. Ao implementar implicitamente essas duas interfaces, apenas um método existe na implementação. Com a implementação explícita da interface, ambas as interfaces têm sua própria implementação. A Listagem abaixo mostra como implementar uma interface explicitamente.

classProgram

{

staticvoid Main(string[] args)

{

Machine1 m1 = new Machine1();

Machine3 m3 = new Machine3();

m1.Start();

m3.Start();

((IControls\_2)m3).Start();

Console.ReadKey();

}

}

publicclassMachine1 : IControls

{

publicvoid Start()

{

Console.WriteLine("Machine1...Start");

}

}

publicclassMachine3 : IControls, IControls\_2

{

publicvoid Start()

{

Console.WriteLine("Implementation of IControls.Start()...");

}

int IControls\_2.Start()

{

Console.WriteLine("Implementation of IControls\_2.Start...");

return 6;

}

}

interfaceIControls

{

void Start();

}

interfaceIControls\_2

{

int Start();

}

**Sumário**

* O encapsulamento é importante no software orientado a objetos. Oculta detalhes internos e melhora a usabilidade de um tipo.
* Os dados podem ser encapsulados com propriedades e indexadores
* As propriedades podem ter um acessador get e um set que podem executar código adicional, comumente conhecido como getters e setters.
* Tipos e elementos de tipo podem ter modificadores de acesso para restringir a acessibilidade.
* Os modificadores de acesso são públicos, internos, protegidos, protegidos internos e privados.
* A classe derivada pode herdar todos os dados de sua classe base, exceto uma que é mencionada com o especificador de acesso privado.
* O C # não permite herança múltipla, mas permite herança em vários níveis.
* Uma classe deve implementar todos os membros de uma interface e todos os métodos abstratos de uma classe abstrata.
* O método abstrato só pode ser escrito dentro de uma classe abstrata.

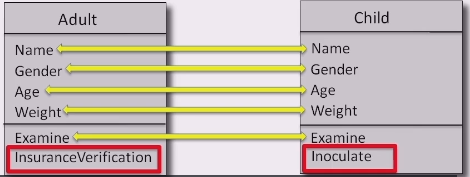
Criar e implementar uma hierarquia de classe

* Projetar e implementar uma interface; herdar de uma classe base; criar e implementar classes com base nas interfaces IComparable, IEnumerable, IDisposable e IUnknown

HERANÇA

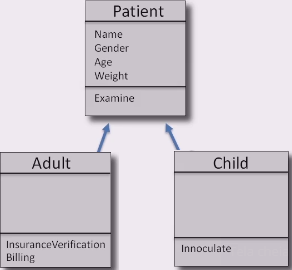
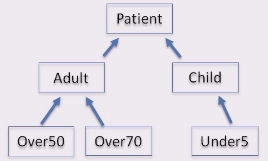
Herança, juntamente com o encapsulamento e polimorfismo, é uma das três características primárias (ou pilares) da programação orientada a objeto. A herança permite que você crie novas classes que podem ser reutilizadas e/ou estendidas. Podemos ver exemplos de herança nas classes do próprio .Net Framework, Por exemplo a classe CollectionBase, que contém propriedades e métodos utilizados pela maioria das classes de Collection (coleções). Ela contém propriedades como Capacity e Count e métodos como Clear() e RemoveAt().

Para utilizar a classe base (superclasse) em outra classe (classe derivada ou subclasse) você faz referência à classe base, colocando o símbolo : (dois pontos) após o nome da classe derivada e em seguida coloque o nome da classe base. Quando você define uma classe para derivar de outra classe, a classe derivada implicitamente ganha todos os membros da classe base, exceto seus construtores e destrutores. A herança permite criar novas classes que reutilizam, estendem e modificam o comportamento definido em outras classes podendo assim reutilizar o código na classe base. Isso elimina a duplicação de elementos nas classes reduzindo a necessidade de escrever o mesmo código em duas classes diferentes facilitando a manutenção do código. Observe nas classes abaixo que existem muitas duplicações de propriedades e métodos e existem apenas duas diferenças:



A herança geralmente é implementada em um modelo denominado “general down to specific” em que se é criado uma classe base, cujos membros são herdados, e classes que herdam esses membros são chamadas de classes derivadas. Uma classe derivada pode ter apenas uma classe base direta. No entanto, a herança é transitiva. Se ClassC é derivada da ClassB e ClassB é derivada da ClassA, ClassC herda os membros declarados em ClassB e ClassA.

Conceitualmente, uma classe derivada é uma especialização da classe base. Poderíamos criar uma classe Patient que contasse com todas as propriedades e métodos que pertencem a todos os pacientes e as classes Adult e Childteriam as suas propriedades específicas com métodos mais especializados. Em que:



* Uma classe só pode herdar de outra classe
* O Adulto não pode herdar do Patient e de outra classe.
* Mais de uma classe pode herdar do Patient.
* A herança é transitiva; uma classe herdada do Adult (Over50, Over70) também tem acesso a propriedades e métodos no Patient

classProgram

{

staticvoid Main(string[] args)

{

Patient p = new Patient();

p.Examine("Bob"); // use the Patient functionalities

Child c = new Child();

c.Examine("Billy"); // Use both Patient and Child functionalities

c.Innoculate();

}

}

publicclassPatient

{

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

publicstring Gender { get; set; }

publicint Age { get; set; }

publicint Weight { get; set; }

publiclong SSN {get; set;}

publicvoid Examine(string pname)

{

// Add code for patient examination...

Console.WriteLine("Exmination of " + pname + " completed...");

}

publicvoid Billing(long ssn)

{

// Add code to retrive patient billing/balance info

Console.WriteLine("Billing completed...");

}

}

publicclassChild : Patient // Use colon to establish inheritance from Patient

{

publicvoid Innoculate()

{

// Add code to record innoculation

Console.WriteLine("Child has been innoculated...");

}

}

Se na classe base tivessámos um construtor que aceitasse parâmetros como abaixo:

public Patient(string FirstName, string LastName)

{

// Validate the first and last names.

if ((FirstName == null) || (FirstName.Length < 1))

thrownew ArgumentOutOfRangeException( "firstName", FirstName, "FirstName must not be null or blank.");

if ((LastName == null) || (LastName.Length < 1))

thrownew ArgumentOutOfRangeException( "lastName", LastName, "LastName must not be null or blank.");

this.FirstName = FirstName;

this.LastName = LastName;

}

Para garantir que o construtor da classe derivada Childaproveite dessas validações, primeiro precisamos atribuir um construtor à classe Child com a mesma assinatura, ou seja, quantidade, tipos e nomes do argumentos iguais. A seguir, após a lista de argumentos, colocamos dois pontos, a palavra-chave base e quaisquer parâmetros que deseja passar para o construtor da classe base, como no exemplo abaixo:

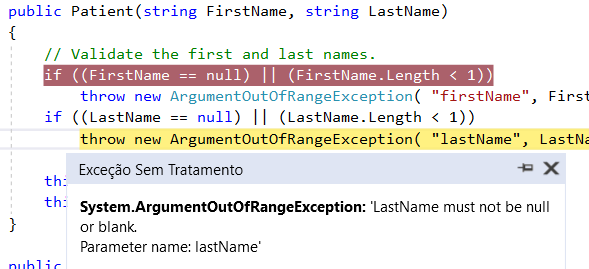
public Child(string firstName, string lastName) : base(firstName, lastName)

{

}

Quando um construtor usa a palavra-chave base para chamar um construtor da classe base, o construtor da classe base é executado antes que o corpo do construtor da classe filho seja executado.

Child pctor = new Child("Daryl", null);



Agora imagine uma situação onde tivéssemos muitos campos dentro de uma classe e precisássemos criar vários construtores que setassem determinados campos, como no exemplo abaixo:

public Patient(string FirstName, string LastName, string Gender)

{

this.FirstName = FirstName;

this.LastName = LastName;

this.Gender = Gender;

}

public Patient(string FirstName, string LastName, string Gender, int Age)

{

this.FirstName = FirstName;

this.LastName = LastName;

this.Gender = Gender;

this.Age = Age;

}

Para evitar essa repetição de código dentro dos construtores temos a opção de fazer um construtor chame um segundo construtor através do uso da palavra-chave "this" em vez da palavra-chave base. O código a seguir mostra como o segundo construtor da classe Patientpode invocar seu primeiro construtor:

public Patient(string FirstName, string LastName, string Gender)

{

this.FirstName = FirstName;

this.LastName = LastName;

this.Gender = Gender;

}

public Patient(string FirstName, string LastName, string Gender, int Age) : this(FirstName, LastName, Gender)

{

this.Age = Age;

}

Quando um construtor usa a palavra-chave “this” para chamar um segundo construtor na mesma classe, o segundo construtor é executado antes que o corpo do primeiro construtor seja executado.

Agora que entendemos o uso da palavra-chave “this” e “base”, podemos aplicar estas duas funcionalidades em conjunto na classe derivada Child incluindo mais um campo contendo o nome do pai, como se segue:

publicstring Nome\_Pai { get; set; }

public Child(string firstName, string lastName) : base(firstName, lastName)

{

}

public Child(string firstName, string lastName, string nomepai)

: this(firstName, lastName)

{

Nome\_Pai = nomepai;

}

Princípio da substituição de Liskov

A herança é uma técnica poderosa, mas deve ser usada com cautela. Como já mencionado, a herança deve ser usada apenas quando você está lidando com um relacionamento “é uma espécie de”. O princípio de substituição de Liskov afirma que uma subclasse deve ser utilizável em cada local em que você pode usar uma das classes base. Eles não devem mudar repentinamente o comportamento do qual os usuários dependeriam. É fácil violar esse princípio. Considere o código abaixo.

classRectangle

{

publicvirtualint Height { get; set; }

publicvirtualint Width { get; set; }

publicint Area

{

get { return Height \* Width; }

}

}

Ao olhar para essa classe Rectangle, você diria que um Square é uma espécie de Rectangle? Em matemática, isso seria verdade. Sabemos que um quadrado é um tipo especial de retângulo. Você pode modelar isso usando uma relação de herança:

classSquare : Rectangle

{

publicoverrideint Width

{

get { returnbase.Width; }

set

{

base.Width = value;

base.Height = value;

}

}

publicoverrideint Height

{

get { returnbase.Height; }

set

{

base.Height = value;

base.Width = value;

}

}

}

Como você sabe que está lidando com um quadrado, ajude o usuário da classe modificando as propriedades Width e Height juntas. Dessa forma, o retângulo sempre será um quadrado.Suponha que você queira usar a classe como mostrado:

Rectangle rect\_squa = new Square();

rect\_squa.Width = 10;

rect\_squa.Height = 5;

Console.WriteLine("rect\_squa Area:" + rect\_squa.Area); // rect\_squa Area:25

Esse código produzirá 25. O usuário acha que está lidando com um Rectangle com uma Areacalculada, mas como o Rectangle está apontando para um Square, apenas o valor mais recente de Height é armazenado. Este é um exemplo típico de violação do princípio de substituição de Liskov. A classe Square não pode ser usada em todos os lugares onde você normalmente usaria um Rectangle.

Rectangle rect\_rect = new Rectangle();

rect\_rect.Width = 10;

rect\_rect.Height = 5;

Console.WriteLine("rect\_rect Area:" + rect\_rect.Area); // rect\_rect Area:50

Square squa\_squa = new Square();

squa\_squa.Width = 10;

squa\_squa.Height = 5;

Console.WriteLine("squa\_squa Area:" + squa\_squa.Area); // squa\_squa Area:25

INTERFACES

Uma interface contém definições para um grupo de funcionalidades relacionadas que uma classe ou uma estrutura deve fornecer. A Interfacefornece o mais alto nível de abstração, é como se fosse a versão de software de um contrato, um padrão, que exige que uma classe implemente funcionalidades específicas, mas não diz à classe como fazê-lo, seus membros não têm detalhes de implementação. Quando uma classe implementa uma interface ela:

* Concorda com os termos do contrato
* Ele deve fornecer uma implementação para todos os membros definidos na interface.
* Informa outras partes do programa que a classe possui esses recursos, para que o código possa invocá-los.
* Classes não podem herdar de mais de uma classe, mas podem implementar mais de uma interface

Todos os detalhes da implementação são definidos por classes que implementam (herdam) interfaces. A interface pode definir métodos, propriedades, eventos, indexadores fornecendo funcionalidade semelhante para nossas classes garantindo consistência entre a chamadas de métodos de classe e polimorfismo.

Isso fornece um tipo de polimorfismo semelhante à maneira como as classes pois, por exemplo, suponha que a classe Child herda da classe Patient e implemente a interface ICloneable. Nesse caso, um programa poderia tratar um objeto Employee como se fosse um objeto Child, Patient ou ICloneable.

Observação:

* Não pode ser instanciada
* Os membros da interface são automaticamente públicos
* Os membros da interface não podem incluir modificadores de acesso

Às vezes, escrever todos os métodos definidos por uma interface pode ser bastante trabalhoso. A menos que a interface esteja bem documentada, pode ser difícil definir quais propriedades, métodos e eventos são necessários. Felizmente, o Visual Studio fornece uma ferramenta que cria código para implementar uma interface para você. Para usar a ferramenta, escreva a declaração da classe e especifique a interface:

interfaceIControls

{

void Start();

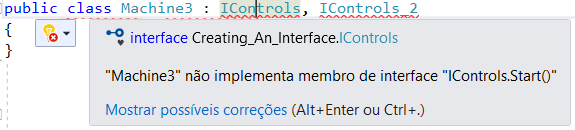
}

interfaceIControls\_2

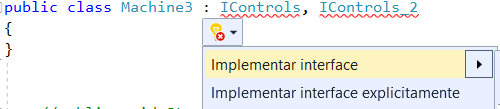
{

int Start();

}



Neste ponto, o Visual Studio sabe que você não implementou a interface. Clique com o botão direito do mouse na declaração da interface para exibir um menu de contexto.



Abra o item Implementar interface e selecione Implementar interface ou Implementar explicitamente para fazer o Visual Studio inserir stubs de código que satisfaçam a interface.O código a seguir mostra o resultado produzido por esta ferramenta se você escolher o item Implementar explicitamente a interface:

publicclassMachine3 : IControls, IControls\_2

{

voidIControls.Start()

{

thrownew NotImplementedException();

}

intIControls\_2.Start()

{

thrownew NotImplementedException();

}

}

Se você selecionar Implementar Interface, o código não incluirá as partes IControls e IControls\_2, mostradas em destaque no código anterior.Além da diferença de sintaxe, há uma diferença funcional entre as duas maneiras de implementação da interface:

Implicitamente:

Interfaces implícitas são implementadas publicamente. É implementado quando a definição explícita dos membros de cada interface não é necessária.

* Não use especificadores de acesso com os membros da interface.
* Não defina a definição de membros da interface.
* Propriedade Auto, indexador, método e evento podem ser usados como um membro de uma interface.
* A classe deve implementar a definição completa dos membros da interface. Caso contrário, poderá ocorrer um erro no tempo de compilação/execução.
* A classe pode implementar mais de uma interface.

interfaceIVehicle

{

int Wheels { get; }

}

classBike : IVehicle

{

privateint wheels;

publicint Wheels

{

get

{

return wheels;

}

}

}

Explicitamente(Ver também, mais atrás, no último tópico de Encapsulamento)

Interfaces explícitas são implementadas em particular. Nós os implementamos explicitamente quando é necessária uma definição separada do membro de cada interface. Por exemplo, quando uma classe implementa mais de uma interface que compartilha o nome de um membro comum, é necessária uma implementação explícita da interface para separar a definição de cada membro.

interfaceIEnglish

{

int Marks { get; }

}

interfaceIMath

{

int Marks { get; }

}

classStudent : IEnglish, IMath

{

int english = 10;

int math = 9;

int IMath.Marks

{

get

{

return english;

}

}

int IEnglish.Marks

{

get

{

return math;

}

}

}

Student std = new Student();

int english = ((IEnglish)std).Marks;

int math = ((IMath)std).Marks;

// English Marks = 9 Math Marks = 10

Console.WriteLine("English Marks = {0} Math Marks = {1}",english, math);

Explicação

* IMath.Marks e IEnglish.Marks usado para implementar a interface explicitamente, definindo o nome do nome de gravação da interface antes do nome do membro.
* Nenhum especificador de acesso usado para implementar interfaces explícitas.
* (IEnglish) std) .Marks; usado para acessar a propriedade "Marks" da interface "IEnglish".

Delegando Interfaces

Tendo duas classes Student e TeachingAssistant que implementam a interface IStudent, portanto, elas incluem código para fornecer os recursos da interface.Você pode evitar duplicar esse código delegando o trabalho de implementar a interface para um objeto Student dentro da classe TeachingAssistant. Sempre que o objeto TeachingAssistant precisar executar alguma tarefa especificada pela interface IStudent, ele faz com que o objeto Student faça o trabalho.

publicinterfaceIStudent

{

// The student's list of current courses.

List<string> Courses { get; set; }

// Print the student's current grades.

void PrintGrades();

}

publicclassStudent : IStudent

{

// Implement IStudent.Courses.

// The student's list of current courses.

public List<string> Courses { get; set; }

// Implement IStudent.PrintGrades.

// Print the student's current grades.

publicvoid PrintGrades()

{

// Do whatever is necessary...

Console.WriteLine("Student.PrintGrades");

}

}

// Delegate IStudent to a Student object.

publicclassTeachingAssistant2 : IStudent

{

// A Student object to handle IStudent.

private Student MyStudent = new Student();

public List<string> Courses

{

get { return MyStudent.Courses; }

set { MyStudent.Courses = value; }

}

publicvoid PrintGrades()

{

MyStudent.PrintGrades();

}

}

Isso pode parecer um trabalho extra, mas permite manter todo o código para implementar a interface na classe Student. Agora, se você precisar alterar o código, poderá fazê-lo nesse local.

IMPLEMENTAR INTERFACES COMUNS

O .NET Framework inclui muitas interfaces que ajudam as classes do Framework a realizar seus trabalhos. Por exemplo, se uma de suas classes implementar a interface IComparable, o método Array.Sort pode classificar uma matriz dessa classe. As seções a seguir explicam como você pode implementar algumas das interfaces mais úteis definidas pelo .NET Framework.

IComparable

Se uma classe implementa IComparable, ele fornece um método CompareTo que permite que um programa compare duas instâncias da classe e determine qual pertence antes da outra na ordem classificada. Por exemplo, suponha que você queira criar uma classe Persone classificar os objetos Personpor suas idades. Nesse caso, você pode fazer com que a classe Personimplemente a interface IComparable e use Array.Sort para classificar uma matriz de objetos Person.

A interface IComparable vem em duas versões: uma versão simples e uma versão genérica. A versão genérica do IComparable é mais simples que a versão não genérica e fornece forte verificação de tipo, portanto você deve usá-lo. A versão não genérica é fornecida para compatibilidade com versões mais antigas do .NET Framework.

A forte verificação de tipo fornecida pela versão genérica impede que você cometa o erro de tentar comparar objetos do tipo errado. Se você usar a versão simples, o método CompareTo usa dois objetos não específicos como parâmetros e o código deve convertê-los em objetos Personantes de comparar as idades. O método CompareTo retorna um valor int que mostra como dois elementos estão relacionados. A Tabela abaixo mostra os possíveis valores que o método CompareTo retorna.

|  |  |
| --- | --- |
| **Valor** | **Significado** |
| Menos que zero | A instância atual precede o objeto especificado pelo método CompareTo na ordem de classificação. |
| Zero | Essa instância atual ocorre na mesma posição na ordem de classificação que o objeto especificado pelo método CompareTo. |
| Maior que zero | Essa instância atual segue o objeto especificado pelo método CompareTo na ordem de classificação. |

O código a seguir mostra a classe Personcom esse tipo de método CompareTo:

classPerson : IComparable

{

publicstring Name { get; set; }

publicint Age { get; set; }

publicint CompareTo(object obj)

{

if (!(obj is Person))

thrownew ArgumentException("Object is not a Person");

Person next = (Person)obj;

returnthis.Age.CompareTo(next.Age);

}

}

O código a seguir mostra a classe Personimplementando a versão genérica da interface IComparable:

classPerson\_Generic : IComparable<Person\_Generic>

{

publicstring Name { get; set; }

publicint Age { get; set; }

publicint CompareTo(Person\_Generic other)

{

returnthis.Age.CompareTo(other.Age);

}

}

Nesta versão, o nome da interface é seguido por um parâmetro genérico informando o tipo de objeto com o qual a classe pode se comparar, neste caso Person\_Generic. O parâmetro do método CompareTo é um objeto desse tipo, portanto, o CompareTo não precisa verificar se o objeto é um Person\_Generice não precisa converter o objeto em um Person\_Generic.

ArrayList people\_array = new ArrayList(){

new Person { Name = "Sundus", Age = 21 },

new Person { Name = "Ali", Age = 22 },

new Person { Name = "Hogi", Age = 12 },

};

//sort list of persons

people\_array.Sort();

foreach (Person person in people\_array)

{

//12 Hogi 21 Sundus 22 Ali

Console.WriteLine(person.Age + " " + person.Name);

}

IComparer

O programa de exemplo IComparable descrito na seção anterior usou uma classe Person que implementa IComparável, para que ele possa classificar uma variedade de objetos Person por suas idades, mas o método CompareTo pode classificar apenas um campo de cada vez, não sendo possível classificar a classe Person em propriedades diferentes, como o nome, por exemplo.

Uma classe que implementa a interface IComparer deve fornecer um método Compare que compara dois objetos. Por exemplo, você pode criar uma classe sortAge e outra classe SortName que implemente IComparer e que tenha um método Compare que compara objetos Person por idade e nome. Como no exemplo a seguir:

classPerson

{

publicstring Name { get; set; }

publicint Age { get; set; }

}

classsortAge : IComparer

{

publicint Compare(object x, object y)

{

Person first = (Person)x;

Person second = (Person)y;

return first.Age.CompareTo(second.Age);

}

}

classSortName : IComparer

{

publicint Compare(object x, object y)

{

Person first = (Person)x;

Person second = (Person)y;

return first.Name.CompareTo(second.Name);

}

}

ArrayList people = new ArrayList();

people.Add(new Person { Name = "Sundus", Age = 21 });

people.Add(new Person { Name = "Ali", Age = 22 });

people.Add(new Person { Name = "Hogi", Age = 12 });

//sort list according to age

people.Sort(new sortAge());

foreach (Person person in people)

{

// 12 Hogi 21 Sundus 22 Ali

Console.WriteLine(person.Age + " " + person.Name);

}

Console.WriteLine();

//sort list according to name

people.Sort(new SortName());

foreach (Person person in people)

{

// Ali 22 Hogi 12 Sundus 21

Console.WriteLine(person.Name + " " + person.Age);

}

Porém esta implementação tem muitas limitações. Aparentemente só funciona com ArrayList, pois ao tentar com Array não aceita parâmetro IComparer (só há sobrecarga com Array) e ao tentar com List<Person> há erro em relação a que nosso IComparer não é genérico. Como vimos na implementação da interface IComparable. A versão IComparer genérica também é mais simples e fornece forte verificação de tipo, portanto devemos usá-la. Então nossa classes IComparer ficariam assim:

classsortAge\_Generic : IComparer<Person>

{

publicint Compare(Person x, Person y)

{

return x.Age.CompareTo(y.Age);

}

}

classSortName\_Generic : IComparer<Person>

{

publicint Compare(Person x, Person y)

{

return x.Name.CompareTo(y.Name);

}

}

Isso já resolve parcialmente o problema de classificação por nome e idade mas, e se tivesse muitas outras propriedades da classe Person que gostariamos de classificar, teríamos de criar várias classes de comparadores uma para cada classificação. Uma opção seria criar uma única classe PersonComparer e dar a ela um campo que o programa possa definir para informar qual campo de carro usar ao comparar objetos de carro. O código a seguir mostra uma classe PersonComparer que demonstra essa abordagem:

classPersonComparer : IComparer<Person>

{

// The field to compare.

publicenumCompareField { Name, Age }

public CompareField SortBy = CompareField.Name;

publicint Compare(Person x, Person y)

{

switch (SortBy)

{

case CompareField.Name: return x.Name.CompareTo(y.Name);

case CompareField.Age: return x.Age.CompareTo(y.Age);

}

return x.Name.CompareTo(y.Name);

}

}

A classe começa com uma enumeração que define os tipos de classificação que essa classe pode fornecer. Seu campo SortBy indica o campo Personque a classe deve usar ao classificar. O método Compare examina o valor SortBy e compara dois objetos Personadequadamente. O programa de exemplo abaixo, usa essa classe PersonComparer para classificar os objetos Personpor Nome eIdade e poderia ser por muitas outras propriedades:

var arr\_per = new Person[]

{

new Person { Name = "Sundus", Age = 21 },

new Person { Name = "Ali", Age = 22 },

new Person { Name = "Hogi", Age = 12 }

};

var sortPerson = "Age";

PersonComparer per\_comparer = new PersonComparer();

if (sortPerson == "Name")

per\_comparer.SortBy = PersonComparer.CompareField.Name;

elseif (sortPerson == "Age")

per\_comparer.SortBy = PersonComparer.CompareField.Age;

Array.Sort(arr\_per, new sortAge\_Generic());

Array.Sort(arr\_per, new SortName\_Generic());

Array.Sort(arr\_per, per\_comparer);

Esse método cria um objeto PersonComparer e define seu valor SortBy de acordo com o valor selecionado por algum controle do programa (aqui definido com uma constante sortPerson). Em seguida, você pode passar um objeto PersonComparer para o método Array.Sort e ele pode usar esse objeto para classificar uma matriz de objetos Person. Caso você estivesse trabalhando com uma lista de Person bastaria convertê-la usando o método ToArray, como abaixo:

List<Person> people\_list = new List<Person>() {

new Person { Name = "Sundus", Age = 21 },

new Person { Name = "Ali", Age = 22 },

new Person { Name = "Hogi", Age = 12 }

};

Array.Sort(people\_list.ToArray(), per\_comparer);

IEquatable

IEquatable <T> é uma interface implementada por tipos cujos valores podem ser equacionados (por exemplo, as classes numéricas e de string). A interface IEquatable fornece esse recurso exigindo que uma classe forneça um método Equals. O código a seguir mostra a classe Person:

classPerson\_Equals : IEquatable<Person\_Equals>

{

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

publicbool Equals(Person\_Equals other)

{

return ((FirstName == other.FirstName) && (LastName == other.LastName));

}

}

Para a maioria dos tipos de referência, o uso do IEquatable é evitado porque, se você o fizer, precisará substituir os métodos Object.Equals(Object) e GetHashCode, pois seu comportamento é consistente com o método IEquatable.Equals. Implementando todas validações na classe Person ficaria assim:

classPerson : IEquatable<Person>

{

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

publicint Age { get; set; }

publicbool Equals(Person other)

{

if ((FirstName == other.FirstName) && (LastName == other.LastName))

returntrue;

elsereturnfalse;

}

publicoverridebool Equals(object obj)

{

Person other = (Person)obj;

returnthis.Equals(other);

}

publicoverrideint GetHashCode()

{

//custom implementation of hashcode

string hash = this.FirstName + this.LastName;

return hash.GetHashCode();

}

publicstaticbooloperator ==(Person person1, Person person2)

{

if (((object)person1) == null || ((object)person2) == null)

return Object.Equals(person1, person2);

return person1.Equals(person2);

}

publicstaticbooloperator !=(Person person1, Person person2)

{

if (((object)person1) == null || ((object)person2) == null)

return !Object.Equals(person1, person2);

return !(person1.Equals(person2));

}

}

Classes de coleção genérica, como Lista, Dicionário, Pilha e Fila, fornece Contains e outros métodos que comparam objetos para igualdade. A Microsoft recomenda que seja implementado o IEquatable em qualquer classe que esteja em qualquer dessas coleções genéricas.

Isso permite criar uma lista de objetos Person que usa o método Contains da lista para verificar se a pessoa já está na lista. Se você tentar criar uma Pessoa com o mesmo nome e sobrenome que uma Pessoa criada anteriormente, o programa exibirá uma mensagem de erro.

Person person1 = new Person() { Age = 21, FirstName = "Ali", LastName = "Dali" };

Person person2 = new Person() { Age = 45, FirstName = "Ali", LastName = "Dali" };

Console.WriteLine(person1 == person2); // True

List<Person> People = new List<Person>();

People.Add(person1);

if (People.Contains(person2)) // True

Console.WriteLine("The list already contains this person.");

else People.Add(person2);

O método Contains da lista usa o fato de que a classe Person implementa IEquatable para decidir se dois objetos são iguais. Se você comentar a parte: IEquatable da declaração da classe Person, a classe não implementa mais o IEquatable; portanto, a lista trata dois objetos diferentes como diferentes, mesmo que eles tenham os mesmos valores de primeiro e último nome.

ICloneable

Uma classe que implementa a interface ICloneable deve fornecer um método Clone que retorna uma cópia do objeto para o qual é chamado. Por exemplo, o código a seguir mostra uma classe Person simples e clonável:

classPerson : ICloneable

{

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

public Person Manager { get; set; }

// Return a clone of this person.

publicobject Clone()

{

Person person = new Person();

person.FirstName = FirstName;

person.LastName = LastName;

person.Manager = Manager;

return person;

}

}

O método Clone desta classe simplesmente cria um novo objeto Person com as mesmas propriedades FirstName, LastName e Manager que o original e, em seguida, retorna o novo objeto. Observe que o método Clone retorna um objeto não específico, não uma Person; portanto, o código de chamada deve converter o resultado em uma Person. O código a seguir mostra como o programa de exemplo ICloneablePerson, disponível para download no site do livro, cria dois objetos Person. e depois clona um deles:

Person ann = new Person() { FirstName = "Ann", LastName = "Archer", Manager = null };

Person bob = new Person() { FirstName = "Bob", LastName = "Baker", Manager = ann };

Person bob2 = (Person)bob.Clone();

Esse código cria uma pessoa chamada Ann Archer e outra chamada Bob Baker. Em seguida, clona a pessoa de Bob Baker para criar um terceiro objeto de pessoa.A classe de método Person.Clone descrita nesta seção é um clone superficial porque define a propriedade Manager do clone igual à propriedade Manager do objeto original.Existem dois tipos de clones:

* **Superficial**: qualquer valor de referência na cópia se refere aos mesmos objetos que os do objeto original
* **Profundo**: os valores de referência do novo objeto são definidos para novos objetos.

A interface ICloneable não especifica se o método Clone deve retornar um clone superficial ou profundo; portanto, você deve fazer o que fizer mais sentido para o seu aplicativo. Se desejar, também é possível criar um segundo método Clone que assume como parâmetro um valor booleano que indica se a cópia deve ser um clone profundo.O código a seguir mostra como a classe Person poderia fornecer clones profundos:

publicobject DeepClone()

{

Person person = new Person();

person.FirstName = FirstName;

person.LastName = LastName;

person.Manager = Manager;

if (Manager != null)

person.Manager = (Person)Manager.Clone();

return person;

}

IUnknown

Antes da existência do .NET, a primeira geração da API do Windows era baseada em uma biblioteca de funções contidas em uma biblioteca de vínculo dinâmico(DLL). Gerações posteriores coletaram essas funções em uma interface Component Object Model(COM). O .NET Framework fornece classes que agrupa muitas dessas APIs em uma versão gerenciada para que você quase nunca toque diretamente em nenhum componente COM.

Normalmente, você apenas adiciona uma referência a um objeto COM e o compilador gera as classes de wrapper necessárias chamadas classes de interoperabilidade COM. Se isso falhar por algum motivo, você precisará criar a classe wrapper; é aqui que a interface IUnknown é usada.

IUnknown é a interface base de todas as outras interfaces COM. Essa interface define três métodos: QueryInterface, AddRefe Release. QueryInterface permite que um usuário de interface solicite ao objeto um ponteiro para outra de suas interfaces. O AddRef e o Release implementam a contagem de referência na interface.

IDisposable (Mais informações em “Gerenciar o ciclo de vida do objeto” mais adiante neste documento)

Outra interface útil no .NET Framework é IDisposable. Como o C # é uma linguagem gerenciada que usa um Garbage Collector para limpar a memória, essa interface é usada somente para facilitar o trabalho com recursos externos não gerenciados, como conexões com o banco de dados ou identificadores de arquivo. O único método que a interface IDisposable possui é Dispose(), conforme mostra a definição da interface IDisposable.

interface pública IDisposable

{

Dispose ();

}

Usando reflexão

* Criar e aplicar atributos; ler atributos; gerar código em tempo de execução usando expressões CodeDom e lambda; usar tipos do namespace System.Reflection, incluindo Assembly, PropertyInfo, MethodInfo, Type

Um aplicativo .NET não contém apenas código e dados; Ele também contém metadados, que são informações sobre dados. No .NET, isso significa que um aplicativo contém o código que define o aplicativo e os dados que descrevem o código. Um atributo é um tipo de metadado que pode ser armazenado em um aplicativo .NET. Outros tipos de metadados contêm informações sobre os tipos, código, montagem e todos os outros elementos armazenados no seu aplicativo. Reflexão é o processo de recuperar esses metadados em tempo de execução. Os dados podem ser inspecionados e usados para tomar decisões.

Criando e usando atributos

O uso de atributos é uma maneira poderosa de adicionar metadados a um aplicativo. Os atributos podem ser adicionados a todos os tipos: montagens, tipos, métodos, parâmetros e propriedades. No tempo de execução, você pode consultar a existência de um atributo e suas configurações e, em seguida, executar as ações apropriadas.Os atributos são usados ​​por vários motivos. Eles podem ser usados ​​para descrever as informações do autor de um assembly ou para fornecer dicas específicas ao compilador sobre como otimizar seu código. Atributos personalizados podem armazenar todos os tipos de dados que você deseja.

No C#, você aplica um atributo colocando-o entre colchetes ([]) acima da declaração à qual deseja aplicar o atributo.Um exemplo de atributo no .NET Framework é o atributo System.SerializableAttribute. Este atributo indica que um tipo pode ser serializado. O .NET Framework verifica a existência desse atributo ao serializar um tipo e garante que todos os membros do tipo também possam ser serializados.

[Serializable]

classPerson

{

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

}

Como você pode ver, a classe real no .NET Framework é chamada SerializableAttribute. Por convenção, o nome é sufixado com Attributepara que você possa distinguir facilmente entre atributos e outros tipos no .NET Framework. Ao usar o atributo, no entanto, você pode pular o sufixo do Attribute.

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. 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 de montagem. A partir desse fluxo, o objeto pode ser armazenado em um banco de dados, um arquivo ou memória. A serialização permite que o desenvolvedor salve o estado de um objeto e o recrie conforme necessário, fornecendo armazenamento de objetos, bem como troca de dados. Através da serialização, um desenvolvedor pode executar ações como:

* Enviando o objeto para um aplicativo remoto usando um serviço web
* Passando um objeto de um domínio para outro
* Passando um objeto através de um firewall como uma seqüência JSON ou XML
* Manutenção de informações de segurança ou específicas do usuário entre aplicativos

[Serializable]

classPerson

{

publicstring Name { get; set; }

publicint Idade { get; set; }

publicdouble Altura { get; set; }

// A field that is not serialized.

[NonSerialized()] publicstring Endereco;

public Person()

{

Name = "Bob";

Idade = 11;

Altura = 1.74159265;

Endereco = "Avenida Brasil";

}

publicvoid Print()

{

Console.WriteLine("Name = '{0}'", Name);

Console.WriteLine("Idade = '{0}'", Idade);

Console.WriteLine("Altura = '{0}'", Altura);

Console.WriteLine("NonSerialized Endereco = '{0}'", Endereco);

}

}

// Creates a new TestSimpleObject object.

Person obj = new Person();

Console.WriteLine("Before serialization the object contains: ");

obj.Print();

// Opens a file and serializes the object into it in binary format.

Stream stream = File.Open("data.xml", FileMode.Create);

BinaryFormatter formatter = new BinaryFormatter();

formatter.Serialize(stream, obj);

stream.Close();

// Empties obj.

obj = null;

// Opens file "data.xml" and deserializes the object from it.

stream = File.Open("data.xml", FileMode.Open);

formatter = new BinaryFormatter();

obj = (Person)formatter.Deserialize(stream);

stream.Close();

Console.WriteLine("");

Console.WriteLine("After deserialization the object contains: ");

obj.Print();

//Before serialization the object contains:

//Name = 'Bob'

//Idade = '11'

//Altura = '1,74159265'

//NonSerialized Endereco = 'Avenida Brasil'

//After deserialization the object contains:

//Name = 'Bob'

//Idade = '11'

//Altura = '1,74159265'

//NonSerialized Endereco = ''

Um tipo pode ter tantos atributos aplicados quanto necessário. Alguns atributos podem até ser aplicados várias vezes. Por exemplo, você pode usar o ConditionalAttribute para indicar ao compilador que uma chamada de método deve ser ignorada, a menos que uma opção específica do compilador seja especificada. O exemplo abaixo mostra como aplicar esse atributo.

[Conditional("CONDITION1"), Conditional("CONDITION2")]

staticvoid MyMethod() { }

Conforme mostrado na listagem, um atributo pode ter parâmetros. Assim como nos tipos regulares, esses parâmetros podem ser nomeados como opcionais. Os valores configurados para um atributo podem ser inspecionados posteriormente em tempo de execução.

O exemplo a seguir demonstra o uso de ConditionalAttribute. O exemplo supõe que a condição seja definida com a opção de compilador /define. Você pode obter resultados diferentes alterando a opção do compilador. Opcionalmente, você pode definir as condições usando pragmas no código de exemplo em vez de identificá-las como opções do compilador.

#define CONDITION1

//#define CONDITION2

#undef CONDITION2 // Ainda imprime o método

using System;

using System.Diagnostics;

namespace Conditional\_Atributo

{

classProgram

{

staticvoid Main(string[] args)

{

Console.WriteLine("Calling Method1");

Method1(3);

Console.WriteLine("Calling Method2");

Method2();

Console.WriteLine("Using the Debug class");

Debug.Listeners.Add(new ConsoleTraceListener());

Debug.WriteLine("DEBUG is defined");

Console.ReadKey();

}

[Conditional("CONDITION1")]

publicstaticvoid Method1(int x)

{

Console.WriteLine("CONDITION1 is defined");

}

[Conditional("CONDITION1"), Conditional("CONDITION2")]

publicstaticvoid Method2()

{

Console.WriteLine("CONDITION1 or CONDITION2 is defined");

}

}

}

/\*

Quando compilado como mostrado, o aplicativo produz a seguinte saída.

Calling Method1

CONDITION1 is defined

Calling Method2

CONDITION1 or CONDITION2 is defined

Using the Debug class

DEBUG is defined

Se comentar as declarações define:

Calling Method1

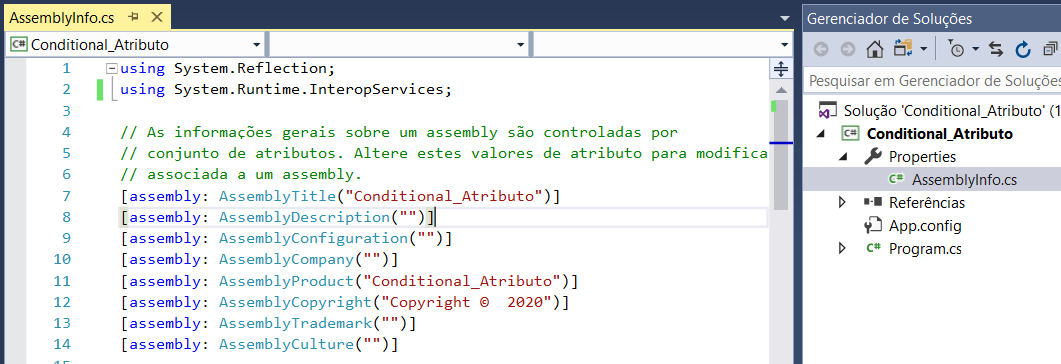
Calling Method2

Using the Debug class

DEBUG is defined

\*/

Um atributo também possui um destino específico ao qual se aplica. Pode ser um atributo aplicado a um assembly inteiro, uma classe, um método específico ou mesmo um parâmetro de um método.Se você olhar para o AssemblyInfo.cs de uma nova biblioteca de classes, poderá ver como o destino é explicitamente especificado.



Esses atributos são todos aplicados à montagem atual e descrevem alguns metadados sobre a montagem.

Atributos de leitura (Veja mais exemplos na seção Reflexion, mais abaixo)

A aplicação de um atributo não é útil se você não puder recuperá-lo. Felizmente, o .NET Framework oferece suporte para leitura de atributos por meio de um processo chamado reflexão. A classe System.Attribute, da qual todos os outros atributos herdam, define alguns métodos estáticos que podem ser usados para verificar se um atributo é aplicado e para obter a instância atual de um atributo para que você possa inspecioná-lo ainda mais.

Suponha que você queira verificar se uma classe tem o atributo Serializable aplicado. Você pode fazer isso chamando o método estático IsDefined no Attribute, como mostra abaixo.

if (Attribute.IsDefined(typeof(Person), typeof(SerializableAttribute)))

{

Console.WriteLine("Attribute.IsDefined as SerializableAttribute");

}

Você também pode recuperar a instância específica de um atributo para poder ver suas propriedades. O código abaixo mostra como você pode obter o ConditionalAttribute do exemplo anterior Conditional\_Atributo.

publicclassClassTeste

{

[ConditionalAttribute("CONDITION1")]

publicstaticvoid MetodoVoid(string txt)

{

Debug.WriteLine("DEBUG method.Invoke.MetodoVoid: " + txt);

}

}

//Call methods with reflection

MethodInfo[] methods = typeof(DLL\_Example.ClassTeste).GetMethods(BindingFlags.Static | BindingFlags.Public);

foreach (MethodInfo method in methods)

{

ConditionalAttribute conditional =

Attribute.GetCustomAttribute(method, typeof(ConditionalAttribute)) as ConditionalAttribute;

if (conditional != null)

{

//I can get the string of the condition;

string conditionString = conditional.ConditionString;

//If the method has a ConditionalAttribute

if (conditionString != null)

{

method.Invoke(null, newobject[] { conditionString });

}

}

}

Os métodos GetAttribute e GetAttributes têm várias sobrecargas para que você possa inspecionar atributos de um assembly, método, módulo ou parâmetro.

Atributo personalizado (Custom)

Os Attributes são um tipo de metadados para marcar o código C# (tipos, métodos, propriedades e assim por diante). Os Attributes podem ser usados com reflexão para consultar o código C# em tempo de execução, para geração de código ou no editor em tempo de compilação de várias maneiras (por exemplo, para ocultar/procurar controles de forma de janela na barra de ferramentas).

Em termos de programação, os atributos são classes C#, herdadas do tipo "Attribute". Ao criar um atributo customizado, é uma regra sufixar seu nome de classe com "Attribute". Uma classe de atributo customizado pode ter também propriedades podem ser declaradas. Somente propriedade pública com get; set; pode declarar na classe de atributo.

Um construtor pode ser declarado em uma classe de atributo personalizado da mesma maneira que é declarado em qualquer classe C#. O construtor pode conter um parâmetro que também pode ser um parâmetro opcional. Ao especificar um parâmetro opcional no construtor, podemos ter a vantagem de passar um valor ou descartá-lo. Esse recurso do uso do construtor no atributo customizado é útil ao fornecer informações opcionais.

Por exemplo, veja o trecho de código abaixo, onde MyCustomAttribute é o nome de um atributo personalizado que herda uma classe "Attribute".

//Tells MyCustomAttribute can only be use on a Class, Method and Property

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method | AttributeTargets.Property)]

classMyCustomAttribute : Attribute

{

publicstring Nome { get; set; }

publicint Idade { get; set; }

publicdecimal Altura { get; set; }

//public MyCustomAttribute()

//{

//}

public MyCustomAttribute(string name, int age = -1)

{

this.Nome = name;

this.Idade = age;

}

}

Os valores das propriedades podem ser atribuídos quando uma instância do atributo personalizado é anexada a qualquer elemento de código C#.

[MyCustom(Nome = "Ali Asad", Idade = 30)]

No parâmetro AttributeUsage, usamos AttributeTargets para restringir um atributo personalizado a ser aplicado apenas nessas enumerações (classe, método, propriedade etc.). Use um tubo de barra vertical '|' para adicionar mais de um AttributeTargets no construtor de AttributeUsage. A seguir, são apresentadas algumas enumerações de AttributeTargets usadas com freqüência, úteis para aplicar restrições no código C #.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| All | Class | Constructor | Delegate | Enum |
| Field | Interface | Method | Property | Struct |

Abaixo temos uma implementação onde exemplifica o uso do atributo personalizado:

namespace Reflection\_CustAttribute

{

classMyCustomAttribute : Attribute

{

}

[MyCustom] //Class, Marked with Custom Attribute

classPerson

{

//Property, Without Custom Attribute

publicint ID { get; set; }

[MyCustom] //Property, Marked with Custom Attribute

publicint Age { get; set; }

//Method, Without Custom Attribute

publicvoid Bye()

{

Console.WriteLine("Bye, world!");

}

[MyCustom] //Method, Marked with Custom Attribute

publicvoid Hi()

{

Console.WriteLine("Hi, world!");

}

}

//Class, Without Custom Attribute

classMachine

{

publicint ID { get; set; }

}

classProgram

{

staticvoid Main(string[] args)

{

Assembly assembly = Assembly.GetExecutingAssembly();

//Get all types that are marked with 'MyCustomAttribute'

var types = (from t in assembly.GetTypes()

where t.GetCustomAttributes<MyCustomAttribute>().Count() > 0

select t);

foreach (var type in types)

{

Console.WriteLine(type.Name);

//Get all properties which are marked with 'MyCustomAttribute'

var properties = (from p in type.GetProperties()

where p.GetCustomAttributes<MyCustomAttribute>().Count() > 0

select p);

foreach (var property in properties)

{

Console.WriteLine("\tProperty Name: {0}", property.Name);

}

//Get all methods which are marked with 'MyCustomAttribute'

var methods = (from m in type.GetMethods()

where m.GetCustomAttributes<MyCustomAttribute>().Count() > 0

select m);

foreach (var method in methods)

{

Console.WriteLine("\tMethod Name: {0}()", method.Name);

}

Console.ReadKey();

}

}

}

}

Person

Property Name: Age

Method Name: Hi()

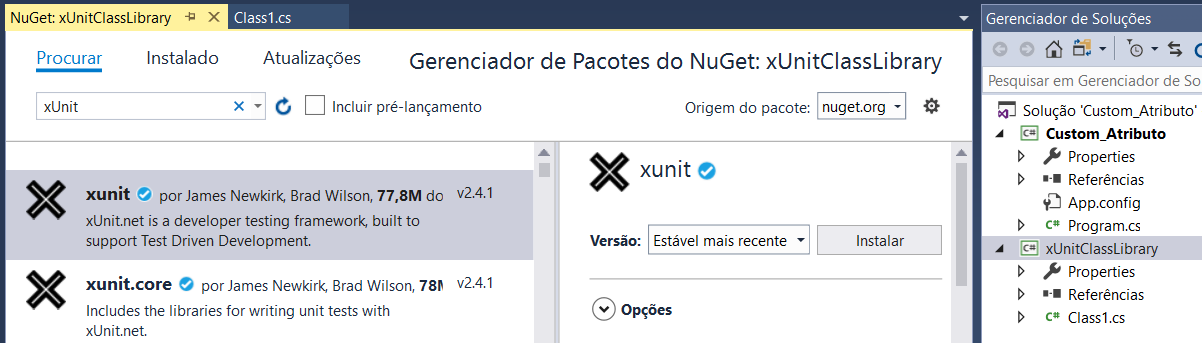
Ali Asad - -1

Observe que [MyCustom] = [MyCustomAttribute] porque o .NET Framework já sabe que "Attribute" é um sufixo, portanto, é um recurso do C# que permite ignorar o sufixo. No código acima, um atributo personalizado do nome "MyCustomAttribute" é criado. Este atributo [MyCustom] está marcado em uma classe, propriedade e método. No método principal, usando reflexão, todos os tipos, propriedades e métodos que foram marcados com "MyCustomAttribute" pode ser encontrado usando o método GetCustomAttributes<TAttribute>(). Os atributos que não estão marcados são ignorados e não são impressos no final.

Usando o método Attribute.GetCustomAttribute(), o valor armazenado nas propriedades de um atributo pode ser recuperado. Para recuperar uma instância de atributo customizado de uma classe, precisamos especificar que tipo de atributo é e a que tipo de classe ele está associado, usando o operador typeof ou o método getType().

Atributos xUnit

O xUnit (uma estrutura popular de teste de unidade) permite categorizar seus testes de unidade aplicando um atributo a eles.



Este pacote (xunit) é chamado de meta-pacote; isto é, é um pacote que existe apenas para que você possa obter referências a vários outros pacotes. Em particular, ele traz pacotes que incluem a estrutura de teste de unidade principal e a estrutura de asserção. Se você abrir o packages.config, verá todos os pacotes que são referenciados:

<?xmlversion="1.0"encoding="utf-8"?>

<packages>

<packageid="xunit"version="2.4.1"targetFramework="net461" />

<packageid="xunit.abstractions"version="2.0.3"targetFramework="net461" />

<packageid="xunit.analyzers"version="0.10.0"targetFramework="net461" />

<packageid="xunit.assert"version="2.4.1"targetFramework="net461" />

<packageid="xunit.core"version="2.4.1"targetFramework="net461" />

<packageid="xunit.extensibility.core"version="2.4.1"targetFramework="net461" />

<packageid="xunit.extensibility.execution"version="2.4.1"targetFramework="net461" />

</packages>

Quando você criou o projeto, o Visual Studio criou automaticamente um arquivo chamado Class1.cs e o abriu para você. Dentro desta classe, adicione alguns testes:

namespace xUnitClassLibrary

{

publicclassClass1

{

[Fact]

publicvoid PassingTest()

{

Assert.Equal(4, Add(2, 2));

}

[Fact]

publicvoid FailingTest()

{

Assert.Equal(5, Add(2, 2));

}

int Add(int x, int y)

{

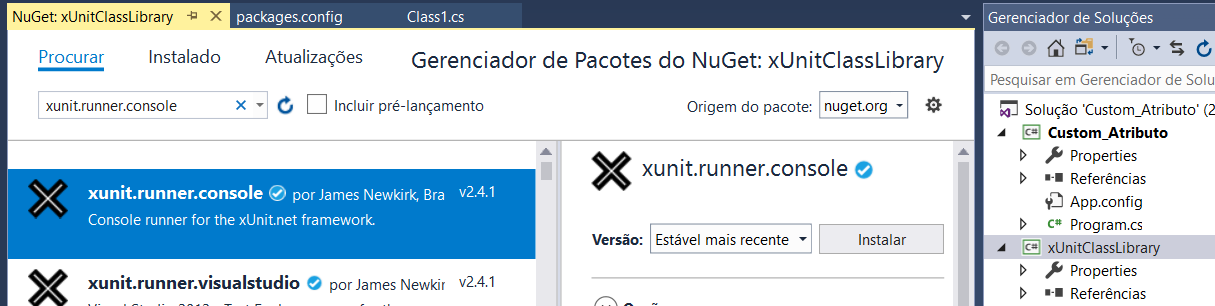
return x + y;

}

}

}

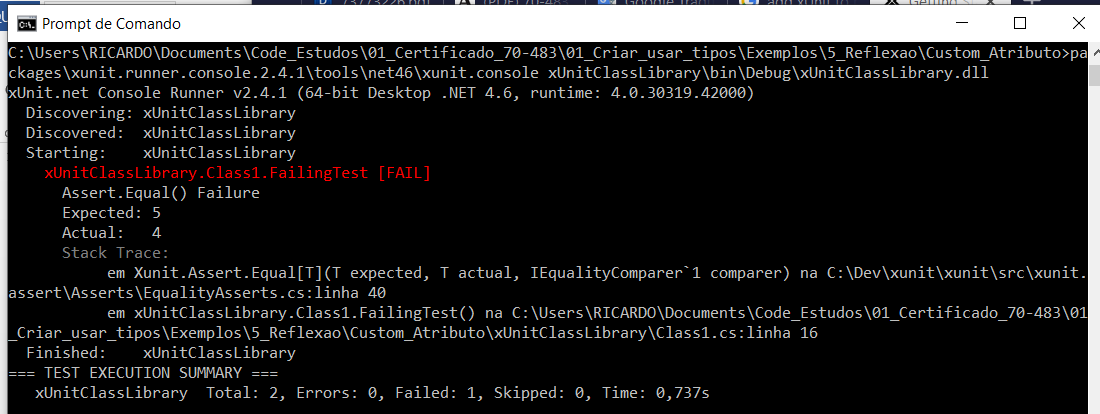
Compile a solução para garantir que o código seja compilado. Agora que você escreveu o primeiro teste, precisamos de uma maneira de executá-lo. Vamos instalar o pacote NuGet com o console runner.



Diferente do pacote anterior (que adicionou referências à estrutura de teste de unidade), esse pacote é conhecido como pacote em nível de solução. Em vez de ter assemblies para referência, ele adiciona algumas ferramentas na pasta da solução. Usaremos uma dessas ferramentas - o console runner- para executar seus testes de unidade.

Abra um prompt de comando ou a janela de comando do PowerShell. Na janela, navegue até a pasta raiz da sua solução. Para executar o console runner, use um comando como o destacado abaixo. Você deve ver uma saída semelhante a esta:

packages\xunit.runner.console.2.4.1\tools\net46\xunit.console xUnitClassLibrary\bin\Debug\xUnitClassLibrary.dll



O console runner tem várias opções de linha de comando, que incluem opções para paralelização, filtragem de teste e relatório de resultados. Para saber mais sobre as opções do corredor do console, execute o corredor do console sem opções de linha de comando. Agora que executamos seus primeiros testes de unidade, vamos apresentar mais uma maneira de escrever testes: usando teorias.

Escreva sua primeira teoria

Você deve estar se perguntando por que seus primeiros testes de unidade usam um atributo chamado [Fact] em vez de um com um nome mais tradicional como Test. O xUnit.net inclui suporte para dois tipos principais de testes de unidade: fatos e teorias. Ao descrever a diferença entre fatos e teorias, gostamos de dizer:

* Fatos são testes sempre verdadeiros. Eles testam condições invariantes.
* Teorias são testes que só são verdadeiros para um conjunto específico de dados.

Um bom exemplo deste teste de algoritmos numéricos. Digamos que você queira testar um algoritmo que determine se um número é ímpar ou não. Se você estiver escrevendo os testes do lado positivo (números ímpares), a inserção de números pares no teste pode causar falhas, e não porque o teste ou o algoritmo esteja errado. Vamos adicionar uma teoria aos nossos fatos existentes (incluindo alguns dados incorretos, para que possamos ver falhas):

//=== SECOND TEST EXECUTION SUMMARY ===

[Theory]

[InlineData(3)]

[InlineData(5)]

[InlineData(6)]

publicvoid MyFirstTheory(int value)

{

Assert.True(IsOdd(value));

}

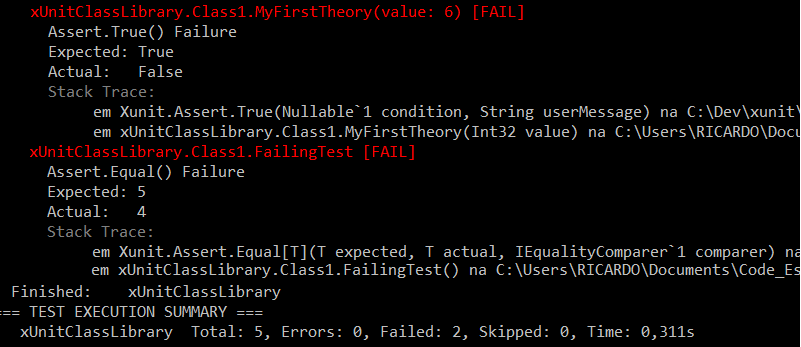
bool IsOdd(int value)

{

return value % 2 == 1;

}

Desta vez, quando compilamos e executamos os testes, vemos uma falha, para nossa teoria que foi dada 6:

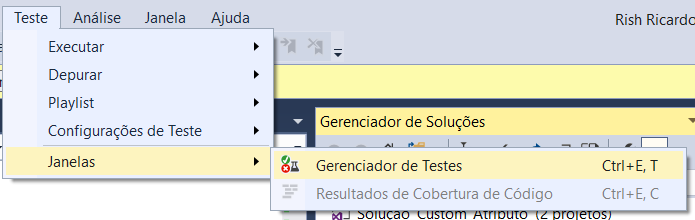


Embora tenhamos escrito apenas três métodos de teste, o corredor do console realmente executou 5 testes; isso ocorre porque cada teoria com seu conjunto de dados é um teste separado. Observe também que o corredor informa exatamente qual conjunto de dados falhou, porque inclui os valores dos parâmetros no nome do teste.

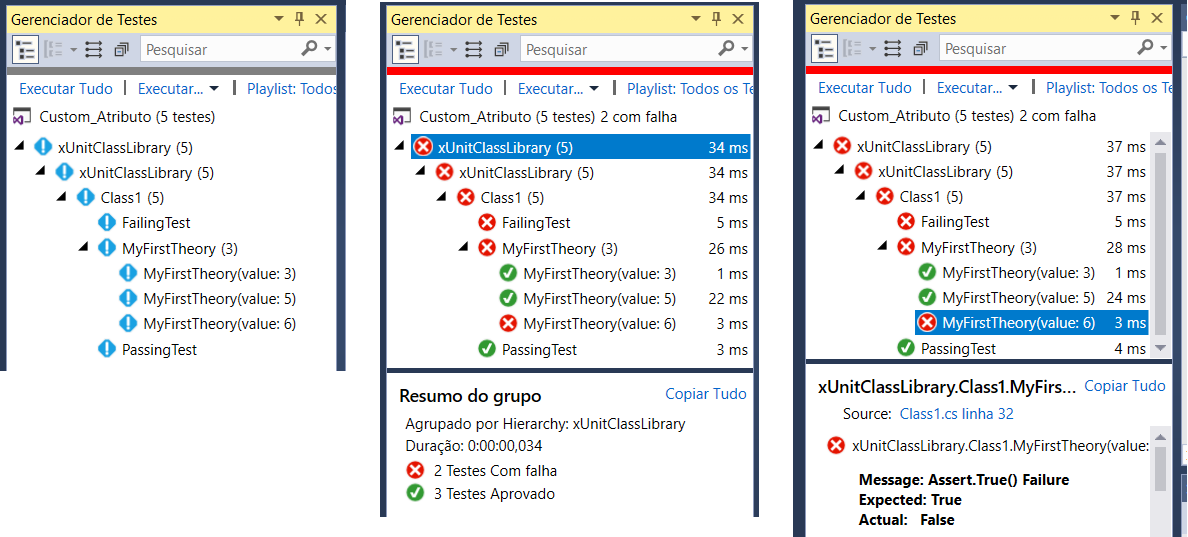
Se você possui o Visual Studio Community (ou uma versão paga do Visual Studio), pode executar os testes do xUnit.net no executor de testes interno do Visual Studio (denominado Test Explorer). Infelizmente, isso não inclui as edições Express do Visual Studio (você deve atualizar para o Community Edition).



Verifique se o Test Explorer está visível (vá para Teste> Windows> Test Explorer).



Toda vez que você compila seu projeto, o corredor descobre testes de unidade no seu projeto. Após um momento de descoberta, você deverá ver a lista de testes descobertos:



Você pode clicar em um teste com falha para ver a mensagem de falha e o rastreamento da pilha. Você pode clicar nas linhas de rastreamento da pilha para levá-lo diretamente para a linha de código com falha.

Às vezes, precisamos apenas executar um tipo específico de testes e não outros. Para poder fazer isso, precisamos de algum mecanismo para categorizar nossos testes. O xUnit possui um mecanismo interno para esse chamado Trait. O atributo Trait recebe dois argumentos. O primeiro é o nome real da categoria e o segundo argumento é a subseção que inclui a categoria. Por exemplo, temos a categoria Category como chave e, para o valor, temos "Unit Test" e "Integration Test".

[Fact]

[Trait("Category", "Unit Test")]

publicvoid PassingTest()

{

Assert.Equal(4, Add(2, 2)); //OK

}

[Fact]

[Trait("Category", "Integration Test")]

publicvoid FailingTest()

{

Assert.Equal(5, Add(2, 2)); //Failed

}

O uso do atributo Trait funciona perfeitamente, exceto que você precisa digitar a Categoria e o Valor manualmente sempre. Esse é um trabalho repetitivo e propenso a erros que você deseja evitar o máximo possível.

Expressões CodeDom

Além de inspecionar tipos em tempo de execução por meio de reflexão, o C# também oferece suporte para gerar código em tempo de execução. Uma maneira de fazer isso é através do CodeDOM. Você pode usar o CodeDOM para criar um gráfico de objeto que pode ser convertido em um arquivo de origem ou em um assembly binário que pode ser executado.

Os cenários de uso típicos para usar o CodeDOM envolvem a geração de código para ASP.NET, serviços da Web, assistentes de código ou designers. Com o CodeDOM você pode representar a estrutura lógica de um trecho de código independente da sintaxe específica do idioma usada. Por exemplo, você pode usar o CodeDOM para criar um arquivo de origem na sintaxe do Visual Basic, C# ou JScript.

Toda vez que você cria o mesmo código repetidamente com algumas pequenas modificações, ou seja, precisa automatizar as tarefas de codificação repetitivas ou aplicar padrões em seus projetos, vai utilizar o CodeDOM. Esta seção demonstrará como criar um arquivo de classe usando o CodeDOM que contém campos, propriedades e métodos. A estrutura típica de uma classe no C# contém os seguintes elementos:

* Arquivo de texto que contém o código da classe
* Conjunto de instruções using
* Declaração de namespace
* Declaração de nome de classe
* Conjunto de campos e propriedades
* Conjunto de métodos, que contêm lógica com estruturas em loop e expressões lógicas, como instruções if e switch.

O CodeDOM está localizado no namespace System.CodeDom. Você pode pensar no seu arquivo de origem como uma árvore com contêineres. Você tem um contêiner superior (chamado CodeCompileUnit) que contém outros elementos, como namespaces, classes, métodos e instruções individuais. Se você deseja gerar um aplicativo simples Hello World, é necessário criar um CodeCompileUnit, um namespace, uma classe e o método Main do seu programa que chamará Console.WriteLine como no exemplo abaixo.

staticvoid Main(string[] args)

{

CodeCompileUnit compileUnit = new CodeCompileUnit();

CodeNamespace myNamespace = new CodeNamespace("MyNamespace");

myNamespace.Imports.Add(new CodeNamespaceImport("System"));

CodeTypeDeclaration myClass = new CodeTypeDeclaration("MyClass");

CodeEntryPointMethod start = new CodeEntryPointMethod();

CodeMethodInvokeExpression cs1 = new CodeMethodInvokeExpression(

new CodeTypeReferenceExpression("Console"), "WriteLine", new CodePrimitiveExpression("Hello World!"));

compileUnit.Namespaces.Add(myNamespace);

myNamespace.Types.Add(myClass);

myClass.Members.Add(start);

start.Statements.Add(cs1);

CSharpCodeProvider provider = new CSharpCodeProvider();

using (StreamWriter sw = new StreamWriter("HelloWorld.cs", false))

{

IndentedTextWriter tw = new IndentedTextWriter(sw, " ");

provider.GenerateCodeFromCompileUnit(compileUnit, tw,

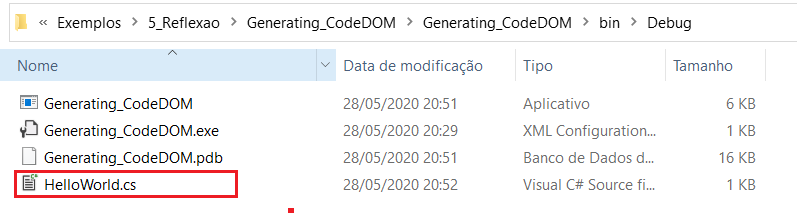
new CodeGeneratorOptions());

tw.Close();

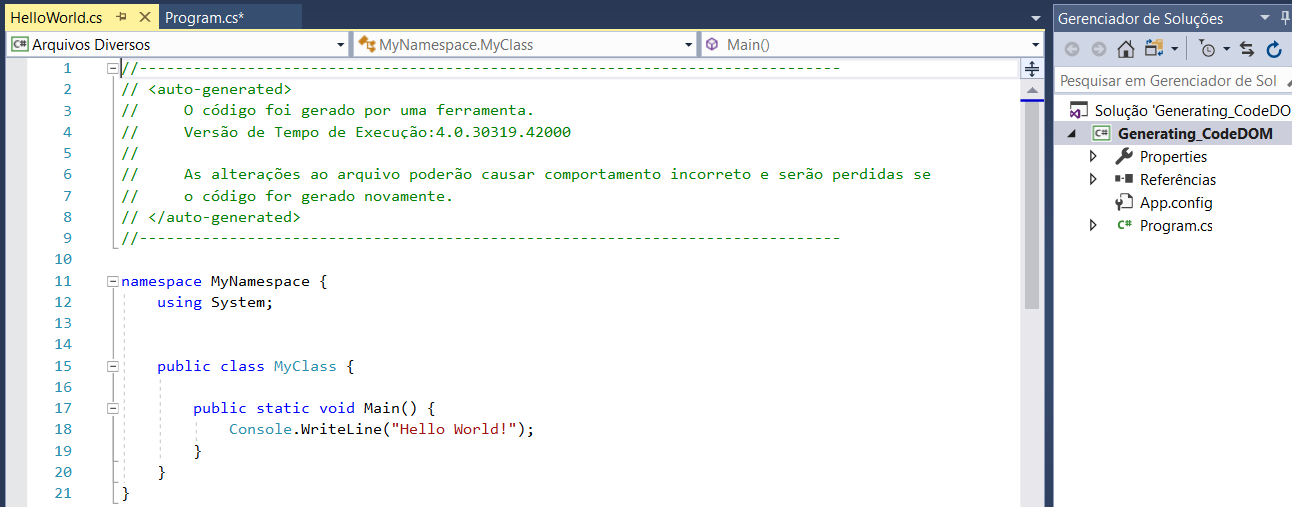
}

Console.ReadKey();

}



A saída gerada no HelloWorld.cs é mostrada abaixo:



Neste próximo exemplo, iremos examinar passo-a-passo como utilizar o CodeDOM para gerar a seguinte classe:

using System;

using System.CodeDom;

using System.CodeDom.Compiler;

using System.IO;

using System.Reflection;

classCalculator

{

privatedouble \_x;

privatedouble \_y;

publicdouble X

{

get { returnthis.\_x; }

set { this.\_x = value; }

}

publicdouble Y

{

get { returnthis.\_y; }

set { this.\_y = value; }

}

publicdouble Divide()

{

if (this.Y == 0)

return 0;

else

returnthis.X / this.Y;

}

publicdouble Exponent(double power)

{

return Math.Pow(this.X, power);

}

}

Essa é uma classe simples chamada Calculator, que está no espaço de nome Reflexão, contém dois campos, duas propriedades e dois métodos. As seções a seguir demonstram quais classes no CodeDOM você deve usar para criar a classe dinamicamente.

CodeCompileUnit

A classe CodeCompileUnit é a classe de nível superior que é o contêiner para todos os outros objetos da classe que você deseja gerar. Pense nisso como a classe que representa o arquivo que contém seu código. O código a seguir é usado para criar uma instância da classe CodeCompileUnit:

CodeCompileUnit codeCompileUnit = new CodeCompileUnit();

CodeNamespace e CodeNamespaceImport

O próximo passo é adicionar o namespace. A classe CodeNamspace é usada para representar o namespace . O construtor usa o namespace como parâmetro.

CodeNamespace codeNamespace = new CodeNamespace("Reflection");

Agora que você possui um namespace, pode anexar as instruções using. Normalmente, quando você cria um arquivo de classe, as instruções using estão acima da declaração do namespace, mas ainda funcionam se você as adicionar após o namespace. A classe CodeNamespaceImport é usada para definir o namespace que você gostaria de importar. Em C# você usa a palavra-chave using, mas no VB.NET você usaria a palavra-chave imports. Ao usar o CodeDOM, você não precisa se preocupar com a palavra-chave correta.

codeNamespace.Imports.Add(new CodeNamespaceImport("System"));

codeNamespace.Imports.Add(new CodeNamespaceImport("System.CodeDom"));

codeNamespace.Imports.Add(new CodeNamespaceImport("System.CodeDom.Compiler"));

codeNamespace.Imports.Add(new CodeNamespaceImport("System.IO"));

codeNamespace.Imports.Add(new CodeNamespaceImport("System.Reflection"));

CodeTypeDeclaration

O próximo passo é declarar a classe. Isso é feito usando a classe CodeTypeDeclaration.

CodeTypeDeclaration targetClass = new CodeTypeDeclaration("Calculator");

targetClass.IsClass = true;

targetClass.TypeAttributes = TypeAttributes.Public;

codeNamespace.Types.Add(targetClass);

codeCompileUnit.Namespaces.Add(codeNamespace);

O código anterior cria uma instância da classe CodeTypeDeclaration e define o atributo IsClass como true, que informa ao .NET Framework para gerar uma declaração de classe. A propriedade TypeAttributes permite definir atributos como público, privado e estático. Eles podem ser combinados usando o operador bit a bit (|). Depois que a classe é definida, você precisa adicioná-lo à coleção Types do namespace. O código anterior produz a seguinte saída:

classCalculator

{

}

CodeMemberField

O próximo passo é adicionar os campos à turma. Isso é feito usando a classe CodeMemberField. Você simplesmente cria uma instância da classe e define sua propriedade Name, define a propriedade Type e a adiciona à coleção Members do objeto CodeTypeDeclaration. O código a seguir cria dois campos, \_x e \_y, ambos declarados como um duplo:

//Create the fields

CodeMemberField xField = new CodeMemberField();

xField.Name = "\_x";

xField.Type = new CodeTypeReference(typeof(double));

targetClass.Members.Add(xField);

CodeMemberField yField = new CodeMemberField();

yField.Name = "\_y";

yField.Type = new CodeTypeReference(typeof(double));

targetClass.Members.Add(yField);

O código anterior produz a seguinte saída:

privatedouble \_x;

privatedouble \_y;

CodeMemberProperty

O próximo passo é criar as propriedades para os campos xe y. Você usa uma classe CodeMemberProperty para criar uma propriedade e gerar os métodos get e set. O código a seguir cria as propriedades X e Y na classe Calculadora:

//Create the properties

CodeMemberProperty xProperty = new CodeMemberProperty();

xProperty.Attributes = MemberAttributes.Public | MemberAttributes.Final;

xProperty.Name = "X";

xProperty.HasGet = true;

xProperty.HasSet = true;

xProperty.Type = new CodeTypeReference(typeof(System.Double));

xProperty.GetStatements.Add(new CodeMethodReturnStatement(new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), "\_x")));

xProperty.SetStatements.Add(new CodeAssignStatement(new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), "\_x"), new CodePropertySetValueReferenceExpression()));

targetClass.Members.Add(xProperty);

CodeMemberProperty yProperty = new CodeMemberProperty();

yProperty.Attributes = MemberAttributes.Public | MemberAttributes.Final;

yProperty.Name = "Y";

yProperty.HasGet = true;

yProperty.HasSet = true;

yProperty.Type = new CodeTypeReference(typeof(System.Double));

yProperty.GetStatements.Add(new CodeMethodReturnStatement(new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), "\_y")));

yProperty.SetStatements.Add(new CodeAssignStatement(new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), "\_y"), new CodePropertySetValueReferenceExpression()));

targetClass.Members.Add(yProperty);

A classe CodeMemberProperty possui duas propriedades (HasGet e HasSet) que você precisa definir como true para que o gerador de código possa criar os acessadores Get e Set. A propriedade de coleção GetStatements é usada para adicionar o código ao acessador Get. Neste exemplo, o método Get retorna o campo this.x. A classe CodeThisReferenceExpression é usada porque em C # você usa isso; no VB você me usa. O gerador de código sabe qual palavra-chave usar ao gerar o código. A propriedade de coleção SetStatements contém o código para definir o campo this.x. Nesse caso, você precisa criar um CodeAssignStatement junto com o CodePropertySetValueReferenceExpression. O código precedente produz a seguinte saída:

publicdouble X

{

get { returnthis.\_x; }

set { this.\_x = value; }

}

publicdouble Y

{

get { returnthis.\_y; }

set { this.\_y = value; }

}

CodeMemberMethod

O próximo passo é criar o método Divide. Para criar métodos usando o CodeDOM, você precisa usar a classe CodeMemberMethod. O código a seguir cria uma instância da classe CodeMemberMethod, nomeia o método Divide, define o tipo de retorno como duplicado e define seus atributos como público e final. Se você deseja definir outros atributos, como estático, virtual ou novo, use o operador bit a bit para concatenar os atributos.

CodeMemberMethod divideMethod = new CodeMemberMethod();

divideMethod.Name = "Divide";

divideMethod.ReturnType = new CodeTypeReference(typeof(double));

divideMethod.Attributes = MemberAttributes.Public | MemberAttributes.Final;

Agora que a assinatura do método está definida, você precisa criar o código para o corpo do método. Ométodo Divide verifica se a propriedade Y é 0 e retorna 0 ou o quociente. Se a lógica for criada usando a classe CodeConditonStatement.

CodeConditionStatement ifLogic = new CodeConditionStatement();

ifLogic.Condition = new CodeBinaryOperatorExpression(new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), yProperty.Name), CodeBinaryOperatorType.ValueEquality,

new CodePrimitiveExpression(0));

ifLogic.TrueStatements.Add(new CodeMethodReturnStatement(new CodePrimitiveExpression(0)));

ifLogic.FalseStatements.Add(new CodeMethodReturnStatement(

new CodeBinaryOperatorExpression(

new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), xProperty.Name),

CodeBinaryOperatorType.Divide,

new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), yProperty.Name))));

divideMethod.Statements.Add(ifLogic);

targetClass.Members.Add(divideMethod);

Como você pode ver, a classe CodeConditonStatement possui uma propriedade Condition que é uma classe CodeBinaryOperatorExpression. Esta classe é usada para criar uma expressão binária. Neste exemplo, a expressão é igual a (this.Y == 0). A classe CodeBinaryOperatorExpression também possui uma propriedade TrueStatements e FalseStatements que permite criar o código que será gravado para as condições verdadeiras e falsas. O código anterior cria a seguinte saída:

publicdouble Divide()

{

if (this.Y == 0)

return 0;

else

returnthis.X / this.Y;

}

CodeParameterDeclarationExpression e CodeMethodInvokeExpression

O próximo passo é criar o método Exponent. Esse método pega um parâmetro chamado power e retorna this.Y aumentado para esse power.

CodeMemberMethod exponentMethod = new CodeMemberMethod();

exponentMethod.Name = "Exponent";

exponentMethod.ReturnType = new CodeTypeReference(typeof(double));

exponentMethod.Attributes = MemberAttributes.Public | MemberAttributes.Final;

CodeParameterDeclarationExpression powerParameter = new CodeParameterDeclarationExpression(typeof(double), "power");

exponentMethod.Parameters.Add(powerParameter);

CodeMethodInvokeExpression callToMath = new CodeMethodInvokeExpression(

new CodeTypeReferenceExpression("System.Math"),

"Pow", new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), xProperty.Name), new CodeArgumentReferenceExpression("power"));

exponentMethod.Statements.Add(new CodeMethodReturnStatement(callToMath));

targetClass.Members.Add(exponentMethod);

Você usa a classe CodeParameterDeclarationExpression para criar o parâmetro power. A classe CodeMethodInvokeExpression é usada para chamar um método e passar um parâmetro para o método. O código anterior produz a seguinte saída:

publicdouble Exponent(double power)

{

return Math.Pow(this.X, power);

}

CodeDOMProvider

O último passo é gerar o arquivo de classe. Você usa a classe CodeDOMProvider para criar o arquivo em C #, VB ou JScript. Essa classe possui um método chamado GenerateCodeFromCompileUnit que usa uma classe CodeCompileUnit, TextWriter e CodeGeneratorOptions como parâmetros. A classe CodeGeneratorOptions possui propriedades que permitem controlar a formatação do seu código gerado automaticamente. O exemplo a seguir informa ao compilador para usar o espaçamento de linha única entre as declarações do membro. Definir a propriedade BracingStyle como "C" coloca os colchetes, {}, em linhas separadas.

CodeDomProvider provider = CodeDomProvider.CreateProvider("CSharp");

CodeGeneratorOptions options = new CodeGeneratorOptions();

options.BlankLinesBetweenMembers = false;

options.BracingStyle = "C";

using (StreamWriter sourceWriter = new StreamWriter(@"c:\CodeDom\Calculator." + provider.FileExtension))

{

provider.GenerateCodeFromCompileUnit(codeCompileUnit, sourceWriter, options);

}

Expressões Lambda

Funções Lambda foram introduzidas no C# 3.0. Você pode pensar em uma expressão lambda como um método compacto para criar um método anônimo. O método anônimo é essencialmente um método sem nome, pode escrever um método anônimo para não ter o trabalho de criar a assinatura em métodos pequenos e usados em escopo limitado. Além disso, você pode passar um método anônimo para outros métodos para alterar dinamicamente como esses métodos se comportam. Este conceito é extremamente importante para entender antes de abordar o conceito de LINQ. Expressões lambda são usadas em qualquer lugar no LINQ.

Delegates e Lambda

Antes de explorar as expressões lambda, comece com o básico. Um delegado é um tipo que faz referência a um método. Ao declarar um delegado, você especifica a assinatura do método que deseja referenciar. Por exemplo, crie uma nova classe chamada LambdaExpressione adicione o seguinte método que pega um parâmetro de string e o grava na janela do console:

publicclassLambdaExpression

{

publicstaticvoid WriteToConsoleForward(string stringToWrite)

{

Console.WriteLine("This is my string: {0}", stringToWrite);

}

}

Se você quiser fazer referência a esse método, primeiro crie um delegado com a mesma assinatura. Observe que o tipo de retorno é nulo e o tipo do parâmetro é uma string que corresponde à assinatura do método WriteToConsoleForward. Agora que você tem um representante, você precisa associar uma variável desse tipo ao método

delegatevoidMyFirstDelegate(string s);

staticvoid Main(string[] args)

{

MyFirstDelegate myFirstDelegate = new MyFirstDelegate(LambdaExpression.WriteToConsoleForward);

myFirstDelegate("Hello World"); // This is my string: Hello World

}

A variável myFirstDelegate contém essencialmente uma referência ao método. Agora você pode chamar o método usando a variável myFirstDelegate e passando um parâmetro. Agora crie um segundo método na classe LambdaExpression que usa uma string como parâmetro e grava a string para trás no console.

publicstaticvoid WriteToConsoleBackwards(string stringToWrite)

{

char[] charArray = stringToWrite.ToCharArray();

Array.Reverse(charArray);

Console.WriteLine(string.Format("Write backwards: {0}", newstring(charArray)));

}

Ambos os métodos têm a mesma assinatura, portanto, você pode criar um único representante para fazer referência a qualquer método. Agora crie outro método chamadoWriteToConsole que usa o delegado como parâmetro e chame o método.

delegatevoidMyFirstDelegate(string s);

staticvoid Main(string[] args)

{

var myLocalString = "Hello World";

WriteToConsole(LambdaExpression.WriteToConsoleForward, myLocalString); // Hello World

WriteToConsole(LambdaExpression.WriteToConsoleBackwards, myLocalString); // dlroW olleH

}

staticvoid WriteToConsole(MyFirstDelegate myDelegate, string stringToWrite)

{

myDelegate(stringToWrite);

}

Você também pode adicionar ou remover um método de uma lista de chamadas usando o operador de atribuição de incremento (+ ou +=) e decremento (- ou -=).

MyFirstDelegate myFirstDelegate2 = LambdaExpression.WriteToConsoleForward;

myFirstDelegate2 += LambdaExpression.WriteToConsoleBackwards;

myFirstDelegate2(myLocalString);

Quando você atribui um método a um delegado, a assinatura do método não precisa corresponder exatamente ao delegado. Isso é chamado de covariância e contravariância.

* **Covariância**: permite que você tenha um método com um tipo de retorno mais derivado do que o tipo de retorno do delegado. Portanto, o tipo de retorno do delegado pode ser uma classe base, e o tipo de retorno do método pode ser um tipo que herda da classe base.
* **Contravariância**: permite tipos de parâmetro menos derivados que os tipos de parâmetro do delegado. Portanto, os parâmetros do delegado podem ser uma classe base, mas os parâmetros do método podem ser uma classe derivada da classe base.

O exemplo abaixo mostra um exemplo de covariância.

publicdelegate TextWriter CovarianceDel();

staticvoid Main(string[] args)

{

CovarianceDel del\_covarinace;

del\_covarinace = MethodStream;

del\_covarinace = MethodString;

var retorno = del\_covarinace();

}

publicstatic StreamWriter MethodStream() { returnnull; }

publicstatic StringWriter MethodString() { returnnull; }

Como o StreamWriter e o StringWriter herdam do TextWriter, você pode usar o CovarianceDel com os dois . TextWriter é a classe base abstrata de StreamWriter e StringWriter, que gravam caracteres em fluxos e cadeias, respectivamente. Crie uma instância de TextWriter para gravar um objeto em uma cadeia de caracteres, gravar cadeias de texto em um arquivo ou serializar XML. Você também pode usar uma instância de TextWriter para gravar texto em um repositório de backup personalizado usando as mesmas APIs que você usaria para uma cadeia de caracteres ou um fluxo, ou para adicionar suporte para formatação de texto.

Um exemplo de contravariância pode ser visto abaixo.

publicdelegatevoidContravarianceDel(StreamWriter tw);

staticvoid Main(string[] args)

{

ContravarianceDel del\_contravariance = DoSomething;

del\_contravariance((StreamWriter)retorno);

Console.ReadKey();

}

publicstaticvoid DoSomething(TextWriter tw) { }

Como o método DoSomething pode funcionar com um TextWriter, certamente também pode funcionar com um StreamWriter. Por causa da contravariância, você pode chamar o delegado e passar uma instância do StreamWriter para o método DoSomething.

Métodos anônimos e Lambda

Os métodos anônimos são semelhantes aos delegados, exceto que você não precisa criar o método. Você ainda cria o delegado, mas pode atribuir o método tudo dentro da mesma linha de código. Uma diferença entre um método anônimo e um delegado é que você pode fazer referência a variáveis locais que não são passadas como parâmetros. Por exemplo, o exemplo a seguir cria uma variável delegada chamada forward que não possui parâmetrose faz referência ao corpo de um método. O método pode ter quantas linhas você desejar. Em seguida, ele cria uma variável local e um método anônimo que usa a variável.

delegatevoidMyAnonymousMethod();

staticvoid Main(string[] args)

{

var myLocalString = "Hello World";

MyAnonymousMethod forward = delegate ()

{

Console.WriteLine(string.Format("This is my AnonymousMethod: {0}", myLocalString));

};

forward();

}

As expressões Lambda permitem criar uma função anônima usando sintaxe abreviada. Considere o seguinte:

delegatedoublesquare(double x);

staticvoid Main(string[] args)

{

square myLambdaExpression = x => x \* x;

Console.WriteLine("X squared is {0}", myLambdaExpression(5)); // 25

}

A expressão lambda é x => x \* x. Ao ler o código, você diria que x vai para (goes to) x vezes x. O => é chamado “goes to” operador. O lado esquerdo do operador vai para os parâmetros de entrada do seu método. O corpo do seu método vai no lado direito do vai para o operador. Nesse caso, o método pode enquadrar qualquer número que seja passado para o método. Se você precisar passar vários parâmetros, use a seguinte sintaxe:

delegateboolGreaterThan(double x, double y);

staticvoid Main(string[] args)

{

GreaterThan gt = (x, y) => x > y;

Console.WriteLine("Is 6 greater than 5. {0}", gt(6, 5)); // True

}

Ao trabalhar com lambdas, você também precisará conhecer os tipos Func<> e Action. Esses tipos genéricos foram adicionados para ter alguns tipos de delegados predefinidos no .NET Framework. Você usa Action quando tem um delegado que não retorna um valor e Func quando você deseja retornar um valor. Ambos podem receber até 16 argumentos de tipo no .NET Framework 4.0.Ao combinar lambda e o tipo Func, você pode criar facilmente um tipo que retorne a soma de dois números inteiros, conforme mostrado abaixo.

Func<int, int, int> addFunc = (x, y) => x + y;

Console.WriteLine(addFunc(2, 3)); // 5

O lambda é do tipo Func<int, int, int>, o que significa que ele recebe dois argumentos inteiros e retorna um int como resultado. A notação estranha => pode ser lida como "se torna" ou "para qual". O tipo addFunc pode ser lido como "x, y se torna x + y".

Instruções Lambdas (Statement Lambdas)

Quando o método contém apenas uma única expressão, é chamado de expressão lambda. Quando você precisa de várias instruções no corpo do método, isso é chamado de Instruções lambdas. Instruções Lambdas estão entre colchetes {}. A seguir está uma expressão lambda para o método WriteToConsoleBackwardcom uma variável local:

MyAnonymousMethod myFirstDelegate2 = () =>

{

char[] charArray = myLocalString.ToCharArray();

Array.Reverse(charArray);

Console.WriteLine(string.Format("This is my string backwards: {0}", newstring(charArray)));

};

myFirstDelegate2();

Usando uma instrução de expressão Lambda com um parâmetro

delegatevoidMyFirstDelegate(string s);

MyFirstDelegate myFirstDelegate3 = s =>

{

char[] charArray = s.ToCharArray();

Array.Reverse(charArray);

Console.WriteLine(string.Format("This is my string backwards: {0}", newstring(charArray)));

};

myFirstDelegate3(myLocalString);

Você também pode usar uma expressão lambda para passar uma função para um método. A seguir, uma expressão lambda para chamar o método WriteToConsole:

delegatevoidMyFirstDelegate(string s);

staticvoid Main(string[] args)

{

var myLocalString = "Hello World";

WriteToConsole(x => Console.WriteLine("Esta é minha string {0}", x), myLocalString);

Console.ReadKey();

}

staticvoid WriteToConsole(MyFirstDelegate myDelegate, string stringToWrite)

{

myDelegate(stringToWrite);

}

Como você pode ver, a sintaxe requer menos digitação à medida que você passa de delegar, para função anônima e para expressões lambda.

Árvores de expressão

Ao usar lambdas, você encontrará árvores de expressão, que são representações de código em uma estrutura de dados semelhante a uma árvore. Assim como o CodeDom pode representar o código de maneira semelhante a uma árvore, as árvores de Expressão podem fazer o mesmo; eles também podem ser usados ​​para gerar código.

Uma árvore de expressão descreve o código em vez de ser o próprio código. As árvores de expressão são muito usadas no LINQ. Ao usar o Linq To Entities para consultar um banco de dados, a consulta não é executada (como é o caso no Linq To Objects). Em vez disso, uma árvore de expressão descreve a consulta. Posteriormente, essa árvore de expressão é traduzida em uma instrução SQL que pode ser enviada ao banco de dados.

O namespace System.Linq.Expressions contém todos os tipos necessários para criar uma expressão. Você tem expressões para chamar um método e criar um novo objeto ou mesmo operações básicas, como adição ou subtração. O exemplo Hello World do CodeDOM também pode ser criado como uma árvore de expressão

BlockExpression blockExpr = Expression.Block(

Expression.Call(null, typeof(Console).GetMethod("Write",

new Type[] { typeof(String) }), Expression.Constant("Hello ")),

Expression.Call(null, typeof(Console).GetMethod("WriteLine",

new Type[] { typeof(String) }), Expression.Constant("World!"))

);

Expression.Lambda<Action>(blockExpr).Compile()(); // Hello World!

A expressão é criada primeiro com uma chamada para Console.Write e Console.WriteLine. Após a construção, a expressão é compilada em uma ação (porque não retorna nada) e executada.

As coisas começam a ficar mais complexas quando sua função lambda começa a se referir a variáveis declaradas fora da expressão lambda (ou a esta referência). Normalmente, quando o controle sai do escopo de uma variável, a variável não é mais válida. Mas e se um delegado se referir a uma variável local e depois retornar ao método de chamada? Agora, o delegado tem uma vida útil mais longa que a variável. Para corrigir isso, o compilador gera código que torna a vida útil da variável capturada pelo menos contanto que o delegado que vive mais tempo. Isso é chamado de encerramento (closure).

Reflection

O namespace System.Reflection contém várias classes que permitem criar e usar dinamicamente tipos, métodos e propriedades de um aplicativo em execução. Em termos de processamento, a reflexão custa muita energia do processador porque, ao usar metadados, ela reverte todos os dados binários em dados legíveis.Ou seja, o Reflection converte informações binárias (de baixo nível) em linguagem legível por humanos (alto nível) e permite que humanos (desenvolvedores) manipulem os dados e o comportamento de um aplicativo em tempo de execução.

O Reflection é mais lenta que a execução normal do código estático. No entanto, pode oferecer uma flexibilidade que o código estático não pode fornecer. O exemplo mais básico de reflexão é obter o tipo atual de um objeto que você possui, você pode usar o método GetType () em uma instância do tipo:

int i = 42;

System.Type type = i.GetType();

Ou pode usar a palavra-chave typeof () e passar o nome do tipo:

System.Type myType = typeof (int);

Isso retorna System.Int32 como o tipo de int. System.Type é uma classe no .NET Framework que você pode usar para obter todos os tipos de metadados sobre qualquer tipo. A classe System.Type representa uma classe, interface, matriz, tipo de valor, enumeração, parâmetro, definições de tipo genérico e tipos genéricos construídos abertos ou fechados. Na maioria das vezes, você geralmente usa um Type para obter informações sobre uma classe contida em uma montagem.

//AssmeblyQualifiedName: System.Int32, mscorlib, Version = 4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089

Console.WriteLine("AssmeblyQualifiedName: {0}", myType.AssemblyQualifiedName);

Console.WriteLine("FullName: {0}", myType.FullName); //FullName: System.Int32

Console.WriteLine("IsValueType: {0}", myType.IsValueType); //IsValueType: True

Console.WriteLine("Name: {0}", myType.Name); //Name: Int32

Console.WriteLine("Namespace: {0}", myType.Namespace); //Namespace: System

Agora, suponha que você deseja criar um sistema de plug-in e que você tenha um diretório em seu sistema que contenha todos os plug-ins, você precisa de alguma maneira de encontrar plug-ins, obter algumas informações e executá-los e isso é impossível sem Reflection. Você pode usar Reflection para criar uma interface IPlugin personalizada que expõe membros que fornecem informações sobre o plug-in e a capacidade de carregá-lo.

publicinterfaceIPlugin

{

string Name { get; }

string Description { get; }

bool Load(Program application);

}

Agora que você possui uma interface base personalizada, pode criar um plug-in herdando desta interface com uma classe de plug-in específica, como mostra abaixo.

publicclassMyPlugin : IPlugin

{

publicstring Name

{

get { return"MyPlugin"; }

}

publicstring Description

{

get { return"My Sample Plugin"; }

}

publicbool Load(Program application)

{

returntrue;

}

}

Usando a reflexão, agora você pode inspecionar uma montagem e verificar se há plug-ins disponíveis. Os tipos que você recebe podem ser usados para criar uma instância do plug-in e usá-lo. O exemplo abaixo mostra como obter todos os plug-ins de um assembly com uma consulta LINQ e construí-los.

Assembly assembly = Assembly.GetExecutingAssembly();

Assembly pluginAssembly = Assembly.Load("Reflexion\_IPlugin");

Console.WriteLine(assembly.FullName == pluginAssembly.FullName);// True

var DLL\_type = from type in assembly.GetTypes()

wheretypeof(IPlugin).IsAssignableFrom(type) && !type.IsInterface

select type;

foreach (Type pluginType in DLL\_type)

{

IPlugin plugin = Activator.CreateInstance(pluginType) as IPlugin;

if (plugin != null)

Console.WriteLine(plugin.Name + " - " + plugin.Description); // MyPlugin - My Sample Plugin

}

O namespace System.Reflection contém várias classes que podem ser usadas para ler metadados ou invocar dinamicamente o comportamento de um tipo. A Tabela abaixo lista algumas das classes usadas com freqüência no namespace System.Reflection.

|  |  |
| --- | --- |
| **Tipo** | **Descrição** |
| Assembly | Representa um arquivo DLL ou EXE e contém propriedades para o nome do assembly, classes, módulos e outro aplicativo de tempo de execução do idioma de metadados. |
| EventInfo | Representa um evento definido em sua classe e contém propriedades como o nome do evento. |
| FieldInfo | Representa um campo definido em sua classe e contém propriedades como se o campo é público ou privado. |
| MemberInfo | Abstrai os metadados sobre uma classe e pode representar um evento, um campo e assim por diante. |
| MethodInfo | Representa um método definido em sua classe e pode ser usado para invocar o método. |
| Module | O módulo é um arquivo que compõe a montagem. Geralmente, esse é um arquivo DLL ou EXE. |
| ParameterInfo | Representa uma declaração de parâmetro para um método ou construtor. Isso permite determinar o tipo de parâmetro, seu nome e outras propriedades. |
| PropertyInfo | Representa uma propriedade definida em sua classe e contém propriedades como o nome e o tipo da propriedade |

Obter informações do assembly (propriedades, métodos, eventos, interfaces, campos)

A classe System.Reflection.Assembly contém métodos e propriedades usadas para ler e manipular informações de um assembly em tempo de execução.

namespace Reflection\_DemoAssembly

{

classA { publicint Random { get; set; } }

classB : A { }

classProgram

{

publicint Age { get; set; }

publicstring Name { get; set; }

staticvoid Main(string[] args)

{

//Get current loaded assembly

Assembly assembly = Assembly.GetExecutingAssembly();

//Get Full Name of the current Assembly

string assemblyName = assembly.FullName;

//Getall types defined in an assembly

Type[] types = assembly.GetTypes();

//Get information of each type

foreach (Type type in types)

{

//Return name of a type and its base type

Console.WriteLine("Type Name:{0}, Base Type:{1}", type.Name, type.BaseType);

Lista\_Propriedades(type);

}

AssemblyName[] referencedAssemblyNames = assembly.GetReferencedAssemblies();

foreach (AssemblyName assemblyRef in referencedAssemblyNames)

{

Console.WriteLine(string.Format("Assembly Name: {0}", assemblyRef.Name));

Console.WriteLine(string.Format("Assembly Version: {0}", assemblyRef.Version));

}

//Reflection\_DemoAssembly, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

Console.WriteLine(assemblyName);

Console.ReadKey();

}

publicstaticvoid Lista\_Propriedades(Type type)

{

//Get all properties defined in a type

PropertyInfo[] properties = type.GetProperties();

foreach (PropertyInfo property in properties)

{

Console.WriteLine("\t{0} has {1} type",

property.Name, property.PropertyType);

}

}

}

}

Quando um aplicativo está em execução, ele obtém os metadados do assembly atual e lê seu nome completo. O nome completo da montagem sempre contém 4 partes, que descrevem o nome da montagem, o número da versão da montagem e a cultura da montagem, e informa se a montagem tem um nome forte (se tiver um token de chave pública associado a ela).

Uma montagem pode estar em um dos três contextos ou em nenhum contexto:

1. Contexto de carga, que contém os conjuntos encontrados pela análise. Sondagem é o processo de procurar no GAC, no armazenamento do conjunto host, na pasta do conjunto de saída ou na pasta bin privada do conjunto em execução para encontrar o conjunto.
2. Contexto de carregamento. Isso contém os assemblies localizados no caminho passado para o método LoadFrom.
3. Contexto somente de reflexão, que contém os assemblies carregados com os métodos ReflectionOnlyLoad e ReflectionOnlyLoadFrom

A classe Type contém métodos e propriedades usadas para ler e editar valores de um tipo especificado. No exemplo acima, assembly.GetTypes() retorna uma matriz de tipos não estáticos. Usando propriedades como type.Name e type.BaseType, podemos obter o nome de um tipo e seu tipo base, respectivamente.

Os métodos GetExportedTypes e GetTypes são todos usados para obter referências aos tipos definidos na montagem. (A classe System.Type será explicada em mais detalhes na próxima seção.) A diferença entre GetExportedTypes e GetTypes é que GetExportedTypes retorna apenas os tipos que são públicos. O seguinte trecho de código exibe todos os tipos definidos no assembly atualmente em execução:

No exemplo acima, criamos um método estático Lista\_Propriedades que recebe um Type para, durante o tempo de execução, usarmos a reflexão para a ler todas as informações de um tipo em uma montagem, incluindo métodos, propriedades, eventos, etc.A classe PropertyInfo é usada para armazenar informações de uma propriedade. Ele contém o método e as propriedades usadas para ler e editar dados de uma propriedade. Por padrão, Type.GetProperties() retorna todas as propriedades públicas não estáticas de um tipo. Ao executarmo o código, teríamos a saída:

Type Name:A, Base Type:System.Object

Random has System.Int32 type

Type Name:B, Base Type:Reflection\_DemoAssembly.A

Random has System.Int32 type

Type Name:Program, Base Type:System.Object

Age has System.Int32 type

Name has System.String type

Reflection\_DemoAssembly, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

O GetReferencesAssemblies é usado para descobrir as referências para o assembly. Isso pode ser útil ao solucionar problemas de implantação.

Assembly Name: mscorlib

Assembly Version: 4.0.0.0

Assembly Name: System

Assembly Version: 4.0.0.0

Da mesma forma que fizemos com o método estático Lista\_Propriedades, podemos criar outros métodos que retornasse outras informações úteis definidos na classe System.Type como métodos, eventos, interfaces, campos etc. Por exemplo, para obtermos informações sobre os métodos:

publicstaticvoid Lista\_Metodos(Type type)

{

//Get all non-static methods of a type

MethodInfo[] methods = type.GetMethods();

foreach (MethodInfo method in methods)

{

Console.WriteLine("\tMethod Name:{0}, Return Type:{1}",

method.Name, method.ReturnType);

}

}

MethodInfo é uma classe que armazena informações de um método. A classe MethodInfo contém métodos e propriedades que são usadas para ler e editar dados de um método. Por padrão, Type.GetMethods() retorna todos os métodos públicos não estáticos de um tipo. A saída desse método ficaria:

Type Name:A, Base Type:System.Object

Random has System.Int32 type

Method Name:get\_Random, Return Type:System.Int32

Method Name:set\_Random, Return Type:System.Void

Method Name:Equals, Return Type:System.Boolean

Method Name:GetHashCode, Return Type:System.Int32

Method Name:GetType, Return Type:System.Type

Method Name:ToString, Return Type:System.String

A saída retornou também os métodos "ToString", "Equals", "GetHashCode" e "GetType" que não são definidos na classe Program. Esses métodos foram definidos na classe System.Object. Como toda classe herda a classe System.Object, o programa também mostrou esses métodos.

O método GetConstructors retorna uma matriz de objetos ConstructorInfo que você pode usar para obter informações sobre todos os construtores do tipo. O código a seguir imprime os construtores e os parâmetros para um objeto System.DataTable na janela Saída:

DataTable myDataTable = new DataTable();

Type myDataTableType = myDataTable.GetType();

ConstructorInfo[] myDataTableConstructors = myDataTableType.GetConstructors();

for (int i = 0; i <= myDataTableConstructors.Length - 1; i++)

{

ConstructorInfo constructorInfo = myDataTableConstructors[i];

Debug.Print(string.Format("\nConstructor #{0}", i + 1));

Console.WriteLine(string.Format("\nConstructor #{0}", i + 1));

ParameterInfo[] parameters = constructorInfo.GetParameters();

Debug.Print(string.Format("Number Of Parameters: {0}", parameters.Length));

Console.WriteLine(string.Format("Number Of Parameters: {0}", parameters.Length));

foreach (ParameterInfo parameter in parameters)

{

Debug.Print(string.Format("Parameter Name: {0}", parameter.Name));

Debug.Print(string.Format("Parameter Type: {0}", parameter.ParameterType.Name));

Console.WriteLine(string.Format("Parameter Name: {0}", parameter.Name));

Console.WriteLine(string.Format("Parameter Type: {0}", parameter.ParameterType.Name));

}

}

Quando o objeto Type representa uma enumeração, os métodos GetEnum permitem determinar todos os nomes e valores em uma enumeração.

privateenumMyCustomEnum { Red = 1, White = 2, Blue = 3 }

for (int i = 1; i <= 3; i++)

{

string enumName = myCustomEnumType.GetEnumName(i);

Debug.Print(string.Format("{0}: {1}", enumName, i));

Console.WriteLine(string.Format("{0}: {1}", enumName, i));

}

Ler, Gravar e Invocar método de classe

O Reflection também pode ser usado para ler e gravar o valor real armazenado em uma propriedade da instância de alguma classe em tempo de execução. Para obter um valor da propriedade de um objeto especificado, utilize o operador typeof ou o método GetType para armazenar o tipo de Objeto. Depois especifique o metadados da propriedade com GetProperty. Por fim, use o método GetValue() para obter o valor da propriedade.

Para definir um valor da propriedade, são necessárias as seguintes etapas de pegar o tipo com o GetType e a da propriedade com GetProperty. Por fim, use o método SetValue().

Já para invocar qualquer método definido, invés de usar o GetProperty devemos utilizar o método GetMethod e o especificar o método dentro da classe e o método Invoke() para invocar-lo. Ao usar o método InvokeMember da classe System.Type, o segundo parâmetro é BindingFlags.InvokeMethod, que aciona o método InvokeMember para invocar o método.

namespace Reflection\_Read\_Write

{

classPerson

{

publicint Age { get; set; }

publicstring FirstName { get; set; }

publicint Show()

{

Console.WriteLine("FirstName = {0} - Age = {1}", FirstName, Age);

return Age;

}

}

classProgram

{

staticvoid Main(string[] args)

{

var personobj = new Person { FirstName = "Sundus", Age = 21 };

var personobj2 = new Person { FirstName = "Ali", Age = 22 };

//Store Metadata of Person Type in Type's Object

//return Type of 'Person' class

Type persontype = typeof(Person);

persontype = personobj.GetType();

//Specify which property information is required

//Return metadata of specified property

PropertyInfo nameproperty = persontype.GetProperty("FirstName");

//Specify 'instance' (personobj) of 'Type' (Person)

//Whose 'property' (nameproperty) value is about to change

nameproperty.SetValue(personobj, "Lakhtey");

//Specify 'instance' (personobj) of 'Type' (Person)

//Whose 'property' (nameproperty) value is required

var value = nameproperty.GetValue(personobj);

Console.WriteLine("{0} = {1}", nameproperty.Name, value); //FirstName = Lakhtey

//Specify which method's information is required

//Return metadata of specified method

MethodInfo methodinfo = persontype.GetMethod("Show");

//Provide instance (personobj) name whose method is about to invoke

//pass parameter value 'null' if specified method has parameter

var returnValue = methodinfo.Invoke(personobj,null);

Console.WriteLine("Age = {0}", returnValue); // Age = 21

returnValue = (double)persontype.InvokeMember("Multiply", BindingFlags.InvokeMethod, null, personobj, newobject[] { 4, 5 });

Console.WriteLine(string.Format("InvokeMember Return Value: {0}", returnValue)); // InvokeMember Return Value: 20

Console.ReadKey();

}

}

}

FirstName = Lakhtey

FirstName = Lakhtey - Age = 21

Age = 21

Obter membros privados e membros estáticos de classe

Por padrão, a reflexão é usada para obter todos os membros públicos, mas com alguns ajustes de código também pode ser útil encontrar membros particulares de um tipo. Para obter o membro privado, especificamos a enumeração BindingFlags.NonPublic no parâmetro dos métodos Type.GetFields() e Type.GetMethods(), etc.

GetProperties() é usado para retornar informações de propriedade usando as enumerações BindingFlags; esse método pode retornar o tipo especificado de propriedades. Essas enumerações indicam que uma propriedade deve ser não pública e não estática, etc. Ao passar bandeiras de encadernação, use o pipe de barra vertical '|' para adicionar mais de um BindingFlag no método GetProperties().

Ao chamar GetFields, você usa a enumeração BindingFlags e pode especificar mais de um valor usando o operador bit a bit.Para obter os membros estáticos públicos de um tipoutilize BindingFlags.Public e BindingFlags.Static juntos.

namespace Reflection\_Get\_Private

{

classPerson

{

publicstaticstring company = "Microsoft";

privateint Age { get; set; }

privatestring FirstName { get; set; }

public Person(int age, string name)

{

this.Age = age;

this.FirstName = name;

}

}

classProgram

{

staticvoid Main(string[] args)

{

var personobj = new Person(21, "Sundus");

var personobj2 = new Person(22, "Ali");

//Store Metadata of Person Type in Type's Object

//return Type of 'Person' class

Type persontype = personobj.GetType();

//Pass BindingFlags to specify what kind of data member you want.

//NonPublic = Private || Non-Static = Instance

PropertyInfo[] props = persontype.GetProperties(BindingFlags.NonPublic | BindingFlags.Instance);

foreach (PropertyInfo prop in props)

{

Console.WriteLine("{0} = {1}", prop.Name, prop.GetValue(personobj));

}

//BindingFlags.Static = Static Member || BindingFlags.Public = Public Membe

FieldInfo[] fields = persontype.GetFields(BindingFlags.Public | BindingFlags.Static);

foreach (FieldInfo field in fields)

{

Console.WriteLine("{0} = {1}", field.Name, field.GetValue(personobj));

}

Console.ReadKey();

}

}

}

Age = 21

FirstName = Sundus

company = Microsoft

O objeto FieldInfo também possui um método SetValue que permite alterar o valor do campo, mesmo que seja privado ou protegido. Para demonstrar, adicione o seguinte acessador da classe Person:

privatestring \_privateField = "Hello";

publicstring PrivateField

{

get { return \_privateField; }

}

O código a seguir altera o valor da variável privateField e imprime seu valor na janela Saída. O método SetValue é utilizado da mesma maneira que foi feito anteriormente para setar um novo valor para propriedade.

Type personsetprivate = typeof(Person);

var setvalue = personsetprivate.GetField("\_privateField", BindingFlags.NonPublic | BindingFlags.Instance);

setvalue.SetValue(personobj, "My New Value");

//Private Field Value: My New Value

Console.WriteLine(string.Format("Private Field Value: {0}", personobj.PrivateField));

Gerenciar o ciclo de vida do objeto

* Gerenciar recursos não gerenciados; implementar IDisposable, incluindo interação com finalização; gerenciar IDisposable usando a instrução Using; gerenciar finalização e coleta de lixo

O ciclo de vida de um objeto é simplesmente o tempo entre quando um objeto é criado na memória e quando é destruído a partir dele. Fundamentalmente, o ciclo de vida de um objeto envolve as duas etapas a seguir:

1. **Criação de um objeto:** Usamos a palavra-chave new para instanciar um novo objeto.

Person person = new Person();

Um bloco de memória está alocado. Esse bloco de memória é grande o suficiente para armazenar o objeto (o CLR lida com a alocação de memória para objetos gerenciados). O bloco de memória é convertido em um objeto inicializado na memória (podemos controlar esta etapa implementando um construtor).

1. **Exclusão de um objeto:** Usamos a destruição para recuperar quaisquer recursos usados ​​por esse objeto. O objeto é limpo, por exemplo, liberando quaisquer recursos não gerenciados usados ​​pelo aplicativo, como identificadores de arquivo e conexões com o banco de dados (podemos controlar esta etapa implementando um destruidor). C # fornece duas maneiras de lidar com o gerenciamento de memória

* **Garbage Collection (Coleta de lixo):** o CLR libera a memória usada pelos objetos gerenciados
* **Disposal:** é necessário liberar manualmente a memória implementando IDisposable para objetos não gerenciados.

Perdas de memória (Memory Leaks)

Se um aplicativo não liberar o recurso alocado na memória após a conclusão do uso, ele criará um vazamento de memória porque a mesma memória alocada não está mais sendo usada pelo aplicativo. Se vazamentos de memória não forem gerenciados corretamente, o sistema acabará ficando sem memória; consequentemente, o sistema começa a ficar lento e o usuário não consegue fechar o aplicativo, sendo necessário é reiniciar o computador. A seguir, algumas causas comuns de vazamento de memória:

1. **Mantendo referências a objetos gerenciados por um longo tempo:** Se as referências de um objeto gerenciado permanecerem mais longas do que o necessário, os contadores de desempenho poderão mostrar um aumento constante no consumo de memória e poderá ocorrer uma OutOfMemoryException. Isso pode ocorrer devido a um escopo global variável, porque o GC não pode destruir uma variável ativa, mesmo que ela não esteja mais sendo usada por um aplicativo. O desenvolvedor precisa lidar com isso informando quanto tempo uma variável pode manter uma referência e destruí-la depois que ela não é mais necessária.
2. **Não foi possível gerenciar o recurso não gerenciado:** O coletor de lixo não pode liberar a memória do recurso não gerenciado. O desenvolvedor precisa liberar explicitamente recursos de recursos não gerenciados. Para fazer isso, o desenvolvedor precisa implementar uma interface IDisposable em tipos que usam recursos não gerenciados.
3. **Referência estática:** Se um objeto for referenciado por um campo estático, ele nunca será liberado. Tais objetos se tornam duradouros. O desenvolvedor precisa garantir que objetos de campo estático desnecessários sejam destruídos quando terminarem de ser usado pelo aplicativo
4. **Evento com a falta de inscrição:** Se um manipulador de eventos estiver inscrito (+=), o publicador do evento manterá uma referência ao assinante por meio do delegado do manipulador de eventos (assumindo que o delegado seja um método de instância). Se o editor viver mais que o assinante, ele manterá o assinante ativo, mesmo quando não houver outras referências ao assinante. Essa é a causa do vazamento de memória quando a desinscrição de um evento não está definida. O desenvolvedor precisa sempre cancelar a inscrição (-=) do evento com um manipulador igual.

## GARBAGE COLLECTOR

Quando programamos em **C** ou **C++**, muitos objetos requerem que façamos a alocação de seus recursos em sua declaração, antes dos objetos poderem ser utilizados com segurança. Liberar estes recursos de volta ao pool de memória livre também é responsabilidade do programador. Se os recursos não forem liberados, dizemos que o código está com vazamento de memória e mais recursos serão consumidos sem necessidade. Por outro lado, se os recursos forem liberados prematuramente, perda de dados, áreas de memória corrompidas e erros de ponteiros nulos podem ocorrer.

O **Garbage Collector** é um processo que automático que cuida do gerenciamento de memória liberando memória de objetos que não mais estão em uso. Características:

* Objetos que não são mais referenciados no programa serão destruídos. Porém não significa que o GC acontece imediatamente quando um objeto fica órfão
* Ele não especifica quando o objeto será destruído. O GC ocorre periodicamente com base na memória disponível, na alocação de memória atual e no tempo desde a última coleta O atraso entre as coleções pode variar de segundos a dias.
* Apenas os objetos não utilizados são destruídos: um objeto nunca é destruído se ele mantém a referência de um outro objeto.
* Nem todos os objetos órfãos são recuperados a cada coleção
* Existem alguns casos em que é necessário ajudar o GC
  + Você pode forçar a coleta de lixo chamando GC.Collect();

Você não pode destruir um objeto explicitamente no código, ou seja, você não pode determinar quando o método destruidor será chamado. Também, quando um objeto perde o escopo no final do método Main(), o processo de destruição não é necessariamente invocado. A decisão de recorrer ao processo de destruição é feita exclusivamente pelo Garbage Collector.

O **Garbage Collector** foi implementado em uma *thread* em separado, sempre rodando no *back end*. Como uma *thread* separada ocasiona um *overhead* extra, a *thread* do **Garbage Collector** tem a prioridade mais baixa. Mas quando o Sistema não encontra mais espaço no *heap gerenciado*, a thread do **Garbage Collector** recebe prioridade **REALTIME** (é a mais alta prioridade no **Windows**) e sai recolhendo todos os objetos que não são mais utilizados.

## Benefícios do Garbage Collector

* Permite que os desenvolvedores escrevam aplicativos sem se preocupar em liberar memória manualmente.
* Aloca memória em um heap gerenciado. Essa alocação de objetos é feito com o máximo de eficiência.
* Recolhe objetos que não estão sendo utilizados e limpa a memória.
* Ativa a segurança da memória, mantendo os objetos em memória em seu devido lugar, impedindo que um objeto utilize conteúdo de outro objeto.

## O heap gerenciado

Há dois lugares na memória onde o **CLR** (**Commom Language Runtime**) armazena itens enquanto o código é executado. Um é a stack; o outro é o heap. Em uma simplificação exagerada, ostack controla o que está executando no seu código e o heap controla os objetosgrandes com um ponteiro na stack apontado para ele. Mas, basicamente, essa é a distinção entre os dois tipos de memória. A stack é limpa automaticamente pelo CLR no final de um método, já o heap é gerenciada pelo Garbage Collector. Depois que o **Garbage Collector** é inicializado pelo **CLR**, elechama um método VirtualAlloc do Win32 que aloca um segmento contígua deespaço de endereçode memória para armazenar objetos ou processos. Esta memória é chamada de **heap gerenciado**.

* Existe um *heap gerenciado* para cada *processo gerenciado*.
* Todas as *threads* de um *processo* alocam memória no mesmo *heap*.

Enquanto os espaços de endereços estão disponíveis no heap gerenciado, o runtime continua a alocar espaço para novos objetos. O heap é dividido em dois: uma parte com objetos grandes (a partir de 85Kbytes, normalmente arrays) e outro com os menores. Quanto menos objetos alocados no heap, menos trabalho o Garbage Collector terá. Por isso, quando você alocar objetos, tente alocar apenas a quantidade de memória necessária.Por exemplo, não aloque uma matriz de 10 bytes quando você precisou apenas de uma matriz de 5 bytes.

No entanto, a memória não é infinita. Eventualmente, o Garbage Collector deve executar uma coleta, a fim de liberar memória. O motor de otimização do Garbage Collector determina o melhor momento para realizar a coleta, com base nas alocações feitas. Quando uma coleta é disparada, o Garbage Collectorchama um método VirtualFree do Win32que limpa a memória ocupada por objetos sem uso. Este processo “compacta” objetos em uso, os reorganiza para ficarem juntos e o espaço excedente é removido. Assim o heap fica menor, ocupando menos memória. Isto assegura que objetos alocados juntos continuem juntos no heap gerenciado, preservando sua localização.

A frequência e duração da intervenção das coletas é resultado do volume de alocações e da quantidade de memória restante no *heap gerenciado*.

## Gerações

Os objetos que são alocados na heap acabam sendo divididos em três grupos, também chamados de gerações, para poder lidar com **objetos de vida útil longa** e **objetos de vida útil curta**. A coleta ocorre primeiro com os objetos de vida útil curta, que normalmente ocupam uma pequena parte do *heap*.Existem 3 gerações de objetos no *heap*:

* **Geração** 0: é a primeira geração onde um objeto é alocado. Assim que algo é alocado na heap, o objeto alocado é colocado imediatamente como sendo parte da geração 0; Um exemplo de objeto de vida curta são variáveis temporárias. A coleta ocorre mais frequentemente nesta geração.
* **Geração** 1: trata-se de uma geração de transição, onde objetos que são utilizados de maneira “média”,de vida curta, são alocados. Serve como um *buffer* entre objetos de vida curta e objetos de vida longa.
* **Geração** 2: é uma geração que contém objetos que são utilizados por mais tempo, de vida longa, e que, por isso, precisam existir na memória por um tempo maior. Um exemplo deste tipo é um objeto de uma aplicação servidor que contém dados estáticos que se mantém ativos durante o processo.

## O ciclo de trabalho do Garbage Collector na heap

O que o Garbage Collector faz basicamente é migrar os objetos entre estas três áreas distintas e eliminar as áreas de memória associadas a objetos que não são mais utilizados. Seguindo o seguinte ciclo de análise:

1. Suspenda todos os threads gerenciados, exceto o thread que acionou o Garbage Collection.
2. Encontre uma lista de todos os objetos ativos.
3. Remova os objetos mortos e recupere sua memória.
4. Compacte os objetos sobreviventes e promova-os para uma geração mais antiga.

Objetos novos alocados formam uma nova geração de objetos e são implicitamente de geração 0, a não ser que sejam objetos grandes. Neste caso eles vão diretamente para o *heap de grandes objetos* na geração 2.A maioria dos objetos são coletados na geração 0 e não sobrevivem para a próxima geração.

A coleta ocorre em gerações específicas quando as condições permitem.É importante notar também que os ciclos de análise ocorrem na geração que é alvo do ciclo e nas gerações anteriores, mais novas. Por exemplo: se o *Garbage Collector* precisa analisar a geração 0, somente ela é analisada. Se o *Garbage Collector* precisa analisar a geração 1, as gerações 1 e 0 são analisadas. Se o *Garbage Collector* precisa analisar a geração 2, as gerações 2, 1 e 0 são analisadas. Em decorrência desse funcionamento, o ciclo de análise na geração 2 ganha o nome de ***coleta completa ou total***, pois analisa todos os objetos de todas as gerações (ou seja, todos os objetos do heap gerenciado). Essa análise do *Garbage Collector* pode ocorrer em situações pré-definidas:

* Quando o sistema operacional ficar sem memória física disponível;
* O tamanho das gerações é estourado, ou seja, objetos alocados na memória precisam de mais espaço;
* O método GC.Collect() é invocado manualmente, o que caracteriza uma chamada explícita para o processo de análise do *Garbage Collector* dentro da aplicação.

O *Garbage Collector* trabalha em ciclos de análise sobre as gerações que existem na *heap*. O que diferencia o funcionamento do *Garbage Collector* sobre as gerações é a peridiocidade da inspeção: a *geração 0*, por ser menor na maioria das vezes, sofre análises do *Garbage Collector* mais frequentes do que as gerações 1 e 2 por exemplo. Nestas análises, o que o *Garbage Collector* faz é verificar se os integrantes das gerações estão ainda sendo utilizados e, caso algum integrante não seja mais necessário, este é removido, fazendo com que a área de memória correspondente seja liberada e fique disponível para novas alocações.

## Sobreviventes e promoções

Os objetos são inicialmente alocados na geração 0 – portanto, na geração que sofre coletas mais rápidas – porque a CLR supõe que estes objetos não serão mais necessários muito rapidamente, o que faria com que estes objetos fossem removidos rapidamente da memória. E isso é verdade para a maioria dos cenários. Porém, alguns objetos podem precisar sobreviver por mais de um ciclo de análise do *garbage collector.*

Objetos não recolhidos durante uma coleta são conhecidos como **sobreviventes** e são **promovidos** para a próxima geração, sofrendo ciclos mais espaçados de análise do *garbage collector*. Objetos sobreviventes de uma coleta na geração 0 são promovidos para a geração 1, objetos sobreviventes de uma coleta na geração 1 são promovidos para a geração 2 e objetos que sobreviventes a uma coleta na geração 2 permanecem na geração 2.

Esse processo de promoção pode tornar a gerência de memória um processo muito lento, já que esse deslocamento de gerações pode exigir uma quantidade de processamento computacional considerável, dependendo da quantidade de objetos alocados. Por isso, o CLR sempre está “supervisionando” o trabalho do *Garbage Collector*. Se o *Garbage Collector* passa a detectar que a taxa de sobrevivência de objetos em uma determinada geração é muito alta, a CLR aumenta o tamanho da geração em questão, evitando que as gerações tenham seu tamanho estourado frequentemente. A CLR sempre tenta equilibrar os ciclos de análise do *Garbage Collector* e o tamanho das gerações – as gerações também não podem ser sempre expandidas, pois isso deixaria o sistema operacional sem memória em um curto espaço de tempo. Ou seja, O CLR faz o balanço continuamente de duas prioridades: não deixar que o conjunto de trabalho de um aplicativo fique muito grande e não deixar a coleta tomar muito tempo.

## Gerações e segmentos efêmeros

O CLR e o *Garbage Collector* supõe que os objetos que fazem parte das gerações 0 e 1 terão um ciclo de vida muito curto, sendo eliminados rapidamente da memória. Por isso, estas gerações também são conhecidas como **gerações efêmeras**, pelo fato dos objetos neles serem de vida curta.

**Gerações efêmeras** devem ser alocados no segmento de memória que é conhecido como segmento efêmero. Cada novo segmento adquirido pelo **Garbage Collector** se torna um novo segmento efêmero e contém os elementos que sobreviveram a coleta da geração 0. O antigo segmento efêmero se torna o novo segmento geração 2.

O tamanho do segmento efêmero varia dependendo do sistema (32 ou 64 bits) e do tipo de **Garbage Collector** que está em execução. Valores padrão estão na tabela seguinte.

|  |  |  |
| --- | --- | --- |
| Tipo | 32-bit | 64-bit |
| GC de estação | 16 MB | 256 MB |
| GC de servidor | 64 MB | 4 GB |
| GC de servidor com mais de 4 CPUs lógicos | 32 MB | 2 GB |
| GC de servidor com mais de 8 CPUs lógicos | 16 MB | 1 GB |

Segmentos efêmeros podem incluir objetos geração 2. Objetos geração 2 podem utilizar múltiplos segmentos (quantas seu processo precisar e a memória suportar).

A quantidade de memória liberada é proporcional ao espaço ocupado pelos objetos inativos.A linguagem C# fornece métodos especiais que são usados ​para liberar a instância de uma classe a partir da memória, são eles: **Finalize() e Dispose().**

Gerenciando recursos não gerenciados

Tudo isso seria suficiente se você estivesse trabalhando apenas com recursos gerenciados. Manter objetos como seqüências de caracteres, números e outros tipos gerenciados é manuseado completamente pelo coletor de lixo. Mas recursos não gerenciados como uma conexão de rede, identificador de arquivo, identificador de janela, é necessário liberar explicitamente esses itens. Caso contrário, você receberá erros como "Este arquivo está em uso" ou não poderá se conectar ao seu banco de dados porque todas as conexões estão em uso.

Finalize ~

Por esse motivo, o C# suporta o conceito de finalização. Esse mecanismo permite que um tipo seja limpo antes da coleta de lixo. É importante entender que um finalizador de C# não é o mesmo que um destruidor de C++. Os destruidores de C++ podem ser chamados determinísticos, pois você sabe quando eles serão executados. Em C#, no entanto, você não sabe quando o GC chamará o método Finalize de um objeto, portanto, esse processo é chamado de finalização não determinística.

O método destruidor Finalize()é um método especial que é chamado a partir de uma classe para qual ele pertence ou a partir de classes derivadas. Um finalizador é o oposto de um construtor, enquanto o construtor é a primeira oportunidade de trabalhar com um objeto o Finalize é a última oportunidade de trabalhar com um objeto. Um finalizador em C# requer alguma sintaxe especial, assim como um construtor. Você precisa prefixar o nome da classe com um til (~) para criar um finalizador.

publicclassSomeType

{

~SomeType()

{

// This code is called when the finalize method executes

}

}

Este destruidor é convertido no seguinte método Finalize:

protectedoverridevoid Finalize()

{

try

{

// Free unmanaged resources here. ...

}

finally

{

base.Finalize();

}

}

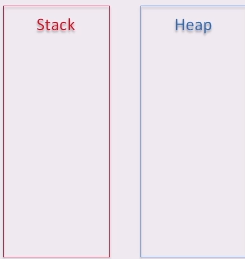
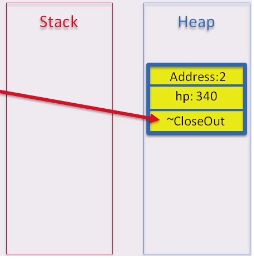
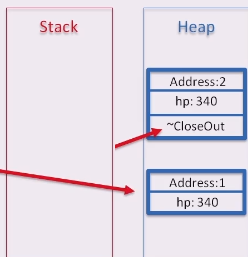
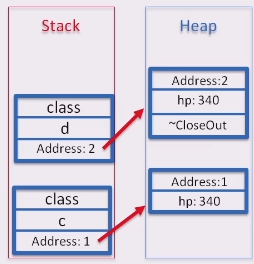
Você não pode substituir explicitamente o método Finalize no código C #. Isso também é bom porque seu código não pode chamar diretamente o método Finalize da classe base. (Veja a lista anterior de regras destruidoras.)

Várias regras se aplicam a destruidores que não se aplicam a outros métodos. A lista a seguir resume essas regras:

* Destrutores podem ser definidos apenas em classes, não em estruturas.
* Uma classe pode ter no máximo um destruidor.
* Os destruidores não podem ser herdados ou sobrecarregados.
* Destrutores não podem ser chamados diretamente.
* Destruidores não podem ter modificadores ou parâmetros.

O importante é entender que um finalizador aumenta a vida útil de um objeto. Como o código de finalização também precisa ser executado, o .NET Framework mantém uma referência ao objeto em uma fila de finalização especial. Um encadeamento adicional executa todos os finalizadores em um momento considerado apropriado com base no contexto de execução. Isso atrasa a coleta de lixo para tipos que têm um finalizador.

1. Os objetos utilizam um ponteiro na pilha que referencia os dados armazenados na pilha. Quando o GC é executado, ele vê que os dois objetos ainda têm referências e, portanto, nenhuma ação é tomada.
2. Na próxima vez que o GC for executado, o Endereço: 1 e o Endereço: 2 não terão mais referências. Como o endereço: 1 não possui um finalizador, ele é removido.
3. Como o Endereço: 2 possui um Finalizador, ele é colocado em uma fila e mantido vivo.
4. O thread do Finalizador é executado e identifica objetos na fila e executa seus métodos de finalização. Após a finalização, o objeto fica órfão. Na próxima vez que o GC ocorrer, o objeto será excluído



Os finalizadores devem ser usados somente quando necessário, pois:

* Usa muitos recursos do sistema
* Reduza a alocação e a coleta de memória
* Prolongar a vida útil dos objetos, não limpa a memória imediatamente.
* Impossível prever a ordem na qual os finalizadores de um conjunto de objetos serão chamados
* Se o código em um finalizador bloquear, referenciar outros objetos finalizáveis ou lançar exceções, a finalização não ocorrerá.

Pode ser utilizado durante o uso de qualquer recurso não gerenciado como um Stream de arquivo declarado no nível de classe. Podemos não estar sabendo em que fase ou etapa deve ser apropriada para fechar o arquivo. Este objeto está sendo usado em muitos lugares no aplicativo. Portanto, neste cenário o Finalize pode ser apropriado onde o recurso não gerenciado pode ser liberado.

Disposal

O método Dispose() é chamado para liberar recursos que GC não é capaz de liberar uma vez que não dispõe de informações para reivindicar a memória desses recursos como manipuladores de arquivos, manipuladores de janelas, sockets de rede, conexões de banco de dados, etc, tão logo o objeto usando o recurso não esta mais sendo usado.

Por exemplo, se você abrir um arquivo no programa e não fechá-lo após o processamento, o arquivo não estará disponível para outras operações ou ele está sendo usado por outra aplicação que não podem abrir ou modificar o arquivo. Para este fim classe FileStream fornece o método Dispose. Temos de chamar esse método após o processamento do arquivo ser concluído, caso contrário, ocorrerá uma exceção de acesso negado ou de que o arquivo está sendo usado por outro programa.

StreamWriter stream = File.CreateText("temp.dat");

stream.Write("some data");

File.Delete("temp.dat"); // Throws an IOException because the file is already open.

Os métodos finalizers são chamados no final do ciclo de varredura do Garbage Collector e realizam a limpeza de recursos não gerenciados. É uma alternativa usada quando o objeto que consome o recurso não chama o Dispose. Você pode forçar isso adicionando uma chamada ao GC.Collect, conforme mostrado no código a seguir. A linha WaitForPendingFinalizers garante que todos os finalizadores tenham sido executados antes que o código continue. O Garbage Collector é bastante inteligente no gerenciamento de memória e não é recomendável fazer a chamada manualmente para o GC.

StreamWriter stream = File.CreateText("temp.dat");

stream.Write("some data");

GC.Collect();

GC.WaitForPendingFinalizers();

File.Delete("temp.dat");

Ao executar esse trecho de código no modo Release, o Garbage Collector verá que não há mais referências ao fluxo e liberará qualquer memória associada à instância do StreamWriter. Isso executará o finalizador, que, por sua vez, liberará qualquer identificador de arquivo para o arquivo temp.dat (no modo de depuração, o compilador garantirá que a referência não seja coletada como lixo até o final do método).

No geral, você não deve depender do Garbage Collector para executar um finalizador para fechar seu arquivo. Para oferecer a você a oportunidade de liberar explicitamente recursos não gerenciados, o C# oferece a idéia da interface IDisposable.

publicinterfaceIDisposable

{

void Dispose();

}

A interface IDiposable oferece umúnico método: Dispose, que libera qualquer recurso não gerenciado imediatamente. Diferente do método Finalize() o método Dispose() não é chamado automaticamente e você precisa chamá-lo explicitamente a partir de uma aplicação cliente quando um objeto não mais for necessário.O exemplo anterior também poderia ter sido escrito utilizando o método Dispose().

StreamWriter stream = File.CreateText("temp.dat");

stream.Write("some data");

stream.Dispose();

File.Delete("temp.dat");

Mas e se uma exceção ocorresse antes que stream.Dispose() fosse chamado? Para garantir que seus recursos sejam sempre limpos, você precisa agrupar todos os tipos que implementam IDisposable em uma instrução try/finally.

Try-finally

É o jeito mais antigo de consumir recursos desde que using foi implementado no .NET Framework. Uma regra básica ao se consumir um recurso é garantir o sucesso de quem for utilizá-lo. No bloco finally vai a chamada explícita ao método Dispose. É seguro, pois o bloco finally sempre será executado.

FileStream fs = null;

string path = @"C:\temp\Teste.txt";

try

{

fs = new FileStream(path, FileMode.CreateNew, FileAccess.ReadWrite);

}

catch (IOException)

{

if (File.Exists(path))

{

var newPath = path.Replace("Teste", "outro");

//tenta criar o arquivo com novo path

fs = new FileStream(newPath, FileMode.CreateNew, FileAccess.ReadWrite);

}

}

finally

{

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

}

Using

Como o uso dos tipos que implementam IDisposable em uma instrução try/finally é muito comum, o C# possui uma instrução especial para simplicar isso: a instrução using. A instrução using é convertida pelo compilador em uma instrução try/finally que faz uma chamada implícita ao método Dispose no objeto. Por esse motivo, a instrução using pode ser usada apenas com tipos que implementam IDisposable, como StreamReader/StreamWriter, DataTable.

|  |
| --- |
| string path = @"C:\temp\outro.txt";  string destPath = @"C:\temp\Teste.txt";  using (TextReader txt = new StreamReader(path))  {  if (!File.Exists(path))  {  File.Copy(path, destPath);  Console.WriteLine("Arquivo copiado com sucesso.");  }  } |

Após descartar um item, você não poderá mais usá-lo. O uso de um item descartado resultará em uma ObjectDisposedException. Abaixo um exemplo de classe implementando IDisposable utilizando using.

classmyClass : IDisposable

{

public StreamReader reader;

publicvoid Dispose()

{

//Cleanup unmanaged resources

if (reader != null)

reader.Dispose();

GC.SuppressFinalize(this);

Console.WriteLine("Disposed");

}

}

using (myClass obj = new myClass()) { }

Console.WriteLine("End");

Implementando IDisposable e um finalizador

O método Finalize não pode ser chamado explicitamente no código. Só pode haver a implementação da interface IDisposable, via declaração do destruidor, e deixar o GC realizar o seu serviço que é o de chamar o Finalize quando os objetos se tornam inacessíveis. É recomendável implementar os métodos Finalize e Dispose em conjunto, se você precisar implementar o método Finalize. Após a compilação o destruidor se torna o método Finalize.

Por exemplo, suponha que você tenha uma classe de wrapper em torno de um recurso de arquivo não gerenciado. Você implementa IDisposable para que os usuários da sua classe possam limpar imediatamente, se quiserem. Você também implementa um finalizador, caso eles se esqueçam de chamar Dispose. A exemplo abaixo mostra como fazer isso.

classUnmangedWrapper : IDisposable

{

bool disposed = false;

public FileStream Stream { get; privateset; }

public UnmangedWrapper()

{

this.Stream = File.Open("temp.dat", FileMode.Create);

}

~UnmangedWrapper()

{

Dispose(false);

}

publicvoid Close()

{

Dispose();

}

publicvoid Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

publicvoid Dispose(bool disposing)

{

if (disposed) return;

if (disposing)

{

if (Stream != null)

{

Stream.Close();

}

}

disposed = true;

}

}

Há algumas coisas a serem observadas sobre essa implementação:

* Foi criado uma flag que vai controlar se o recurso já foi liberado ou não (bool disposed);
* O finalizador chama Dispose apenas passando false para disposing.
* O método Dispose extra com o argumento booleano faz o trabalho real. Este método deve ser protected virtual por causa da hierarquia de herança. É assinado com protected para limitar sua visibilidade por outras classes fora da hierarquia. Este método verifica se está sendo chamado em um Dispose explícito ou se está sendo chamado a partir do finalizador:
* Se o finalizador chamar Dispose, você não fará nada porque o objeto Stream também implementa um finalizador e o Garbage Collector se encarrega de chamar o finalizador da instância do Stream. Você não pode ter certeza se já foi chamado, por isso é melhor deixar isso para o coletor de lixo.
* Se Dispose for chamado explicitamente, feche o FIleStream subjacente. É importante ser defensivo ao codificar esse método; sempre verifique qualquer fonte de possíveis exceções. Pode ser que Dispose seja chamado várias vezes e isso não cause erros.
* O método Dispose regular chama GC.SuppressFinalize (this) para garantir que o objeto seja removido da lista de finalização que o Garbage Collector está acompanhando. A instância já foi executada, portanto, não é necessário que o Garbage Collector chame o finalizador.
* Por fim, utilize o finalizer apenas se ele não foi implementado no seu código dentro de um bloco using ou try-finally com uma chamada ao Dispose.

É importante saber a diferença entre implementar o IDisposable e um finalizador. Ambos limpam seu objeto, mas um finalizador é chamado pelo Garbage Collector e o método Dispose pode ser chamado a partir do código.

|  |  |
| --- | --- |
| **Finalize** | **Dispose** |
| Usado para liberar recursos não gerenciados como arquivos, conexões de banco de dados, recursos COM, etc. retidos por um objeto antes que objeto seja destruído. | Ele é usado para liberar recursos não gerenciados como arquivos, conexões de banco de dados, etc COM a qualquer momento. |
| Internamente é chamado pelo Garbage Collector e não pode ser chamado pelo código do usuário. | É chamado explicitamente pelo código do usuário, a classe que o define deve implementar a interface IDisposable. |
| Pertence a classe Object | Pertence à interface IDisposable. |
| Implementar quando você tem recursos não gerenciados em seu código, e quer ter certeza de que esses recursos são liberados quando a coleta de lixo acontecer. | Implementar quando você está escrevendo uma classe personalizada que será usada por outros usuários. |
| Há custo de desempenho associado. | Não há custo de desempenho associado |

A lista a seguir resume as regras e conceitos de gerenciamento de recursos:

* Se uma classe não contém recursos gerenciados e não gerenciados, não é necessário implementarIDisposable ou ter um destruidor.
* Se a classe tiver apenas recursos gerenciados, deverá implementar IDisposable, mas não precisa de um destruidor. O destruidor deve liberar apenas recursos não gerenciados. (Quando o destruidor é executado, você não pode ter certeza de que os objetos gerenciados ainda existem, portanto não pode chamar os métodos Dispose de qualquer maneira.)
* Se a classe tiver apenas recursos não gerenciados, precisará implementar IDisposable e precisará de um destruidor, caso o programa não chame Dispose.
* O método Dispose deve ser seguro para executar mais de uma vez. Você pode conseguir isso usando uma variável para acompanhar se ela já foi executada antes.
* O método Dispose deve liberar recursos gerenciados e não gerenciados.
* Após liberar recursos, o destruidor deve chamarGC.SuppressFinalize, para que o objeto possa pular a fila de finalização.
* Para evitar confusão, idealmente, uma classe não deve incluir recursos gerenciados e não gerenciados. Se a classe tiver recursos não gerenciados, deverá gerenciar apenas um recurso

WeakReference - Referências fracas

Um WeakReference permite que o coletor de lixo colete o objeto enquanto ainda permite que o aplicativo o acesse. Uma referência fraca é válida somente durante o período indeterminado até que o objeto seja coletado quando não há nenhuma referência forte. Quando você usa uma referência fraca, o aplicativo ainda pode obter uma referência forte para o objeto, o que impede que ele seja coletado. No entanto, sempre há o risco de o coletor de lixo obter o objeto primeiro antes de uma referência forte ser restabelecida.

|  |  |
| --- | --- |
| **Propriedades** | **Descrição** |
| [IsAlive](https://docs.microsoft.com/pt-br/dotnet/api/system.weakreference.isalive?view=netcore-3.1#System_WeakReference_IsAlive) | Obtém uma indicação se o objeto referenciado pelo objeto [WeakReference](https://docs.microsoft.com/pt-br/dotnet/api/system.weakreference?view=netcore-3.1) atual passou pela coleta de lixo. |
| [Target](https://docs.microsoft.com/pt-br/dotnet/api/system.weakreference.target?view=netcore-3.1#System_WeakReference_Target) | Obtém ou define o objeto (o destino) referenciado pelo objeto [WeakReference](https://docs.microsoft.com/pt-br/dotnet/api/system.weakreference?view=netcore-3.1) atual. |
| [TrackResurrection](https://docs.microsoft.com/pt-br/dotnet/api/system.weakreference.trackresurrection?view=netcore-3.1#System_WeakReference_TrackResurrection) | Obtém uma indicação se o objeto referenciado pelo objeto [WeakReference](https://docs.microsoft.com/pt-br/dotnet/api/system.weakreference?view=netcore-3.1) atual é acompanhado depois de finalizado. |

static WeakReference \_weak;

staticvoid Main(string[] args)

{

\_weak = new WeakReference(new StringBuilder("perls"));

// See if weak reference is alive.

if (\_weak.IsAlive)

Console.WriteLine((\_weak.Target as StringBuilder).ToString());

// Invoke GC.Collect.

GC.Collect();

// Check alive.

if (\_weak.IsAlive)

{

Console.Write("stillAlive: ");

Console.WriteLine((\_weak.Target as StringBuilder).ToString());

}

Console.ReadKey();

}

Você pode criar uma referência fraca curta ou uma referência fraca longa:

* **Short:** O destino de uma referência fraca curta se torna null quando o objeto é recuperado pela coleta de lixo. A referência fraca é um objeto gerenciado e está sujeita à coleta de lixo assim como qualquer outro objeto gerenciado. Uma referência fraca curta é o construtor sem parâmetros para WeakReference.
* **Long:** Uma referência fraca longa é mantida após o método Finalize do objeto ter sido chamado. Isso permite que o objeto seja recriado, mas o estado do objeto permanece imprevisível. Para usar uma referência longa, especifique true no construtor WeakReference.

Se o tipo de objeto não tiver um método Finalize, a funcionalidade de referência fraca curta é aplicada e a referência fraca é válida apenas até o destino ser coletado, o que pode ocorrer a qualquer momento após o finalizador ser executado.

Para estabelecer uma referência forte e usar o objeto novamente, converta a propriedade Target de um WeakReference para o tipo do objeto. Se a propriedade Target retornar null, o objeto terá sido coletado, caso contrário, continue usando o objeto, pois o aplicativo recuperou uma referência forte para ele.

Às vezes, você precisa trabalhar com objetos grandes que exigem muito tempo para serem criados. Por exemplo, uma lista de objetos que precisam ser recuperados de um banco de dados. Seria bom se você pudesse apenas manter os itens na memória; no entanto, isso aumenta a carga de memória do seu aplicativo e talvez a lista não seja mais necessária. Mas se a coleta de lixo ainda não ocorreu, seria bom se você pudesse reutilizar a lista que criou.

static WeakReference data;

publicstaticvoid Run()

{

object result = GetData();

// GC.Collect(); Uncommenting this line will make data.Target null

result = GetData();

}

privatestaticobject GetData()

{

if (data == null)

{

data = new WeakReference(LoadLargeList());

}

if (data.Target == null)

{

data.Target = LoadLargeList();

}

return data.Target;

}

privatestaticobject LoadLargeList()

{

thrownew NotImplementedException();

}

A função GetData verifica se o WeakReference ainda contém dados. Caso contrário, os dados serão carregados novamente e salvos no WeakReference. O interessante é que descomentar a linha GC.Collect () libera a memória que o WeakReference aponta. Se a coleta de lixo não ocorreu, os dados dentro de WeakReference.Target podem ser acessados e retornados ao chamador.

Usar o WeakReference não é uma solução completa para um cenário de armazenamento em cache. Se você deseja implementar um cache, defina um algoritmo que decida quais itens devem ser removidos do cache. Ao remover, você transforma uma referência em WeakReference e deixa para o coletor de lixo.

Manipular cadeias de caracteres

* Manipular cadeias de caracteres usando as classes StringBuilder, StringWriter e StringReader; pesquisar cadeias de caracteres; enumerar métodos de cadeias de caracteres; formatar cadeias de caracteres; usar interpolação de cadeia($)

Strings são diferentes de outros tipos de dados. Os programas geralmente os tratam como se fossem qualquer outro dado de valor, mas nos bastidores a classe string é notavelmente complexa. Por exemplo, se você entender Como as strings são armazenadas, saberá quando seria melhor usar a classe StringBuilder em vez de simplesmente concatenar as strings juntas.

No C#, a palavra-chave string é um alias para System.String cujo valor é text; portanto, quando você cria uma variável de string, na verdade, você está criando um objeto String. Internamente, o texto é armazenado como uma coleção sequencial de somente leitura de objetos Char. O .NET Framework representa caracteres como a versão Unicode UTF-16, um formato que usa 16 bits para armazenar cada caractere. Isso permite que um caractere Unicode represente muito mais caracteres do que os fornecidos em um teclado americano padrão. (A versão mais recente do Unicode define valores para mais de 110.000 caracteres em mais de 100 scripts.).

Uma das características mais incomuns do Strings é que eles são imutáveis. Isso significa que o conteúdo de uma String não pode ser alterado após a criação da String. Em vez disso, métodos que parecem modificar o valor de uma String, como Replace e ToUpper, na verdade retornam um novo objeto String que contém o valor modificado. As operações de string no .NET são altamente otimizadas, portanto:

* Na maioria das condições, seu uso não afetará o desempenho
* Porém, as funções de loop que executam centenas ou milhares de vezes executando operações de string podem afetar o desempenho

Para economizar memória, o CLR mantém uma tabela chamada pool interno que contém uma única referência a cada único valor de texto usado por um programa. Qualquer variável String que se refere a uma parte específica do texto é na verdade uma referência ao pool interno. Múltiplas Strings que representam o mesmo valor referem-se à mesma entrada no pool interno.Tudo isso requer alguma sobrecarga, portanto, trabalhar com Strings não é tão rápido quanto trabalhar com tipos de valor. Se um programa deve executar um grande número de concatenações, cada um cria uma nova instância de String que deve ser internada e que leva tempo. Nesse caso, o uso da classe StringBuilder pode oferecer melhor desempenho.

Construtores de String

Três das maneiras mais comuns de inicializar uma variável String são:

* Setagem por uma string literal. (string mystring = "test";)
* Setagem por um controle TextBox ou ComboBox. (string mystring = TextBox.Text)
* Setagem por resultado de um cálculo. (string mystring = (1 + 2).ToString())

O último deles inclui métodos que formatam uma variável para produzir uma String, como usar o método ToString ou o método String.Format. Além desses métodos, a classe String fornece vários construtores que às vezes podem ser úteis:

1. Inicializa a String a partir de uma matriz unidimensional de caracteres.

var hello = newchar[] { 'H', 'e', 'l', 'l', 'o', '!' };

string ctor1 = newstring(hello);

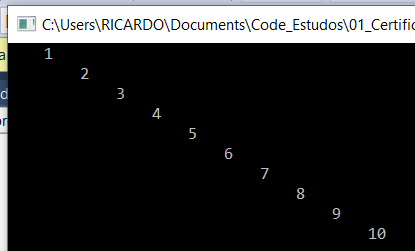
Console.WriteLine(ctor1); //Hello!

1. Um segundo construtor usa apenas parte de uma matriz de caracteres, tomando como parâmetros a matriz, uma posição inicial e o comprimento dos caracteres a serem usados.

var hello = newchar[] { 'H', 'e', 'l', 'l', 'o', '!' };

string ctor2 = newstring(hello, 0, 4);

Console.WriteLine(ctor2); //Hell

1. Um terceiro construtor usa como parâmetro um caractere e o número de vezes que você deseja repetir esse caractere na nova String. Isso pode ser particularmente útil se você quiser recuar uma sequência com um certo número de espaços ou caracteres de tabulação.

Por exemplo, o código a seguir exibe os números de 1 a 10 em linhas separadas, com cada linha recuada mais quatro espaços que o anterior:

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

{

string indent = newstring(' ', 4 \* i);

Console.WriteLine(indent + i.ToString());

}

A maioria dos valores de String é criada por literais de string, texto digitado pelo usuário ou pelos resultados de cálculos, mas os construtores de String às vezes podem ser úteis.

Campos e propriedades da sequência

A classe String fornece apenas três campos e propriedades:

* **String.Empty:** que retorna um objeto que representa uma sequência vazia. Você pode usar esse valor para definir o valor de uma string ou para ver se uma string contém um valor vazio. (Como alternativa, você pode usar a string vazia literal "".)
* **Length:** que retorna o número de caracteres na string.
* **Indexador read-only:** que retorna os caracteres na String. Por ser um indexador, você pode obter seus valores adicionando um índice ao nome de uma variável String. Por exemplo, a instruçãousername[4] retorna o caractere 4 da stringusername. O indexador é somente leitura, portanto, você não pode definir um dos caracteres da String com uma declaração como username[4] = 'x'. Se você precisar fazer algo assim, poderá usar os métodos String descritos na próxima seção.

Se for mais fácil tratar a String como se fosse uma matriz de caracteres de leitura/gravação, você pode usar o método ToCharArray para converter String em uma matriz de caracteres, manipule-os e crie uma nova String passando o construtor pela matriz modificada. Por exemplo, o código a seguir usa uma matriz para alternar os caracteres de uma string entre maiúsculas e minúsculas:

var text = "text em maiuscula e minusculas";

char[] characters = text.ToCharArray();

for (int i = 0; i < characters.Length; i++)

if (i % 2 == 0) characters[i] = char.ToUpper(characters[i]);

else characters[i] = char.ToLower(characters[i]);

text = newstring(characters);

Console.WriteLine(text); //TeXt eM MaIuScUlA E MiNuScUlAs

Você também pode usar o indexador como fonte de iteração em um loop foreach:

text = "The quick brown fox jumps over the lazy dog.";

int[] counts = newint[26];

text = text.ToUpper();

foreach (char ch in text)

{

Console.Write(ch + " - ");

if (char.IsLetter(ch))

{

int index = (int)ch - (int)'A';

counts[index]++;

}

}

Imutabilidade

String é usada para armazenar valores de texto. String é imutável, o que significa que, uma vez que uma variável de string armazena algum texto, ela não pode editá-lo novamente; o texto é armazenado como uma coleção somente leitura de objetos Char. Portanto, sempre que o valor de uma variável de string é atualizado, ele recria uma instância para literais de string, o que não é bom em termos de memória e consumo de processo. O código abaixo parece inocente, mas criará uma nova string para cada iteração no seu loop. Ele usa muita memória desnecessária e mostra por que você deve ter cuidado ao trabalhar com seqüências de caracteres.

Stopwatch watch = new Stopwatch();

//Record how much time

watch.Start();

string mystring = "test";

for (int i = 1; i < 100000; i++)

{

mystring += i;

}

//Stop Recording time

watch.Stop();

float miliToSec = watch.ElapsedMilliseconds / 1000;

Console.WriteLine("Total time: {0}s", miliToSec); // Total time: 19s

Esse código será executado 100.000 vezes e sempre que criará uma nova string. A referência "s" apontará apenas para o último item, para que todas as outras seqüências estejam prontas imediatamente para a coleta de lixo. Como o C# está ciente desse problema, o compilador tenta otimizar o trabalho com seqüências de caracteres para você. Ao criar dois string idênticos em uma unidade de compilação, o compilador garante que apenas um objeto de string seja criado pelo CLR. Isso é chamado de cadeia de caracteres interna (string interning), que é feita apenas no tempo de compilação. Fazer isso em tempo de execução acarretaria uma penalidade de desempenho excessiva (pesquisar todas as strings toda vez que você criar uma nova resulta em muita perda de performance).

Ao trabalhar com um número tão grande de operações de string, você deve ter em mente que a string é imutável e que o .NET Framework oferece algumas classes auxiliares especiais ao lidar com strings.

Stringbuilder

StringBuilder é uma classe de System.Text que fornece melhor desempenho ao manipular dados de texto de uma maneira muito melhor que um System.String tradicional. StringBuilder é mutável, o que significa que os dados de texto podem ser editável. Seu método Append ajuda a concatenar os dados de texto, pois cria um buffer de sequência de caracteres que fornece melhor desempenho nessas situações.

A classe StringBuilder pode ser usada quando você estiver trabalhando com seqüências de caracteres em um loop restrito. Em vez de criar uma nova sequência repetidamente, você pode usar o StringBuilder, que usa um buffer de seqüências internamente para melhorar o desempenho. A classe StringBuilder ainda permite alterar o valor de caracteres individuais dentro de uma string, adicionar uma nova linha ao texto existente ou limpar todo o conteúdo.

//// Let's look at string concatenation

string s1 = "This is an example of s1 ";

s1 = s1 + "string concatenation.";

//Isso realmente fará com que um novo objeto s1 seja criado

//This is an example of s1 string concatenation.

Console.WriteLine(s1);

// StringBuilder pode realmente acrescentar dados a uma string

StringBuilder sb2 = new StringBuilder("This is an example of sb2 ");

sb2.Append("string concatenation.");

Console.WriteLine(sb2); // This is an example of sb2 string concatenation

// O StringBuilder pode adicionar uma nova linha ao texto existente ou limpar todo o conteúdo

sb2.AppendLine();

sb2.Append("Using StringBuilder...");

Console.WriteLine(sb2);// This is an example of sb2 string concatenationUsing StringBuilder...

// StringBuilder pode substituir o conteúdo no objeto

sb2.Replace("StringBuilder", "the StringBuilder class...");

Console.WriteLine(sb2);

// This is an example of sb2 string concatenation.

// Using the StringBuilder class......

sb2[0] = 'N';

sb2[1] = 'Ó';

Console.WriteLine(sb2);

// NÓis is an example of sb2 string concatenation.

// Using the StringBuilder class......

Se executar novamente a análise de performance realizada no System.String anteriormente teremos:

Stopwatch watch = new Stopwatch();

//Record how much time

watch.Start();

StringBuilder mystring = new StringBuilder("test");

for (int i = 1; i < 100000; i++)

{

mystring.Append(i);

}

//Stop Recording time

watch.Stop();

float miliToSec = watch.ElapsedMilliseconds / 1000;

Console.WriteLine("Total time: {0}ms", miliToSec); // Total time: 0ms

Console.ReadKey();

Na minha máquina, foram necessários 0 milissegundos para acrescentar um texto no StringBuilder, enquanto no exemplo anterior System.String levou 19 segundos. Portanto, o StringBuilder é mais rápido que o System.String. Mas um aspecto a ter em mente é que o StringBuilder nem sempre oferece melhor desempenho. Ao concatenar uma série fixa de seqüências, o compilador pode otimizar isso e combinar operações de concatenação individuais em uma única operação. Quando você está trabalhando com um número arbitrário de strings, como no exemplo de loop, um StringBuilder é uma escolha melhor (neste exemplo, você poderia ter usado também umnew String (“x”, 10000) para criar a string; ao lidar com com dados mais variados, isso não será possível).

Lembre-se também de que os tempos envolvidos para algumas operações de String são pequenos. O uso de um StringBuilder para concatenar 10 seqüências pode ser um pouco mais rápido do que executar 10 concatenações simples, mas a quantidade total de tempo economizada é medida em milissegundos. A menos que o programa repita essa operação muitas vezes ou faça concatenações muito mais longas, talvez seja melhor sacrificar alguns milissegundos para manter o código mais fácil de entender.

A Tabela a seguir descreve as propriedades mais úteis da classe StringBuilder.

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| Capacity | Obtém ou define o número de caracteres que podem ser mantidos pelo StringBuilder. Se a quantidade de texto armazenada no StringBuilder excede esse valor, o objeto aloca mais espaço. Se você sabe que o StringBuilder precisa armazenar pelo menos um certo número de caracteres, você pode usar essa propriedade para fazer com que o objeto pré-aloque memória em vez de alocar memória de forma incremental. Algumas versões sobrecarregadas do construtor da classe permitem especificar uma capacidade inicial. |
| Length | Obtém ou define o número atual dos caracteres armazenados no StringBuilder. Se você definir esse valor como menor que o comprimento atual, o texto no StringBuilder será truncado |

A Tabela abaixo descreve os métodos mais úteis da classe StringBuilder.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Append | Anexa uma representação de string de um objeto ao final do texto doStringBuilder |
| AppendFormat | Formata uma série de objetos e anexa o resultado ao final do texto do StringBuilder |
| EnsureCapacity | Garante que o StringBuilder tenha pelo menos uma determinada capacidade |
| Insert | Insere uma representação de seqüência de caracteres de um objeto em uma determinada posição no texto do StringBuilder |
| Remove | Remove um intervalo de caracteres do texto do StringBuilder |
| Replace | Substitui todas as instâncias de um caractere ou sequência por um novo caractere ou sequência |
| ToString | Retorna uma representação String normal do texto do StringBuilder |

Algumas APIs no .NET Framework esperam que um TextWriter ou TextReader funcione. Essas APIs não podem funcionar diretamente com uma string ou com o StringBuilder. Por isso, o .NET Framework adiciona uma classe StringReader e StringWriter. Essas classes adaptam a interface do StringBuilder para que possam ser usadas em locais onde é esperado um TextWriter ou TextReader.

StringReader

StringReader é uma classe do System.IO usada para ler linhas de uma string. Com StringReader, podemos ler um caractere com o método Read ou ReadAsync e uma string inteira com o método ReadToEnd ou ReadToEndAsync. Esse tipo ajuda a acessar dados da string por meio de uma interface orientada a fluxo (Stream).A classe StringReader fornece uma implementação TextReader que lê partes de dados obtidas de um StringBuilder subjacente. Ele fornece métodos que facilitam a leitura seqüencial de partes de texto de uma string.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Peek | Retorna o próximo caractere nos dados, mas não avança para o seguinte caractere. |
| Read | Retorna o próximo caractere nos dados e avança para o seguinte caractere. Uma versão sobrecarregada pode ler um bloco de caracteres. |
| ReadAsync | Lê assincronamente caracteres do StringReader em um buffer. |
| ReadBlock | Lê até um número máximo de caracteres do StringReader em um buffer começando em um índice especificado. |
| ReadBlockAsync | Lê assincronamente até um número máximo de caracteres do StringReader em um buffer começando em um índice especificado. |
| ReadLine | Lê caracteres do StringReader até encontrar o final da linha. |
| ReadLineAsync | Lê assincronamente caracteres do StringReader até encontrar o final da linha. |
| ReadToEnd | Retorna o texto restante do StringReader como uma String. |
| ReadToEndAsync | Retorna assincronamente o texto restante do StringReader como uma String |

//'@' It's a verbatim string literal. It ignores escape sequence

string text = @"Hi I'm Ali Asad.

I can help you in C# Certification Exam.

I've helped many individuals like you in their exam prep.

I believe if we work together, you can become:

Microsoft Certified Professional & Specialist in C#";

StringReader reader = new StringReader(text);

int currentLine = 0;

string line = "";

//return each line of string to 'line'

while ((line = reader.ReadLine()) != null)

{

Console.WriteLine("line{0}: {1}", ++currentLine, line.Trim());

}

StringWriter

StringWriter é uma classe do System.IO. É usado para gravar em uma classe StringBuilder. Com StringWriter, podemos escrever um caractere/string com o método Write ou WriteAsync e uma linha inteira de string com o método WriteLine ou WriteLineAsync. É uma maneira eficiente de usar o StringBuilder com o StringWriter para manipular a string. A classe StringWriter fornece métodos que facilitam a gravação sequencial de valores em uma string. A Tabela a seguir descreve os métodos mais úteis do StringWriter.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Flush | Libera todos os dados armazenados no StringWriter subjacente. |
| ToString | Retorna o conteúdo atual do objeto como uma String. |
| Write | Anexa um item aos dados da sequência. As versões sobrecarregadas anexam char, string, int, double e muitos outros tipos de dados. |
| WriteAsync | Anexa de forma assíncrona um caractere, sequência ou matriz de caracteres ao final dos dados da sequência. |
| WriteLine | Anexa um item aos dados da string da mesma forma que Write e adiciona uma nova linha. |

StringBuilder builder = new StringBuilder();

StringWriter swriter = new StringWriter(builder);

swriter.Write("Ali Asad");

//Ali Asad

Console.WriteLine(builder.ToString());

Um dos métodos no .NET Framework que espera uma instância do TextWriter é o XmlWriter.Create. Normalmente, você transmite uma instância do StreamWriter para poder criar um novo arquivo XML. Mas quando você deseja o XML resultante apenas na memória, pode passar um StringWriter.

var stringWriter = new StringWriter();

using (XmlWriter writer = XmlWriter.Create(stringWriter))

{

writer.WriteStartElement("book");

writer.WriteElementString("price", "19.95");

writer.WriteEndElement();

writer.Flush();

}

string xml = stringWriter.ToString();

//<?xml version="1.0" encoding="utf-16"?><book><price>19.95</price></book>

Console.WriteLine(xml);

Ao usar o XmlReader, você pode analisar um pouco de XML e acessar os elementos individuais. XmlReader espera uma instância do TextWriter, para que você possa passar um StringReader

var stringReader = new StringReader(xml);

using (XmlReader readerxml = XmlReader.Create(stringReader))

{

readerxml.ReadToFollowing("price");

// Make sure that you read the decimal part correctly

decimal price = decimal.Parse(readerxml.ReadInnerXml(), new CultureInfo("en-US"));

Console.WriteLine(price); //19,95

}

O método Serialize da classe XmlSerializer envia a saída para um TextWriter. Se você deseja serializar em uma string, pode enviar a saída para um StringWriter e depois usar o método ToString do StringWriter para obter o resultado. Se você precisar manipular os dados da string subjacente de outras maneiras, como remover ou substituir caracteres, o StringBuilder fornece mais flexibilidade.

Internamente, StringWriter e StringReader usam um StringBuilder. A única coisa que eles fazem é adaptar a interface do StringBuilder à do TextWriter e TextReader. Conforme pode ser visto com mais clareza no seguinte exemplo:

classProgram

{

staticvoid Main(string[] args)

{

StringReadWrite srw = new StringReadWrite();

}

}

publicclassStringReadWrite

{

StringBuilder sb = new StringBuilder();

public StringReadWrite()

{

WriteData();

ReadData();

Console.ReadKey();

}

publicvoid WriteData()

{

// Note that we are passing the StringBuilder sb object to the StringWriter

StringWriter sw = new StringWriter(sb);

Console.WriteLine("Please enter your first and last name...");

string name = Console.ReadLine();

// Write the name to the StringBuilder sb object

sw.WriteLine("Name: " + name);

// Close the sw stream object

sw.Flush();

sw.Close();

}

publicvoid ReadData()

{

// Note we are converting the sb object to a string and passing it to the StringReader

StringReader sr = new StringReader(sb.ToString());

Console.WriteLine("Reading the information...");

// Use Peek to see if another character exists in sb

while (sr.Peek() > -1)

{

// Read a line from the string and display it

Console.WriteLine(sr.ReadLine());

}

Console.WriteLine("Thank you!");

sr.Close();

}

}

//Please enter your first and last name...

//Rick Bob

//Reading the information...

//Name: Rick Bob

//Thank you!

Procurando por Strings

Ao trabalhar com cadeias de caracteres, você costuma procurar uma substring dentro de outra cadeia de caracteres, para analisar algum conteúdo ou verificar a entrada válida do usuário ou outro cenário.A classe String oferece alguns métodos que podem ajudá-lo a executar todos os tipos de ações de pesquisa. Os mais comuns são IndexOf, LastIndexOf, StartsWith, EndsWith e SubString.

Uma coisa a ter em mente é que os métodos de string podem ser sensíveis à cultura. É por isso que a maioria dos métodos aceita uma instância da enumeração StringComparison. Ao trabalhar com seqüências de caracteres, sempre tente evitar os métodos que não usam um valor explícito de StringComparison.

|  |  |  |
| --- | --- | --- |
| **Método** | **Descrição** | **Exemplo value = "<mycustominput>"** |
| IndexOf | Retorna o índice da primeira ocorrência de um caractere ou substring em uma string. Se o valor não puder ser encontrado, ele retorna -1. | value.IndexOf('y'); // 2 |
| LastIndexOf | O mesmo que IndexOf, exceto que esse método começa a pesquisar no final de uma sequência e passa para o início | value.LastIndexOf('m'); // 8 |
| StartsWith | Verifica se o string inicia com um determinado valor. Retorna verdadeiro ou falso. | if(value.StartsWith("<")) { } |
| EndsWith | Verifica se o string termina com um determinado valor. Retorna verdadeiro ou falso. | if(value.EndsWith(">")) { } |
| SubString | Recupera uma cadeia parcial de outra cadeia | value.Substring(3, 6); //custom |

Outra maneira de pesquisar uma string é usando uma expressão regular, que usa uma notação de correspondência de padrões que pode analisar rapidamente grandes quantidades de texto procurando um formato específico. Expressões regulares podem ser úteis ao validar a entrada do usuário (como um endereço de email, CEP ou data). O código abaixo retira todos os títulos dos nomes que você passa. Imagine quanto trabalho teria sido criar isso ao usar IndexOf e SubString.

string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";

string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels", "Abraham Adams", "Ms. Nicole Norris" };

foreach (string name in names)

Console.WriteLine(Regex.Replace(name, pattern, String.Empty));

//Henry Hunt

//Sara Samuels

//Abraham Adams

//Nicole Norris

Enumerando Strings

Uma string é uma matriz de caracteres. Você pode enumerar uma sequência como se fosse uma coleção típica. Como uma string implementa IEnumerable e IEnumerable<Char>, expõe o método GetEnumerator que você pode usar para iterar sobre uma string.

Você pode usar uma sequência em um loop foreach para verificar todos os caracteres individuais, como abaixo:

string value = "My Custom Value";

foreach (char c in value)

Console.WriteLine(c);

Também é possível dividir uma sequência de palavras e iterá-las. A seguinte linha divide a sentença em espaços; retorna um IEnumerable que pode ser iterado:

foreach (string word in"My sentence separated by spaces".Split(' '))

Console.WriteLine(word);

Um exemplo usando apenas StringBuilders (sem Strings), podemos escrer um programa que exiba todas as subsequências iniciais das letras do alfabeto A, AB, ABC e assim por diante.

StringBuilder letters = new StringBuilder("ABCDEFGHIJKLMNOPQRSTUVWXYZ");

StringBuilder line = new StringBuilder();

// Create the result StringBuilder.

StringBuilder result = new StringBuilder();

// Loop over the letters.

for (int i = 0; i < 26; i++)

{

// Add the next letter to line.

line.Append(letters[i]);

// Add line to the result.

result.AppendLine(line.ToString());

}

As String possuem muitos métodos e propriedades que ajudam a manipular dados de texto. Alguns deles estão listados abaixo.

|  |  |  |
| --- | --- | --- |
| **Método** | **Descrição** | **Exemplo value = "<mycustominput>"**  **string value2 = "Ali";** |
| Clone | Use para criar clone de sequência no tipo de objeto. | value.Clone() asstring; |
| CompareTo | Compare dois valores de sequência e retorne o valor inteiro. Retorna 0 para true e -1 para false | string value2 = "asad";  if ((value.CompareTo(value2)) == 0) |
| Equals | Compare duas strings e retorne true se forem iguais. | value.Equals(value2)//False |
| IndexOf | Retorne o número do índice da primeira ocorrência de um caractere especificado. | value.IndexOf('t') //6 |
| ToLower | Retorne a minúscula da string. | value2.ToLower() //ali |
| ToUpper | Retorne a caixa alta da string. | value.ToUpper()//<MYCUSTOMINPUT> |
| Insert | Retorna uma nova string na qual um novo caractere/string é inserido em um índice especificado de uma string. | value2.Insert(3, " Asad") //Ali Asad |
| Remove | Retorne uma nova string excluindo todos os caracteres de um índice especificado até o final. | value.Remove(3) //<my |
| Replace | Retorna uma nova string na qual a ocorrência de caracteres especificados é substituída por outra especificada personagens. | value2.Replace("Ali","Asad") //Asad |
| Split | Divida uma sequência em uma matriz de sequências baseada nos caracteres que ocorrem em uma sequência. | value2.Split(' ') |
| ToCharArray | Retornar uma nova matriz de caracteres que contém um caractere de uma sequência | value2.ToCharArray() |
| Trim | Remova os espaços em branco do início e do final de uma string. | value2.Trim() |

Formatando strings

Ao exibir seqüências de caracteres para o usuário, você deseja garantir que elas estejam no formato correto. Especialmente ao trabalhar com dados sensíveis à cultura, como DateTime ou números, é importante garantir que a sequência seja exibida de maneira adequada aos usuários atuais e suas configurações.

Formatar um valor para exibição é uma conversão de tipo particularmente importante. Até converter um DateTime, decimal ou double em algum tipo de String, você não pode exibi-lo para o usuário. Dois dos métodos mais úteis para realizar a “Formatação composta” de valores como strings são os métodos ToString e String.Format. Outra forma de formatação é a interpolação de strings que veremos mais adiante. As cadeias de formatação se enquadram em duas grandes categorias:

* **Formatação padrão:** permitem determinar como você deseja que um valor seja exibido em um nível alto. As cadeias de formatação padrão reconhecem a localidade, portanto, permitem que o programa produza uma saída apropriada para a localidade do computador. Por exemplo, a sequência de formato de data “d” indica um padrão de data curto e produz um resultado semelhante a 14/03/2014 nos Estados Unidos ou 14/03/2014 na França.
* **Formatação personalizadas:** permitem criar formatos que não são fornecidos pelas seqüências de formatação padrão. Por exemplo, a instrução a seguir produz um resultado semelhante ao: São agora 14 horas. Console.WriteLine (string.Format ("São agora {0: HH} horas", DateTime.Now)).

ToString

A classe de objeto fornece um método ToString que todas as outras classes herdam. Por padrão, esse método retorna o nome do tipo de um objeto como uma String, mas a maioria das classes para as quais faz sentido substitui esse método para retornar o valor do objeto como uma String. Por exemplo, se uma variável flutuante mantiver o valor 1,23, seu método ToString retornará o valor "1,23" como uma sequência. Por outro lado, se você definir uma classe Employee, por padrão, o método ToString retornará o nome da classe, semelhante ao Formatting.Employee. Se você usar o método ToString de uma variável sem parâmetros, obterá uma representação padrão de seu valor.

classEmployee

{

publicstring Name { get; set; }

publicdecimal Salary { get; set; }

}

decimal inteiro = newint();

decimal value = 1.23m;

Employee e = new Employee();

Console.WriteLine(inteiro); // 0

Console.WriteLine(value); // 1,23

Console.WriteLine(e); // Formatting.Employee

Console.WriteLine(e.ToString()); // Formatting.Employee

O método ToString também pode levar como parâmetros um provedor de formato, uma string de formatação ou ambos. Usando a sequência de formatação, você pode personalizar o texto resultante. Por exemplo, se o custo variável for flutuante, a instrução value.ToString ("0,000000") produzirá uma sequência contendo o valor do custo exibido mais casas decimais.

decimal value = 1.23m;

Console.WriteLine(value.ToString("0.000000")); // 1,230000

Console.WriteLine(value.ToString("0,000000")); // 0.000.001

Quando um objeto possui várias representações de seqüência de caracteres, a substituição de ToString não é suficiente. Por exemplo, um objeto Temperatura pode exibir sua temperatura em graus Fahrenheit, Celsius ou Kelvin. Um valor inteiro também pode ser exibido de várias maneiras. Talvez represente um número de telefone ou uma quantia em dinheiro. Para habilitar esse tipo de comportamento, você pode usar seqüências de caracteres de formato, que descrevem como um objeto deve ser exibido. O .NET Framework os utiliza para tipos numéricos, datas, horas e enumerações, como mostra abaixo.

double cost = 1234.56;

Console.WriteLine(cost.ToString("C", new CultureInfo("en-US"))); // $1,234.56

Você pode usar a mesma abordagem ao exibir um valor de data e hora. Dependendo da cultura, a saída formatada pode ser completamente diferente. A código a seguir mostra como usar cadeias de formato diferentes com um DateTime para uma cultura inglesa.

DateTime d = new DateTime(2013, 4, 22);

CultureInfo provider = new CultureInfo("en-US");

Console.WriteLine(d.ToString("d", provider)); // Displays 4/22/2013

Console.WriteLine(d.ToString("D", provider)); // Displays Monday, April 22, 2013

Console.WriteLine(d.ToString("M", provider)); // Displays April 22

Fornecer o CultureInfo correto é importante ao formatar valores. Ele contém todas as informações necessárias sobre como um tipo específico é exibido nessa cultura. Da mesma forma, é importante garantir que, quando você salvar valores em um banco de dados, por exemplo, você faz isso de uma maneira que não agride a cultura. Se os dados insensíveis à cultura forem carregados, eles poderão ser formatados dependendo do usuário que estiver visualizando os dados.

Você também pode implementar essa formatação personalizada em seus próprios tipos. Você faz isso criando um método ToString (string) no seu tipo. Overridingo ToString é uma boa prática. Se você não fizer isso, o ToString retornará por padrão o nome do seu tipo. Ao overrideo ToString, você pode atribuir um valor mais significativo, como mostra a exemplo.

classPerson

{

public Person(string firstName, string lastName, int age)

{

this.FirstName = firstName;

this.LastName = lastName;

this.Age = age;

}

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

publicint Age { get; set; }

publicoverridestring ToString()

{

return FirstName + " " + LastName;

}

publicstring ToString(string format)

{

if (string.IsNullOrWhiteSpace(format)) format = "FL";

format = format.Trim().ToUpperInvariant();

switch (format)

{

case"FL":

return FirstName + " " + LastName;

case"LF":

return LastName.ToUpper() + ", " + FirstName;

case"FSA":

return FirstName + " " + LastName + "(" + Age + ")";

default:

thrownew FormatException(String.Format(

"The '{0}' format string is not supported.", format));

}

}

}

Person p = new Person("John", "Doe", 24);

Console.WriteLine(p); // John Doe

Console.WriteLine(p.ToString()); // John Doe

Console.WriteLine(p.ToString("LF")); // DOE, John

Console.WriteLine(p.ToString("fsa")); // John, Doe(24)

IFormatProvider e IFormattable

Ao formatar seqüências de caracteres, você também pode usar um IFormatProvider. O IFormatProvider possui um método, GetFormat (Type), que retorna informações de formatação específicas para formatar um tipo. Todos os objetos CultureInfo implementam IFormatProvider. O objeto CultureInfo retorna um NumberFormatInfo ou DateTimeFormatInfo específico da cultura se uma sequência de caracteres ou DateTime estiver formatada. Dessa forma, você pode formatar uma string como cultura específica, passando um objeto CultureInfo para o método ToString.

Ao implementar seu próprio método de formatação ToString em um tipo, você também pode optar por aceitar um IFormatProvider. Ao fazer isso, você pode implementar a interface IFormattable. O uso do IFormattable garante que você possa integrar-se ao .NET Framework quando se trata de formatar seqüências de caracteres.

Ao implementar o IFormattable, você tem suporte para conversão de string pela classe Convert (que possui uma sobrecarga que aceita um objeto e o IFormatProvider). Você também pode suportar a Formatação Composta, na qual seu tipo é usado para criar uma sequência composta com outros tipos.

O método ToString permite converter o valor de uma única variável em uma String. O método Format estático da classe String permite criar uma String que pode conter os valores de muitas variáveis formatadas de maneiras diferentes. O método String.Format possui algumas versões sobrecarregadas, mas o mais comum toma como parâmetros uma string de formatação e um ou mais argumentos usados para preencher itens na cadeia de formatação. O código a seguir mostra um exemplo simples.

int a = 1;

int b = 2;

string result = string.Format("a: {0}, b: {1}", a, b);

Console.WriteLine(result); // Displays ‘a: 1, b: 2’

Os métodos ToString e String.Format entendem centenas de seqüências de caracteres de formato padrão e personalizadas. Objetos são de vários tipos e alguns são raramente usados. Para cada tipo, os dados são armazenados ou exibidos em um formato diferente. Alguns dos formatos estão listados abaixo:

Formatos numéricos padrão

|  |  |  |
| --- | --- | --- |
| **Especificador** | **Descrição** | **Exemplo** |
| "C" ou "c" | Usado para formatar o valor da moeda | $12,345.67 |
| "D" ou "d" | Usado para formatar dígitos inteiros com sinal negativo opcional | 12345 |
| "E" ou "e" | Usado para formatar notação exponencial | 1.234567E+004 |
| "F" ou "f" | Usado para o especificador de precisão para definir o valor flutuante fixo | 12345.67 |
| "G" ou "g" | Geral (ponto fixo ou científico, o que for mais curto | 12345.67 |
| "N" ou "n" | Usado para formatar números por separadores de grupos(com separadores decimais e milhares) | 12,345.67 |
| "P" ou "p" | Usado para exibir porcentagem com número(multiplicado por 100 e% adicionada) | 0.12 torna 12.00% |
| "X" ou "x" | Usado para exibir o valor hexadecimal(somente tipos inteiros) | 3039 |

Alguns desses formatos podem ter um especi ﬁ cador de precisão opcional que controla o número de dígitos exibidos. Para a maioria desses tipos, o especi ﬁ cador de precisão indica o número de dígitos a serem exibidos após o ponto decimal. Por exemplo, se o valor for 12345.67, então value.ToString ("C4") produz $ 12.345,6700. Para notação científica, o especificador de precisão indica o número de dígitos após o ponto decimal na mantissa. Por exemplo, se o valor for 12345.67, então value.ToString ("E2") produz 1,23E + 004.

Formatos DateTime padrão

|  |  |  |
| --- | --- | --- |
| **Formato** | **Descrição** | **Exemplo** |
| d | Short date | 3/14/2014 |
| D | Long date | Friday, March 14, 2012 |
| f | “Full” with short time | Friday, March 14, 2012 2:15 PM |
| F | “Full” with long time | Friday, March 14, 2012 2:15:16 PM |
| g | “General” with short time | 3/14/2014 2:15 PM |
| G | “General” with long time | 3/14/2014 2:15:16 PM |
| M ou m | Month/day | March 14 |
| t | Short time | 2:15 PM |
| T | Long time | 2:15:16 PM |
| Y ou y | Year/month | March, 2014 |

Além desses formatos padrão, a estrutura DateTime fornece quatro métodos que produzem resultados semelhantes aos especificadores dos formatos d, D, te T. Esses métodos são ToShortDateString, ToLongDateString, ToShortTimeString e ToLongTimeString

Controle de espaçamento

O espaçamento é útil para formatar a saída. String pode ajudar a formatar o espaçamento.

string name = "Ali";

int age = 22;

//Name Ali | Age 22

Console.WriteLine("Name {0,10} | Age {1, 10}", name, age);

Alinhadores de controle

Por padrão, as strings estão alinhadas à direita. Para criar uma sequência alinhada à esquerda em um campo, você precisa usar um sinal negativo, como {0, -5} para definir um campo alinhado à direita de 5 caracteres.

//- Ali | end

string name = "Ali";

Console.WriteLine("- {0,-8} |end", name);

# Interpolação de Strings

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. Este recurso está disponível a partir de C# 6. 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:

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.");

// Hello, Mark! Today is Sunday, it's 12:20 now.

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.

A estrutura de um item com uma expressão de interpolação é a seguinte:

{<interpolationExpression> [, <alignment>] [: <formatString>]}

Elementos em colchetes são opcionais. A tabela a seguir descreve cada elemento:

|  |  |
| --- | --- |
| **Elemento** | **Descrição** |
| interpolationExpression | A expressão que produz um resultado a ser formatado. A representação em cadeia de null é String.Empty. |
| alignment | A expressão constante cujo valor define o número mínimo de caracteres na representação de sequência do resultado da expressão. Se positivo, a representação da string está alinhada à direita; se negativo, está alinhado à esquerda. |
| formatString | Um formato string suportada pelo tipo do resultado da expressão. |

O exemplo a seguir usa os componentes opcionais de formatação descritos acima:

Console.WriteLine($"|{"Left",-7}|{"Right",7}|"); //| Left | Right |

constint FieldWidthRightAligned = 20;

// 3,14159265358979 - default formatting of the pi number

Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");

// 3,142 - display only three decimal digits of the pi number

Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi number");

Especifique uma cadeia de caracteres de formato compatível com o tipo do resultado de expressão seguindo a expressão de interpolação com dois-pontos (":") e o format string.

{<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 data = new DateTime(1731, 11, 25);

Console.WriteLine($"On {data:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote {Math.E:F5} in a letter to Christian Goldbach.");

Você especifica a largura mínima do campo e o alinhamento do resultado da expressão formatada seguindo a expressão de interpolação com uma vírgula (",") e a expressão constante:

{<interpolationExpression>, <alignment>}

Se o valor do alinhamento for positivo, o resultado da expressão formatada será alinhado à direita; se negativo, está alinhado à esquerda. Se você precisar especificar o alinhamento e uma sequência de formato, comece com o componente de alinhamento:

{<interpolationExpression>, <alignment>: <formatString>}

O exemplo a seguir mostra como especificar o alinhamento e usa caracteres de barra vertical ("|") para delimitar os campos de texto:

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}|"); //| Arithmetic | 3,500 |

Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a \* b),ValueAlignment:F3}|"); // | Geometric | 3,464 |

Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 / b),ValueAlignment:F3}|"); // | Harmonic | 3,429 |

Como mostra a saída de exemplo, se o comprimento do resultado da expressão formatada exceder a largura do campo especificada, o valor do alinhamento será ignorado.

Como usar seqüências de escape em uma sequência interpolada

Seqüências de caracteres interpoladas suportam todas as seqüências de escape que podem ser usadas em literais de seqüência de caracteres comuns. Para obter mais informações, consulte Sequências de escape de string.

Para interpretar seqüências de escape literalmente, use uma string literal literal. Uma sequência literal 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 verbais interpoladas válidas.

Para incluir um colchete, "{" ou "}", em uma sequência de resultados, use dois colchetes, "{{" ou "}}". Para obter mais informações, consulte a seção Escapando chaves do tópico Formatação composta.

O exemplo a seguir mostra como incluir chaves em uma sequência de resultados e construir uma sequência interpolada literal:

var xs = newint[] { 1, 2, 7, 9 };

var ys = newint[] { 7, 9, 12 };

Console.WriteLine($"Find the intersection of the {{{string.Join(", ", xs)}}} and {{{string.Join(", ", ys)}}} sets.");

var userName = "Jane";

var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";

var verbatimInterpolated = $@"C:\Users\{userName}\Documents";

Console.WriteLine(stringWithEscapes);

Console.WriteLine(verbatimInterpolated);

Como usar um operador condicional ternário

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-o entre parênteses, como mostra o exemplo a seguir:

var rand = new Random();

for (int i = 0; i < 7; i++)

{

Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" : "tails")}");

}

Como criar uma string de resultados específica da cultura com interpolação de string

Por padrão, uma cadeia de caracteres interpolada usa a cultura atual definida pela propriedade CultureInfo.CurrentCulture para todas as operações de formatação. Use a conversão implícita de uma seqüência de caracteres interpolada em uma instância System.FormattableString e chame seu método ToString (IFormatProvider) para criar uma sequência de resultados específica da cultura. O exemplo a seguir 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 data2 = DateTime.Now;

var number = 31\_415\_926.536;

FormattableString message = $"{data2,20}{number,20:N3}";

foreach (var culture in cultures)

{

var cultureSpecificMessage = message.ToString(culture);

Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");

}

Como o exemplo mostra, você pode usar uma instância FormattableString para gerar várias seqüências de resultados para várias culturas.

Como criar uma sequência de resultados usando a cultura invariável

Junto com o método FormattableString.ToString (IFormatProvider), você pode usar o método estático FormattableString.Invariant para resolver uma seqüência de caracteres interpolada em uma seqüência de resultados da InvariantCulture. O exemplo a seguir mostra como fazer isso:

string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture: {DateTime.Now}");

Console.WriteLine(messageInInvariantCulture);

Caracteres especiais

Para incluir uma chave, "{" ou "}", no texto produzido por uma cadeia de caracteres interpolada, use duas chaves, "{{" ou "}}". Como os dois-pontos (":") têm um significado especial em um item de expressão de interpolação, para usar um operador condicional em uma expressão de interpolação, coloque essa expressão entre parênteses.

O seguinte exemplo mostra como incluir uma chave em uma cadeia de caracteres de resultado e como usar um operador condicional em uma expressão de interpolação:

string nome = "Horace";

int age = 34;

Console.WriteLine($"He asked, \"Is your name {nome}?\", but didn't wait for a reply :-{{");

Console.WriteLine($"{nome} is {age} year{(age == 1 ? "" : "s")} old.");

//He asked, "Is your name Horace?", but didn't wait for a reply :-{

//Horace is 34 years old.