Nome: João Vitor T. de Campos

Construindo seus próprios tipos com Programação Orientada a Objetos

Falando sobre OOP (Programação Orientada a Objetos):

A programação orientada a objetos é um paradigma de programação que organiza o código em objetos que interagem entre si. Em C#, você cria classes para representar objetos e usa conceitos como encapsulamento, herança e polimorfismo para modelar o mundo real de forma eficiente e modular.

Um objeto no mundo real é uma coisa, como um carro ou uma pessoa, enquanto um objeto na programação muitas vezes representa algo no mundo real, como um produto ou conta bancária, mas também pode ser algo mais abstrato.

Em C#, usamos as palavras-chave **class** e **record** (principalmente) ou **struct** (às vezes) para definir um tipo de objeto.

EX:

```
public class Imovel
{
    public int ImovelId {get; set;}
    public string Nome {get; set;}
    public string Discretion {get; set;}
    public double Valor {get; set;}
    public int Comodos {get; set;}
    public LocalityType LocalityType {get; set;}
    public BusinessType BusinessType {get; set;}
}
```

Criação de biblioteca de classe .NET reutilizável:

1. Use sua ferramenta de codificação preferida para criar um novo projeto, conforme definido na lista a seguir:

- Modelo do projeto: Biblioteca de Classes (Class Library/classlib)
- Arquivo e pasta do projeto: PacktLibraryNetStandard2
- Arquivo e pasta da solução/workspace: Chapter05

2. Abra o arquivo PacktLibraryNetStandard2.csproj e observe que, por padrão, as bibliotecas de classes criadas pelo SDK .NET 7 têm como alvo o .NET 7 e, portanto, só podem ser referenciadas por outros assemblies compatíveis com o .NET 7, conforme mostrado no seguinte trecho de código:

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>net7.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
    </PropertyGroup>
</Project>
```

3. Modifique o framework para ter como alvo o .NET Standard 2.0, adicione uma entrada para usar explicitamente o compilador C# 11 e importe estaticamente a classe System.Console para todos os arquivos C#, conforme destacado no seguinte trecho de código:

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>netstandard2.0</TargetFramework>
        <LangVersion>11</LangVersion>
        <Nullable>enable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
    </PropertyGroup>
        <ItemGroup>
            <Using Include="System.Console" Static="true" />
        </ItemGroup>
    </Project>
```

Definindo uma classe com namespace: Para usar uma classe definida em um namespace em outro arquivo C# ou em outro projeto, você precisará adicionar uma referência usando a diretiva **using** no topo do código.

Compreendendo os membros:

Os membros podem ser campos, métodos ou versões especializadas de ambos. Aqui está uma descrição deles:

Campos: São usados para armazenar dados e dentro destes campos existem três categorias especializadas:

- Constante: Os dados nunca mudam. O compilador literalmente copia os dados para qualquer código que os lê.
- Somente leitura: Os dados não podem ser alterados após a classe ser instanciada, mas os dados podem ser calculados ou carregados de uma fonte externa no momento da instanciação.
- Evento: Os dados referenciam um ou mais métodos que você deseja executar quando algo acontece, como clicar em um botão ou responder a uma solicitação de algum outro código.

Métodos: São utilizados para executar instruções e mesmo dentro dos métodos existem categorias especializadas:

- Construtor: As instruções são executadas quando você usa a palavra-chave new para alocar memória e instanciar uma classe.
- Propriedade: As instruções são executadas quando você obtém ou define dados. Os dados são comumente armazenados em um campo, mas podem ser armazenados externamente ou calculados em tempo de execução. As propriedades são a maneira preferida de encapsular campos, a menos que o endereço de memória do campo precise ser exposto.
- Indexador: As instruções são executadas quando você obtém ou define dados usando a sintaxe de "array" [].
- Operador: As instruções são executadas quando você usa um operador como + e / em operandos do seu tipo.

Dividindo classes usando parciais:

Parciais são usados para dividir uma classe em várias partes físicas em arquivos diferentes. Isso é útil para separar a implementação de uma classe em várias partes, facilitando a manutenção. Voce pode estar fazendo isto usando a palavra-chave **partial**.

Imagine que queremos adicionar declarações à classe Person que são automaticamente geradas por uma ferramenta como um mapeador objeto-relacional que lê informações do esquema de um banco de dados. Se a classe for definida como partial, então podemos dividir a classe em um arquivo de código gerado automaticamente e um arquivo de código editado manualmente.

Em Person.cs, adicione a palavra-chave partial, como mostrado destacado no código a seguir:

```
public partial class Person
```

No projeto/pasta PacktLibraryNetStandard2, adicione um novo arquivo de classe chamado PersonAutoGen.cs. Adicione declarações ao novo arquivo, conforme mostrado no seguinte código:

EX:

```
namespace Packt.Shared;

public partial class Person

{

}
```

O restante do código que escreveremos para este capítulo será escrito no arquivo PersonAutoGen.cs.

EX2:

```
// Arquivo1.cs

public partial class Pessoa {

    public string Nome { get; set; }

}


// Arquivo2.cs

public partial class Pessoa {

    public int Idade { get; set; }
```

}

Armazenando um valor usando um tipo enum:

Às vezes, um valor precisa ser uma de um conjunto limitado de opções. Por exemplo, existem sete maravilhas do mundo antigo, e uma pessoa pode ter uma favorita. Em outros momentos, um valor precisa ser uma combinação de um conjunto limitado de opções. Por exemplo, uma pessoa pode ter uma lista de desejos das maravilhas do mundo antigo que deseja visitar. Podemos armazenar esses dados definindo um tipo enum.

Um tipo enum é uma maneira muito eficiente de armazenar uma ou mais opções porque, internamente, ele usa valores inteiros em combinação com uma tabela de pesquisa de descrições de string.

Ex:

```
namespace Packt.Shared;

public enum WondersOfTheAncientWorld
{
 GreatPyramidOfGiza,
 HangingGardensOfBabylon,
 StatueOfZeusAtOlympia,
 TempleOfArtemisAtEphesus,
 MausoleumAtHalicarnassus,
 ColossusOfRhodes,
 LighthouseOfAlexandria
}
```

- Em person.cs, adicione:

```
public WondersOfTheAncientWorld FavoriteAncientWonder;
```

- Em program.cs, adicione:

```
bob. FavoriteAncientWonder = WondersOfTheAncientWorld. StatueOfZeusAtOlympia;
WriteLine (
   format: "{0}'s favorite wonder is {1}. Its integer is {2}.",
```

arg0: bob.Name,

arg1: bob. FavoriteAncientWonder,

arg2: (int)bob. FavoriteAncientWonder);

- Execute o código e veja o resultado:

O mundo antigo favorito de Bob Smith é StatueOfZeusAtOlympia. Seu inteiro é 2.

O valor enum é armazenado internamente como um int para eficiência. Os valores inteiros são atribuídos automaticamente começando em 0, então a terceira maravilha do mundo em nosso enum tem um valor de 2.

Controlando o acesso com propriedades e indexadores:

Uma propriedade é simplesmente um método (ou um par de métodos) que age e se parece com um campo quando você quer obter ou definir um valor, simplificando assim a sintaxe. Indexadores permitem que você acesse elementos de uma coleção por meio de uma sintaxe semelhante à de um array.

Definindo propriedades com capacidade de atribuição:

Para criar uma propriedade com capacidade de atribuição, você deve usar a sintaxe mais antiga e fornecer um par de métodos - não apenas uma parte get, mas também uma parte set:

public string? FavoriteIceCream { get; set; }

Embora você não tenha criado manualmente um campo para armazenar o sorvete favorito da pessoa, ele está lá, automaticamente criado pelo compilador para você.

- Exigindo que propriedades sejam definidas durante a instanciação

O C# 11 introduz o modificador required. Se você o usar em uma propriedade ou campo, o compilador garantirá que você defina a propriedade ou campo para um valor quando o instanciar. Isso requer o direcionamento do .NET 7 ou posterior, então precisamos criar uma nova biblioteca de classes primeiro.

Vamos definir um indexador para simplificar o acesso aos filhos de uma pessoa:

1. Em PersonAutoGen.cs, adicione declarações para definir um indexador para obter e definir um filho usando o índice do filho, conforme mostrado no seguinte código:

```
public Person this[int index]
{
    get
{
    return Children[index];
 }
    set
 {
    Children[index] = value;
 }
}
```

Você pode sobrecarregar indexadores para que diferentes tipos possam ser usados para seus parâmetros. Por exemplo, além de passar um valor int, você também poderia passar um valor string.

2. Em PersonAutoGen.cs, adicione declarações para definir um indexador para obter e definir um filho usando o nome do filho, conforme mostrado no seguinte código:

```
public Person this[string name]
{
    get
 {
    return Children.Find(p => p.Name == name);
 }
    set
 {
    Person found = Children.Find(p => p.Name == name);
    if (found is not null) found = value;
```

```
 }
}
```

3. Em Program.cs, adicione declarações para adicionar dois filhos a Sam e, em seguida, acessar o primeiro e o segundo filho usando tanto o campo Children mais longo quanto a sintaxe do indexador mais curta, conforme mostrado no seguinte código:

```
sam.Children.Add(new() { Name = "Charlie", DateOfBirth = new(2010, 3, 18)
});
sam.Children.Add(new() { Name = "Ella", DateOfBirth = new(2020, 12, 24)
});
WriteLine($"Sam's first child is {sam.Children[0].Name}.");
WriteLine($"Sam's second child is {sam.Children[1].Name}.");
WriteLine($"Sam's first child is {sam[0].Name}.");
WriteLine($"Sam's second child is {sam[1].Name}.");
WriteLine($"Sam's child named Ella is {sam["Ella"].Age} years old.");
```

Mais sobre métodos:

Sobrecarga de Método: Sobrecarga de método permite que você defina vários métodos com o mesmo nome, mas com diferentes parâmetros.

```
public class Calculadora {
   public int Somar(int a, int b) {
      return a + b;
   }

   public double Somar(double a, double b) {
      return a + b;
   }
}
```

Métodos Estáticos: Métodos estáticos são métodos que pertencem à classe em vez de instâncias individuais da classe.

```
public class Matematica {
    public static int Dobro(int num) {
        return num * 2;
    }
}
```

Métodos de Extensão: Métodos de extensão permitem adicionar novos métodos a tipos existentes sem modificar a definição do tipo original.

```
public static class StringExtensions {
    public static string PrimeiraMaiuscula(this string input) {
        if (string.IsNullOrEmpty(input))
            return input;
        return char.ToUpper(input[0]) + input.Substring(1);
    }
}
```

Métodos Assíncronos: Métodos assíncronos permitem que você escreva código que pode ser executado de forma assíncrona, o que é útil para operações de E/S intensivas ou operações de longa duração que não bloqueiam a thread principal.

```
public async Task<string> ObterDadosAsync() {
    HttpClient client = new HttpClient();
    String result = await client.GetStringAsync();
    return result;
}
```

Correspondência de padrões com objetos:

A correspondência de padrões é uma técnica usada para extrair informações de objetos de forma concisa e segura.

Switch com Correspondência de Padrões:

```
object obj = new Pessoa("João", 30);
string mensagem = obj switch {
    Pessoa p when p.Idade < 18 => "Menor de idade",
    Pessoa p => $"Adulto com {p.Idade} anos",
    _ => "Objeto não reconhecido"
};
```

Padrões de Propriedade:

```
var pessoa = new Pessoa("Maria", 25);

if (pessoa is { Nome: "Maria", Idade: 25 }) {
    Console.WriteLine("É a Maria com 25 anos!");
}
```

Padrões de Tipo com Propriedades:

```
if (obj is Pessoa { Idade: > 18 }) {
    Console.WriteLine("Esta pessoa é maior de idade!");
}
```

Trabalhando com registros:

Eles são introduzidos no C# 9 e são úteis para modelar dados que não mudam após a criação. Registros são definidos de forma concisa e geram automaticamente construtores, propriedades e suporte para comparação estrutural. Eles suportam padrões de desconstrução e clonagem imutável, tornando-os eficazes em várias situações.

```
using System;
public record Pessoa(string Nome, int Idade);
class Program
{
    static void Main(string[] args)
```

```
    {
        Pessoa pessoa = new Pessoa("João", 30);
        Console.WriteLine($"Nome: {pessoa.Nome}, Idade: {pessoa.Idade}");
        var (nome, idade) = pessoa;
        Console.WriteLine($"Desconstruído - Nome: {nome}, Idade: {idade}");
        Pessoa novaPessoa = pessoa with { Idade = 35 };
        Console.WriteLine($"Nova idade: {novaPessoa.Idade}");
    }
}
```