

Aula 2

Programação I

Prof. Alan Matheus Pinheiro Araya

1

Conversa Inicial

2

Coleções e *generics*

- Nesta aula vamos abordar os principais conceitos e estruturas de dados da linguagem, além de explorarmos um conceito fantástico chamado de "*Generics*"

3

Tipos e Interações Básicas com Arrays

4

Arrays

- No C# podemos encontrar o tipo básico "Array" como uma implementação nativa da CLR
 - Um array é uma estrutura que armazena um tamanho fixo de elementos do mesmo tipo, um conjunto contíguo de elementos na memória
- Todo Array no C# herda da classe System.Array
 - Ela é a classe base para Arrays de uma ou mais dimensões

5

- Além dos Arrays você terá à sua disposição uma série de classes para lidar com dados de forma mais especializada
 - "As "Collections" (Coleções) provêm implementações para os mais variados cenários, como: Listas, Filas, Pilhas, Listas encadeadas, HashSets, entre outras

6

- A classe **Array** unifica os **types**, assim todas as coleções compartilham métodos e comportamentos previstos no **System.Array**
- Mesmo o **System.Array** implementa algumas Interfaces (comportamentos) do “universo” de “Collections”

7

▪ Exemplos de arrays nativos no C#

```
//Inicializando um array nativo
int[] inteiros = new int[10]; //um array de inteiros contendo 10 posições
char[] caracteres = new char[2] { 'o', 'i' }; //um array de caracteres com 2
posições já inicializadas na construção

var versoes = new Version[5]; //um array de uma classe qualquer
var matriz = new int[3,2]; //uma matriz 3x2 (array bi-dimensional)
var matriz3D = new int[3,3,3]; //uma matriz 3x3x3 (array tri-dimensional)
```

8

- Vamos explorar melhor os Arrays na prática?

9

Generics no C#

10

- Mecanismos para facilitar/auxiliar na reutilização de código:
 - Herança (inheritance) e Genéricos (generics)
- Com o Generics o C# é capaz de criar um “Template” (modelo) que reduz muito a carga de boxing, unboxing e casting

11

- O uso de uma classe com Generics é reconhecido pela sintaxe: **MinhaClasse<T>**
 - Onde o “<>” aponta para a utilização de Generics
 - E o T é apenas uma letra que pode representar qualquer Type

12

- Vamos ver exemplos práticos de uso da nossa classe "CustomStack"!

13

- O <T> pode ser substituído por qualquer Type
- Mas somente em tempo de compilação, isto é
 - Quando você instanciar uma variável, deve-se definir seu Type
- Atenção: ser genérico não significa ser dinâmico!

14

Cada instância deve ter seu Type bem definido

```
var customStack1 = new CustomStack<int>();
var customStack2 = new CustomStack<string>();
var customStack3 = new CustomStack<DateTime>();
var customStack4 = new CustomStack<object>();
var customStack5 = new CustomStack<enum>();
```

Após definida um Type para a instância ela somente aceitará elementos desse Type

```
var customStack1 = new CustomStack<int>();
customStack1.Push(1); //ok
customStack1.Push("Hello"); //Error
customStack1.Push(new object()); //Error
customStack1.Push(DateTime.Now); //Error
```

15

Se todos herdam de Object, por que Generics?

- Os Generics possibilitam escrever classes que podem trabalhar com qualquer tipo

16

- No exemplo ao lado, criamos uma nova classe
- Observe bem que ela trabalha com um array de objects e não possui "<>", logo não utiliza Generics

```
public class ObjectStack
{
    int position;
    object[] data = new object[10];
    public void Push(object obj)
    {
        data[position++] = obj;
    }
    public object Pop()
    {
        return data[--position];
    }
}
```

17

- Vamos observar um exemplo de uso de nossa classe "ObjectStack"

18

Generics Constraints

- No C# os *Generics* suportam um conceito chamado de "*Generics Constrains*" (restrições de genéricos)
- Este recurso é muito poderoso e faz com que as classes genéricas possam limitar seu escopo

19

As principais notações de restrição (*constraints*) são

- `class MinhaClasse where T : OutraClasseQualquer`
- `class MinhaClasse where T : interface`

20

- `class MinhaClasse where T : class`
 - Indica que o Type passado no T deve ser do tipo "Reference Type"
- `class MinhaClasse where T : struct`
 - Indica que o Type passado no T deve ser do tipo "Value Type"

21

A seguir podemos ver um exemplo de "Generics Constraints"

```
public class Classe1 { }  
public interface Interface1 { }  
  
public class GenericClass<T, U> where T : Classe1, Interface1  
    where U : struct  
{  
}
```

- **Vamos ver mais alguns exemplos práticos**

22

Arrays, Lists, Sets, Dictionarys e Collections

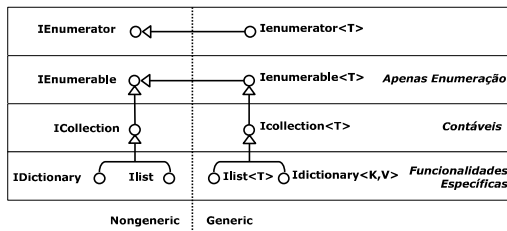
23

As Interfaces de coleções

- Para que várias classes possam ter abstrações comuns, o C# possui um conjunto de interfaces coleções essenciais

24

- Todas elas herdam de IEnumerator:



25

IEnumerator<T> e IEnumerable<T>

- A interface IEnumerator<T> define o protocolo (comportamento) básico de baixo nível pelo qual os elementos em uma coleção são percorridos, ou seja, enumerados, de maneira sequencial apenas. Sua declaração é a seguinte (Albahari, 2017, p. 302)

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

26

- Todas as coleções implementam por sua vez a interface IEnumerable<T>
- Esta interface basicamente retorna um IEnumerator<T>
- Isso significa que: "ser enumerável é poder ter vários IEnumerator<T> simultâneos"

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

27

- O bloco de loop "for" e "foreach" percorre um IEnumerator de forma nativa
- Isso por que a classe Array implementa a interface IEnumerable
- E o C# abstrai a inicialização do Enumerator
- Sempre que você inicia um novo "foreach" o C# cria um Enumerator que irá percorrer o array

28

ICollection<T>

- A interface ICollection<T> define comportamentos bem comuns para a maior parte das coleções

- Como a propriedade "Count" que armazena a quantidade de elementos, o método "Add", "Remove" e "Clear"
- Observe que a interface ICollection<T> herda de IEnumerable<T>

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool IsReadOnly { get; }

    void Add(T item);
    bool Remove(T item);
    void Clear();
}
```

29

30

ICollection<T>

- A interface **ICollection<T>** é sem dúvida uma das interfaces mais completas e úteis do C#

31

- Seus comportamentos trazem a coleção que irá implementar, além de todas as funcionalidades da **ICollection<T>**, funcionalidades extras como:
- Acessar um elemento em uma posição específica do vetor (**lista[pos]**), pesquisar o índice de um elemento (**IndexOf**), inserir um elemento em uma posição (**Insert**) e remover um elemento de uma posição (**RemoveAt**)

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    T this[int index] { get; set; }
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
}
```

32

- As classes concretas (que implementam as interfaces de coleções) podem implementar várias interfaces
- É o caso da classe **Array** ou **List**, por exemplo
- Vamos ver na prática?

33

HashSet<T>

- Outro tipo de coleção interessante é o **HashSet<T>**
- Essa coleção implementa um conceito de **Hashtable**
- É um tipo de algoritmo matemático para buscas muito performáticas

34

- O interessante dos **HashSets** é que eles implementam as interfaces **ICollection<T>** e **IEnumerable<T>**
- O que significa que você pode usar todos os métodos destas interfaces em um **HashSet** (além de enumerá-los)

35

- Por definição, um **HashSet** não armazena elementos duplicados
- Quando você adicionar um elemento duplicado ele irá automaticamente ignorá-lo
- Um dos principais métodos do **HashSet** é o **Contains**
- Nele a coleção realiza uma busca ótima de performance na notação **Big-O: O(1)**

36

Dictionary<TKey, TValue>

- Os dicionários (Dictionary) são coleções parecidas com HashSets, mas das quais você é responsável por controlar a "chave" e o "valor"
- Dicionários trabalham com uma struct chamada de "KeyValuePair"

37

- Todo elemento do dicionário será uma instância dessa struct
- Dentro dela você terá sempre uma chave e um valor

```
public struct KeyValuePair<TKey, TValue>
{
    public TKey Key { get; }
    public TValue Value { get; }
}
```

38

- Podemos destacar os principais métodos dos Dictionarys como:
 - ContainsKey
 - Retorna um booleano (true/false) indicando se a chave já existe ou não dentro do dicionário
 - TryGetValue
 - Retorna um booleano (true/false) indicando se o elemento existe e foi recuperado com sucesso da coleção
 - Além de retornar o elemento em si via interface de parâmetros out do C#

39

- TryAdd
 - Tenta adicionar um elemento na coleção
 - Caso já exista uma chave na coleção conflitante com a nova chave fornecida, retorna false
 - Caso consiga adicionar com sucesso, retorna true

40

Conversões entre Collections

41

Array é a base

- Para que você possa converter coleções de um tipo para outro, sem perder seu conteúdo, é importante lembrar
 - TODAS as coleções invariavelmente armazenarão seus dados em um Array
 - Isso pois o "[" oferece um modelo nativo de armazenamento compreendido e gerenciado pela CLR

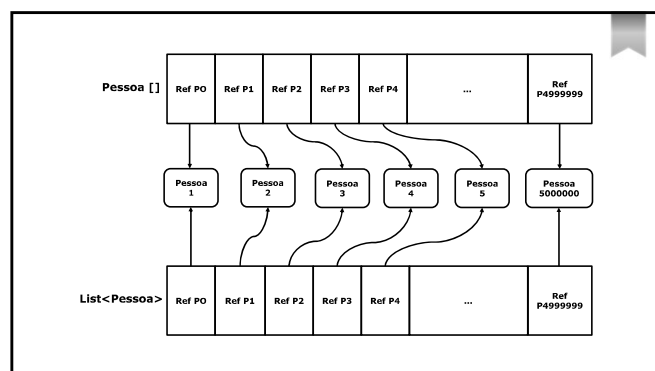
42

- Você pode converter coleções de quase todos os tipos umas para as outras
- Veremos na próxima aula que utilizando-se do LINQ é ainda mais simples fazer isso
- Resumindo
 - ✓ HashSets podem virar List e vice-versa, List pode virar Array e Dictionary pode virar List ou Array

43

- Vamos exercitar melhor esse conceito

44



45

Métodos de Extensão x Coleções

46

Métodos de Extensão

- Os métodos de extensão permitem que uma classe existente seja estendida com novos métodos sem alterar sua definição original

47

- Um método de extensão é um método estático de uma classe estática, onde o modificador "this" é aplicado ao primeiro parâmetro
- Observe o destaque no "this"

```
public static class StringHelper
{
    public static bool IsCapitalized(this string s)
    {
        if (string.IsNullOrEmpty(s))
        {
            return false;
        }
        return char.IsUpper(s[0]);
    }
}
```

48

- O uso do "static" na classe e "this" diz ao compilador para "adicionar" esse método dentro da classe
- **TODOS** os métodos de extensão devem ser estáticos e estarem definidos em classes estáticas
 - Em nosso exemplo, criamos um método chamado:
 - ✓ "IsCapitalized" para a classe string

49

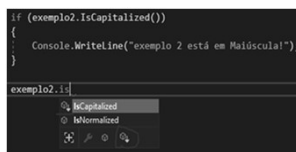
- **Veja a aplicação do método de extensão:**

```
bool isCaps = "aula 2".IsCapitalized();
Console.WriteLine(isCaps); // false

string exemplo2 = "String Caps";
if (exemplo2.IsCapitalized()) // true
{
    Console.WriteLine("exemplo 2 está em Maiúscula!");
}
```

50

- Você pode reconhecer um método de extensão no Visual Studio ou no VS Code através do ícone destacado na imagem abaixo:



51

Métodos de Extensão LINQ em Collections

- Como veremos em detalhes na próxima aula, o LINQ trouxe várias novidades ao C# enquanto linguagem
 - Uma delas são um conjunto de métodos de extensão para conversão entre coleções


52

- O namespace "System.Linq" adiciona uma série de métodos de extensão às coleções
 - Inclusive de conversão entre elas
 - Sua utilização é muito simples e chega a ser natural na maioria dos casos

53

- **Veja a seguir alguns exemplos de utilização dos métodos de extensão de conversão entre coleções**

54



```
//Lista de strings
var listaExemplo = new List<string>();

//Conversão de uma List para um HashSet utilizando métodos de extensão
var hashSetExemplo = listaExemplo.ToHashSet();

//Conversão de uma Stack para um List
var stack = new Stack<string>(listaExemplo);
listaExemplo = stack.ToList();

//Conversão de um Array para um List
var arrayExemplo = new string[100];
listaExemplo = arrayExemplo.ToList();

//Conversão de uma List para um Array
arrayExemplo = listaExemplo.ToArray();
```