Aula 1

Programação I

Prof. Alan Matheus Pinheiro Araya

1

Conversa Inicial

2

Introdução ao C#

- **Introdução ao C# e à plataforma .NET**
- **Sintaxe básica**
- **Propriedades, modificadores de acesso, interfaces e outros**
- ***Objects, strucys* e *types***
- ***Exceptions***

3

Introdução ao C# e à Plataforma .NET

4

História da linguagem e plataforma

- **O C# nasceu em 2002 como uma iniciativa da Microsoft para criar uma linguagem mais produtiva e amigável ao C++ e VB**
- **Junto da linguagem, nasceu também a plataforma .NET**

5

- **A história da linguagem C# também esta ligada à plataforma .NET**
- **Completa 20 anos em 2022**
  - **Possui um *roadmap* público e ativo**

6

- Em 2016, o C# e a plataforma .NET foram reconstruídos do zero
  - Neste ano, foi lançado o .Net Core
  - Código aberto (*open source*) e multiplataforma (Windows, Linux, Mobile, Mac, IoT, etc)
- O Net Core chegou em sua versão final, a versão 3.1, em 2020

7

- A mudança de nome .NetFramework para .Net Core não agradou muito a comunidade
- Em 2020, a plataforma foi rebatizada e foi lançado o .NET 5
- A partir de agora, as próximas versões seguem com o nome ".NET X" (em que X é a versão)

8

- A imagem abaixo traduz o atual nível de maturidade da plataforma .NET, suportando, em multiplataforma, várias linguagens, como o C#, o F# e o VB.Net

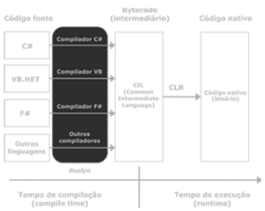
9

A máquina virtual do C#

- Assim como no Java, o C# executa sobre uma máquina virtual da linguagem
  - No C# a *Common Language Runtime* (CLR)
- O compilador do C# (o Roslyn) gera a *Common Intermediate Language* (CIL)
- A CLR traduz a CIL para binário (código executado pelo computador)

10

- C# e F#: Roslyn
  - CIL
- CLR: Código nativo



11

Sintaxe Básica

12

- O C#, assim como no Java, possui dois tipos básicos de variáveis
  - *Value type* (tipo por valor)
  - *Reference type* (tipo pro referência)

13

Tipos básicos (*value types*)

- *Value types* também são conhecidos como "tipos primitivos"
- Eles herdam diretamente de System.ValueType no C#. Seu principal tipo é a "struct"

Tipo	Bytes na memória	Limites
byte	1	De 0 a 255
sbyte	1	De -128 a 127
short	2	De -32,768 a 32,767
ushort	2	De 0 a 65,535
int	4	De -2 bilhões a 2 bilhões
uint	4	De 0 a 4 bilhões
long	8	-9 quatrilhões a 9 quatrilhões
ulong	8	0 a 18 quatrilhões
float	4	Números até 10 elevados a 38
double	8	Números até 10 elevados a 308
char	2	Caracteres unicode
decimal	24	Números com até 28 casas decimais
bool	1	True ou false

14

Declaração de variáveis

- O C# possui um modelo simples e conciso de declaração de variáveis
- Vamos ver os principais tipos primitivos e seu modo de declaração

```
//bool:
bool a = true;
bool b = false;
bool resultBool = a && b; // true AND false = false – operação básica com booleanos.
```

15

```
//int (inteiros):
int v1 = 10;
int v2 = -20;
int resultInt = v1 + v2; //10 -20 = -10
uint uv3 = 10 + 10; //20
```

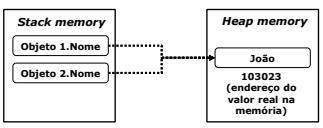
```
//char
char o = 'o';
char i = 'i';
char exc = (char)33; //o código 33 representa o char: !

//Não podemos "somar dois caracteres", pois isso seria uma string.
//Então vamos inicializar uma string a partir de um array (lista) de caracteres:
string resultChar = new string(new char[] { o, i, exc });
Console.WriteLine(resultChar); //Output no console: oi!
```

16

Tipos por referência

- Os *reference types* (tipos por referência) são também conhecidos como "*objects*" (objetos)
- Todos os objetos no C# herdam da classe System.Object
- Objetos sempre apontam para uma referência, como na esquema abaixo



17

Exemplo de instância de objetos

- Repare o uso da palavra-chave "*new*" para instanciar um novo objeto

```
//string:
string s1 = "oi!";
string s2 = new string('x', 5); // "xxxxx"
string s3 = s1 + s2; // "oi!" + "xxxxx" = "oi!xxxxx"
//objetos
object obj1 = new object();
Console.WriteLine(obj1.ToString()); // System.Object -
 este é um objeto vazio (com um valor default na memória)
object obj2 = s3;
Console.WriteLine(obj2.ToString()); // "oi!xxxxx" -
 este objeto aponta para a mesma referência da string s3
```

18

Blocos de decisão – *if*

```
//Blocos de decisão if:
int val1 = 10;
int val2 = 100;
if (val1 + val2 > 0)
{
  //executa: "Condição 1 verdadeira“
  Console.WriteLine("Condição 1 verdadeira");
}
val2 = int.MinValue;
if (val1 + val2 > 0)
{
  Console.WriteLine("Condição 2 verdadeira");
}
else
{
  //executa: "Condição 2 falsa“
  Console.WriteLine("Condição 2 falsa");
}
```

19

Blocos de decisão – *switch*

```
int teste1 = 1;
bool csharpRocks = true; //teste lógico, se 1 > 0 e a variável csharpRocks
for verdadeira
//decidirá em qual "case" cair
switch (teste1 > 0 && csharpRocks)
{
  case true:
        //executa:
       Console.WriteLine("C# Rocks!");
       break;
  case false:
       Console.WriteLine("C# é ruim!");
       break;
  default:
//repare que o compilador lhe ajuda marcando embaixo da
palavra console
//pois como é uma condição lógica, só temos true e false
//o default do bloco switch/case nunca será executado para
esta situação
       Console.WriteLine("Não pode cair aqui!");
       break;
}
```

20

Blocos de loop – *for*

```
//blocos de loop: for
for (int counter = 0; counter < 5; counter++)
{
     Console.WriteLine($"Olá, este é um for e esta é a
iteração: {counter}");
}

//resultado:
//Olá, este é um for e esta é a iteração: 0
//Olá, este é um for e esta é a iteração: 1
//Olá, este é um for e esta é a iteração: 2
//Olá, este é um for e esta é a iteração: 3
//Olá, este é um for e esta é a iteração: 4
```

21

Blocos de loop – *foreach*

```
string stringForLoop = "c# is the best!";
foreach (char caractere in stringForLoop)
{
     Console.WriteLine(caractere);
}
//resultado:
//c
//#
//
//i
//s
//
//t
//h
//e
//
//b
//e
//s
//t
//!
```

22

Blocos de loop – *while*

```
string stringForWhile = "c# rocks!";

while (stringForWhile.Length > 0)
{
    Console.WriteLine(stringForWhile);
    stringForWhile = stringForWhile.Substring(0, stringForWhile.Length-1);
    if (stringForWhile.EndsWith("#"))
    {
        stringForWhile = string.Empty;
    }
}
/*
c# rocks!
c# rocks
c# rock
c# roc
c# ro
c# r
c#
*/
```

23

Namespaces e classes

- **As classes podem estar organizadas dentro de *namespaces*, como "pastas", em uma estrutura de arquivos**
  - **Podem criar um conceito de níveis**
  - **Sempre separado por um ponto**
- **Você pode ter classes com o mesmo nome em *namespaces* diferentes**

24

Propriedades, Modificadores de Acesso e Outros

25

Propriedades

- Propriedades (*properties*) encapsulam comportamentos fundamentais de qualquer variável
  - *Get*: recuperar o valor da variável
  - *Set*: atribuir um valor a variável

26

- Exemplo de um *field* e uma *property*
  - A variável "nome" é um campo (*field*)
  - A variável "idade" é uma propriedade (*property*)

```
class Pessoa
{
    string nome;
    int Idade { get; set; }
}
```

27

- Propriedades podem implementar controles sobre o *get* e o *set*
  - Campos (*fields*) não possuem esse controle
- O *get* e o *set* são métodos implícitos

28

- Uma propriedade pode assumir duas formas: *full* e *simple*. Observe os dois exemplos

Full

```
string nomeCompleto;
public string NomeCompleto
{
    get
    {
        return nomeCompleto;
    }
    set
    {
        if (value.Contains(" "))
        {
            nomeCompleto = value.Replace(" ", "-");
            return;
        }
        nomeCompleto = value;
    }
}
```

Simple

```
public string NomeCompleto
{
    get;
    set;
}
```

29

Interfaces

- Declaram comportamentos, ou seja, métodos
- Seu objetivo é expor comportamentos comuns entre classes
- Auxiliam nos conceitos de abstração da orientação a objetos

30

- **É uma convenção utilizarmos a letra "I" na frente**
- **Sufixo "*able/iable*" (em inglês)**
- **O ideal é que seu nome reflita seu conjunto de comportamentos**
  - **Métodos**
  - **Propriedades**
  - **Retornos de valores constantes**

```
interface Icomestivel
{
    void Mastigar(object comida);
    int MaximoComida { get;}
    string Descrever()
    {
        return "Possibilita o comportamento de mastigar um alimento";
    }
}
```

31

Modificadores de acesso

- **Modificadores de acesso são palavras-chaves adicionadas antes de um tipo (*type*) e que podem limitar sua acessibilidade a outros tipos (classes, *strutcs*, etc.)**

32

- **Exemplos de modificadores de acesso do C#**
  - ***Public***
  - ***Internal***
  - ***Private***
  - ***Protected***
  - ***Protected internal***

33

Var: variáveis locais de tipo implícito

- **Podemos usar a palavra "var" para abstrair o tipo da variável local**
- **Não afeta a performance**
- **Não torna o tipo dinâmico, permitindo a ele mudar de tipagem**

34

- **Facilitam a leitura de tipos extensos, em especial com "*generics*"**
- **O var sempre é introduzido do lado esquerdo da declaração da variável**

```
//sem var
Dictionary<string, string> dicionario = new Dictionary<string, string>();

/com var
var dicionario = new Dictionary<string, string>();
```

35

Objects, Structs e *Types*

36

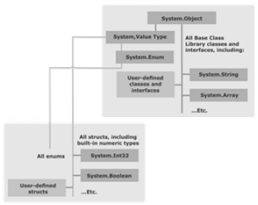
Types

- A palavra "*type*" ("tipo"), tem um significado especial no C#, pois todos os objetos e tipos primitivos são de um "tipo", ou seja, de um *type*
- Todos os objetos possuem um método "GetType()"
  - Descreve metadados do objeto
  - Ajuda a entender qual o *type* desse objeto
- O *type* define o objeto (nome, métodos, propriedades, atributos, etc.)

37

Objects

- Todos os objetos (*reference types*) herdam de um único *type*: "*object*" (System.Object)
  - Permite o *boxing* e o *unboxing*
  - Facilita as abstrações e o polimorfismo



38

- Observe o exemplo, no qual declaramos um int e o convertemos para *object* e depois para int novamente

```
int x = 9;
object obj = x; // "box"(empacota) o int dentro de um objeto


// Desempacota a operação acima. Fazendo um "cast" (conversão) do objeto para
// seu tipo original (int) que neste caso é um value type.
int y = (int)obj; // Unbox  (desempacota)
```

39

- *Boxing* não requer nenhum "cuidado" especial
- *Unboxing* requer um processo de conversão, chamado de c*asting*
  - Pode gerar exceções caso o tipo final não seja igual ao esperado no *cast*

```
// O processo de Unboxing requer um cast "explicito". O CLR (runtime) verifica
// que o value type é do mesmo tipo do objeto. Caso ele não seja, uma exceção
// do tipo InvalidCastException é lançada.
// O exemplo abaixo lançaria uma exeção, pois o value type "long" não é do mesmo tipo que o int
object obj = 9; // 9 é um int (quando apenas digitamos o valor "cru" na IDE)

long x = (long)obj; // InvalidCastException será lançada em tempo de execução pelo CLR
```

40

Structs

- São *value types*. Não herdam diretamente de *object*
- Todos os tipos primitivos são *structs*
- Não suportam herança
- Implicitamente é um *object*, mas é um tipo especial de *object*
  - Interpretado de forma diferente pela CLR
  - Seu valor nunca é uma referência, como outros *objects*

41

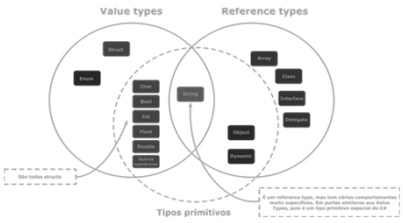
Enums

- São *value types*, assim como as *structs*
- Possuem a funcionalidade de "enumerar" listas tipadas e rígidas
- Cada item do enum possui implicitamente um valor int

```
public enum Direcoes
{
    Cima,
    Baixo,
    LadoEsquerdo,
    LadoDireito = 6
}
```

42

Estrutura de Types no .NET



43

Exceptions

44

Exceptions (exceções)

- São estruturas mensageiras de erros internos da linguagem
- São lançadas quando a CLR encontra um situação de erro/falha inesperada
- Encerram o programa de forma abrupta quando o código não possuir um bloco "*try/catch*"

45

- *Exceptions* são classes que carregam consigo metadados do erro. Existem três propriedades comuns e importantes nelas
  - *Message*
    - Uma *string* contendo a descrição do erro. Ajuda a entender qual é o problema

46

- *StrackTrace*
  - Uma *string* que representa todos os métodos que o código executou até o erro. Ajuda a entender como o erro aconteceu
- *InnerException*
  - Caso sua exceção possua outras exceções como causa raiz, essa propriedade estará preenchida com a exceção original

47

Try/catch/finally

- Blocos *try/catch/finally* são blocos que podem deter um erro inesperado
- O bloco *try* deve estar sempre acompanhado de um bloco *catch* ou um bloco *finally* ou de ambos

48