PROGRAMAÇÃO I

AULA 1

Prof. Alan Matheus Pinheiro Araya

CONVERSA INICIAL

O objetivo desta aula será introduzir os conceitos iniciais da plataforma .NET e da linguagem de programação C#.

Vamos abordar as principais estruturas de programação da linguagem para que você faça uma transição tranquila de qualquer outra linguagem imperativa para o C#.

TEMA 1 – INTRODUÇÃO AO C# E A PLATAFORMA .NET

O C# nasceu em 2002 de uma iniciativa da Microsoft em tentar unificar as diversas linguagens de programação que possuía e criar uma plataforma de desenvolvimento mais amigável e produtiva que o C++ e o VB (Visual Basic). No início, a Microsoft tentou usar o Java, mas, por problemas de incompatibilidade com bibliotecas de código nativo do Windows e licenciamento com a Sun, ela teve de criar sua própria linguagem e plataforma. Chamou, para isso, Anders Hejlsberg, um dos principais nomes por trás do Delphi. Logo, o C# surgiria com muita herança e conceitos comuns ao Java e ao Delphi.

Completando (em 2022) 20 anos de história, o C# e a plataforma .NET oferecem muitos recursos aos desenvolvedores. Possibilitando o desenvolvimento multiplataforma (para ambientes Windows, Linux, Mac e IoT) e ocupando um papel de destaque em grandes empresas do mundo inteiro.

O C# como linguagem de programação roda em cima da **plataforma** .NET, que está em constante evolução, desde o surgimento da linguagem C#. Porém, o .NET cresceu tanto que “abriga” dentro de si outras linguagens, como o VB.Net (parecido com o antigo VB mas suportado pela plataforma .Net) e o F#, que é uma linguagem funcional. Atualmente, a **plataforma .NET** tem um ciclo de evolução **específico** e **diferente** da **linguagem C#**. Como podemos ver na imagem a seguir, considerando a plataforma e linguagem em linha temporal:

Figura 1 – História do .NET e C#

Fonte: Araya, 2021.

1.1 .NET Framework, .NET Core e .NET 5/6

Uma confusão comum quando iniciamos no mundo .NET são suas nomenclaturas e evoluções. Quando o .NET surgiu ele nasceu como uma Framework, focado em desenvolvimento para Windows e de código fechado (proprietário Microsoft). Porém, em 2016 a Microsoft lançou uma nova versão do .NET 100% *open source* (de código aberto) e o batizou de .NET Core. Como ele não possuía ainda todas as funcionalidades da última versão disponível naquele momento (o .Net 4.6), veio o nome "*Core*" como uma forma de separar o "antigo" .NET Framework do novo. Além disso, o .NET Framework tinha problemas como: não ser modular, estar atrelado à distribuição/instalação via "*services pack*", entre outros.

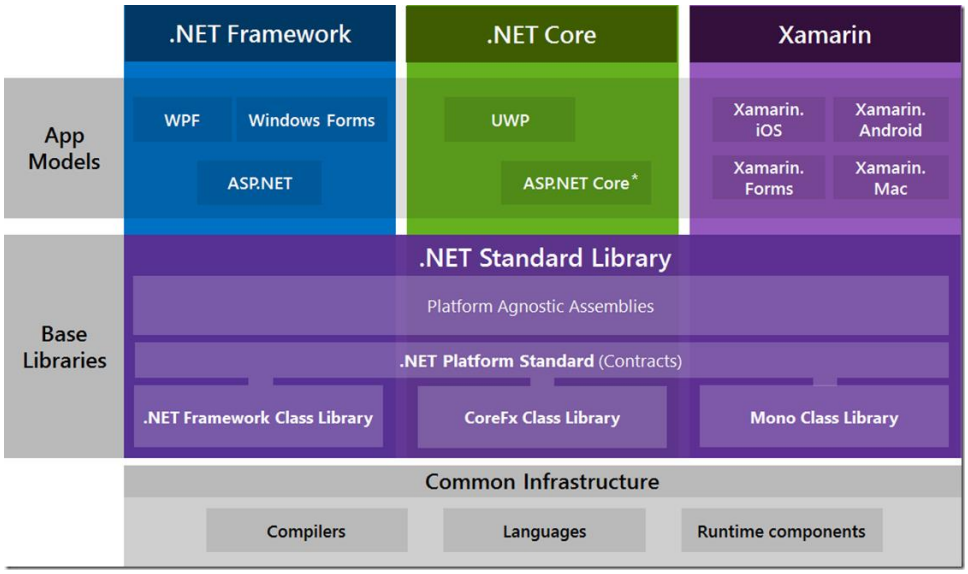
Com o lançamento do Core, a Microsoft catapultou muitos desenvolvedores iniciantes ou que estavam migrando entre linguagens para dentro do universo .NET.

No entanto, após o lançamento do .NET Core 3.1, a Microsoft percebeu que essas diferenças de **nomenclatura** entre *framework* e *core* **não estavam ajudando muito**, em especial os novatos. Então, ela decidiu que as próximas versões se chamariam apenas *NET X* (em que *X* é a versão).

Em novembro de 2020 foi lançado o .NET 5. O primeiro framework 100% open source, contendo, pela primeira vez, todas as *features* do .Net Core além das *features* do .NET Framework 4.8 (última versão do "antigo" framework).

A seguir podemos ter uma "visão de blocos" de como era estruturada a plataforma em 2016, quando foi conceituada a base do .Net Core (também utilizada no .NET). Podemos ver que na base dos componentes temos os compiladores e as linguagens (C#, F#, VBNet). Em cima dessa bloco comum, temos também a .Net Standard, que é um conjunto de interfaces e APIs de código que provem funcionalidades gerais a toda e qualquer Plataforma. Aqui está a essência do Framework. Todos os namespaces e classes-base utilizadas no seu dia a dia. As camadas de plataforma Core, Framework e Xamarin eram até então implementações em cima do NetStandard

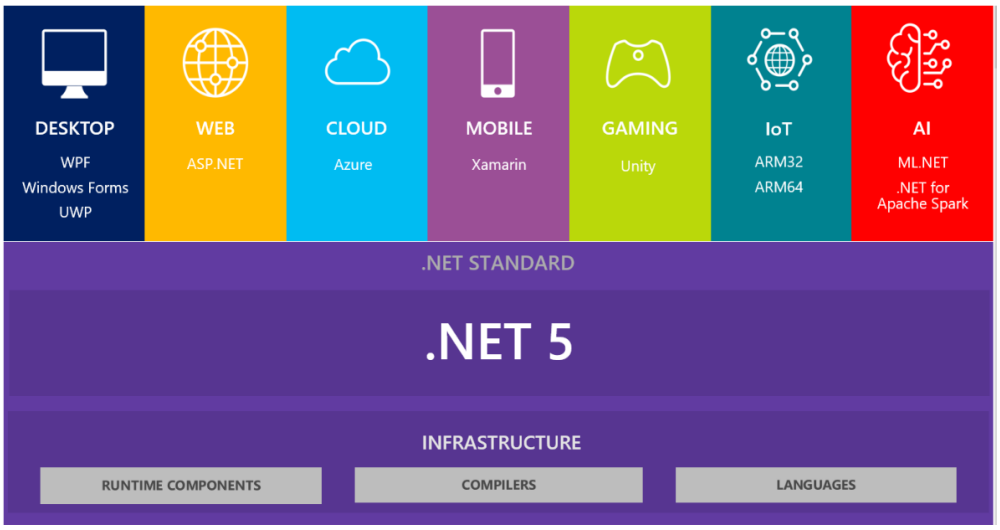
Figura 2 – Camadas da plataforma



Fonte: Torre, 2016.

Na figura a seguir, podemos ver como ficaram estruturadas as plataformas em cima do .NET 5 (que será a mesma base do 6). Temos todas as plataformas: Web, Mobile, Cloud, Game e mesmo IA em cima do .NET Standard (que é a base do framework). Isso permite que as plataformas evoluam de forma independente do Framework. E que a base do framework (NetStandard) também possa evoluir focando em novas funcionalidades comuns às plataformas.

Figura 3 - .NET 5



Fonte: Torre, 2016.

Em nossas aulas práticas, trabalharemos com a sua última versão, o .NET 6, que estará disponível em versão final LTS em novembro de 2021, mas que

pode ser utilizada em forma de preview sem nenhum problema, pois já é uma versão bem estável.

Saiba mais

LANDER, R. Announcing .NET 6 Preview 3. **Microsoft DevBlogs**. 2021. Disponível em: <https://devblogs.microsoft.com/dotnet/announcing-net-6-preview-3>. Acesso em: 31 ago. 2021.

1.2 A máquina virtual do .NET

Várias linguagens de programação possuem o conceito de “máquina virtual de execução”, que encapsulam o programa que está executando dentro de um ambiente virtual “protegido”, o qual é responsável por executar as tarefas com o sistema operacional.

O .NET e, por sua vez, o C#, utilizam este conceito, assim como o Java. Entre o sistema operacional e a aplicação existe uma **camada extra** responsável por "traduzir", mas não apenas isso, o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento (seja Windows ou Linux).

Você irá notar que uma máquina virtual é um conceito bem mais amplo que o de um interpretador. Seu objetivo é isolar o código da dependência com o sistema operacional, possibilitando, assim, múltiplos sistemas operacionais e ambientes de execução. Ela é responsável por gerenciar memória, threads, I/O de arquivos, a pilha de execução, comunicação com a rede etc.

A aplicação executa sem nenhum envolvimento direto com o sistema operacional. Sempre conversando **apenas** com a máquina virtual do C#: a *Common Language Runtime* (CLR). A **CLR** é o ambiente de execução para todas as linguagens da plataforma .NET, não apenas para o C#. Certamente isso não foi uma revolução. Como já foi mencionado, esse conceito não é um mérito do C#, sendo que o Java também possui sua própria máquina virtual, a JVM.

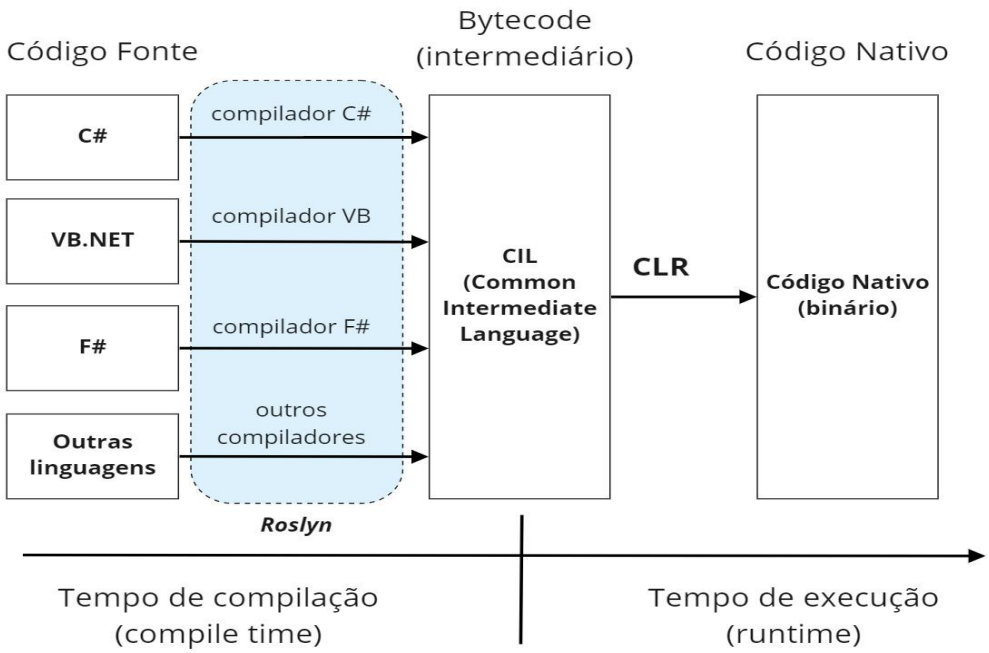
A CLR **isola** totalmente a aplicação do sistema operacional. Se uma aplicação rodando no CLR termina abruptamente, ela **não afetará** as outras máquinas virtuais e nem o sistema operacional.

Como a máquina virtual deve trabalhar com diversas linguagens de programação diferentes (C#, F#, VB.Net), a CLR não pode executar diretamente o código do C#, ela precisa executar uma linguagem intermediária comum a

todas as linguagens da plataforma .NET: a “Common Intermediate Language” (CIL). Para gerar o **CIL** que será executado pela CLR, precisamos passar o código C# por um compilador da linguagem, neste caso o “*Roslyn”* (veja mais nas referências*)*. O compilador lê o arquivo com o C# e o traduz para o **código intermediário** (CIL), que será executado pela máquina virtual (CLR).

Na figura a seguir, podemos ver como isso ocorre.

Figura 4 – CLR e CIL no .NET



Fonte: Araya, 2021.

TEMA 2 – SINTAXE BÁSICA

Agora, vamos apreender algumas noções básicas sobre a sintaxe do C#. No C#, assim como no Java, temos dois conjuntos básicos de “tipos” de variáveis dentro da linguagem, tipos de **valor** e tipos de **referência**. Dentre os tipos de valor, podemos destacar os tipos “primitivos”.

Tabela 1 – Tipos “primitivos”

Tipo	Bytes na memória	Limites
byte	1	0 a 255
sbyte	1	-128 a 127
short	2	-32,768 a 32,767
ushort	2	0 a 65,535
int	4	-2 bilhões a 2 bilhões
uint	4	0 a 4 bilhões

long	8	-9 quatrilhões a 9 quatrilhões
ulong	8	0 a 18 quatrilhões
float	4	Números até 10 elevado a 38
double	8	Números até 10 elevado a 308
char	2	Caracteres Unicode
decimal	24	Números com até 28 casas decimais
bool	1	True ou false

Os tipos primitivos são derivados das structs. Abordaremos mais sobre eles nos próximos temas.

A definição usada para tipos por valor e por referência no C# é similar ao Java: as variáveis de tipo por valor (value type) contêm dentro delas um valor, enquanto as variáveis por referência (reference types) contêm uma referência ao valor real. Podemos resumir da seguinte forma: value stypes derivam de System.ValueType enquanto reference types derivam de System.Object.

2.1 Declaração de variáveis e inicialização

No C#, a declaração de variáveis está fortemente atrelada ao type dela, ou seja, uma vez que declaramos um tipo numérico, não podemos atribuir um valor booleano nele, por exemplo.

Vemos seguir um exemplo de declaração de variáveis de alguns dos tipos primitivos (value types).

```
//Declaração de tipos primitivos:

//bool:
bool a = true;
bool b = false;
bool resultBool = a && b; // true AND false = false - operação básica com booleanos
Console.WriteLine(resultBool);

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

//long:
long l1 = 100000000;
long l2 = -5000;
long resultLong = l1 + (l2 * l1); //100000000 - 5000 * 100000000 = -499900000000
Console.WriteLine(resultLong);

//float:
float f1 = 10.5f; //note usamos o final (f) para indicar o compilador que trata-
se de um float
float pi = 3.1415926535f;
float resultFloat = pi * f1; // 10,5 * 3.1415926535 = 32.9867249
resultFloat = resultFloat / v2; // -1.64933622
Console.WriteLine(resultFloat);

//double
double d1 = 2000000.1231231313;
double d2 = 100;
double resultDouble = d1 * d2; // 2000000.1231231313 * 100 = 200000012.31231311

//* Note que aqui usamos o resultado de um float para multiplicar por um double
//* O compilador realiza um processo boxing e converte o tipo float para um tipo double,
resultando em um novo double
resultDouble = resultDouble * resultFloat; // -329867264.07392859
Console.WriteLine(resultDouble);

//char
char o = 'o';
char i = 'i';
char exc = (char)33;

//Não podemos "somar dois caracteres", pois isso corresponde a uma string
//então vamos inicializar uma string a partir de um array (lista) de caracteres:
string resultChar = new string(new char[] { o, i, exc });

//escrevendo no console:
Console.WriteLine(resultChar);
```

Exemplo de declaração de variáveis de alguns tipos de referência (*reference types*) ou de forma mais simples, **objetos**.

```
//Declaração de referência types, ou seja, objetos:

//string:
string s1 = "oi!";
string s2 = new string('x', 5); // "xxxxx"
string s3 = s1 + s2; // "oi!" + "xxxxx" = "oi!xxxxx"
Console.WriteLine(s3);

//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
```

Fica claro no exemplo citado que o “obj2” tem o mesmo valor que “s3”, pois strings são objetos e objetos, por serem “*reference types*” possuem apenas uma referência ao valor verdadeiro. Outra observação importante é o uso do “new” como inicializado de um objeto.

2.2 Blocos de decisão e loop

O C# possui inúmeras palavras reservadas (também chamadas de ***keywords***) para que você instrua o compilador de que o trecho de código deve ter um comportamento pré-determinado.

Blocos de decisão – IF / ELSE.

No C#, podemos executar código condicional utilizando a construção ***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");
}
```

Repare que no segundo bloco o código executado cairá no “else”.

Blocos de decisão – Switch/Case.

```
//bloco switch/case
int teste1 = 1;
bool csharpRocks = true;

//teste lógico, se 1 > 0 e a variável csharpRocks for veradeira
//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;
}
```

A sintaxe de switch/case do C# é igual à do Java. Variando apenas em opções diferentes que o bloco Switch pode receber para condicionar. No C# temos variações do Switch que aceitam strings, pattern matching, entre outras.

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
```

O bloco ***for*** no C# é igual às demais linguagens imperativas. Recebendo uma variável numérica, estabelecendo um ponto de checagem/parada e um ponto de incremento da variável numérica.

Blocos de loop – foreach.

```
//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
//!
```

A sintaxe do bloco ***foreach*** no C# é igual às demais linguagens imperativas. No caso do C#, recebemos uma variável que implementa a interface ***IEnumerable***. A partir disso, o código irá iterar sobre itens desta variável (vamos abordar isso de forma mais detalhada no próximo tema).

No exemplo, o código iterou sobre os **caracteres** que compõem nossa *string* armazenada na variável stringForLoop.

Blocos de loop – while.

```
//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#
 c#
*/
```

A sintaxe do bloco **while** no C# é igual às demais linguagens imperativas. O while receberá uma variável booleana ou o resultado de um método/operação booleano. A partir disso, entrará em repetição. A principal diferença dele para outros blocos é que você pode manipular a variável ou ponto de parada presente no: while (X). Veja no exemplo anterior, pudemos manipular a string quando ela chegou a conter apenas os caracteres: “c#”, nós substituímos sua referência para a referência de uma string vazia (string.empty). Logo na próxima condição, o comprimento da variável não satisfez mais a condição.

```
using BibliotecaComum.DAL;

namespace ConsoleApp.Aula1
{
    public class ClasseExemplo
    {
        SubDir.Item item = new SubDir.Item();
    }

    public class ClasseExemplo2
    {
        ClasseExemplo3 exemplo3 = new ClasseExemplo3();
    }

    public class ClasseExemplo3
    {
        ClasseExemplo exemplo1 = new ClasseExemplo();
    }
}


namespace ConsoleApp.Aula1.SubDir
{
    public class Item
    {
        ClasseExemplo3 exemplo3 = new ClasseExemplo3();


         BibliotecaComum.DAL.Comum.Repositorio Repositorio = new BibliotecaComum.DAL
.Comum.Repositorio();

         RepositorioConfiguracao configuracao = new RepositorioConfiguracao();

    }
}
namespace BibliotecaComum.DAL.Comum
{
    public class Repositorio
    {

    }
}

namespace BibliotecaComum.DAL
{
    public class RepositorioConfiguracao
    {

    }
}
```

2.3 Namespaces e classes

Como podemos ver no bloco que apresentamos, um namespace pode ter várias classes. Esse é o intuito dele. Organizar as várias classes dentro de uma espécie de diretório. Em que você pode acessar “níveis” usando o “.”. Sendo

assim, podemos estruturar nosso sistema para que as classes fiquem em níveis de organização, como se fossem pastas do sistema operacional.

No exemplo dado vemos dois *namespaces* e podemos ver que na classe "*ClasseExemplo*" estamos inicializando uma instância da classe "Item". Repare que se faz necessário o uso do prefixo "SubDir". Isso pois o *namespace* onde esta classe item está fica um nível abaixo do *namespace* da *ClasseExemplo.* Repare a classe item, ao inicializar uma instância da "ClasseExemplo3" não precisamos utilizar nenhuma referência ao namespace.

Podemos observar, também, a classe item referenciando a classe "Repositorio", que está dentro de uma namespace bem diferente dela (classe item). Ela pode fazer isso usando um caminho completo ou, caso deseje simplificar, pode-se usar a diretiva "using" em cima do namespace para dizer ao compilador que você quer poder acessar qualquer classe daquele nível de namespace sem declarar seu caminho completo, como no exemplo da inicialização da classe "RepositorioConfiguracao".

TEMA 3 – PROPRIEDADES, MODIFICADORES DE ACESSO, INTERFACES E OUTROS

No C#, temos um "tipo de campo" chamado *Propriedade* (property), que é padrão da linguagem e fornece algumas características interessantes. O conceito de Propriedade em linguagens como o C# e Java é muito parecido, mudando pouco a forma de declaração e recursos das linguagens sobre as propriedades. Você notará que propriedades são a forma mais comum e prática de expor dados de suas classes (entre elas).

Vamos conceituar primeiro o que são campos (fields), propriedades (properties) e métodos (method) no C#. A seguir, podemos ver a classe "Pessoa" com os três tipos:

```
class Pessoa
{
    string nome;
    int Idade { get; set; }
    bool PodeAndar() { return true; }
}
```

- A variável “nome” pode ser descrita como um campo (*field*), pois é uma variável local, pertencente somente ao contexto da classe (de acesso privado), sem modificadores de acesso ou getters e setters nativos.
- A variável “Idade” é uma propriedade contendo getters e setters nativos. Vamos explorar os detalhes logo abaixo.
- PodeAndar é um método que retorna um booleano.

Note a diferença de “casing” (modo de escrever usando maiúsculas ou minúsculas) de cada uma. No C# adotamos a seguinte convenção (Albahari, 2017, p. 18):

- Parâmetros, variáveis locais e campos privados (private fields) devem ser escritos utilizando-se a notação **camel case**. Por exemplo, “minhaVariavel” (disponível em: <https://docs.microsoft.com/pt-br/style-guide/capitalization>).
- Demais identificadores, como classes, métodos, propriedades, namespaces, tipos, atributos etc. devem ser escritos utilizando-se a notação **pascal case**. Por exemplo, “MinhaPropriedade”.

3.1 Propriedades

Propriedades no C# devem ser compostas por instruções de *gets* e/ou *sets*. A diferença principal entre propriedades e campos é o modo como vamos expor elas. As propriedades permitem criar um modelo para que a variável receba valores e/ou ceda seus valores.

Uma propriedade precisa ter pelo menos um *get* ou um *set*. Eles possuem o seguinte comportamento:

- Get: é um trecho de código (similar a um método), que tem como objetivo retornar o valor da propriedade. Podendo fazer operações de checagem, atribuição de outros campos internos e inicialização de campos internos antes de retornar seu valor.
- Set: é um trecho de código (similar a um método), que tem como objetivo alterar o valor da propriedade. Geralmente modificando seu valor para o valor passado por parâmetro.

A seguir, podemos ver uma propriedade clássica do C# e um campo com métodos *Get* e *Set* que atuam de forma muito similar às propriedades, como no Java, por exemplo.

```
class Pessoa
{
    string Nome { get; set; }

    int idade;

    void SetIdade(int idadeDesejada)
    {
        idade = idadeDesejada;
    }

    int GetIdade()
    {
        return idade;
    }
}
```

Podemos notar que o comportamento de alterar o valor do campo (*field)* idade depende do método “SetIdade” e “GetIdade”. Já na propriedade Nome, esse comportamento é automático. A linguagem gera (dentro do CIL) o *get e set* necessários para que a propriedade tenha o mesmo comportamento do campo idade, simplificando a necessidade de métodos para isso. O C# cria um campo e os métodos de forma totalmente transparente ao desenvolvedor e implementa o comportamento padrão para eles: sempre que se atribuir o valor na propriedade, ele atribui o mesmo valor/referência no campo, sempre que buscar (pegar) o valor da propriedade, ele retorna o mesmo valor/referência do campo. (Griffiths, 2019, p. 184).

Podemos resumir do seguinte modo: **as propriedades são abstrações para armazenar, recuperar e alterar o valor de um campo**.

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

        nomeCompleto = value;
    }
}
```

A seguir, podemos ver outro exemplo no qual implementamos de forma explícita comportamentos no get e no set de uma propriedade:

Ao implementar de forma explícita o comportamento do get e do set precisamos também de um campo (field) para armazenar a referência de nossa propriedade. E ela se torna uma espécie de “camada” de acesso à referência final, na qual podemos transformar o dado antes de ele ser armazenado ou mesmo antes de ser retornado. Observe que, no exemplo acima, nós checamos o valor da variável especial value e caso ela contenha espaço em branco, nós substituímos esse espaço por “-“ antes de armazenar em um campo. Neste caso, o get não modifica o valor retornado, apenas retornando a referência para o campo “nomeCompleto”. Mas ele poderia, caso você precise, modificar o retorno.

Os modificadores get e set também podem ser suprimidos de forma a criar propriedades com apenas um dos dois comportamentos. Por exemplo, pode-se remover o set e manter apenas o get criando-se assim uma propriedade “somente leitura” (read-only). Isso pode ser muito útil em algumas situações. Sempre que se deseja sobrescrever o comportamento padrão da interface (que é atuar como uma camada para expor um get e set de um campo), deve-se implementá-la de forma explícita (Griffiths, 2019, p. 181).

Como propriedades explícitas são como métodos disfarçados, você pode escrever códigos extensos nelas. Mas isso não é uma boa prática e na medida do possível devem ser evitados comportamentos complexos dentro de propriedades (Griffiths, 2019, p. 182).

3.2 Interfaces

Uma interface é como uma classe, porém, apenas descreve comportamentos, ou seja, métodos e propriedades. Ela não pode implementar nenhum desses, apenas declará-los. Seu objetivo é descrever comportamentos comuns, para que classes que implementam a interface possam utilizar o conceito de abstração da orientação a objetos.

A seguir, podemos ver uma interface clássica:

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

É uma convenção no C# usarmos a leta “I” maiúscula na frente do nome da interface para diferenciá-la de uma classe. Outra convenção é o uso do sufixo “able/iable” em inglês. Esse sufixo é utilizado para dar sentido de que é uma ação possível de ser feita, como Access -> Accessible ou Wear -> Wearable. Em português nem sempre encontramos boas traduções literais para essas palavras. O mais importante ao declarar uma interface é que ela reflita um comportamento, e não uma ação ou um substantivo (Griffiths, 2019, p. 195).

Note que a utilização de interfaces é uma característica comum de muitas linguagens orientadas a objeto. No C# uma classe pode implementar quantas interfaces desejar. Devemos ler isso como “uma classe pode implementar quantos comportamentos desejar”.

```
abstract class Animal : IComestivel
{
    public int MaximoComida => 10;

    public void Mastigar(object comida)
    {
    }
}

class Cachorro : Animal, ICloneable
{
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}

class PetShop
{
    public void Teste()
    {
        Cachorro cachorro = new Cachorro();

        cachorro.Clone();
        cachorro.Mastigar(new object());
        int maximo = cachorro.MaximoComida;
    }
}
```

No exemplo anterior, vemos uma classe implementando a interface IComestivel. Note que, ao implementar a interface, você é obrigado a implementar os métodos e propriedades declaradas na interface. Observe que a classe cachorro herda de animal e ainda implementa outra interface, a IConeable. Essa interface define o comportamento de um objeto poder se clonar; obriga sua classe a implementar o método “Clone”.

```
abstract class Animal : IComestivel
{
    public int MaximoComida => 10;

    public abstract void Mastigar(object comida);
}

class Cachorro : Animal, ICloneable
{
    public object Clone()
    {
        return this.MemberwiseClone();
    }

    public override void Mastigar(object comida)
    {
    }
}
```

Uma interface classe abstrata pode também não implementar um comportamento de forma explícita e delegá-lo a sua implementação concreta. Observe no exemplo que a classe Animal declara o método Mastigar como “abstract” e assim não precisa implementá-lo. Deixando a implementação concreta deste método para outras classes que herdem dela. Isso é chamado de “abstract implementation” no C# e é muito útil para quando você deseja propagar um comportamento de uma interface em classes que herdam de uma classe abstrata. Em nosso exemplo, isso seria interessante, pois cachorros mastigam diferente de vacas. E ambos são animais. Logo, se a classe vaga herdar de Animal ela será obrigada a implementar o método “Mastigar” e esse método poderá ser diferente do método Mastigar da classe cachorro.

Interfaces por padrão declaram comportamentos por meio de métodos ou propriedades de forma pública. Por isso não se faz necessário o uso do modificador de acesso: “public” na frente de seus identificadores, veremos isso melhor no tópico a seguir 3.3.

3.3 Modificadores de acesso

Para promover encapsulamento, um tipo (type) ou membro de um tipo pode limitar sua acessibilidade a outros tipos (classes, variáveis e métodos) adicionando um dos cinco modificadores de acesso existentes antes de sua declaração (Albahari, 2017, p. 128-129).

- **public:**
  - Totalmente acessível. Esse é o comportamento padrão de interfaces e enumeradores. Torna acessível uma classe ou membro desta a outras classes e membros.
- **internal:**
  - Acessível somente dentro do mesmo “assembly” (dll) ou métodos da mesma classe. Esse é o tipo padrão de acessibilidade para “non-nested types” ou tipos não aninhados no c#.
- **private:**
  - Acessível somente ao tipo ou classe onde está contido. Esse é o tipo padrão de membros em classes ou structs.

- **protected:**
    - Acessível somente dentro do tipo/classe ou dentro de suas subclasses. Por exemplo, pode ser acessado dentro da classe mãe e dentro de todas as classes que herdam ela.
- **protected internal:**
    - É a junção do protected com internal. Tornar o tipo acessível a todas as suas subclasses e a todos os outros tipos dentro do mesmo “assembly” (dll) ao mesmo tempo (Griffiths, 2019, p. 300 e Albahari, 2017, p. 129).

Os modificadores de acesso são muito importantes, pois é com eles que você poderá limitar a exposição de dados de suas classes a outras classes. Os tipos de modificadores mais usados são: *public, private e protected.*

Deve-se declarar o modificador de acesso antes da declaração de uma classe ou membro dela (variável ou propriedade). Podendo ser utilizado também dentro do get e set das propriedades, como no exemplo a seguir.

```
public class ModificadoresAcesso
{
    //torna a variável "quantidade" privada, ou seja, de leitura e escrita exclusiv
a da classe "ModificadoresAcesso"
    private int quantidade;

    //isso limita o set para que apenas a classe "ModificadoresAcesso" possa altera
r seu valor, enquanto o get fica público
    //é como ter um método público de get e outro privado de set
    //muito útil para expor o valor de um campo mas limitar seu acesso a escrita à
outras classes
    public bool SetPrivado { get; private set; }

    //torna a variável "nome" pública, ou seja, de leitura e escrita aberta a todas
 as classes
    public string nome;

    //deixa a propriedade idade visível e alterável apenas a suas subclasses
    protected int Idade { get; set; }

    //limita o acesso a todas as classes e tipos que herdem desta classe
    protected internal int RecuperarPeso()
    {
        return 10;
    }

    //limita o acesso apenas a classe "ModificadoresAcesso"
    private void CalcularPeso()
    {
        //inicializa uma classe interna aninhada com a classe "ModificadoresAcesso"
        ClasseInterna objeto = new ClasseInterna();

        //chama um método com visibilidade interna
        objeto.Calcular();

        //Não podemos executar a linha abaixo, pois isso geraria um erro de compi-
lação
        //a variável quantidadeInterna pertence apenas à "ClasseInterna", pois está
 com visibilidade private
        //objeto.quantidadeInterna = 1;
    }

    internal class ClasseInterna
    {
        private int quantidadeInterna;

        //método com visibilidade interna apenas
        internal void Calcular()
        {

        }
    }
}
```

3.4 Var – variáveis locais de tipo implícito

O C#, assim como outras linguagens “tipadas”, obriga que o desenvolvedor sempre declare o tipo de variável que está trabalhando. Na versão 3.0, foi introduzido um modo diferente de declarar variáveis, em que foi introduzida uma nova palavra-chave: **var**.

Ela permite que você abstraia parte da declaração de variáveis para simplificar a escrita. Não produz nenhuma alteração de performance na aplicação ou mesmo de modo como as variáveis se comportam. Essa palavra-chave existe para facilitar a vida do desenvolvedor com tipos que são muito extensos (como vamos ver nas próximas aulas).

Um exemplo de uso do var pode ser visto a seguir.

```
public class TesteVar
    {
        public void Metodo1()
        {
            //declaração sem usar o var
            Dictionary<string, string> dicionario = new Dictionary<string, string>(
);

            //declaração usando o var
            var dicionarioComVar = new Dictionary<string, string>();

            if(dicionario.GetType() == dicionarioComVar.GetType())
            {
                //veja que o tipo das duas variáveis é o MESMO
                //o var apenas ajuda/abstrai a declaração delas
                Console.WriteLine("C# Rocks!");
            }
     }
```

Perceba no exemplo exposto como a declaração da variável “dicionario” é mais longa que a “dicionarioComVar”. Veja também que as duas são iguais em termos de tipo e de comportamento. Ambas terão o mesmo funcionamento dentro do C#.

No exemplo a seguir vemos um modo de como o **var** do C# não tem um comportamento dinâmico, como no Javascript ou em outras linguagens dinâmicas. O **var** do C# não pode mudar de tipo após ter sido declarado. O exemplo a seguir vai resultar em um erro de compilação (Albahari, 2017, p. 74):

```
//neste caso abaixo o compilador irá lançar um erro em tempo de build
var x = 5;
x = "hello";
```

O uso do ***var*** deve ser feito com cuidado apenas em uma situação: na qual ele degrade a legibilidade do código. Como ele remove nossa capacidade de compreender o tipo da variável, quando usado para obter o retorno de um método, por exemplo, nossa variável deve ter um nome muito legível e de fácil compreensão para que não tenhamos um código ambíguo ou de difícil interpretação para o leitor. No caso abaixo, sem passar o mouse em cima do método "*Next()"* e explorar seu tipo de retorno, **não** saberemos o tipo da variável "y" (Albahari, 2017, p. 74):

```
//cuidar com a legibilidade e facilidade de entendimento do código ao usar
 var
//o exemplo abaixo demonstra um típico caso de onde devemos procurar ser o
 mais
//claro possíveis em nossos nomes de variáveis
var r = new Random();
var y = r.Next();
```

TEMA 4 – OBJECTS, STRCUTS E TYPES

No C#, como vimos no Tema 2, existem dois tipos de variáveis: *Reference Types* e *Value Types*. Quando falamos de "types" e objetos, assim como acontece no Java, todos os objetos herdam de um tipo único: "*Object*" (*System.Object).*

Com isso, ganhamos muitas funcionalidades e facilidades para trabalhar com conversões, *boxing e unboxing*. Como todas as instâncias de classes que você pode declarar herdam majoritariamente de um único tipo, isso nos dá uma grande versatilidade.

Como sabemos, um objeto é a instância de uma classe, ou seja, quando usamos a palavra reservada ***new*** para inicializar uma classe, ela passa a ser um objeto. Dentro do C#, as classes têm a seguinte anatomia básica:

```
class MinhaClasse
{
}
```

Podendo ainda ter (Albahari, 2017, p. 79):

- Antes da palavra *class*:
    - Atributos e modificadores de acesso da classe. Os modificadores são: public, internal, abstract, sealed, static, unsafe, and partial

- Logo após o nome da sua classe:
    - Parâmetros de tipos genéricos, uma classe mãe/base e interfaces
- Dentro das chaves:
    - Métodos, propriedades, indexadores, eventos, campos, construtores, operadores de overload, tipos aninhados (subclasses) e finalizadores ou destrutores

A seguir, temos um exemplo bastante completo tentando abordar diversas possibilidades básicas das classes, como modificadores de acesso, herança, implementação de interfaces, construtores, destrutores etc.

```
using System;

namespace ConsoleApp.Aula1
{
    /// <summary>
    /// Nossa classe abstrata mae. Repare que a herança de System.Object fica
 marcada com uma cor opaca pelo Visual Studio.
    /// Isso significa que a herança é redundante, ou seja, todas as classes
herdam de system.object, mesmo não declararando explicitamente.
    /// </summary>
    public abstract class ClasseMae : System.Object
    {
        //propriedade com protected (visível a todas as subclasses)
        protected long Quantidade { get; set; }

        //construtor
        public ClasseMae(long qtd)
        {
            Quantidade = qtd;
        }

        //método abstrato
        public abstract void MetodoAbstrato(string nome, string sobreNome);

        //método estático (não requer inicialização)
        public static long CalcularTicks(int valor)
        {
            return DateTime.Now.Ticks * valor;
        }
    }

    /// <summary>
    /// Classe filha. Repare no "sealed", que significa que a classe não pode
    ser herdada ou modificada. Este é um tipo de modificador de acesso
    /// especial de classes.
    /// Observe que ela HERDA da ClasseMae e implementa a
    /// interface IColonable
    /// </summary>
    public sealed class MinhaClasseCompleta : ClasseMae, ICloneable
    {
        //propriedade privada
        private string Nome { get; set; }

        //campo/field auxiliar interno (privado)
```

```
        string nomeCompletoAuxiliar;

        //Repare que o construtor recebe um valor inteiro e chama o
        //construtor da classe mãe usando a referência "base"
        //passa como parâmetro o valor para um método estático (que não
        //requer que a classe esteja inicializada para executar)
        //passando como retorno deste método o valor para o construtor
        public MinhaClasseCompleta(int qtd) : base(MinhaClasseCompleta.Calcul
arTicks(qtd))
        {
            //iniciliazação de campos e propriedades
            nomeCompletoAuxiliar = string.Empty;
            Nome = nomeCompletoAuxiliar;
        }

        //método público da interface
        public object Clone()
        {
            return MemberwiseClone();
        }

        //implementação do método abstrato
        public override void MetodoAbstrato(string nome, string sobreNome)
        {
            nomeCompletoAuxiliar = string.Join(" ", nome, sobreNome);
            Nome = nomeCompletoAuxiliar;
        }

        //destrutor/finalizador da classe
        ~MinhaClasseCompleta()
        {
            Nome = null;
        }
    }
}
```

4.1 Boxing, unboxing e casting

Como o C# é uma linguagem tipada e todos os seus tipos de referência herdam a classe System.Object por padrão, isso nos permite a utilização de um tipo de comportamento chamado *boxing* (empacotar) e *unboxing* (desempacotar).

Para melhor exemplificar isso, vamos analisar o código a seguir:

```
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)
```

Observe que o int x é convertido para um objeto. Quando isso acontece, uma nova posição na memória é alocada para “encapsular” o *value type*. Essa posição apenas tem um ponteiro de memória para o *value type* original e um

pequeno espaço na memória (16 bits) para representar System.Type (veremos isso no próximo tema) deste objeto. Por isso o nome "*boxing*". Quando convertemos o objeto novamente para int, o nome desta operação é "***cast***". Chamamos o processo de conversão entre tipos (*value types* ou mesmo *reference types*) de "***Casting***". Esse processo pode fazer *unboxing* (desempacotar) o valor original de um tipo (Albahari, 2017, p. 106).

```
// O processo de Unboxing requer um cast "explícito". 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
```

No exemplo dado podemos ver um caso em que aconteceria um erro em tempo de execução. Isso porque o C# checa conversões em tempo de compilação (enquanto você escreve e faz *build* do código) e em tempo de execução (quando o código está rodando, por meio do CLR). **Você não pode fazer unboxing explícito para tipos diferentes**.

```
//O código abaixo funciona sem problemas:
object obj = 9; // 9 é um int
long x = (int)obj; //unbox do int e cast implícito para long

//Assim como o exemplo abaixo também funciona:
object objeto = 3.5; // 3.5 é automaticamente inferido pela IDE como tipo: double

// unbox de objeto para double. Depois cast implícito para int.
// o valor do y será 3 (pois é apenas a parte inteira do valor 3.5)
int y = (int)(double)objeto;
```

Já no exemplo anterior, podemos ver um processo de *casting* chamado de *implict cast* (conversão implícita). Isso porque certos tipos, como no caso tipos numéricos, possuem conversão entre eles. Você pode converter um int para um *long*, pois um *long* armazena mais informação que um int (ver tabela no Tema 2). O contrário, como vimos no exemplo anterior, não é permitido. Neste exemplo também podemos ver o tipo *double* sendo convertido para int. Isso pode ser feito, mas o valor obtido desta conversão sofrerá um processo chamado de "*truncating*" (simplificação). Ao ser convertido para int o *double* perde seu valor decimal e o int recebe apenas a parte inteira (antes da vírgula) (Albahari, 2017, p. 107).

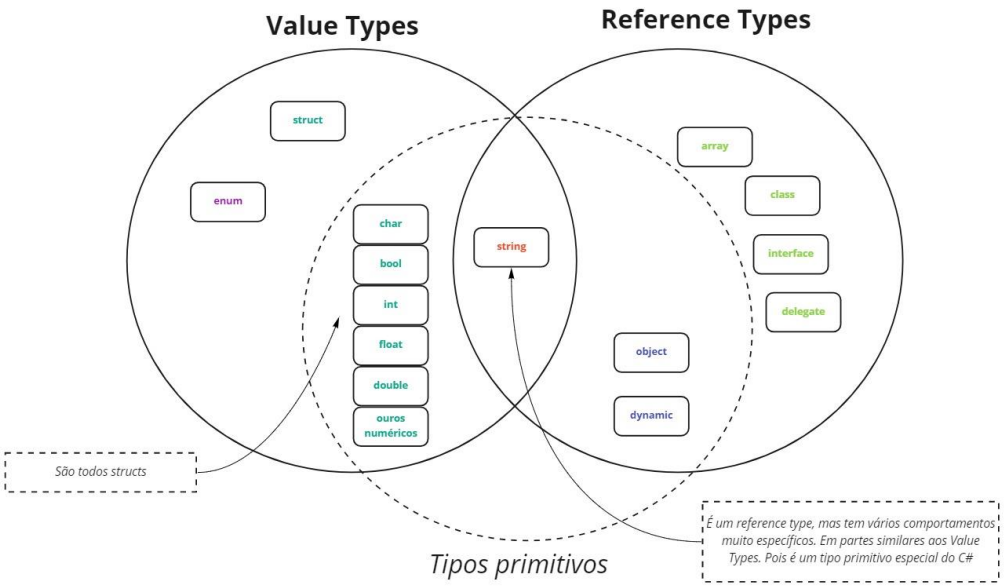
4.2 Types

A palavra *type*, traduzida livremente para “tipo”, tem um significado especial no C#. Isso porque todos os objetos e tipos primitivos são de um “tipo”, ou seja, de um *Type*. A partir de agora, vamos nos referir a *Types* usando a terminologia em inglês, para evitar confusões.

A classe *Type* define literalmente o tipo do objeto. Ela contém uma série de metadados sobre o objeto, podendo nos fornecer informações muito importantes para que o CLR possa distinguir um objeto de outro, afinal, todos os objetos herdam de *System.Object.*

Conforme vemos na figura a seguir, existe uma estrutura muito bem definida de hierarquia dos tipos e como o CLR trata eles:

Figura 4 – Esquema de types do C#



Fonte: Araya, 2021.

Recomendamos muito a leitura do link oficial da documentação do C# sobre o *Common Type Sistem* (CTS), presente na seção de referências da aula.

Logo, quando exploramos a anatomia de um objeto, podemos ver que ele possui um método chamado “*GetType*()”. Todos os objetos possuem esse método, pois ele está na classe *System.Object*. E esse método pode nos prover informações de metadados, ou seja, de como nosso objeto é. Ele serve para descrever nosso objeto para o CLR. Essa descrição é usada para muitos fins dentro da plataforma. Desde saber como armazenar, se é possível fazer *boxing* de um tipo para outro, ou mesmo para que a própria plataforma obtenha dados

de dentro de seu objeto para, por exemplo, permitir que você depure (faça o *debugging*) de seu código.

Observe na imagem a seguir que o *System.Object* possui dentro de si um *Type*. Dessa forma, tudo no C# possui um *Type*, um int, um *double*, uma *interface*, uma classe, uma *struct*, um *enum*, uma *excpetion* etc.

```
namespace System
{
    //
    // Summary:
    //     Supports all classes in the .NET class hierarchy and provides low-level services
    //     to derived classes. This is the ultimate base class of all .NET classes; it is
    //     the root of the type hierarchy.
    public class Object
    {
        //
        // Summary:
        //     Initializes a new instance of the System.Object class.
        public Object();

        ~Object();

        public static bool Equals(Object? objA, Object? objB);
        public static bool ReferenceEquals(Object? objA, Object? objB);
        public virtual bool Equals(Object? obj);
        public virtual int GetHashCode();
        //
        // Summary:
        //     Gets the System.Type of the current instance.
        //
        // Returns:
        //     The exact runtime type of the current instance.
        public Type GetType();
        public virtual string? ToString();
        protected Object MemberwiseClone();
    }
}
```

Fonte: Araya, 2021.

4.3 Structs

O C# permite também a criação de Structs. Structs são tipos de Value Type especiais. Assim como os tipos primitivos numéricos (int, long, double etc.), as Structs herdam de *System.ValueType*. Observe, no entanto, que por ser uma classe, mesmo um Value Type é um objeto (herdando de System.Object), como víamos no tópico anterior.

```
namespace System
{
    public abstract class ValueType
    {
        protected ValueType();

        public override bool Equals(object? obj);
        public override int GetHashCode();
        public override string? ToString();
    }
}
```

Fonte: Araya, 2021.

Structs são similares a objetos, com as seguintes diferenças notáveis (Albahari, 2017, p. 109):

- Uma Struct herda de System.ValueType, enquanto uma Class herda de System.Object, logo, Structs não são Reference Types.
- Structs não suportam herança. Você não pode criar abstrações de Structs como faz com as classes.

Além disso, Structs podem ter todos os membros de uma classe, com exceção de (Albahari, 2017, p. 109):

- Um construtor sem parâmetros.
- Destrutores/Finalizadores.
- Métodos marcados com Virtual ou Proteced.

4.4 Enums

Enums são *Value Types*, assim como as *Structs*. Porém, possuem a função de "enumerar", ou seja, "criar uma lista" de opções tipadas e rígidas. Para facilitar, vamos supor que você deseja criar uma lista de valores que possuam apenas quatro possibilidades para representar direções possíveis, ao invés de criar uma classe e declarar quatro propriedades, você pode usar um *Enum*, veja no exemplo a seguir:

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

Observe que cada item do Enum possui implicitamente um valor inteiro por definição. Seguindo a ordem que foi declarado, ou seja, o item "Cima" possui o valor "0", o item "Baixo" o valor "1" e assim sucessivamente. No trecho de código a seguir podemos ver os *enums* com ordens diferentes e como utilizá-los. Observe que podemos fazer "*casting* implícito" de um *Enum* para um Int. E observe outro caso muito comum de uso dos *Enum*, o bloco Switch/Case, isso pois o *Enum* nos permite uma excelente legibilidade das opções dentro de um Switch/Case, sendo na maioria vezes muito mais interessante que blocos if/else para essas estruturas.

Saiba mais

Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/builtin-types/enum)>. Acesso em: 31 ago. 2021.

```
/// <summary>
/// Observe que foi setado um valor explícito para cada item do enum
/// </summary>
public enum Direcoes
{
    Cima = 1,
    Baixo = 10,
    LadoEsquerdo = 20,
    LadoDireiro = 67
}

public class ExemploEnum
{
    public void MetodoExemploEnum()
    {
        var direcaoCima = Direcoes.Cima;
        var direcaoBaixo = Direcoes.Baixo;

        if ((int)direcaoCima > 20)
        {
            return;
        }

        switch (direcaoCima)
        {
            case Direcoes.Cima:
                break;
            case Direcoes.Baixo:
                break;
            case Direcoes.LadoEsquerdo:
                break;
            case Direcoes.LadoDireiro:
                break;
            default:
                break;
        }

    }
}
```

Todos os *Enums* herdam implicitamente da classe *System.Enum*. Você pode usar o *enum* por meio da palavra reserva ***enum*** ou da classe *Enum*. Ambos são a mesma coisa.

TEMA 5 – EXCEPTIONS

O tema "*exceptions*" (exceções) é comum em linguagens orientadas a objeto. Esse é o caso no C#, no VB, no Javascript (mesmo sendo uma linguagem

dinâmica) e no Java, este último de onde boa parte da base comum do C# foi herdada e aprimorada.

Exceptions ou exceções são estruturas de erro, as quais o framework do .NET (a CLR) cria quando encontra uma situação em tempo de execução da qual não é possível continuar. Por exemplo, vamos supor que você solicitou para seu programa converter uma string para um inteiro. Se a string possuir um caractere numérico em seu valor, o C# irá lançar uma exceção. Essa exceção é um modo da linguagem abortar abruptamente o que está fazendo e lhe avisar de que ocorreu um erro grave dentro de uma operação interna na sua biblioteca (Griffiths, 2019, p. 370).

É importante entender que uma exception interrompe a execução de um método e vai “parar” no primeiro bloco de catch do seu tipo que houver. Caso seu programa inteiro não possua um bloco de try/catch, provavelmente o processo da sua aplicação irá falhar e fechar, encerrando assim seu programa. No entanto, blocos try/catch devem ser evitados em relação ao seu uso “aninhado”, ou seja, um dentro do outro (mesmo que em métodos distintos, quando existe uma cadeia de chamadas entre eles, a CLR criará vários blocos aninhados), pois dessa forma você pode gerar problemas de performance em sua aplicação.

Por boas práticas de design de código é importante que métodos de negócio implementem uma forma de retornar um objeto que contenha indicadores de sucesso ou erro da operação. E evitar o uso de exceções. Mas ao mesmo tempo, em operações que dependam de acesso ao hardware ou a sistemas externos, como ler um arquivo do disco, transferir algum dado pela rede ou buscar alguma informação do banco de dados, podem falhar por eventos externos ao seu sistema. Nesses casos, é essencial proteger esses trechos de código com um bloco try/catch (Griffiths, 2019, p. 370).

A seguir, podemos ver um exemplo de como montar um bloco *try/catch/finally* no C#. Observe que quando um erro acontece e uma *exception* é lançada, a CLR irá mudar seu contexto de execução para dentro do bloco *catch* correspondente e colocar várias informações do erro e de seu contexto dentro da *exception*.

```
try
{
    //ao executarmos o trecho de código abaixo, vamos ver que o C# irá lançar
    //uma exception do tipo System.FormartException
    //pois o parâmetro "texto" não está no formato esperado pelo método Parse
 da classe int
    var texto = "uma string qualquer";
    int i = int.Parse(texto);

    //outros códigos quaisquer....
    //..
    //..
    i = i + 10;
}
catch (FormatException formatEx)
{
    //a CLR irá abortar a execução do bloco acima e o trecho: i= i+10 nunca s
erá executado
    //o objeto formatEx terá a mensagem e detalhes do erro
    Console.WriteLine(formatEx.Message);

    //podemos "relançar a exception" para outro bloco ou método superior ao n
osso, usando a palavra: throw
    //a palavra reservada throw é responsável por avisar o C# que uma exceção
 será lançada.
    //Observe abaixo que temos uma exceção nova sendo criada, usando a mensag
em da exeção formartEx como parâmetro
    throw new Exception(formatEx.Message);
}
catch (ArgumentException argEx)
{
    Console.WriteLine(argEx);
}
catch (Exception ex)
{
    //ao final, o código irá cair aqui
    //O type (classe) base de todas as exceções é a "System.Exception"
    //todas as demais exceções herdam dela

    //Você pode ter quantos catchs quiser em seu bloco try/catch
    //contanto que as exceções estejam em ordem de prevalência (herança), ou
seja,
    //as classes filhas acima das classes pai (exceções)
    //como é o caso da FormartException e ArgumentException acima, que herda
da classe Exception

    Console.WriteLine(ex);

    //Também é possível relançar a exceção original
    //o modo correto de fazer isso é simplesmente usar a diretiva "throw"
    //dessa forma, a StackTrace é mantida e você conseguirá rastrear melhor a
 fonte original do problema
    //esse é o modo correto de re-lançar exceções, quando você não irá trata-
las em seu bloco catch
    //e deixará para outro método/bloco superior ao seu trata-la
    throw;
}
finally
{
    //o bloco finally representa uma instrução para que SEMPRE
    //seja executado esse trecho de código, independente do
    //bloco TRY gerar um erro. Mesmo que passe por vários catchs
```

```
    //o bloco finally representa um bloco que SEMPRE SERÁ EXECUTADO
    //você deve usá-lo para "limpar" alguma variável ou estado
    //da aplicação e permitir que seja possível seu programa continuar
    //a execução, até retestando a execução do seu próprio método
}
```

Vamos praticar o uso de blocos try/catch/finally em nossas aulas práticas, utilizando-os em situações comuns de leitura e escrita de arquivos em disco.

5.1 Propriedades-chave de uma exceção

A classe Exception contém várias propriedades que auxiliam a entender o problema ocorrido em uma aplicação. O uso e interpretação correto de algumas dessas propriedades é essencial para qualquer programador. As três principais propriedades, segundo Albahari (2017, p. 165):

- Message:
    - Uma string contendo a descrição do erro em si. Essa é a principal propriedade para entender **o que é** o problema.
- StackTrace:
    - É uma string que representa todos os métodos que o código executou, da origem ou “inicio” da operação até o bloco catch. Esta é a principal propriedade para entender **como** o problema pode ter acontecido.
- InnerException:
    - Caso sua exceção possua outras exceções como causa, essa propriedade virá preenchida com a exceção original.

Perceba que a Exception não irá carregar com si um “log” detalhado do que aconteceu como causa do erro em si. A StackTrace apenas lhe mostrará a “CallTree”, ou seja, os métodos chamados até a fonte do erro. Mesmo assim, interpretar ela nem sempre é uma tarefa trivial. Por isso, o correto uso de log em métodos críticos é importante, para posterior análise e interpretação de problemas.

FINALIZANDO

Nesta aula, você teve a oportunidade de navegar dentro dos principais temas do universo do C#. Alguns temas abordados foram simplificados para os principais casos de uso encontrados diariamente no desenvolvimento com C#. No entanto, encorajamos fortemente o uso dos materiais de referência, em

especial da documentação oficial da linguagem (link na seção de referências), para aprofundar seus conhecimentos e resolver dúvidas sobre o funcionamento e utilização da linguagem.

O C# e universo .NET são fantásticos, permitindo a você criar programas para qualquer plataforma (Desktop, Mobile, Cloud/Web, IoT, Machine Learning, entre outras). Nas próximas aulas vamos explorar melhor as diversas possibilidades e funcionamentos da linguagem.

As primeiras referências a seguir fazem menção a vídeos e documentações de apoio ao aluno. A consulta habitual desses materiais é altamente recomendada.

REFERÊNCIAS

ALBAHARI, J.; ALBAHARI, B. **C# 7.0 in a Nutshell**. 7. ed. United States of America: O'Reilly Media Inc, 2017.

CHAUHAN, S. **A Brief Version History of .NET Framework**. DotNetTricks. 2020. Disponível em: <https://www.dotnettricks.com/learn/netframework/a-brief-version-history-of-net-framework>. Acesso em: 31 ago. 2021.

CLARION TECHNOLOGIES. **.NET Revolution**: An Overview of the .Net Framework Versions. [S.d.]. Disponível em: <https://www.clariontech.com/blog/the-.net-revolution-an-overview-of-the-.net-framework-versions>. Acesso em: 31 ago. 2021.

DOTNET - What's is .NET? **YouTube**. Disponível em: <https://www.youtube.com/watch?v=eIHKZfgddLM&list=PLdo4fOcmZ0oWoazjhXQzBKMrFuArxpW80&index=1>. Acesso em: 31 ago. 2021.

GRIFFITHS, I. **Programming C# 8.0**. 2. ed. United States of America: O'Reilly Media Inc, 2019.

MICROSOFT. **C# HowTo**. 20 dez. 2017. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/how-to/>. Acesso em: 31 ago. 2021.

MICROSOFT. **C# Version History**. 18 jun. 2021. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-version-history>. Acesso em: 31 ago. 2021.

MICROSOFT. **Common Type System**. 30 mar. 2017. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/standard/base-types/common-type-system>. Acesso em: 31 ago. 2021.

MICROSOFT. **Palavras-chave/ Keywords no C#**. 17 mar. 2021. Disponível em: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/>. Acesso em: 31 ago. 2021.

MICROSOFT. **Tipos de valores no C#**. 22 jan. 2020. Disponível em:<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/builtin-types/value-types>. Acesso em: 31 ago. 2021.

MICROSOFT. **Tipos por referência no C#**. 20 jul. 2015. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/reference-types>. Acesso em: 31 ago. 2021.

TORRE, C. de la. .Net Core, .Net Framework, Xamarin – The “WHAT and WHEN to use it”. **Microsoft DevBlogs**. 2016. Disponível em: <https://devblogs.microsoft.com/cesardelatorre/net-core-1-0-net-framework-xamarin-the-whatand-when-to-use-it/>. Acesso em: 31 ago. 2021.