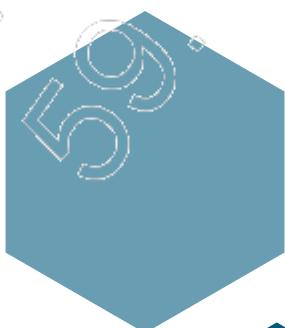


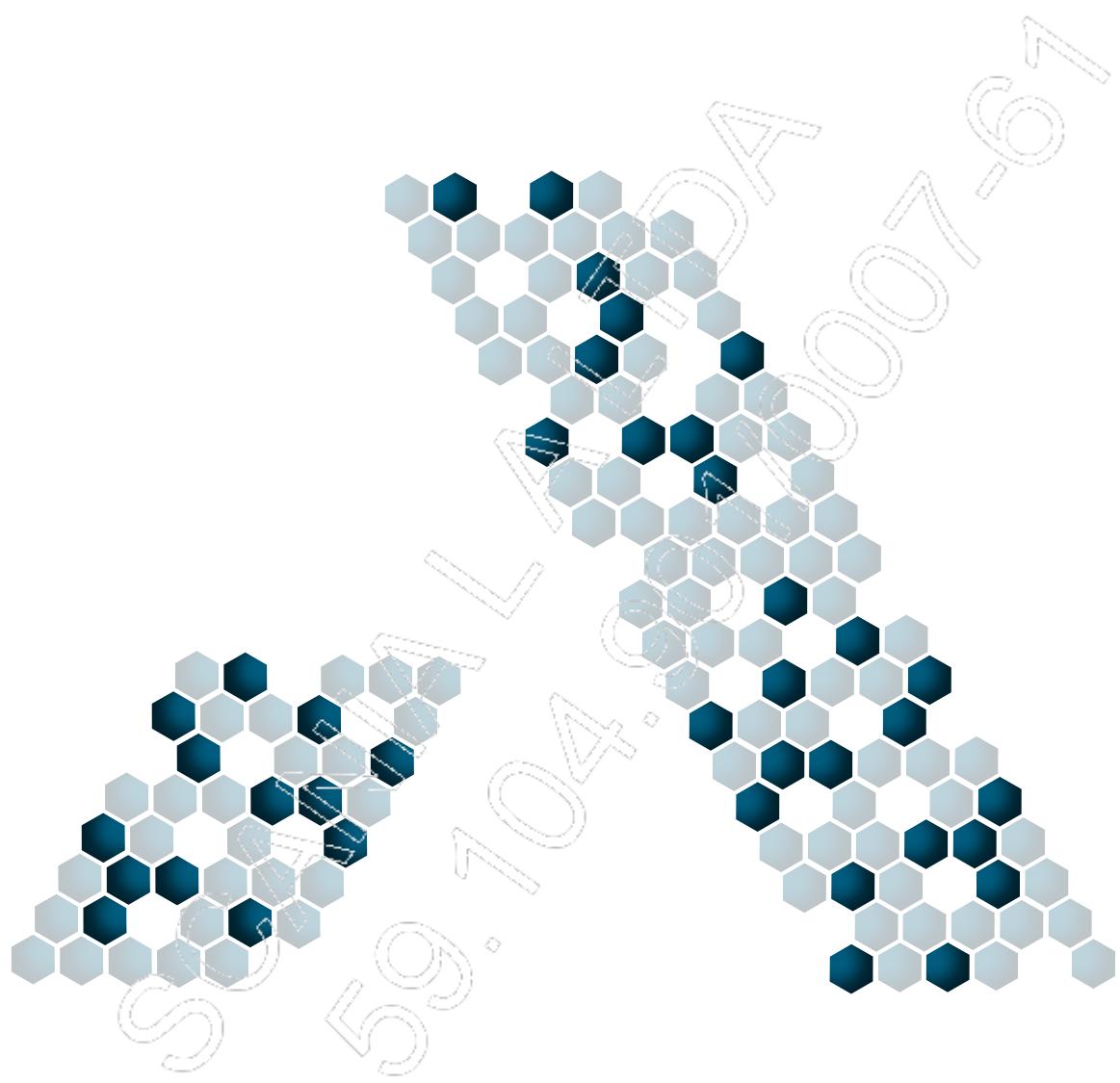


# Programando com C#



Editora  
**IMPACTA**







# Programando com C#





## Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Monte Everest Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

*"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."*

# Programando com C#

## Coordenação Geral

Henrique Thomaz Bruscaqin

Autoria

Autoria

# Revisão Ortográfica e Gramatical

Marcos César dos Santos Silva

## Diagramação

## Diagramação

Edição nº 1 | 1888\_0

Marco/ 2020

# Sumário

<b>Capítulo 1 - Sintaxe da Linguagem C# e o Visual Studio 2019 .....</b>	<b>11</b>
1.1.    Introdução.....	12
1.2.    A linguagem C#.....	12
1.3.    A Plataforma .NET.....	12
1.4.    O Framework .NET.....	13
1.5.    O Visual Studio.....	13
1.5.1.    Processo de instalação do Visual Studio .....	14
1.5.2.    Criação de projetos no Visual Studio.....	16
1.5.3.    Configurando o ambiente de desenvolvimento .....	19
1.6.    Iniciando a programação .....	20
1.6.1.    Elaboração do código .....	21
1.6.2.    Compilando e executando o projeto .....	22
1.7.    Instruções .....	23
1.7.1.    Identificadores.....	24
1.8.    Tipos de dados .....	25
1.9.    Variáveis.....	27
1.9.1.    Declaração de variáveis.....	27
1.10.    Operadores .....	28
1.10.1.    Operador de atribuição.....	28
1.10.2.    Operadores aritméticos .....	29
1.10.3.    Operadores aritméticos de atribuição reduzida.....	30
1.10.4.    Operadores incrementais e decrementais .....	31
1.10.5.    Operadores relacionais ou booleanos .....	33
1.10.6.    Operadores lógicos.....	33
1.10.7.    Operador ternário.....	35
1.10.8.    Operador de coalescência nula .....	36
1.10.9.    Precedência e associatividade .....	37
1.11.    Sequência de escape.....	37
1.12.    Principais formatações de dados.....	38
1.12.1.    A instrução string.Format() .....	38
1.12.2.    Formatação por interpolação .....	44
Pontos principais .....	45
<b>Teste seus conhecimentos.....</b>	<b>47</b>
<b>Mãos à obra!.....</b>	<b>51</b>
<b>Capítulo 2 - Estruturas de Controle da Linguagem C#.....</b>	<b>61</b>
2.1.    Introdução.....	62
2.2.    Estruturas de controle .....	62
2.2.1.    O comando if.....	62
2.2.2.    O comando if /else .....	63
2.2.3.    A instrução switch / case .....	66
2.2.4.    Expressões switch (switch expressions) .....	69
2.3.    Estruturas de repetição .....	71
2.3.1.    Estrutura while (teste no início).....	71
2.3.2.    Estrutura do / while (teste no final).....	73
2.3.3.    Estrutura for .....	75
2.3.4.    Estrutura foreach .....	79
Pontos principais .....	81
<b>Teste seus conhecimentos.....</b>	<b>83</b>
<b>Mãos à obra!.....</b>	<b>87</b>

# Programando com C#

<b>Capítulo 3 - Orientação a objetos .....</b>	<b>101</b>
3.1.    Introdução.....	102
3.2.    Classes .....	102
3.3.    Objetos .....	103
3.4.    Atributos .....	104
3.5.    Modificadores de acesso.....	107
3.6.    Namespaces .....	108
3.7.    Propriedades .....	113
3.7.1.    Encapsulamento .....	115
3.7.2.    Propriedades como membros aptos para expressões.....	117
3.7.3.    Propriedades implícitas.....	117
3.8.    A referência this .....	118
3.9.    Objetos implícitos e anônimos.....	118
3.10.    Enumeradores .....	120
3.11.    Estruturas.....	121
3.11.1.    Diferenças entre classes e estruturas .....	122
3.12.    O modificador static .....	123
3.13.    Métodos .....	125
3.13.1.    Sobrecarga de métodos .....	129
3.13.2.    Métodos estáticos.....	131
3.13.3.    Métodos de extensão.....	132
3.13.4.    Parâmetros de métodos .....	134
3.13.4.1.    Passagem por referência – ref .....	134
3.13.4.2.    Parâmetro de saída: out .....	136
3.13.4.3.    Parâmetros default e opcionais .....	137
3.13.4.4.    Parâmetros nomeados .....	139
3.13.4.5.    Method references .....	140
Pontos principais .....	141
<b>Teste seus conhecimentos.....</b>	<b>143</b>
<b>Mãos à obra!.....</b>	<b>147</b>

<b>Capítulo 4 - Construtores e destrutores .....</b>	<b>175</b>
4.1.    Introdução.....	176
4.2.    Construtor default .....	176
4.3.    Regras sobre construtores .....	178
4.4.    Construtores personalizados (com parâmetros) .....	178
4.5.    Sobrecarga de construtores .....	181
4.6.    Destrutores .....	184
Pontos principais .....	185
<b>Teste seus conhecimentos.....</b>	<b>187</b>
<b>Mãos à obra!.....</b>	<b>191</b>

# Sumário

---

<b>Capítulo 5 - Herança e polimorfismo .....</b>	<b>199</b>
5.1.    Introdução.....	200
5.2.    Herança de classes .....	200
5.3.    Acesso à superclasse com o operador base .....	202
5.4.    Polimorfismo .....	203
5.5.    Tipos de classes .....	206
5.5.1.    Classes e membros abstratos .....	206
5.6.    Chamada a métodos polimórficos.....	208
5.6.1.    Classe sealed.....	211
5.6.2.    Classe estática.....	211
Pontos principais .....	213
<b>Teste seus conhecimentos.....</b>	<b>215</b>
<b>Mãos à obra!.....</b>	<b>219</b>
<b>Capítulo 6 - Interfaces.....</b>	<b>231</b>
6.1.    Introdução.....	232
6.2.    Interfaces .....	232
6.3.    Valores default para membros de interfaces .....	237
Pontos principais .....	238
<b>Teste seus conhecimentos.....</b>	<b>239</b>
<b>Mãos à obra!.....</b>	<b>243</b>
<b>Capítulo 7 - Genéricos.....</b>	<b>251</b>
7.1.    Introdução.....	252
7.2.    Classes genéricas .....	252
7.3.    Métodos genéricos .....	254
7.4.    Genéricos e herança .....	255
7.4.1.    Sobrescrita de métodos genéricos .....	255
7.4.2.    Subclasse genérica .....	256
7.5.    Interfaces genéricas.....	256
7.6.    Restrições ao uso dos genéricos .....	257
7.6.1.    Tabela de restrições .....	258
7.7.    Covariância e contravariância.....	259
7.7.1.    Covariância.....	259
7.7.2.    Contravariância .....	259
Pontos principais .....	261
<b>Teste seus conhecimentos.....</b>	<b>263</b>
<b>Mãos à obra!.....</b>	<b>267</b>

# Programando com C#

<b>Capítulo 8 - Coleções.....</b>	<b>271</b>
8.1.    Introdução.....	272
8.2.    Arrays.....	272
8.2.1.   Construção e instanciação de arrays .....	272
8.2.2.   Passando um array como parâmetro .....	278
8.3.    Coleções.....	281
8.3.1.   Diferenças entre coleções e arrays.....	282
8.4.    A classe ArrayList .....	282
8.5.    A classe Stack.....	285
8.6.    A classe Queue .....	288
8.7.    A classe Hashtable.....	290
8.8.    A classe List<T> .....	292
8.8.1.   Inicializadores de List .....	293
8.9.    As classes HashSet<T> e SortedSet<T> .....	293
8.10.   A interface IEnumerable<T>.....	294
8.10.1.  A palavra reservada yield .....	296
Pontos principais .....	297
<b>Teste seus conhecimentos.....</b>	<b>299</b>
<b>Mãos à obra!.....</b>	<b>303</b>
<b>Capítulo 9 - Tratamento de exceções.....</b>	<b>321</b>
9.1.    Introdução.....	322
9.2.    Tipos de erros .....	322
9.2.1.  Erro de lógica .....	322
9.2.2.  Erro de compilação .....	323
9.2.3.  Erro de execução .....	323
9.3.    O bloco try ... catch .....	324
9.4.    A classe Exception e suas subclasses.....	327
9.4.1.  Propriedades da classe Exception .....	329
9.5.    O comando throw .....	329
9.6.    O bloco finally .....	330
9.7.    O bloco using .....	332
9.7.1.  A interface IDisposable .....	333
Pontos principais .....	334
<b>Teste seus conhecimentos.....</b>	<b>335</b>
<b>Mãos à obra!.....</b>	<b>339</b>
<b>Capítulo 10 - Delegates e expressões lambda.....</b>	<b>351</b>
10.1.   Introdução.....	352
10.2.   Declaração de delegates .....	352
10.2.1. Vinculando o delegate com o operador new.....	353
10.2.2. Vinculando o delegate com o operador +=.....	354
10.3.   Expressões lambda.....	355
10.4.   Delegates genéricos .....	357
10.5.   Delegates predefinidos no framework.....	358
Pontos principais .....	360
<b>Teste seus conhecimentos.....</b>	<b>361</b>
<b>Mãos à obra!.....</b>	<b>365</b>

# Sumário

---

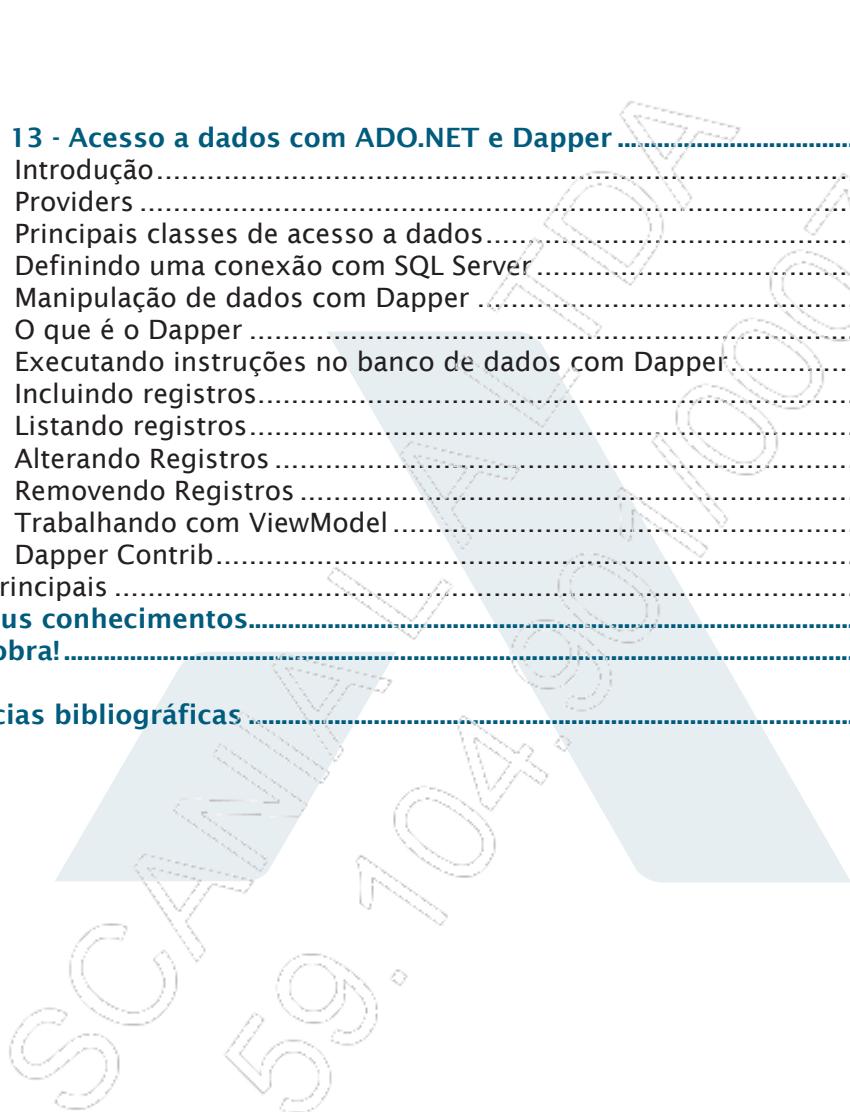
<b>Capítulo 11 - Programação assíncrona .....</b>	<b>373</b>
11.1. Introdução.....	374
11.2. Criação de tarefas – Classe Task .....	374
11.3. Aguardando pelo término de uma tarefa.....	376
11.4. Valor de retorno de uma Task.....	378
11.5. Operações assíncronas .....	379
11.5.1. async e await.....	379
Pontos principais .....	382
<b>Teste seus conhecimentos.....</b>	<b>383</b>
<b>Mãos à obra!.....</b>	<b>387</b>
<b>Capítulo 12 - Arquivos .....</b>	<b>393</b>
12.1. Introdução.....	394
12.2. A classe File.....	394
12.2.1. Lendo dados de arquivos .....	394
12.2.1.1. Método ReadAllText .....	394
12.2.1.2. Método ReadAllLines .....	395
12.2.1.3. Método ReadAllBytes .....	396
12.2.2. Escrevendo dados em arquivos .....	397
12.2.2.1. Método WriteAllText .....	397
12.2.2.2. Método WriteAllLines .....	398
12.2.2.3. Método WriteAllBytes .....	398
12.2.2.4. Método AppendAllText .....	399
12.2.2.5. AppendAllLines .....	399
12.3. Métodos adicionais .....	400
12.4. A classe Directory .....	400
12.4.1. Método CreateDirectory .....	401
12.4.2. Método Delete .....	402
12.4.3. Método Exists .....	402
12.4.4. Método GetDirectories .....	403
12.4.5. Método GetFiles .....	404
12.5. Streams .....	404
12.5.1. Manipulação de Streams .....	405
12.5.2. A classe StreamReader .....	405
12.5.3. A classe StreamWriter .....	407
12.5.4. A classe BinaryReader .....	408
12.5.5. A classe BinaryWriter .....	409
Pontos principais .....	410
<b>Teste seus conhecimentos.....</b>	<b>411</b>
<b>Mãos à obra!.....</b>	<b>415</b>



# Programando com C#

---

<b>Capítulo 13 - Acesso a dados com ADO.NET e Dapper .....</b>	<b>429</b>
13.1.    Introdução.....	430
13.2.    Providers .....	430
13.3.    Principais classes de acesso a dados.....	430
13.4.    Definindo uma conexão com SQL Server .....	431
13.5.    Manipulação de dados com Dapper .....	432
13.5.1.    O que é o Dapper .....	432
13.5.2.    Executando instruções no banco de dados com Dapper.....	433
13.5.3.    Incluindo registros.....	434
13.5.4.    Listando registros.....	435
13.5.5.    Alterando Registros .....	436
13.5.6.    Removendo Registros .....	437
13.6.    Trabalhando com ViewModel .....	437
13.7.    Dapper Contrib.....	440
Pontos principais .....	444
<b>Teste seus conhecimentos.....</b>	<b>445</b>
<b>Mãos à obra!.....</b>	<b>449</b>
 <b>Referências bibliográficas.....</b>	 <b>475</b>



# 1

# Sintaxe da Linguagem C# e o Visual Studio 2019

- A linguagem C#;
- A Plataforma .NET;
- O Framework .NET;
- O Visual Studio;
- Iniciando a programação;
- Instruções;
- Tipos de dados;
- Variáveis;
- Operadores;
- Sequência de escape;
- Principais formatações de dados.

Editora

**IMPACTA**

## 1.1. Introdução

A plataforma .NET é a principal plataforma de desenvolvimento da Microsoft. Ela disponibiliza algumas linguagens de programação, adequadas à experiência do desenvolvedor. A linguagem C# é uma delas, considerada a mais importante e a preferida entre os desenvolvedores.

Neste capítulo, abordaremos os fundamentos da linguagem C#, bem como os principais tipos de projetos que podemos criar no Visual Studio, especificamente na versão 2019. Veremos não só como criar, mas também como compilar e executar um programa em C#. Em capítulos posteriores, conheceremos mais detalhes a respeito dos conceitos da linguagem.

## 1.2. A linguagem C#

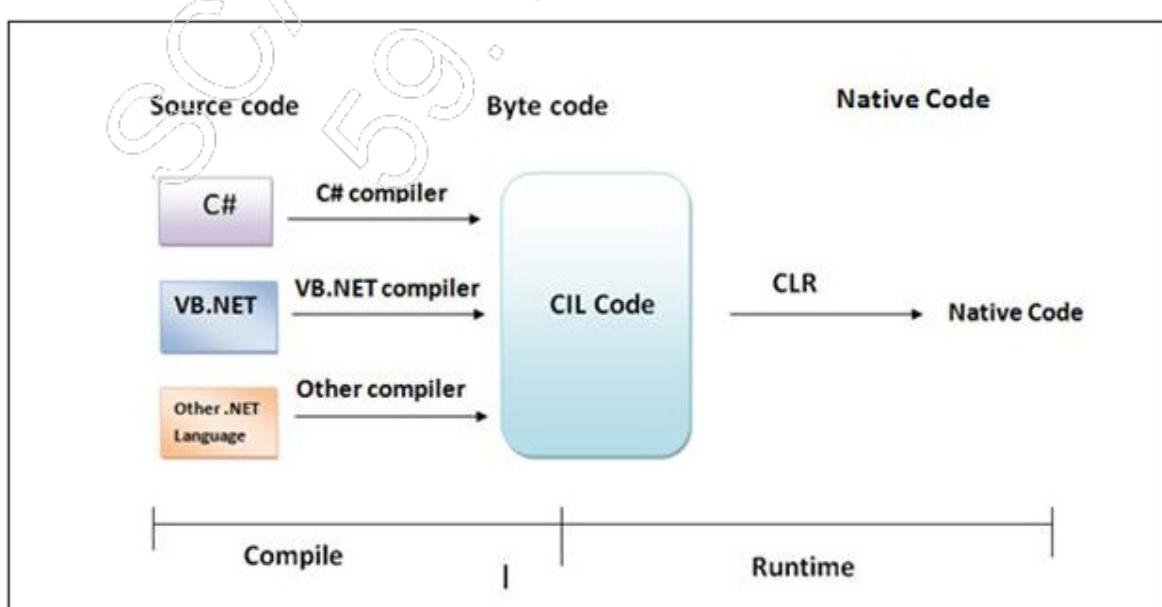
A linguagem C# é utilizada para a criação de vários tipos de aplicativos, é totalmente orientada a objetos e suas instruções requerem declaração explícita de variáveis. Esse recurso dá à linguagem a denominação de **linguagem fortemente tipada**.

## 1.3. A Plataforma .NET

A plataforma .NET é o ambiente de desenvolvimento e execução de diversas aplicações. Qualquer código gerado por essa plataforma poderá ser executado em qualquer dispositivo ou equipamento que possua o framework compatível.

As diferentes tecnologias que compõem a piataforma .NET são chamadas, em conjunto, de **.NET Framework**.

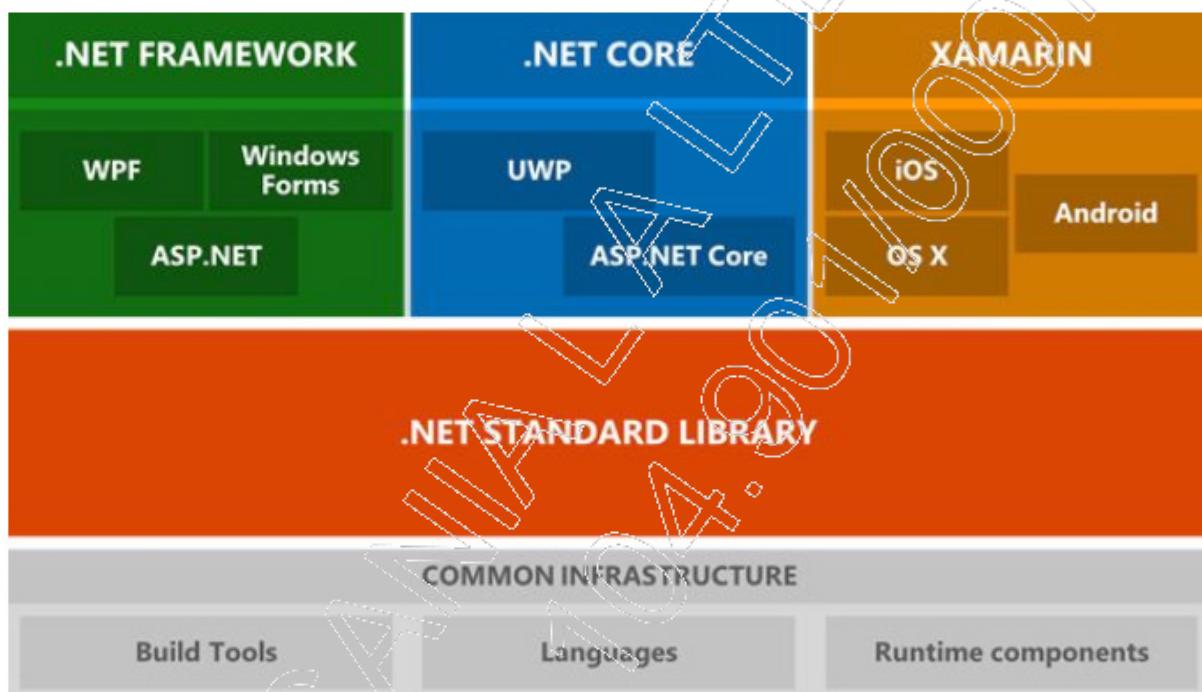
A plataforma inclui, basicamente, as linguagens C# e Visual Basic, e dois componentes fundamentais, o **Common Language Runtime (CLR)** e o **Framework Class Library (FCL)**.



Observando a imagem, podemos ver que o processo de compilação de uma aplicação em .NET leva em conta a linguagem usada, mas, após a compilação, é gerada uma instrução chamada bytecode, que na verdade independe da linguagem. Em linhas gerais, após a compilação, a instrução gerada se torna unificada para a plataforma .NET.

## 1.4. O Framework .NET

A parte mais importante da plataforma .NET é o Framework .NET. Ele consiste em um conjunto de bibliotecas desenvolvidas para finalidades específicas. Se o objetivo é desenvolver uma aplicação que realize acesso a dados, temos a vertente do framework chamada ADO.NET. Analogamente, cada modelo de projeto requer uma parte específica do framework. Veja a imagem a seguir:



Observe que, além das bibliotecas específicas, temos a **.NET Standard Library**. Essa biblioteca está presente em todas as aplicações e constitui a base para qualquer programa.

## 1.5. O Visual Studio

O **Integrated Development Environment (IDE)** do Visual Studio é um ambiente de desenvolvimento utilizado para escrever programas. Por se tratar de um software de programação potente e personalizável, o Visual Studio possui todas as ferramentas necessárias para desenvolver aplicações sólidas de forma rápida e eficiente. Muitas das características encontradas no Visual Studio se aplicam da mesma forma para as linguagens disponíveis.

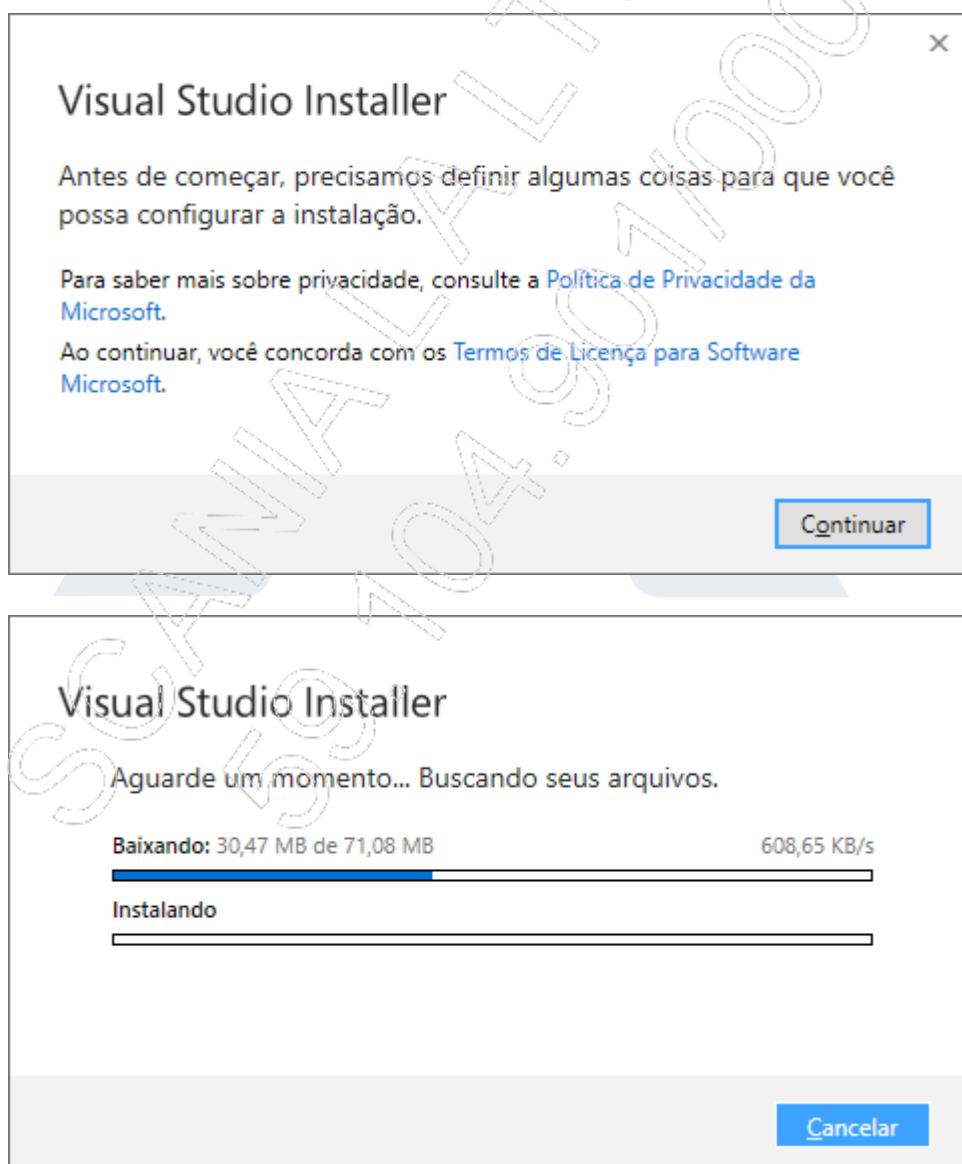
Neste curso, abordaremos o **Visual Studio 2019**. Mesmo que uma nova versão seja lançada, é possível que os projetos mais antigos ainda sejam abertos. O que deve ser compatível é a versão do Framework. Isso permite inclusive que seus projetos sejam abertos nas versões mais antigas do editor.

## 1.5.1. Processo de instalação do Visual Studio

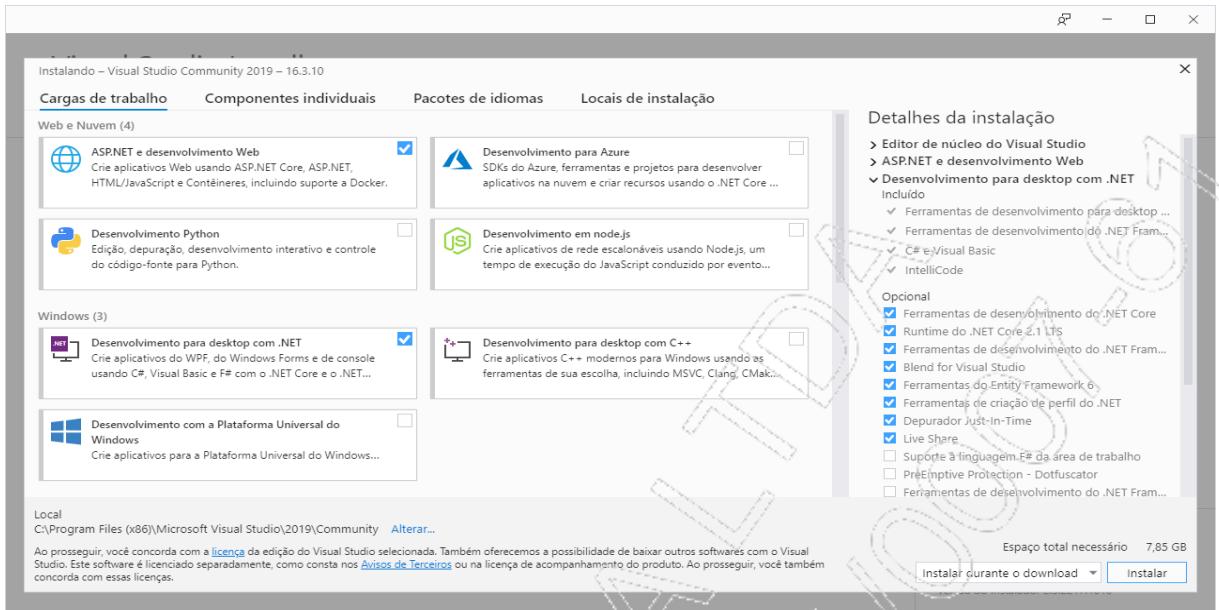
O Visual Studio pode ser obtido no link:

<https://visualstudio.microsoft.com/pt-br/>

Sugerimos a instalação do **Visual Studio Community**, por ser uma versão gratuita. Após realizar o download, execute o programa baixado. Ele vai acionar o **Installer**, programa necessário para o processo de instalação.



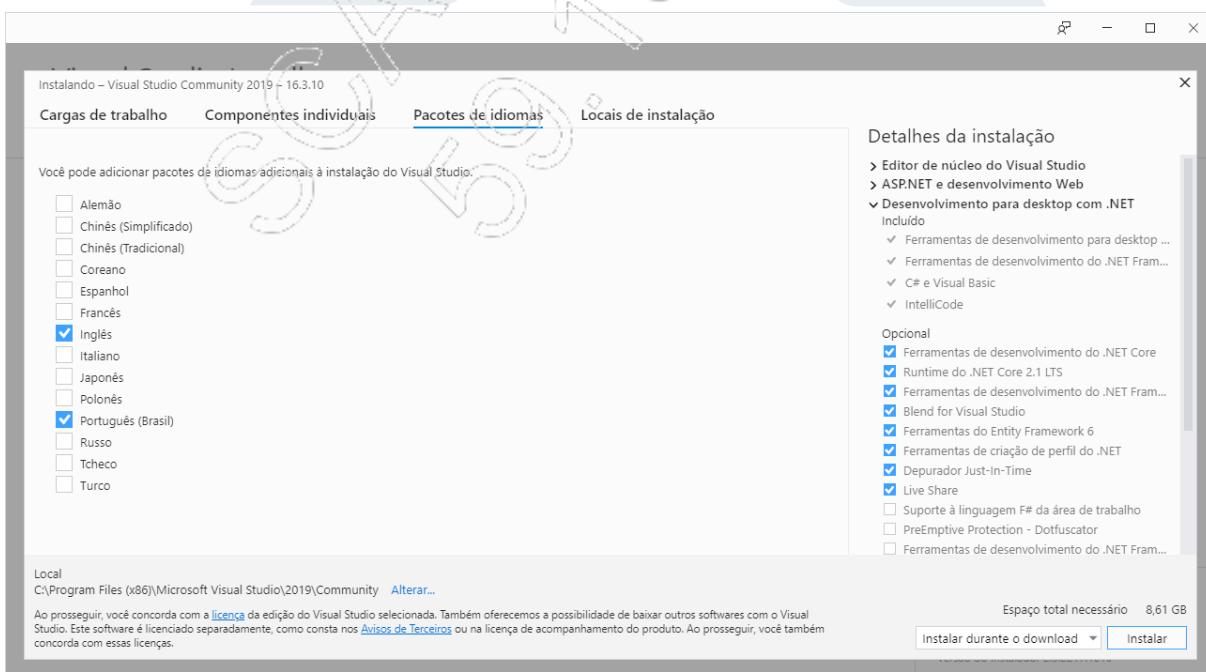
No início da instalação, o primeiro passo é selecionar os recursos desejados. Para desenvolvimento de aplicações desktop, aplicações Web e Webservices, recomendamos marcar os itens a seguir:



Claro que outros recursos podem ser adicionados, como os recursos para desenvolvimento em Python, Node.js, UWP, dentre outros. Fique atento ao espaço requerido para a instalação!

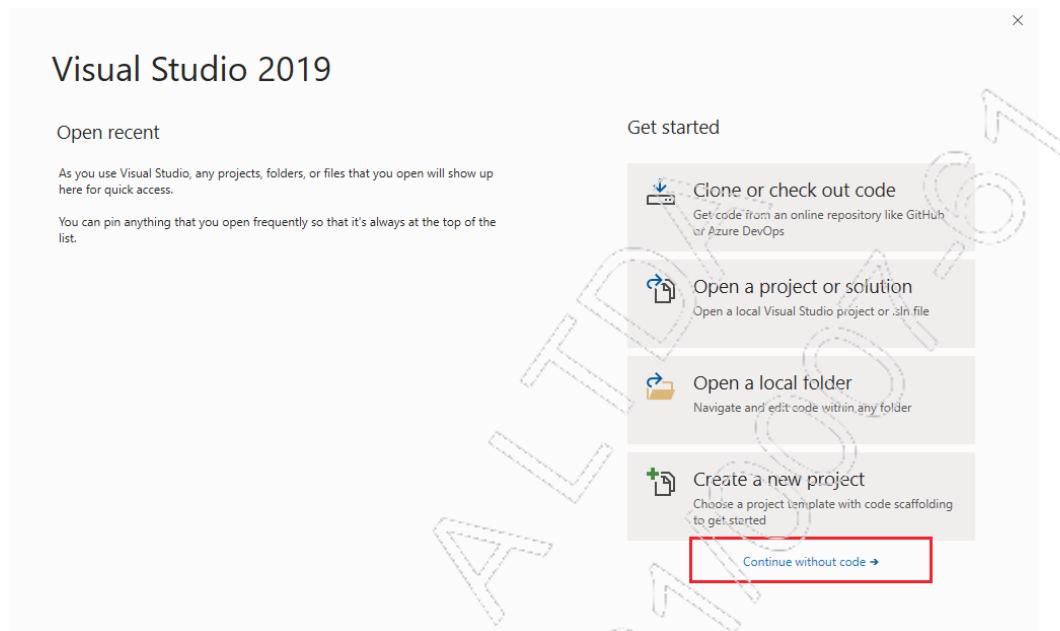
Se for necessário adicionar ou remover algum recurso, basta executar novamente o programa de instalação.

Um recurso importante no processo de instalação é a inclusão do pacote de idiomas. É possível que apenas a versão em português esteja disponível na instalação padrão.



## 1.5.2. Criação de projetos no Visual Studio

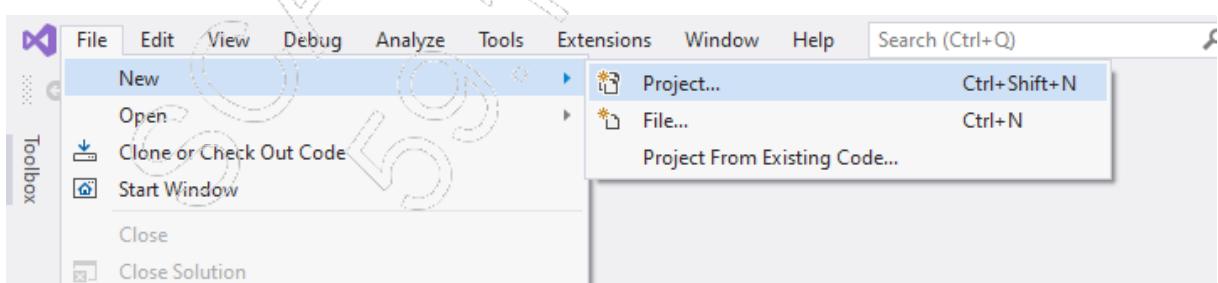
Após a instalação do Visual Studio, o próximo passo é começar a usá-lo. Ao executá-lo pela primeira vez, teremos a tela a seguir:



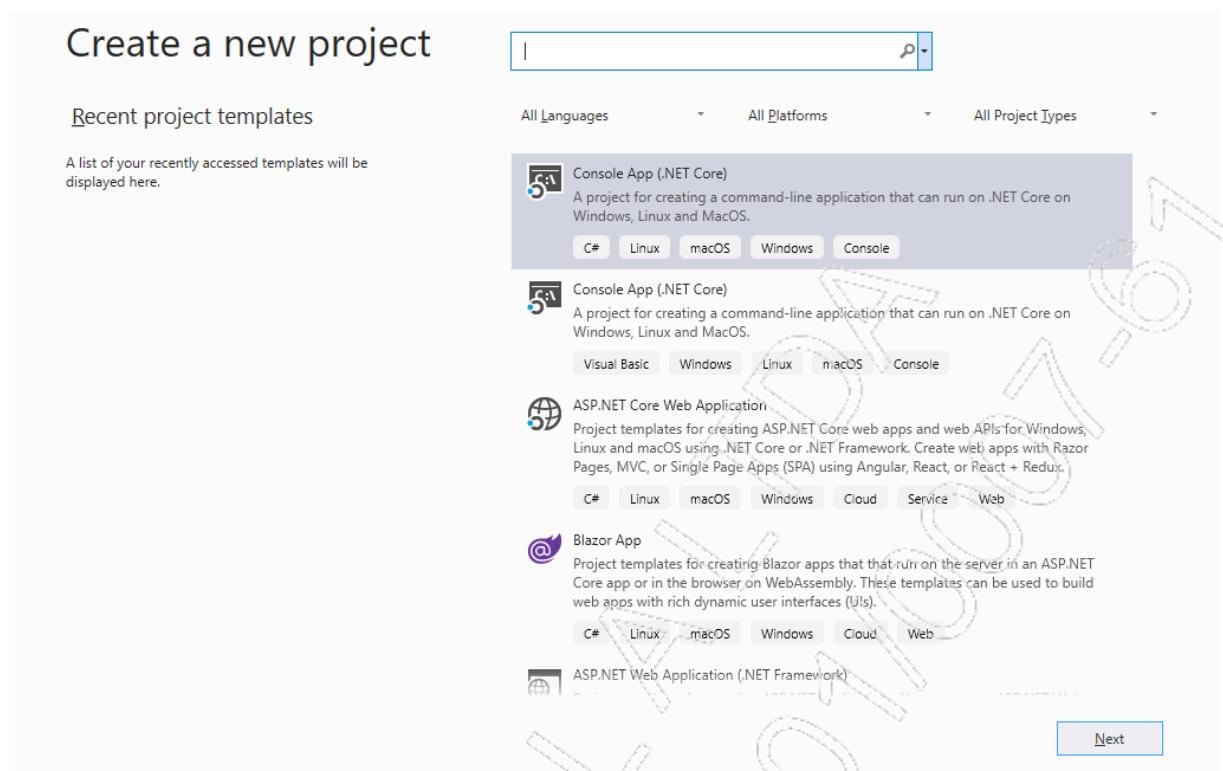
Basta selecionar o item **Continue Without Code**.

Observe que estamos usando a versão em inglês do Visual Studio. Na instalação, se a versão disponível estiver em português, selecione o menu **Ferramentas / Opções**. Selecione então **Configurações Regionais** e escolha o idioma desejado. Ao longo do curso, usaremos a versão em inglês.

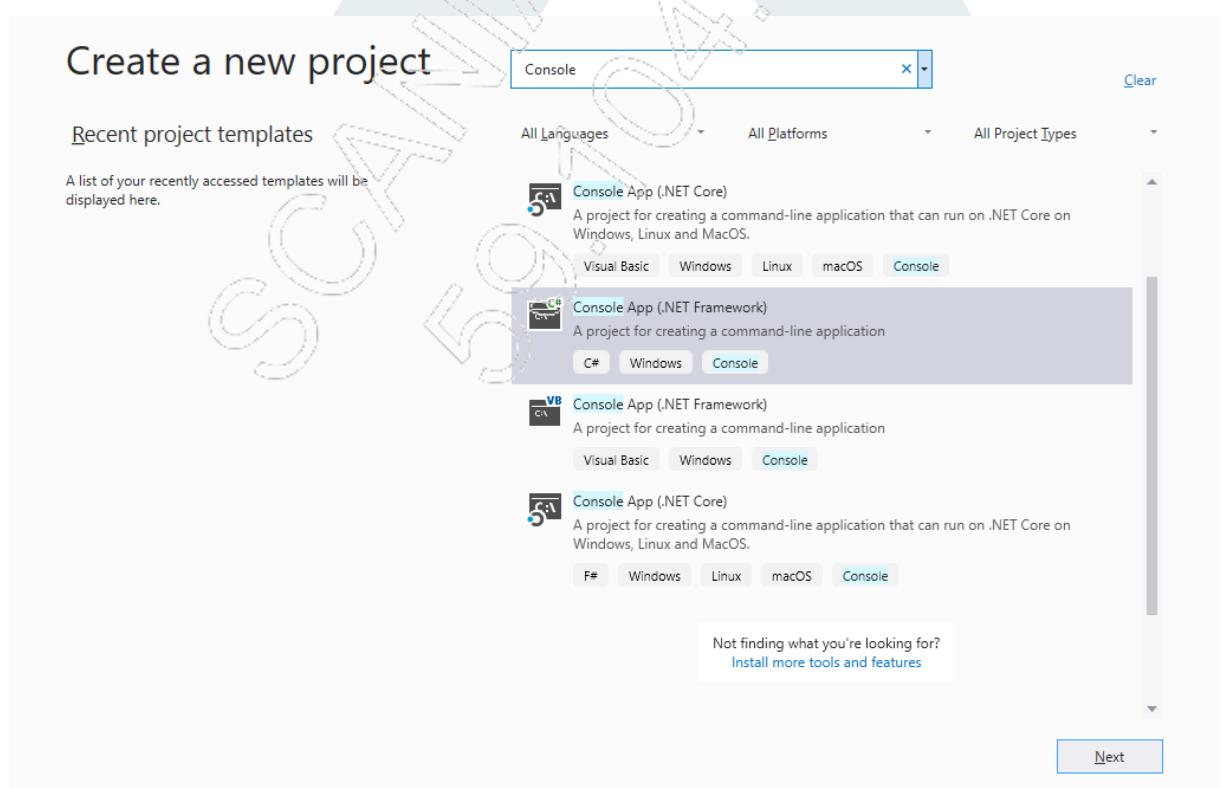
Prossiga com a execução. No menu **File**, selecione **New / Project...**:



Então, será aberta a caixa de diálogo **Create New Project**, onde é possível selecionar um projeto listado ou buscá-lo pelo nome:



Vamos selecionar a opção **Console App (.NET Standard)**. Se não aparecer, selecione-a na lista, como mostrado adiante:

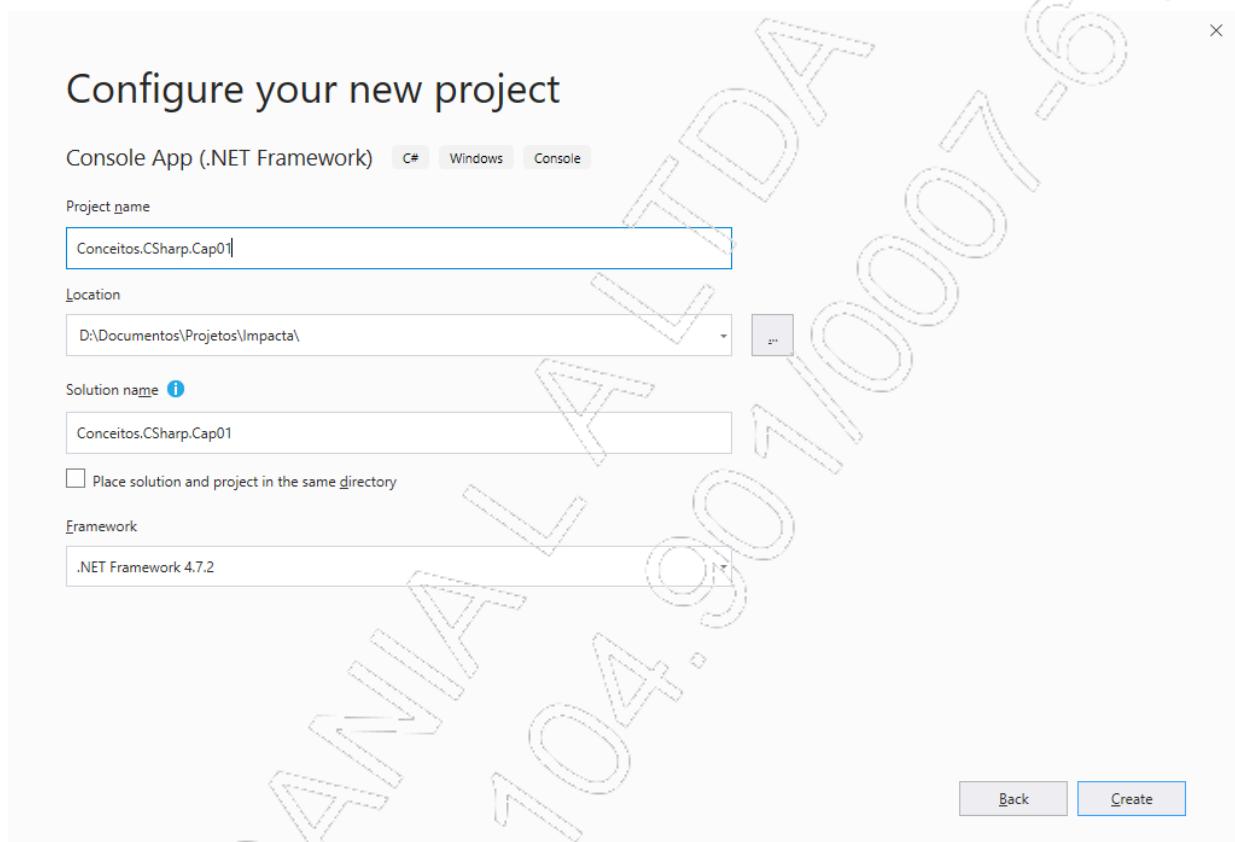


# Programando com C#

No próximo passo, escolhemos:

- O nome do projeto;
- A localização;
- O nome do Solution;
- A versão do Framework.

Escolha uma pasta da sua escolha, e mantenha as demais informações como mostrado a seguir:



Clique em **Create**.

O nome do projeto não pode conter nenhum dos caracteres a seguir:

#	Cerquilha	/	Barra
%	Percentual	\	Barra invertida
&	E comercial	:	Dois-pontos
*	Asterisco	"	Aspas
?	Ponto de interrogação	<	Menor que
	Barra vertical	>	Maior que
	Espaço inicial ou final	'	Apóstrofo

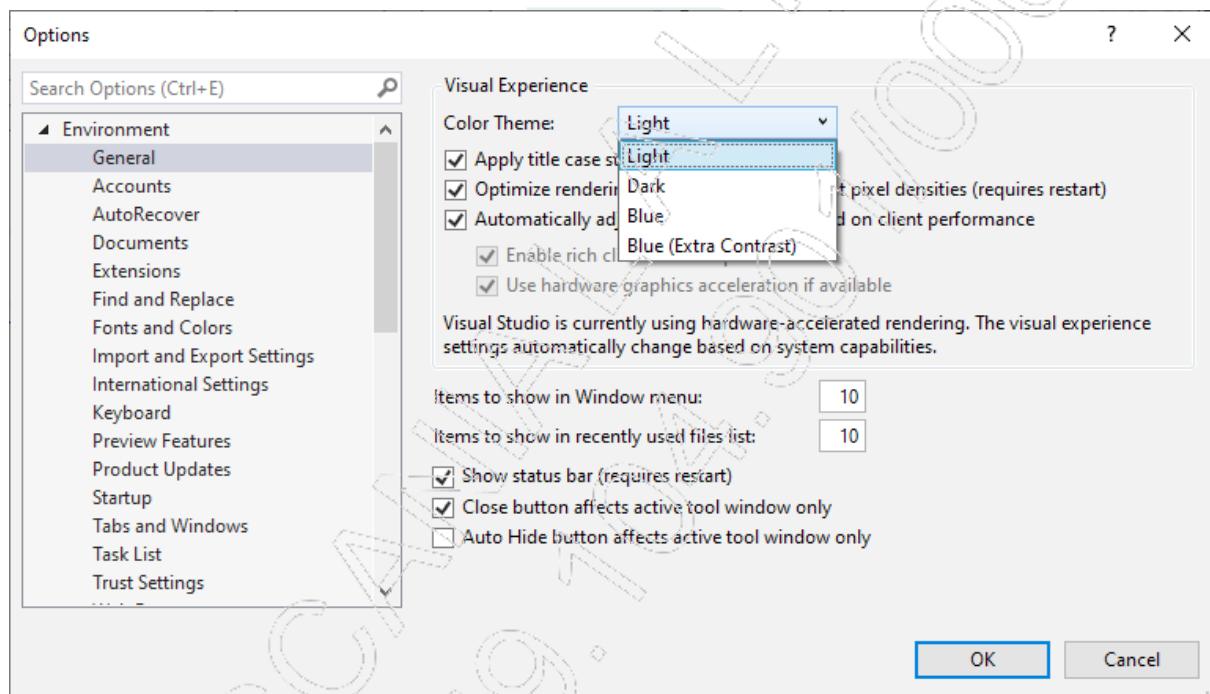
Criamos, neste exemplo, um projeto do tipo **Console App**, mas poderíamos criar quaisquer outros tipos.

Ao longo deste curso, apresentaremos os conceitos da Linguagem C# através de aplicações Console, por conta de sua simplicidade e praticidade. Nos laboratórios, apresentaremos outros projetos.

## 1.5.3. Configurando o ambiente de desenvolvimento

Já mencionamos a alteração do idioma. Outra configuração simples de ambiente é a troca do tema do Visual Studio. Para fazê-la, clique no menu **Tools** e, em seguida, em **Options**.

Na janela **Options**, clique, na guia **Environment**, em **General**, e escolha a opção de tema de cor mais confortável para o seu ambiente de desenvolvimento:

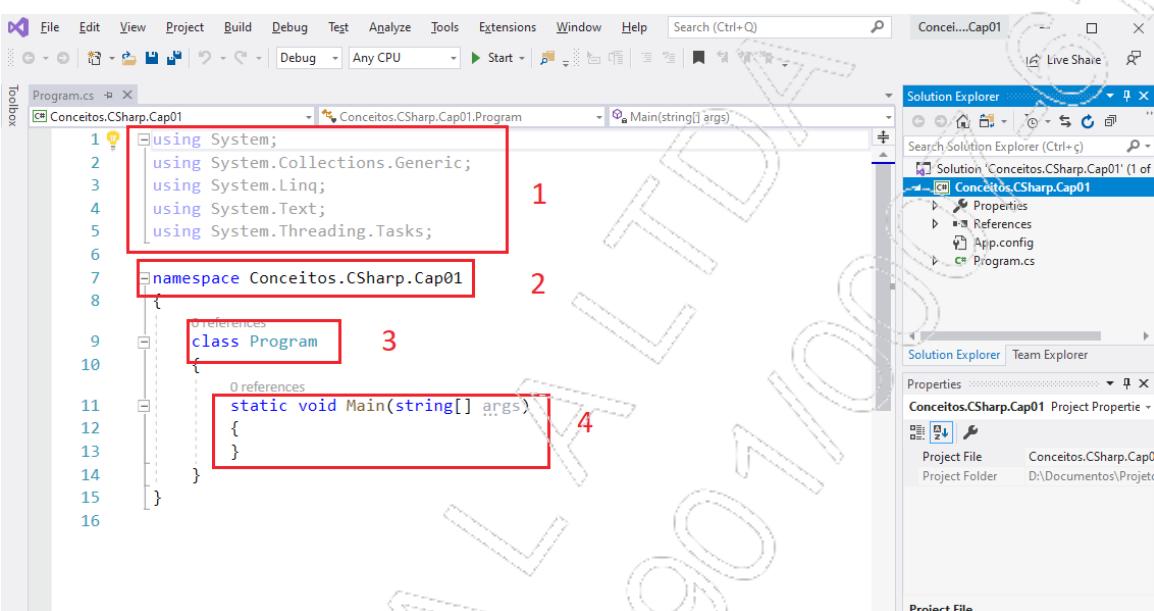


## 1.6. Iniciando a programação

Com o projeto criado, já estamos prontos para iniciarmos a programação em C#.

O objetivo, por enquanto, é apenas dar uma visão geral: conheceremos, entre outros, detalhes sobre classes, métodos e instruções no decorrer deste treinamento.

Considere o código gerado:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Conceitos.CSharp.Cap01
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13        }
14    }
15 }
16
```

Nesse código, temos:

- **1 – Diretivas ou Namespaces:** São os namespaces que o programa irá utilizar e que simplificam a digitação dos nomes das classes ou demais bibliotecas durante a escrita do código;
- **2 – Namespace da Aplicação:** Todo projeto tem um namespace e, normalmente, todos os arquivos do projeto estão contidos nesse mesmo namespace;
- **3 - Definição da classe:** A classe é um dos elementos presentes em uma aplicação. De modo geral, uma aplicação Console App possui uma classe chamada Program;
- **4 – Método Main():** Os métodos são necessários para executar tarefas nos programas. Nesta aplicação, o método Main() é chamado pelo CLR na execução do programa. Em outras palavras, este método é o primeiro a ser executado.

Cada parte do programa possui seu próprio bloco. Um bloco é formado por um par de chaves ({}). Assim, temos o bloco do namespace, o bloco da classe, o bloco do método, e assim por diante.

## 1.6.1. Elaboração do código

A primeira coisa que podemos destacar acerca dos códigos são os comentários. Eles são muito úteis aos desenvolvedores, pois registram o que um programa está fazendo.

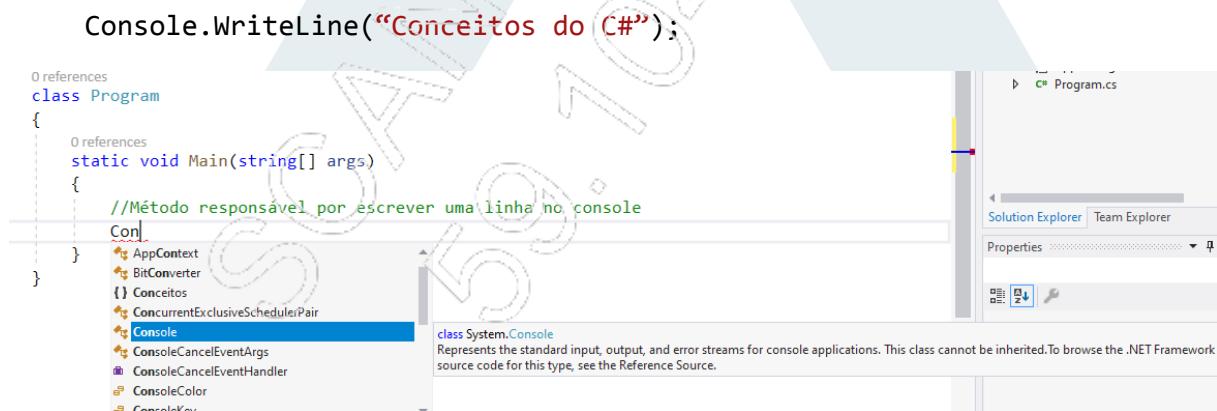
Os comentários são indicados por texto comum após barra dupla (//). O compilador ignora todo conteúdo situado entre as barras e o fim da linha. Comentários com mais de uma linha devem iniciar com uma barra seguida de asterisco /\*) e terminar com um asterisco seguido de barra (\*/). A seguir, temos um exemplo de comentário em código:

```
static void Main(string[] args)
{
    //Comentários de linha

    /*
     * Comentários de bloco,
     * ou várias linhas
     */
}
```

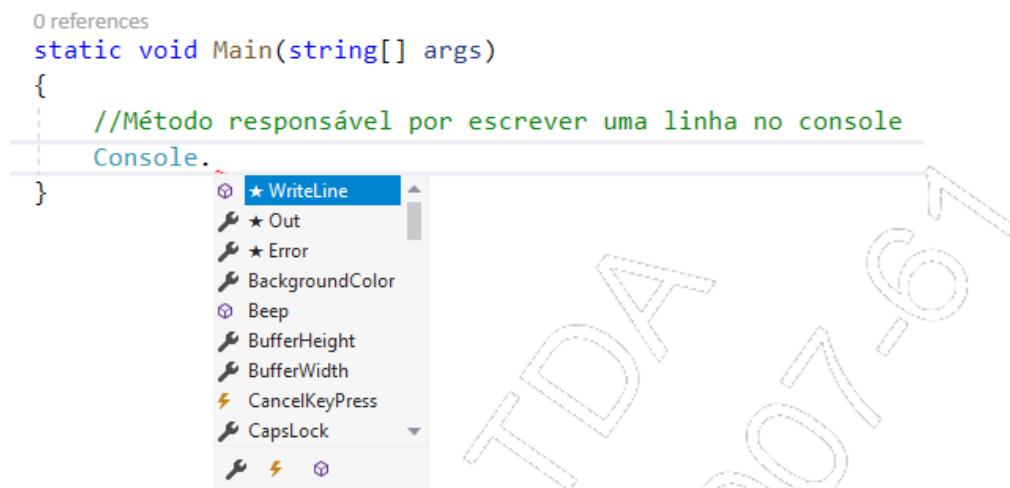
**É fundamental saber que a linguagem C# diferencia letras maiúsculas e minúsculas.**

Vamos escrever as instruções a seguir no método Main(), para testar sua execução. Podemos verificar que o Visual Studio fornece diversas simplificações, como recursos de **Intellisense** que nos auxilia na elaboração do código. Para ilustrar, vamos escrever a instrução:



Quando o Intellisense apresentar o código, nem é preciso continuar a digitação: podemos selecionar o código desejado e pressionar ENTER.

No código anterior, o objetivo é apresentar a classe **Console**. Ao terminarmos de digitar, colocamos o operador ponto (.) e teremos a lista de instruções disponíveis:



Complete o código:

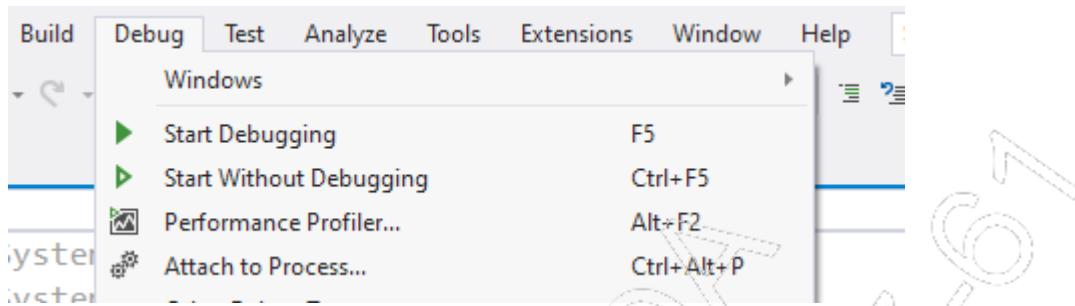
```
class Program
{
    static void Main(string[] args)
    {
        //Método responsável por escrever uma linha no console
        Console.WriteLine("Conceitos do C#");
        Console.ReadKey();
    }
}
```

## 1.6.2. Compilando e executando o projeto

Na **Solution Explorer**, clique com o botão direito do mouse sobre o nome do projeto e selecione **Build** ou **Rebuild**:

- **Build**: Compila somente os módulos que foram alterados;
- **Rebuild**: Recompila todos os módulos do projeto.

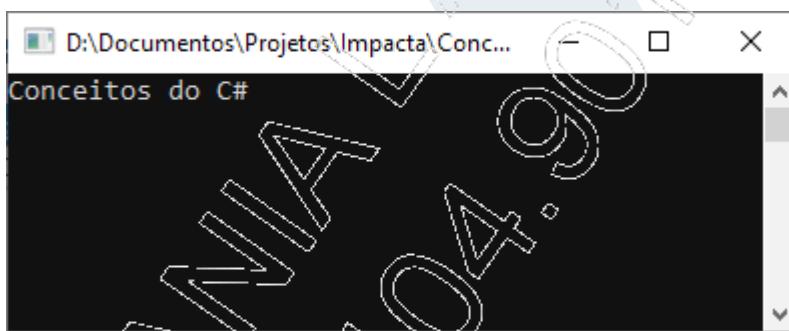
Para executar o código, podemos utilizar a opção **Start Debugging** ou **Start Without Debugging**, do menu **Debug**, ou, ainda, o botão **Start** (▶ Start) na Barra de Ferramentas.



Em que:

- **Start Debugging**: Permite executar o programa passo a passo;
- **Start Without Debugging**: Permite executar o programa integralmente, sem utilizar o debug.

O resultado é este:



## 1.7. Instruções

Consideremos os conceitos a seguir:

- Instrução é um comando responsável pela execução de uma ação;
- Método é uma sequência nomeada de instruções;
- Sintaxe é o conjunto de instruções que obedecem a uma série de regras específicas para seu formato e construção;
- Semântica é a especificação do que as instruções realizam.



Quanto maior o domínio sobre a sintaxe e a semântica de uma linguagem, maior a naturalidade com que se programa e, por consequência, maior a qualidade do trabalho desenvolvido.

## 1.7.1. Identificadores

Os elementos nos programas são representados por identificadores. Sua criação deve seguir duas regras simples: a primeira é a obrigatoriedade de se utilizar apenas letras, números inteiros de 0 a 9 e o caractere sublinhado (\_); e a segunda é que o primeiro elemento de um identificador deve ser sempre uma letra ou o sublinhado, mas nunca um número. Assim, `valortotal`, `_valor` e `preçoDeCusto` são identificadores válidos. Já `valortotal$`, `$valor` e `1preçoDeCusto` não são válidos.

Além dessas regras, devemos lembrar que os identificadores são “case sensitive”, ou seja, diferenciam maiúsculas de minúsculas.

Embora acentos sejam permitidos, evite utilizá-los.

- **Palavras reservadas ou palavras-chave**

Palavras-chave são identificadores exclusivos da linguagem que não podem ser usados para outras finalidades. São elas:

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort
using	using static	virtual	void
volatile	while		

- **Palavras-chave contextuais**

Uma palavra-chave contextual é usada para fornecer um significado específico no código, mas não é uma palavra reservada no C#. Algumas palavras-chave contextuais, como **partial** e **where**, têm significados especiais em dois ou mais contextos:

add	alias	ascending
async	await	by
descending	dynamic	equals
from	get	global
group	into	join
let	nameof	on
orderby	parcial (tipo)	partial (método)
remove	select	set
value	var	when
where	yield	

Essas palavras reservadas foram obtidas em <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/>

## 1.8. Tipos de dados

Em C#, qualquer variável usada dentro de um procedimento precisa ter seu tipo definido.

A seguir, temos uma tabela com diversos tipos predefinidos de C#, com tamanhos conhecidos:

- **Tipos inteiros**

Tipo .NET	Tipo C#	Faixa	Bytes
Sbyte	sbyte	-128 a 127	Com sinal 8-bit.
Byte	byte	0 a 255	Sem sinal 8-bit.
Char	char	U+0000 a U+ffff	Unicode 16-bit character.
Int16	short	-32,768 a 32,767	Com sinal 16-bit.
UInt16	ushort	0 a 65,535	Sem sinal 16-bit.

# Programando com C#

Tipo .NET	Tipo C#	Faixa	Bytes
Int32	int	-2,147,483,648 a 2,147,483,647	Com sinal 32-bit.
UInt32	uint	0 a 4,294,967,295	Sem sinal 32-bit.
Int64	long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	Com sinal 64-bit.
UInt64	ulong	0 a 18,446,744,073,709,551,615	Sem sinal 64-bit.

- Tipos com ponto flutuante

Tipo .NET	Tipo C#	Faixa	Precisão
Single	float	$\pm 1.5e-45$ a $\pm 3.4e38$	7 dígitos.
Double	double	$\pm 5.0e-324$ a $\pm 1.7e308$	15-16 dígitos.
Decimal	decimal	$\pm 1.0 \times 10e-28$ a $\pm 7.9 \times 10e28$	28-29 dígitos.

- Outros

Tipo .NET	Tipo C#	Descrição
String	string	Texto Unicode.
Boolean	bool	true ou false (George Boole - 1815 a 1864).
Object	object	Qualquer tipo de objeto do .NET.

- Tipos anuláveis

Permite a atribuição de valor **null** a variáveis cujo tipo não seja proveniente de classes, ou seja, do tipo **value**.

O caractere responsável por esse comportamento dos tipos anuláveis é interrogação (?).

```
//Valor NULL não é aceito pela variável  
byte idadeAluno1 = null;  
  
//Valor NULL é aceito pela variável  
byte? idadeAluno2 = null;
```

## 1.9. Variáveis

As variáveis podem ser definidas como locais destinados ao armazenamento temporário de informações de diferentes tipos, como números, palavras, propriedades, datas, entre outros, e devem receber um nome único, o qual será utilizado para fazer referência ao valor por elas armazenado.

A seguir, apresentamos algumas convenções importantes:

- Não utilizar sublinhados e não criar variáveis que diferem umas das outras apenas por meio de letras maiúsculas e minúsculas. Por exemplo, uma variável chamada **VariavelTeste** e outra denominada **variavelTeste**;
- Iniciar o nome da variável com letra minúscula. Nos casos de variáveis com mais de uma palavra, empregar maiúscula no início da segunda palavra e nas posteriores. Esse procedimento é conhecido como notação **camelCase**;
- Evitar a utilização de notação húngara, a qual consiste na especificação de prefixos nos nomes das variáveis para facilitar a identificação de seu tipo. Exemplo: **strNome**, **dblSalario**.

É importante salientar que, além de gerar confusões devido à semelhança dos nomes, a criação de variáveis que se diferem umas das outras apenas por meio de letras maiúsculas e minúsculas pode restringir a capacidade de reutilização de classes. Isso ocorre, especificamente, em aplicações desenvolvidas em linguagens como o Visual Basic, que não são case-sensitive, ou seja, não distinguem maiúsculas de minúsculas.

### 1.9.1. Declaração de variáveis

Para declarar uma variável, devemos colocar o seu tipo e, em seguida, o nome da variável. Opcionalmente, podemos, também, atribuir valor a ela.

```
int idade;
string nome = "Mirosmar";
float salario = 25.5f;
double preco = 10.32;
decimal valor = 21.3m;
bool casado = false;
char sexo = 'F';
```

- **Variáveis locais de tipo implícito**

Uma característica que o compilador do C# tem é a capacidade de verificar rapidamente qual o tipo de uma expressão que utilizamos para inicializar uma variável e informar se há correspondência entre o tipo da expressão e o da variável.

Usamos a palavra reservada **var** para permitir que o compilador do C# deduza qual o tipo de uma determinada variável a partir do tipo da expressão que foi utilizada para inicializá-la.

Veja o exemplo a seguir:

```
var x = 10;
string s = x.ToString();
[local variable] int x
```

```
var texto = "123";
s = tex
[local variable] string texto
```

## 1.10. Operadores

Os operadores indicam o tipo de operação que será executada, gerando novos valores a partir de um ou mais operandos.

### 1.10.1. Operador de atribuição

O sinal de igualdade (=) representa a atribuição de um valor a uma variável, em que a variável e o valor atribuído devem ser de tipos compatíveis.

É possível atribuirmos uma variável primitiva a outra variável primitiva. Veja como isso ocorre no exemplo a seguir:

```
int a = 3;
int b = a;
```

Em que:

- **a** recebe o valor 3;
- **b** recebe o valor da variável **a**, logo, **b** contém o valor 3.

Nesse momento, as duas variáveis (**a**, **b**) têm o mesmo valor, porém, se alterarmos o valor de uma delas, a outra não será alterada.

Veja o exemplo a seguir:

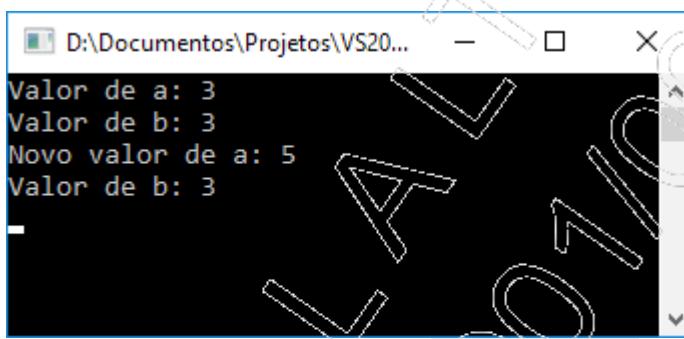
```
static void Main(string[] args)
{
    //O valor 3 é atribuído à variável a
    int a = 3;

    //Exibe o valor de a
    Console.WriteLine("Valor de a: " + a.ToString());

    //O valor de a é atribuído à variável b
    int b = a;
```

```
//Exibe o valor de b  
Console.WriteLine("Valor de b: " + b.ToString());  
  
//Variável a tem seu valor alterado para 5  
a = 5;  
  
//Exibe o novo valor de a  
Console.WriteLine("Novo valor de a: " + a.ToString());  
  
//Exibe o valor de b  
Console.WriteLine("Valor de b: " + b.ToString());  
Console.ReadKey();  
}
```

A compilação e a execução desse código resultarão no seguinte:



## 1.10.2. Operadores aritméticos

Os operadores aritméticos descritos na tabela adiante são utilizados nos cálculos matemáticos:

Operador aritmético	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão)

Devemos estar atentos para o fato de que nem todos os operadores podem ser utilizados com todos os tipos de dados. Embora existam valores que aceitam qualquer um dos operadores aritméticos (a saber: **char**, **decimal**, **double**, **float**, **int** e **long**), há, também, aqueles que não aceitam (como **bool** e **string**, nos quais só podemos aplicar o operador **+**).

Quanto ao resultado de uma operação aritmética, ele depende não só do operador, mas também do operando, ou seja, quando temos operandos de mesmo tipo, obtemos um resultado que também é desse tipo; quando temos operandos de tipos diferentes, o compilador do C#, detectando a incompatibilidade, gera um código que converte um dos valores antes que se execute a operação.

Embora o compilador do C# tenha a capacidade de fazer as devidas conversões antes de executar operações em que os operandos (ou valores) são de tipos diferentes, essa não é uma prática recomendada.

## 1.10.3. Operadores aritméticos de atribuição reduzida

Os operadores aritméticos de atribuição reduzida são utilizados para compor uma operação aritmética e uma atribuição.

A seguir, temos uma tabela em que constam os operadores aritméticos de atribuição reduzida:

Operador aritmético	Descrição
<code>+ =</code>	Soma igual
<code>- =</code>	Subtrai igual
<code>* =</code>	Multiplica igual
<code>/ =</code>	Divide igual
<code>% =</code>	Módulo igual

No exemplo a seguir, é subtraído o valor 3 da variável x:

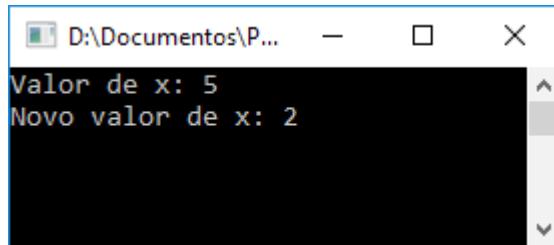
```
static void Main(string[] args)
{
    //Declaração da variável
    int x = 5;

    //Exibe o valor x
    Console.WriteLine("Valor de x: " + x.ToString());

    //Novo valor de x
    //Equivale a x = x - 3
    x -= 3;

    //Exibe o valor x
    Console.WriteLine("Novo valor de x: " + x.ToString());
    Console.ReadKey();
}
```

O resultado será o seguinte:



```
D:\Documentos\P...
Valor de x: 5
Novo valor de x: 2
```

## 1.10.4. Operadores incrementais e decrementais

Os operadores incrementais e decrementais têm a função de aumentar ou diminuir, respectivamente, o valor 1 em uma variável. Eles podem ser pré- ou pós-incrementais e pré- ou pós-decrementais. Veja:

- **Incremental (++)**
  - **Pré-incremental ou prefixo**

Significa que, se o sinal for colocado antes da variável, primeiramente, será somado o valor 1 para essa variável, continuando, em seguida, a resolução da expressão.

- **Pós-incremental ou sufixo**

Significa que, se o sinal for colocado após a variável, primeiro será resolvida toda a expressão (adição, subtração, multiplicação etc.) para, em seguida, ser adicionado o valor 1 à variável.

- **Decremental (--)**
  - **Pré-decremental ou prefixo**

Significa que, se o sinal for colocado antes da variável, primeiramente, será subtraído o valor 1 dessa variável e, em seguida, será dada continuidade à resolução da expressão.

- **Pós-decremental ou sufixo**

Significa que, se o sinal for colocado após a variável, primeiro será resolvida toda a expressão (adição, subtração, multiplicação etc.) para, em seguida, ser subtraído o valor 1 da variável.

# Programando com C#

O código a seguir ilustra a utilização dos operadores incrementais e decrementais:

```
static void Main(string[] args)
{
    int a;

    a = 5;
    Console.WriteLine("Exemplo Pré-Incremental");
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine("2 + ++a = " + (2 + ++a));
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine(new string('-', 50));

    a = 5;
    Console.WriteLine("Exemplo Pós-Incremental");
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine("2 + a++ = " + (2 + a++));
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine(new string('-', 50));

    a = 5;
    Console.WriteLine("Exemplo Pré-Decremental");
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine("2 + --a = " + (2 + --a));
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine(new string('-', 50));

    a = 5;
    Console.WriteLine("Exemplo Pós-Decremental");
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine("2 + a-- = " + (2 + a--));
    Console.WriteLine("valor de a: " + a);
    Console.WriteLine(new string('-', 50));

    Console.ReadKey();
}
```

O resultado será o seguinte:

```
D:\Documentos\Projetos\VS2017\ConsoleApp\ConsoleApp\bin\Debug - X

Exemplo Pré-Incremental
valor de a: 5
2 + ++a = 8
valor de a: 6
-----
Exemplo Pós-Incremental
valor de a: 5
2 + a++ = 7
valor de a: 6
-----
Exemplo Pré-Decremental
valor de a: 5
2 + --a = 6
valor de a: 4
-----
Exemplo Pós-Decremental
valor de a: 5
2 + a-- = 7
valor de a: 4
-----
```

## 1.10.5. Operadores relacionais ou booleanos

Operadores relacionais ou operadores booleanos são aqueles que realizam um cálculo cujo resultado é sempre **true** ou **false**. Dentre os diversos operadores booleanos existentes, o mais simples é o **NOT**, representado por um ponto de exclamação (!), o qual nega um valor booleano, apresentando sempre como resultado um valor oposto.

Na tabela logo adiante, apresentamos o funcionamento desses operadores utilizando, como exemplo, uma variável do tipo **int** que nomearemos como **quantia**:

Operador	Descrição	Exemplo	Se quantia for 50
<code>==</code>	Igual a	<code>quantia == 40</code>	Falso
<code>!=</code>	Diferente de	<code>quantia != 30</code>	Verdadeiro

Já os operadores relacionais comparam um valor a outro de mesmo tipo e retornam um valor booleano (**true** ou **false**) que indica se ele é maior ou menor. Veja quais são esses operadores na tabela a seguir, considerando o mesmo exemplo **quantia**:

Operador	Descrição	Exemplo	Se quantia for 50
<code>&lt;</code>	Menor que	<code>quantia &lt; 60</code>	Verdadeiro
<code>&lt;=</code>	Menor ou igual a	<code>quantia &lt;= 40</code>	Falso
<code>&gt;</code>	Maior que	<code>quantia &gt; 25</code>	Verdadeiro
<code>&gt;=</code>	Maior ou igual a	<code>quantia &gt;= 55</code>	Falso

## 1.10.6. Operadores lógicos

Os operadores lógicos trabalham com operandos booleanos e seu resultado também será booleano (**true** ou **false**). Sua função é combinar duas expressões ou valores booleanos em um só. Utilizados somente em expressões lógicas, eles estão reunidos na seguinte tabela:

Operador lógico	Descrição
<code>&amp;&amp;</code>	AND
<code>  </code>	OR
<code>!</code>	NOT

Embora sejam parecidos, a diferença entre os operadores lógicos e os operadores relacionais e de igualdade é que, não só o valor das expressões em que eles aparecem é verdadeiro ou falso, mas também os valores em que operam.

# Programando com C#

Em um teste lógico utilizando o operador `&&` (AND), o resultado somente será verdadeiro (`true`) se ambas as expressões lógicas forem avaliadas como verdadeiras. Porém, se o operador utilizado for `||` (OR), basta que uma das expressões lógicas seja verdadeira para que o resultado também seja verdadeiro. Já o operador lógico `!` (NOT) é utilizado para gerar uma negação, invertendo a lógica de uma expressão.

O código a seguir ilustra a utilização dos operadores lógicos:

```
static void Main(string[] args)
{
    int a = 30, b = 40, c = 50, d = 60;

    Console.WriteLine("a: " + a);
    Console.WriteLine("b: " + b);
    Console.WriteLine("c: " + c);
    Console.WriteLine("d: " + d);
    Console.WriteLine(new string('-', 20));
    Console.WriteLine("a < b || c == 50 --> " + (a < b || c == 50));
    Console.WriteLine("a < b || c == d --> " + (a < b || c == d));
    Console.WriteLine("a > b || c == 50 --> " + (a > b || c == 50));
    Console.WriteLine("a > b || c == d --> " + (a > b || c == d));
    Console.WriteLine(new string('-', 20));
    Console.WriteLine("a < b && c == 50 --> " + (a < b && c == 50));
    Console.WriteLine("a < b && c == d --> " + (a < b && c == d));
    Console.WriteLine("a > b && c == 50 --> " + (a > b && c == 50));
    Console.WriteLine("a > b && c == d --> " + (a > b && c == d));
    Console.WriteLine(new string('-', 20));

    Console.ReadKey();
}
```

O resultado é o seguinte:

```
D:\Documentos\Projetos\VS2017\Console...
a: 30
b: 40
c: 50
d: 60
a < b || c == 50 --> True
a < b || c == d --> True
a > b || c == 50 --> True
a > b || c == d --> False
-----
a < b && c == 50 --> True
a < b && c == d --> False
a > b && c == 50 --> False
a > b && c == d --> False
```

Quando trabalhamos com os operadores `||` e `&&`, pode ser que eles não façam a avaliação dos dois operandos para que, então, determinem o resultado de uma expressão lógica. Isso ocorre devido a um recurso conhecido como **curto-circuito** que, em alguns casos, ao avaliar o primeiro operando da expressão, já pode determinar seu resultado. Esse resultado baseado em apenas um dos operandos é possível quando:

- O operador que está localizado à esquerda de `&&` for `false`, o que torna desnecessária a avaliação do segundo, uma vez que o resultado será, obrigatoriamente, `false`;
- O operador que está localizado à esquerda de `||` for `true`, o que torna desnecessária a avaliação do segundo, uma vez que o resultado será, obrigatoriamente, `true`.

Ao elaborarmos as expressões de modo que a adoção dessa prática se torne possível, ou seja, posicionando expressões booleanas simples à esquerda, o desempenho do código pode ser otimizado, já que muitas expressões complexas, se posicionadas à direita, não precisarão passar por avaliação alguma.

## 1.10.7. Operador ternário

O operador ternário, ou operador condicional, é composto por três operandos separados pelos sinais `(?)` e `(:)` e tem o objetivo de atribuir um valor a uma variável de acordo com o resultado do teste lógico. Sua sintaxe é a seguinte:

```
teste lógico ? valor se verdadeiro : valor se falso;
```

Em que:

- **teste lógico** é qualquer expressão ou valor que pode ser avaliado como verdadeiro ou falso;
- **valor se verdadeiro** é o valor atribuído se o teste lógico for avaliado como verdadeiro;
- **valor se falso** é o valor atribuído se o teste lógico for avaliado como falso.

O código a seguir ilustra a utilização do operador ternário:

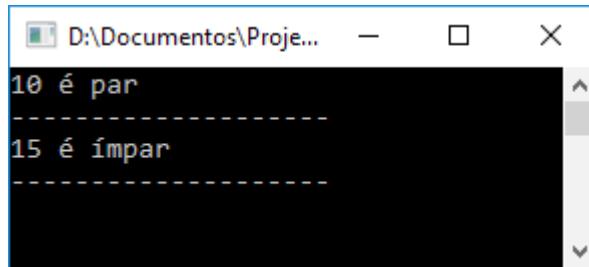
```
static void Main(string[] args)
{
    int numero;

    numero = 10;
    Console.WriteLine(numero + " é " + (numero % 2 == 0 ? "par"
    : "ímpar"));
    Console.WriteLine(new string('-', 20));

    numero = 15;
    Console.WriteLine(numero + " é " + (numero % 2 == 0 ? "par"
    : "ímpar"));
    Console.WriteLine(new string('-', 20));

    Console.ReadKey();
}
```

O resultado será o seguinte:



```
D:\Documentos\Proje... - X
10 é par
-----
15 é ímpar
```

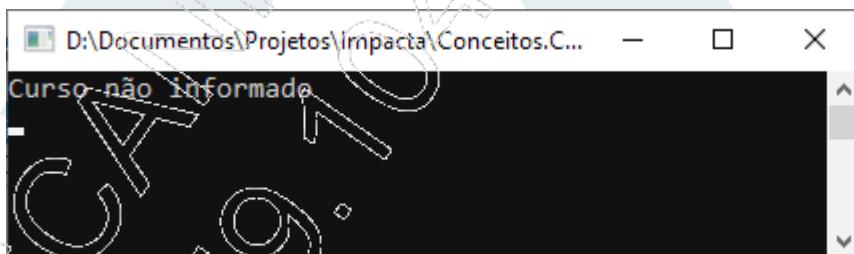
## 1.10.8. Operador de coalescência nula

O operador de coalescência nula (??) retornará o valor do operando esquerdo se não for **null**. Caso contrário, ele avaliará o operando direito e retornará seu resultado. Exemplo:

```
static void Main(string[] args)
{
    string curso = null;
    string texto = curso ?? "Curso não informado";

    Console.WriteLine(texto);
    Console.ReadKey();
}
```

No código anterior, a variável **texto** recebe o valor da variável **curso** somente se esta for diferente de **null**. Veja o resultado:



```
D:\Documentos\Projetos\Impacta\Conceitos.C... - X
Curso não informado
```

O exemplo a seguir utiliza o operador de atribuição de união nula (??=). Ele atribuirá o valor de seu operando à direita para o operando à esquerda somente se o operando esquerdo for avaliado como **null**. Este operador está disponível no C# 8.0. Veja o exemplo:

```
static void Main(string[] args)
{
    List<string> cursos = null;
    (cursos ??= new List<string>()).Add("C#");
    Console.WriteLine(string.Join(" ", cursos));

    Console.ReadKey();
}
```

Neste exemplo, a variável **cursos** recebe uma nova lista de string (**new List<string>**) se ela for nula.

## 1.10.9. Precedência e associatividade

A precedência, ou prioridade, determina em que ordem devem ser avaliados os operadores de uma expressão. Os operadores multiplicativos (\*, /, %), por exemplo, tem precedência sobre os aditivos (+ e -), portanto, em uma expressão em que estejam presentes os dois tipos de operadores, será calculado primeiro o multiplicativo.

O recurso que temos disponível para determinar uma precedência diferente da preexistente são os parênteses ( ).

## 1.11. Sequência de escape

A utilização da barra invertida (\) seguida de um ou mais caracteres específicos funciona como um comando nas strings, permitindo que determinados caracteres possam ser utilizados. Ela pode ser escrita entre aspas ou apóstrofos.

Veja a tabela a seguir:

Sequência	Descrição
\b	Backspace
\n	Próxima linha (ENTER)
\r	Início da linha
\t	Tabulação horizontal
\v	Tabulação vertical
\"	Caractere aspas
\'	Caractere apóstrofo
\\	Caractere barra invertida
\0	Caractere nulo

## 1.12. Principais formatações de dados

Vejamos, nos subtópicos a seguir, as principais formatações de dados.

### 1.12.1. A instrução `string.Format()`

Para apresentarmos dados formatados no .NET, usamos o método `Format` da classe `String`.

A sintaxe básica do método é a seguinte:

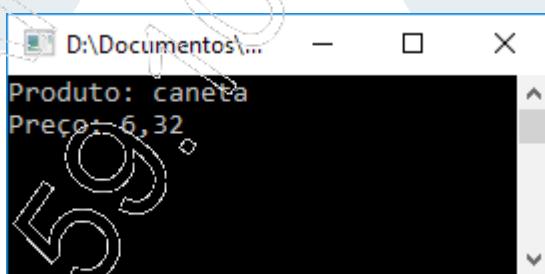
```
String.Format(String, Object)
```

Em que `String` é a máscara do format, e `Object` é o valor que será formatado.

Veja o exemplo:

```
static void Main(string[] args)
{
    string produto = "caneta";
    decimal preco = 6.32m;
    Console.WriteLine(string.Format("Produto: {0}\nPreço: {1}",
    produto, preco));
    Console.ReadKey();
}
```

O resultado é o seguinte:



O valor entre chaves representa a posição do argumento na lista de argumentos. No caso anterior, `produto` é o argumento zero, e `preco` o argumento um.

```
exemploLabel.Text = string.Format("Produto: {0}\nPreço: {1}",
    produto, preco);
    ↑           ↑
    |           |
    |           | argumento 1 da lista
    |           | argumento 0 da lista
```

Para valores inteiros ou não inteiros, temos alguns caracteres que aplicam formatos aos seus argumentos:

Caractere	Formato aplicado
c	Monetário
d	Decimal
e	Notação científica
f	Fixo - ponto flutuante
g	Geral
n	Padrão (ponto flutuante, separador de milhares, sem símbolo monetário)
p	Percentual
r	Arredondado
x	Hexadecimal

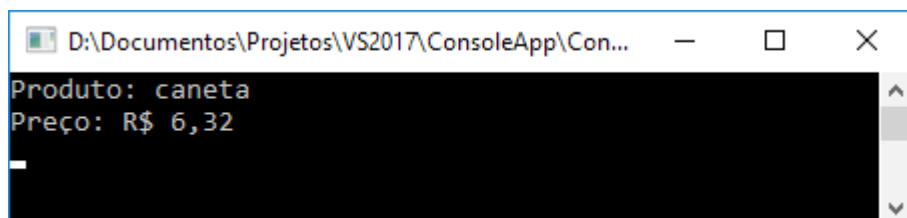
Veja o exemplo:

```
static void Main(string[] args)
{
    string produto = "caneta";
    decimal preco = 6.32m;

    //Formato monetário com duas casas pós vírgula
    //Respeita as configurações regionais do Windows
    Console.WriteLine(string.Format("Produto: {0}\nPreço:
{1:c2}", produto, preco));

    Console.ReadKey();
}
```

Agora, veja o resultado:



# Programando com C#

Podemos obter o mesmo resultado passando a formatação como parâmetro do método `ToString()`.

```
static void Main(string[] args)
{
    string produto = "caneta";
    decimal preco = 6.32m;

    //Formato monetário com duas casas pós vírgula
    //Respeita as configurações regionais do Windows
    Console.WriteLine(string.Format("Produto: {0}\nPreço: {1}",
                                    produto, preco.ToString("C2")));

    Console.ReadKey();
}
```

Temos, ainda, a opção de criar formatos personalizados, usando os caracteres a seguir:

Caractere	Formato aplicado
0	Dígito obrigatório
#	Dígito opcional
.	Separador decimal
,	Separador de milhares
%	Percentual

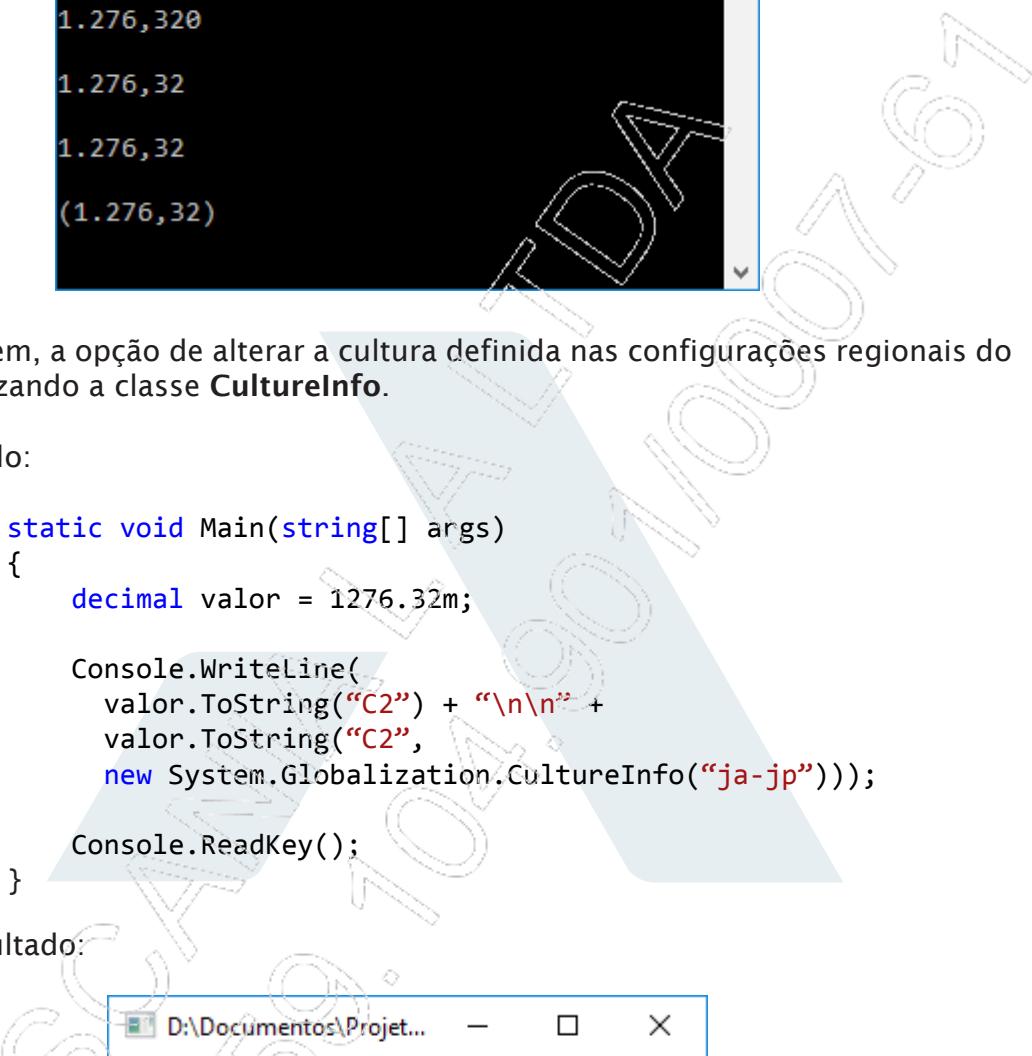
Veja um exemplo:

```
static void Main(string[] args)
{
    decimal valor = 1276.32m;

    Console.WriteLine(
        string.Format("{0:0.000}", valor) + "\n\n" +
        string.Format("{0:#.###}", valor) + "\n\n" +
        string.Format("{0:0,000.000}", valor) + "\n\n" +
        string.Format("{0:#,###.###}", valor) + "\n\n" +
        string.Format("{0:#,##0.00;(#,##0.00);0.00}", valor) +
        "\n\n" +
        string.Format("{0:#,##0.00;(#,##0.00);0.00}", -valor));

    Console.ReadKey();
}
```

Agora, confira o resultado:



```
D:\Documentos\Projetos\VS201...
1276,320
1276,32
1.276,320
1.276,32
1.276,32
(1.276,32)
```

Temos, também, a opção de alterar a cultura definida nas configurações regionais do Windows utilizando a classe **CultureInfo**.

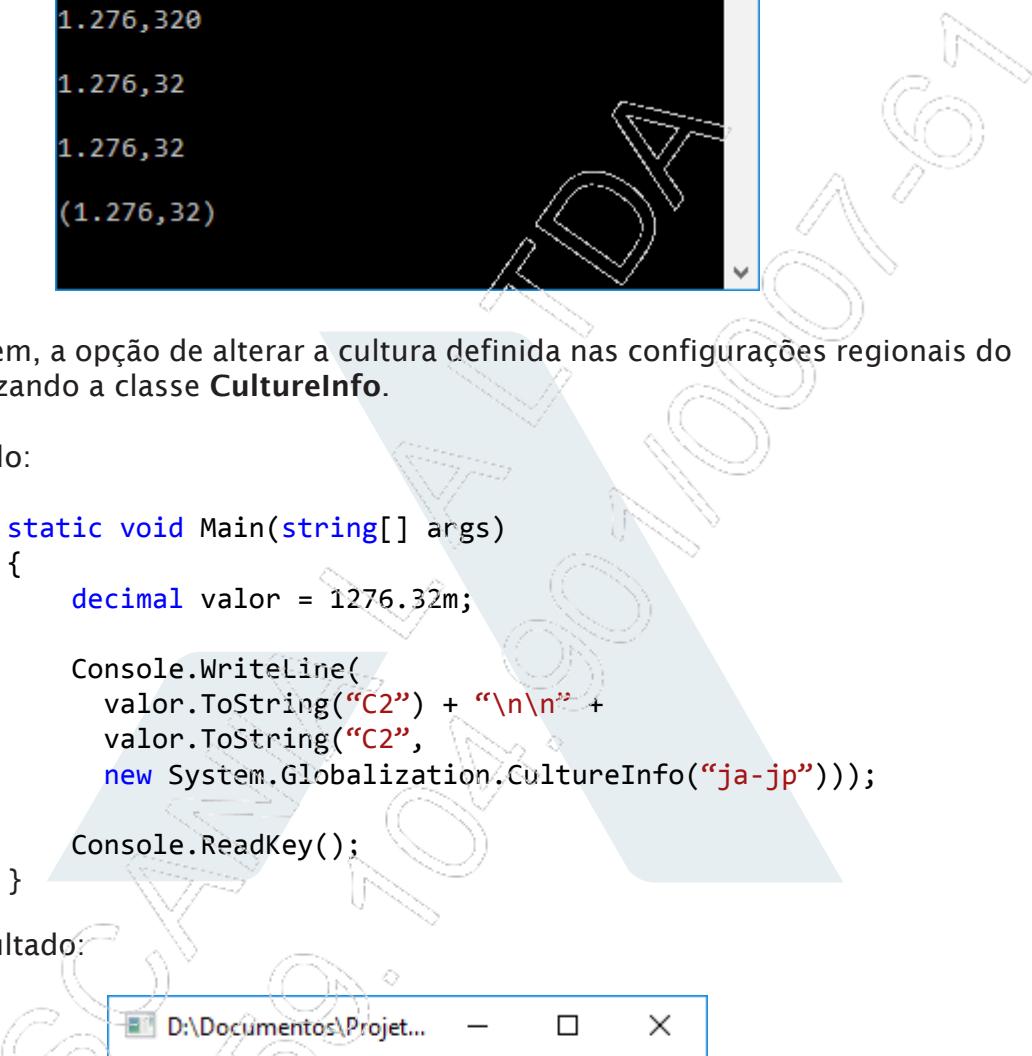
Veja o exemplo:

```
static void Main(string[] args)
{
    decimal valor = 1276.32m;

    Console.WriteLine(
        valor.ToString("C2") + "\n\n" +
        valor.ToString("C2",
            new System.Globalization.CultureInfo("ja-jp")));
}

Console.ReadKey();
```

Confira o resultado:



```
D:\Documentos\Projet...
R$ 1.276,32
¥1,276.32
```

# Programando com C#

Para formatos baseados em datas, temos os caracteres a seguir:

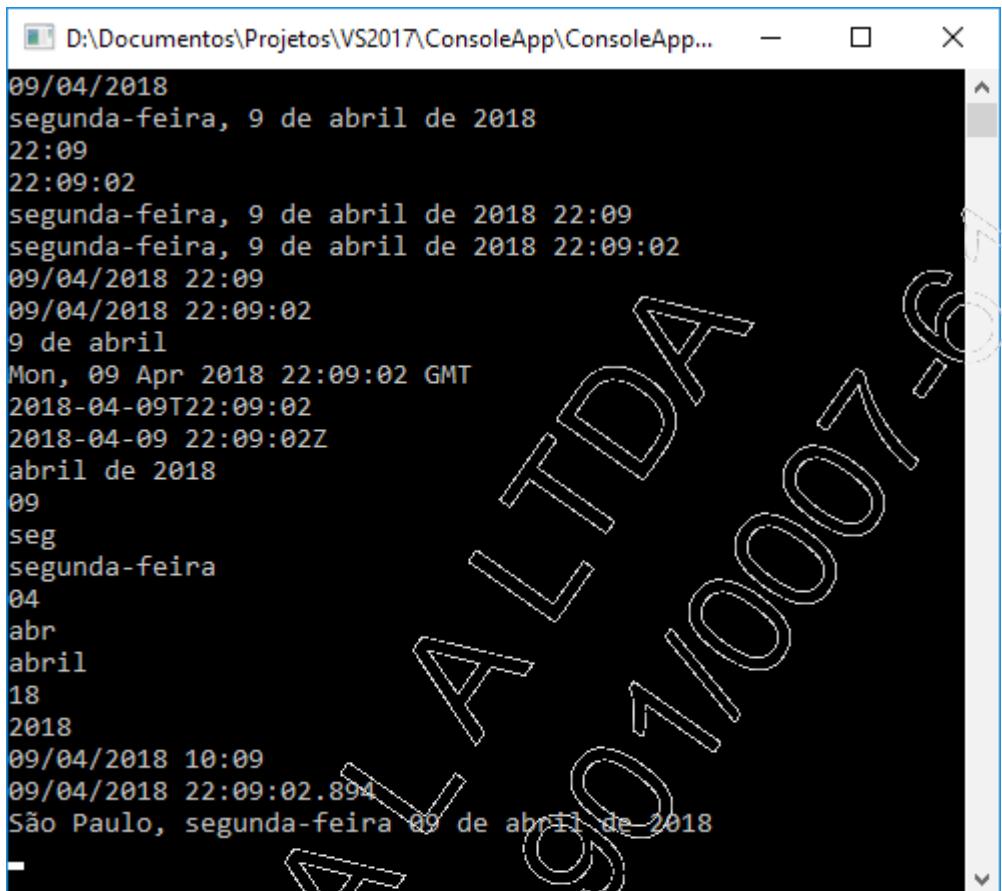
Caractere	Formato aplicado
d	Data curta
D	Data completa
t	Hora curta
T	Hora completa
f	Data e hora
F	Data e hora completa
g	Data e hora padrão
G	Data e hora padrão completa
M	Dia / mês
r	Padrão RFC1123 (Requirements for Internet Hosts)
s	Data e hora sortable
u	Formato universal – Timezone local
Y	Mês / ano
dd	Dia com dois dígitos
ddd	Dia da semana abreviado
ddd	Dia da semana completo
MM	Mês com dois dígitos
MMM	Mês abreviado
MMMM	Mês completo
yy	Ano com dois dígitos
yyyy	Ano com quatro dígitos
h	Hora padrão 12h com um dígito
hh	Hora padrão 12h com dois dígitos
H	Hora padrão 24h com um dígito
HH	Hora padrão 24h com dois dígitos
m	Minuto com um dígito
mm	Minuto com dois dígitos

Caractere	Formato aplicado
s	Segundo com um dígito
ss	Segundo com dois dígitos
fff	Milissegundos
tt	AM/PM

Veja um exemplo:

```
static void Main(string[] args)
{
    Console.WriteLine(
        DateTime.Now.ToString("d") + "\n" +
        DateTime.Now.ToString("D") + "\n" +
        DateTime.Now.ToString("t") + "\n" +
        DateTime.Now.ToString("T") + "\n" +
        DateTime.Now.ToString("F") + "\n" +
        DateTime.Now.ToString("F") + "\n" +
        DateTime.Now.ToString("g") + "\n" +
        DateTime.Now.ToString("G") + "\n" +
        DateTime.Now.ToString("M") + "\n" +
        DateTime.Now.ToString("r") + "\n" +
        DateTime.Now.ToString("s") + "\n" +
        DateTime.Now.ToString("u") + "\n" +
        DateTime.Now.ToString("Y") + "\n" +
        DateTime.Now.ToString("dd") + "\n" +
        DateTime.Now.ToString("ddd") + "\n" +
        DateTime.Now.ToString("ddd") + "\n" +
        DateTime.Now.ToString("MM") + "\n" +
        DateTime.Now.ToString("MMM") + "\n" +
        DateTime.Now.ToString("MMMM") + "\n" +
        DateTime.Now.ToString("yy") + "\n" +
        DateTime.Now.ToString("yyyy") + "\n" +
        DateTime.Now.ToString("dd/MM/yyyy hh:mm") + "\n" +
        DateTime.Now.ToString("dd/MM/yyyy HH:mm:ss.fff") + "\n" +
        DateTime.Now.ToString("São Paulo, dddd dd 'de' MMMM 'de' yyyy")));
    Console.ReadKey();
}
```

Agora, confira o resultado:



```
D:\Documentos\Projetos\VS2017\ConsoleApp\ConsoleApp... - X
09/04/2018
segunda-feira, 9 de abril de 2018
22:09
22:09:02
segunda-feira, 9 de abril de 2018 22:09
segunda-feira, 9 de abril de 2018 22:09:02
09/04/2018 22:09
09/04/2018 22:09:02
9 de abril
Mon, 09 Apr 2018 22:09:02 GMT
2018-04-09T22:09:02
2018-04-09 22:09:02Z
abril de 2018
09
seg
segunda-feira
04
abr
abril
18
2018
09/04/2018 10:09
09/04/2018 22:09:02.894
São Paulo, segunda-feira 09 de abril de 2018
```

## 1.12.2. Formatação por interpolação

Podemos também usar a nova sintaxe de formatação: a interpolação. Veja o exemplo:

```
static void Main(string[] args)
{
    string produto = "caneta";
    decimal preco = 6.32m;
    Console.WriteLine($"Produto: {produto}\nPreço: {preco}");

    Console.ReadKey();
}
```

Este modelo de formatação não utiliza índices, e segue os mesmos padrões do método Format() mostrado anteriormente. Esta opção está disponível a partir do C# 6.0.

Observe que basta iniciar a string a ser formatada com o caractere \$. As variáveis a serem formatadas são inseridas no texto, no lugar dos índices.

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A linguagem C# é utilizada para a criação de vários tipos de aplicativos. Trata-se de uma linguagem orientada a objetos, “fortemente tipada”, estável e versátil que, em conjunto com o Visual Studio, permite ao desenvolvedor um ganho de produtividade característico das ferramentas Microsoft;
- A plataforma .NET visa a uma única plataforma de desenvolvimento e execução de aplicativos e sistemas. Qualquer código gerado por essa plataforma poderá ser executado em qualquer dispositivo ou equipamento que possua o framework compatível;
- O **Integrated Development Environment (IDE)** do Visual Studio é um ambiente de desenvolvimento utilizado para escrever programas. Por se tratar de um software de programação potente e personalizável, o Visual Studio possui todas as ferramentas necessárias para desenvolver programas sólidos de forma rápida e eficiente. Muitas das características encontradas no Visual Studio IDE se aplicam de forma idêntica às características do Visual Basic, Visual C++ e Visual C#;
- Para transformar um código digitado em um programa que pode ser executado, esse código deve ser compilado. Após a compilação do código, podemos executá-lo. Para isso, na janela do Visual Studio, podemos utilizar a opção **Start Debugging** ou **Start Without Debugging**;
- Instrução é um comando responsável pela execução de uma determinada ação;
- Método é uma sequência nomeada de instruções;
- Sintaxe é o conjunto de instruções que obedecem a uma série de regras específicas para seu formato e construção;
- As variáveis podem ser definidas como locais destinados ao armazenamento temporário de informações de diferentes tipos, como números, palavras, propriedades e datas, entre outros;
- Os operadores indicam a operação matemática que será executada, gerando novos valores a partir de um ou mais operandos (itens à direita ou à esquerda do operador);
- A utilização da barra invertida (\) seguida de um ou mais caracteres específicos funciona como um comando nas strings;
- Para apresentarmos dados formatados no .NET, usamos o método **Format()** da classe **String** ou o método **ToString()** dos tipos não string. Podemos usar também a interpolação de string. Para tanto, devemos iniciar a string a ser formatada com o caractere \$ e, dentro da string, incluir as variáveis a serem formatadas no lugar dos índices.

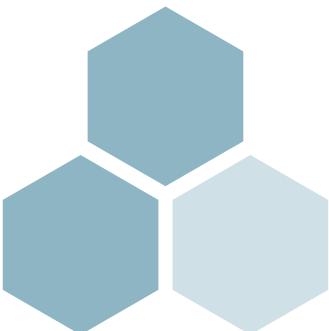




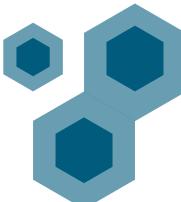
1

# Sintaxe da Linguagem C# e o Visual Studio 2019

Teste seus conhecimentos



Editora  
**IMPACTA**



### 1. O que deve ser feito para transformar um código em um programa executável?

- a) Nada, ele já pode ser executado.
- b) Registrá-lo no computador em que se deseja executá-lo.
- c) Ele deve ser compilado.
- d) Ele deve fazer parte da lista de aplicativos padrão do Windows (AppPath).
- e) Todas as alternativas anteriores estão corretas.

### 2. Qual das alternativas a seguir corresponde a um projeto do tipo Console Application?

- a) Uma lista simples que auxilia o desenvolvedor a escolher o nome das variáveis.
- b) Uma coleção orientada a evento que contém tipos reutilizáveis que podem ser empregados no desenvolvimento de aplicativos.
- c) Uma coleção de controles para serem utilizados na construção das telas.
- d) Conjunto de DLLs contendo milhares de recursos disponíveis para utilizarmos nas nossas aplicações.
- e) Uma aplicação contendo uma classe com o método Main, considerado o ponto de partida do programa.

**3. Considerando as seguintes variáveis e a expressão lógica adiante, qual a resposta correta?**

```
int a = 10, b = 15, c = 10;  
bool d = a == c || b != a && a == b;
```

- a) false
- b) !=
- c) true
- d) &\*
- e) !<>

**4. Considerando o código a seguir, qual será o resultado mostrado em label1?**

```
int a = 4, b = 5, c = 6;  
int d = (a++ / 2) * (++c % 2) * ++b;  
string r = d + (d % 3 == 0 ? " é " : " não é ") + "múltiplo  
de 3";  
  
Console.WriteLine(  
string.Format($"a = {a}, b = {b}, c = {c}, d = {d}\nresultado:  
{r}"));
```

- a) a = 5, b = 6, c = 8, d = 11, 11 não é múltiplo de 3.
- b) a = 4, b = 5, c = 9, d = 15, 15 é múltiplo de 3.
- c) a = 5, b = 6, c = 7, d = 12, 12 não é múltiplo de 3.
- d) a = 5, b = 6, c = 7, d = 12, 12 é múltiplo de 3.
- e) a = 5, b = 6, c = 7, d = 15, 15 é múltiplo de 3.

### 5. A que se refere a sintaxe a seguir?

```
teste lógico ? valor se verdadeiro : valor se falso;
```

- a) A um operador unário.
- b) A um operador binário.
- c) A um operador ternário.
- d) A um operador lógico reduzido.
- e) A nada.



1

# Sintaxe da Linguagem C# e o Visual Studio 2019



Mãos à obra!



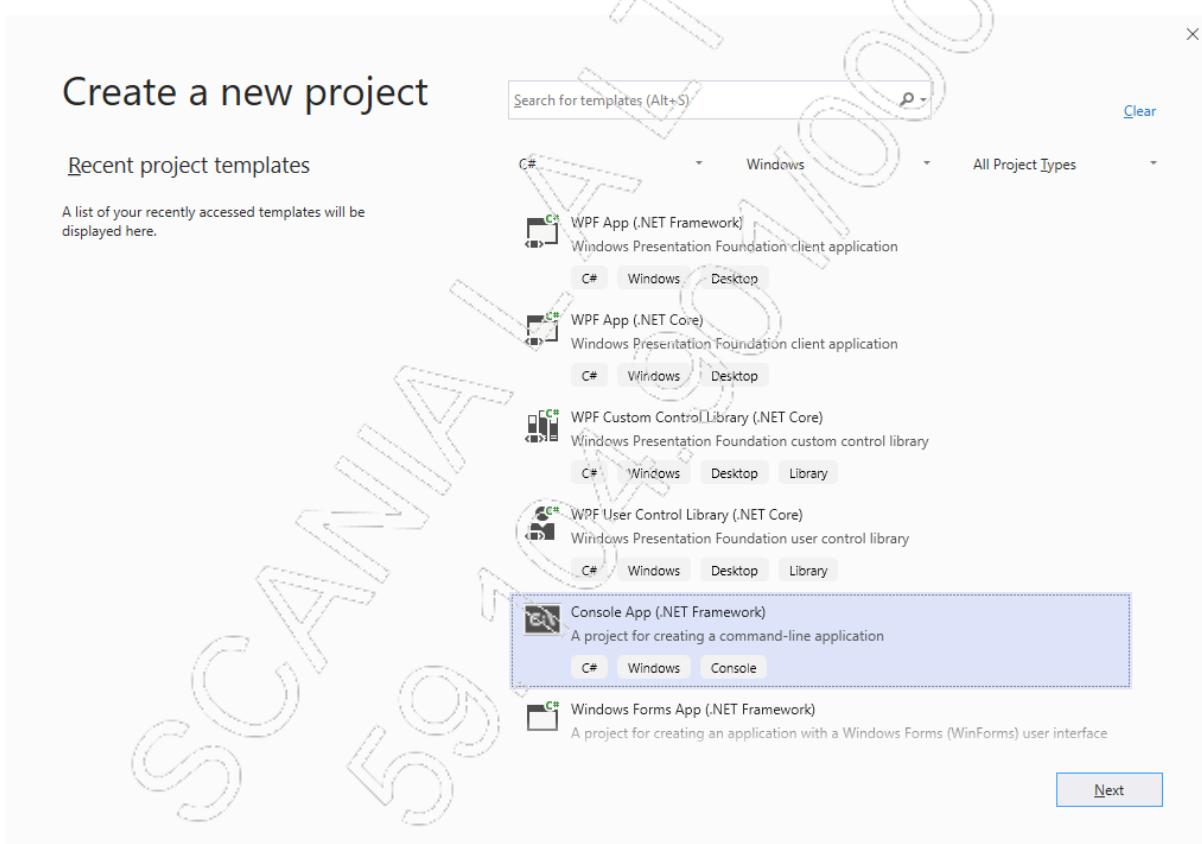
## Laboratório 1

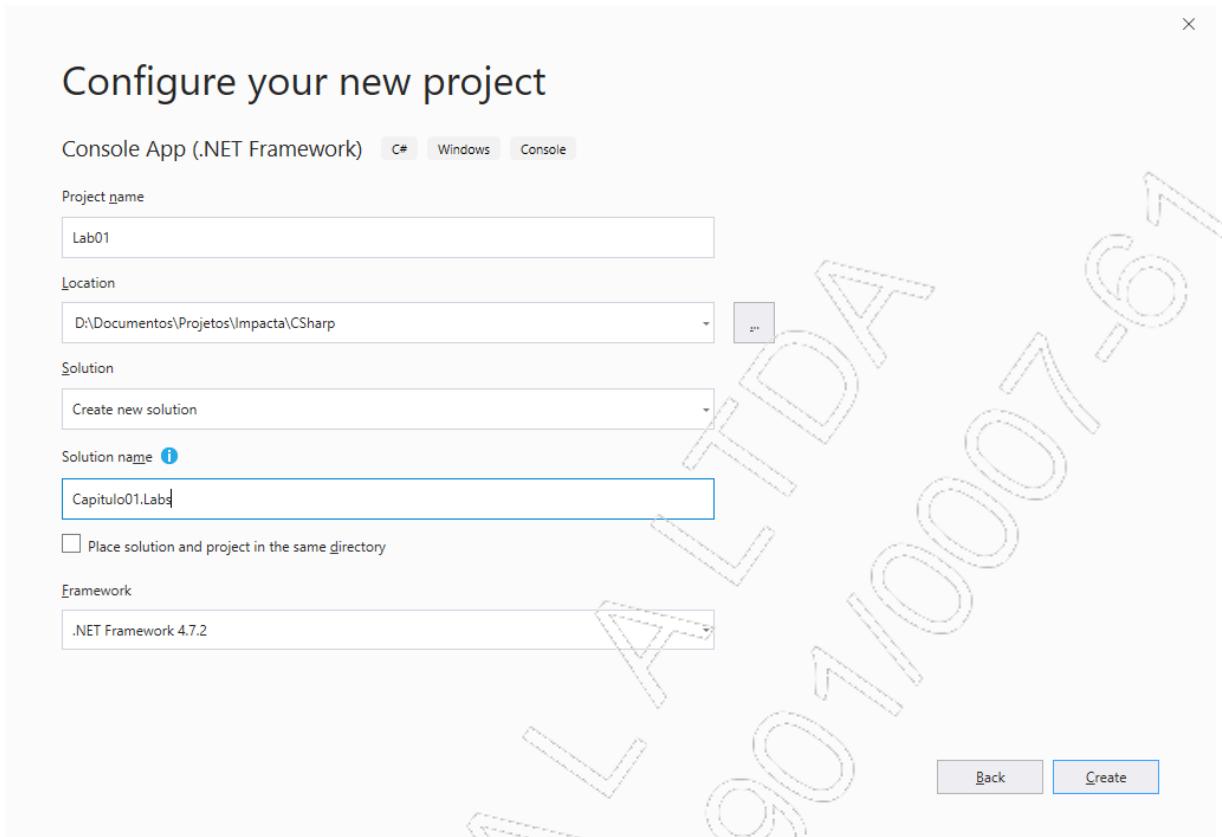
### A - Escrevendo um programa que calcula o quadrado de um número

#### Objetivo:

Escrever um programa que calcula o quadrado de um número fornecido como dado de entrada.

1. Inicie um novo projeto **Console App (.NET Framework)** chamado **Lab01**. Mantenha o nome do solution como **Capítulo01.Labs**:





2. No método **Main**, escreva uma instrução para solicitar um valor real ao usuário:

```
namespace Lab01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Fornecer um valor real: ");
            double valor = Convert.ToDouble(Console.ReadLine());
        }
    }
}
```

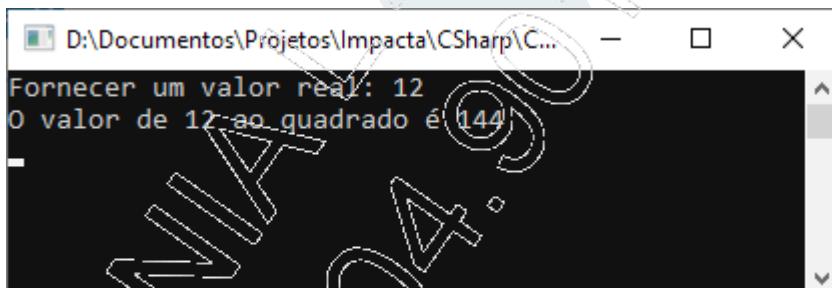
3. Declare uma variável para armazenar o resultado esperado. Em seguida, apresente o resultado:

```
namespace Lab01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Fornecer um valor real: ");
            double valor = Convert.ToDouble(Console.ReadLine());

            double resultado = Math.Pow(valor, 2);
            Console.WriteLine($"O valor de {valor} ao quadrado é
{resultado}");

            Console.ReadKey();
        }
    }
}
```

4. Exemplo de execução:



## Laboratório 2

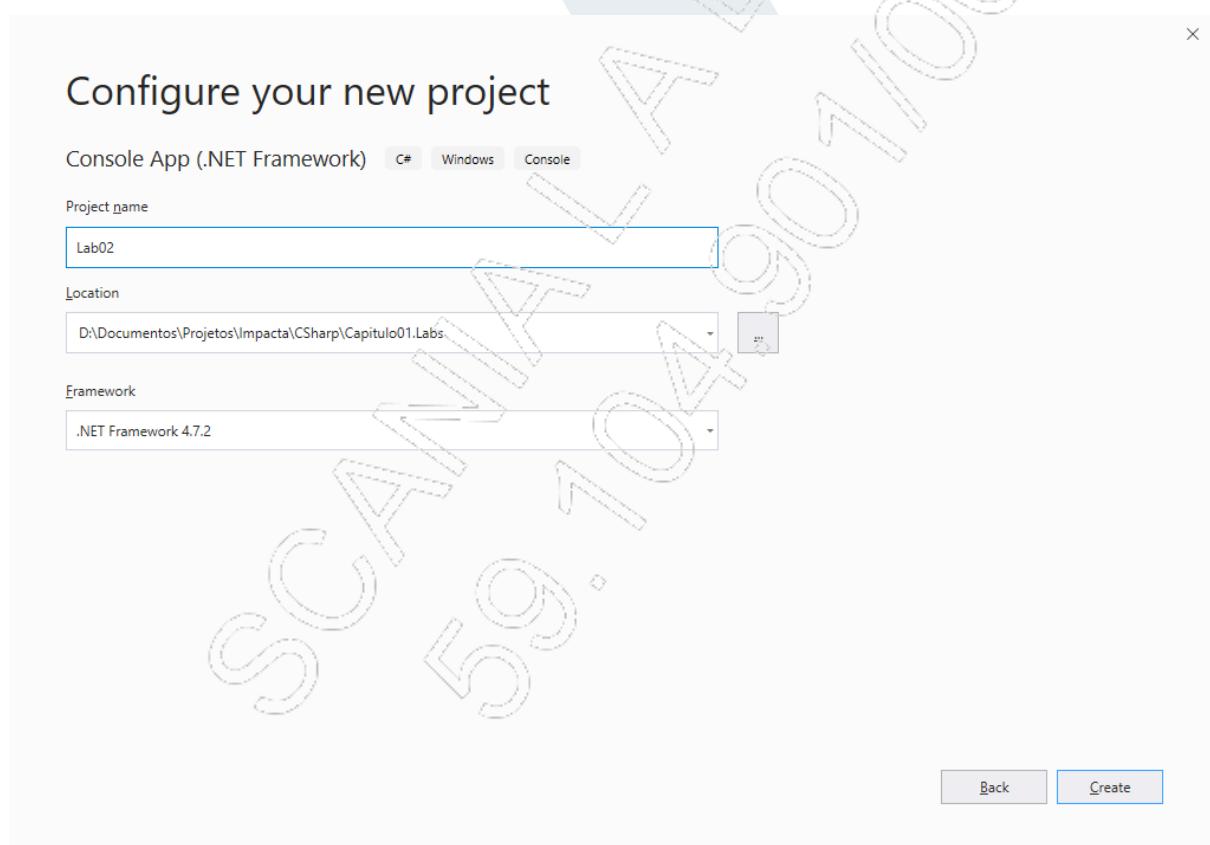
### A – Escrevendo um programa que solicite ao usuário dados de um funcionário

#### Objetivo:

Escrever um programa que solicite ao usuário os seguintes dados de um funcionário: **nome** e **salário**. Considere um desconto de 10% referente a taxas sociais. O programa deve apresentar:

- O nome do funcionário;
- O salário bruto;
- O valor do desconto;
- O salário líquido.

1. Adicione um novo projeto **Console App (.NET Framework)** chamado **Lab02** no solution existente **Capítulo01.Labs**:



# Programando com C#

2. Escreva, no método **Main**, as instruções para solicitar o nome e o salário do funcionário:

```
namespace Lab02
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe o nome do funcionário: ");
            string nome = Console.ReadLine();

            Console.Write("Informe o salário do funcionário: ");
            double salario = Convert.ToDouble(Console.ReadLine());
        }
    }
}
```

3. Escreva as instruções para calcular a taxa social (10%), calcular o salário líquido e, em seguida, apresentar os dados para o usuário:

```
namespace Lab02
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe o nome do funcionário: ");
            string nome = Console.ReadLine();

            Console.Write("Informe o salário do funcionário: ");
            double salario = Convert.ToDouble(Console.ReadLine());

            //Cálculo de 10% sobre o salário fornecido
            double desconto = salario * 10 / 100;

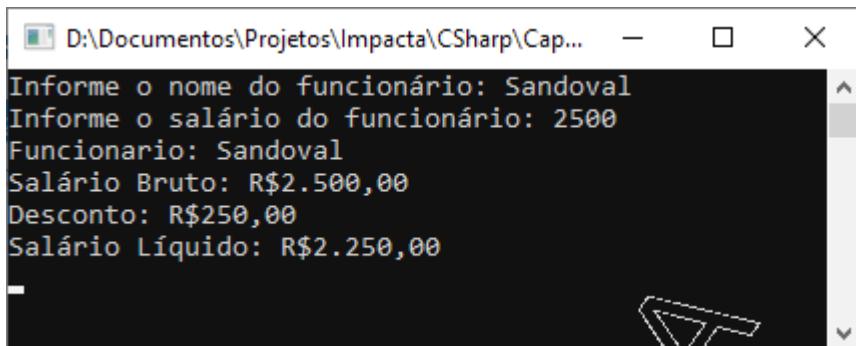
            double salarioLiquido = salario - desconto;

            string resposta =
                $"Funcionário: {nome}\nSalário Bruto: {salario:c}\n" +
                $"Desconto: {desconto:c}\nSalário Líquido: {salarioLiquido:c}";

            Console.WriteLine(resposta);

            Console.ReadKey();
        }
    }
}
```

4. Execute a aplicação. Marque o projeto **Lab02** como **Startup Project**.



```
D:\Documentos\Projetos\Impacta\CSharp\Cap...
Informe o nome do funcionário: Sandoval
Informe o salário do funcionário: 2500
Funcionário: Sandoval
Salário Bruto: R$2.500,00
Desconto: R$250,00
Salário Líquido: R$2.250,00
```

## Laboratório 3

### A – Escrevendo outro programa que solicite dados de um funcionário

#### Objetivo:

Escrever um programa que solicite os dados de um funcionário:

- O nome;
- O salário;
- Valor gasto com transporte por mês.

Considere que o funcionário tem um desconto máximo de 6% sobre seu salário. Se o valor gasto com transporte for superior a 6% do salário, o desconto é calculado com base nessa porcentagem. Porem, se o valor gasto com transporte for inferior a 6% do salário, o desconto é exatamente o valor gasto.

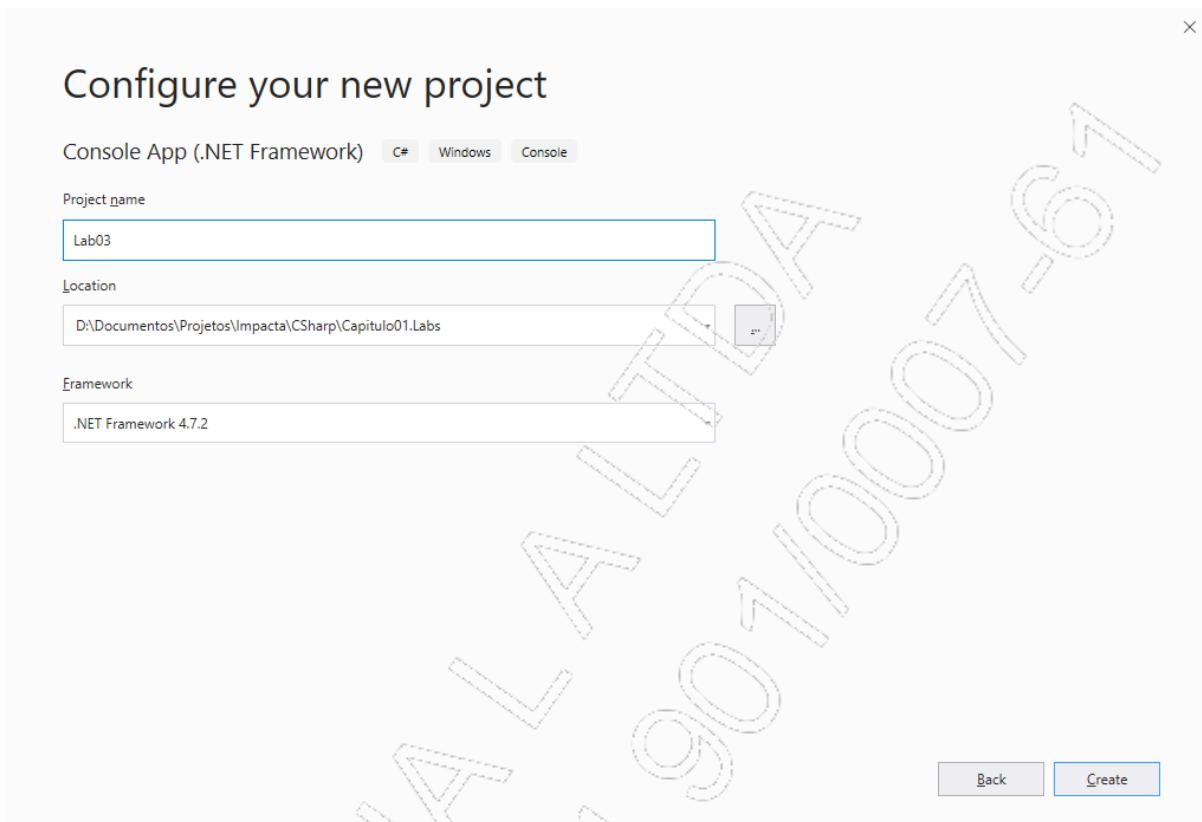
O programa deve apresentar:

- O nome do funcionário;
- O valor do salário;
- O valor descontado pelo transporte.

Usar o operador ternário para decidir pelo valor do desconto com transporte.

# Programando com C#

1. Adicione um projeto **Console App** chamado **Lab03** ao solution existente **Capítulo01.Labs**:



2. No método **Main**, escreva as instruções para solicitar ao usuário os dados de entrada:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe o nome do funcionário: ");
            string nome = Console.ReadLine();

            Console.Write("Informe o salário do funcionário: ");
            double salario = Convert.ToDouble(Console.ReadLine());

            Console.Write("Informe o valor gasto com transporte: ");
            double transporte = Convert.ToDouble(Console.ReadLine());
        }
    }
}
```

3. Escreva o código para calcular 6% sobre o salário. Em seguida, escreva o código para determinar o valor real descontado como vale-transporte:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe o nome do funcionário: ");
            string nome = Console.ReadLine();

            Console.Write("Informe o salário do funcionário: ");
            double salario = Convert.ToDouble(Console.ReadLine());

            Console.Write("Informe o valor gasto com transporte: ");
            double transporte = Convert.ToDouble(Console.ReadLine());

            //cálculo de 6% sobre o salário
            double vt = salario * 6 / 100;

            //verificação do valor real do vale-transporte
            double vt_real = (transporte > vt ? vt : transporte);

        }
    }
}
```

4. Apresentação do resultado:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe o nome do funcionário: ");
            string nome = Console.ReadLine();

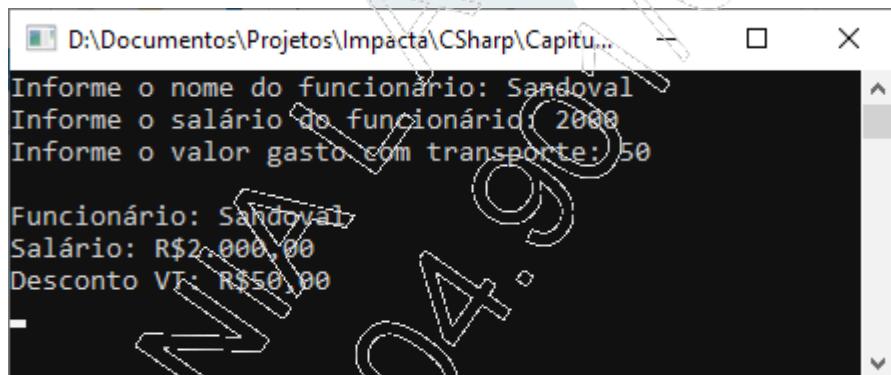
            Console.Write("Informe o salário do funcionário: ");
            double salario = Convert.ToDouble(Console.ReadLine());

            Console.Write("Informe o valor gasto com transporte: ");
            double transporte = Convert.ToDouble(Console.ReadLine());

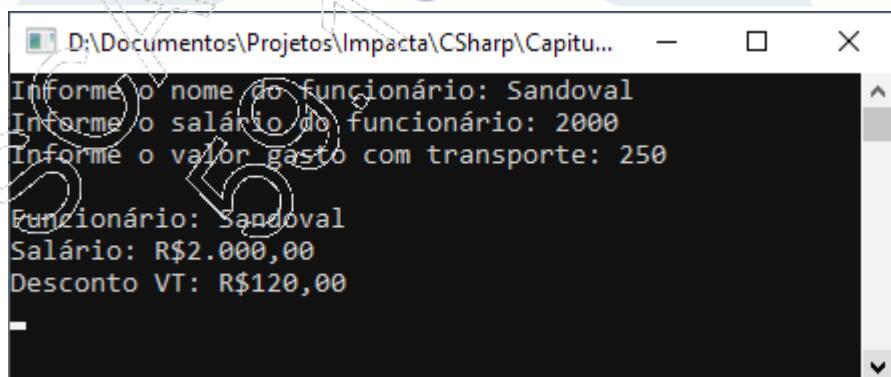
            //cálculo de 6% sobre o salário
            double vt = salario * 6 / 100;
```

```
//verificação do valor real do vale-transporte  
double vt_real = (transporte > vt ? vt : transporte);  
  
string resultado = $"\\nFuncionário: {nome}\\n" +  
    $"Salário: {salario:c}\\n" +  
    $"Desconto VT: {vt_real:c}";  
  
Console.WriteLine(resultado);  
  
Console.ReadKey();  
}  
}  
}
```

5. Marque o projeto **Lab03** como **StartUp Project**. Teste o programa.



```
D:\Documentos\Projetos\Impacta\CSharp\Capitu...  
Informe o nome do funcionário: Sandoval  
Informe o salário do funcionário: 2000  
Informe o valor gasto com transporte: 50  
  
Funcionário: Sandoval  
Salário: R$2.000,00  
Desconto VT: R$50,00
```



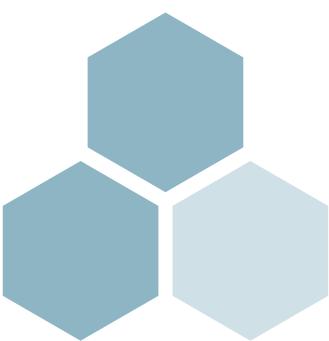
```
D:\Documentos\Projetos\Impacta\CSharp\Capitu...  
Informe o nome do funcionário: Sandoval  
Informe o salário do funcionário: 2000  
Informe o valor gasto com transporte: 250  
  
Funcionário: Sandoval  
Salário: R$2.000,00  
Desconto VT: R$120,00
```



2

# Estruturas de Controle da Linguagem C#

- Estruturas de controle;
- Estruturas de repetição.



### 2.1. Introdução

Na maior parte das vezes, é necessário produzir desvios no programa em vez de simplesmente seguir um fluxo contínuo. Esses desvios podem ocorrer por conta da necessidade de executar um trecho do programa somente se alguma condição for satisfeita. Um exemplo muito simples ocorre no fechamento das notas em uma faculdade. Todo final de ano, no boletim escolar, na frente do nome de cada disciplina aparece escrito se o aluno foi aprovado ou se está de exame ou reprovado. Nesse caso, para cada aluno da faculdade, a partir de suas notas semestrais, o sistema pode realizar uma dentre três opções. A escolha de uma das opções depende da média do aluno. Vale destacar que o sistema escolherá apenas uma opção e automaticamente eliminará as outras.

Em outras situações, uma instrução ou grupo de instruções deve se repetir. É o caso, por exemplo, da geração de relatórios, em que determinados dados devem ser listados de forma repetitiva. O número de repetições pode ser controlado pelo programa. A quantidade de vezes que um conjunto de instruções será executado depende de um teste lógico. As estruturas de repetição em programação também são conhecidas como **loops**.

Neste capítulo, veremos os comandos de decisão e as estruturas de repetição, incluindo possíveis variações na sua elaboração.

### 2.2. Estruturas de controle

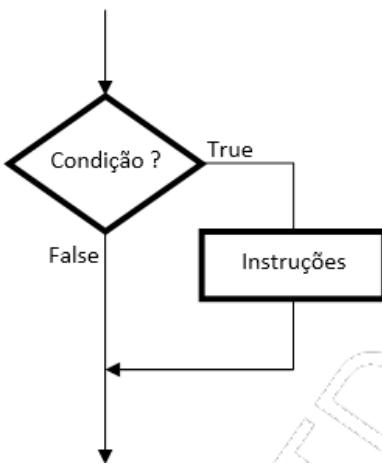
Para decidirmos qual ação deverá ser tomada em determinada parte de um programa, solicitamos que este avalie uma expressão. Caso essa expressão seja avaliada como verdadeira, uma sequência de comandos será executada; caso contrário, uma nova decisão poderá ser tomada ou outra sequência de instruções poderá ser executada, ou, ainda, nada será executado e a próxima linha de comando abaixo do bloco de decisão será executada.

#### 2.2.1. O comando if

Quando utilizamos o comando **if**, apenas expressões com resultados booleanos podem ser avaliadas. Para definir um comando **if**, deve-se utilizar a seguinte sintaxe:

```
if(teste lógico)
{
    Instruções para o teste lógico verdadeiro
}
```

O fluxo de execução deste teste segue o seguinte fluxo:



```
if (teste lógico)
{
    Comandos se o teste lógico for verdadeiro
}
else // é opcional
{
    Comandos se o teste lógico for falso
}
```

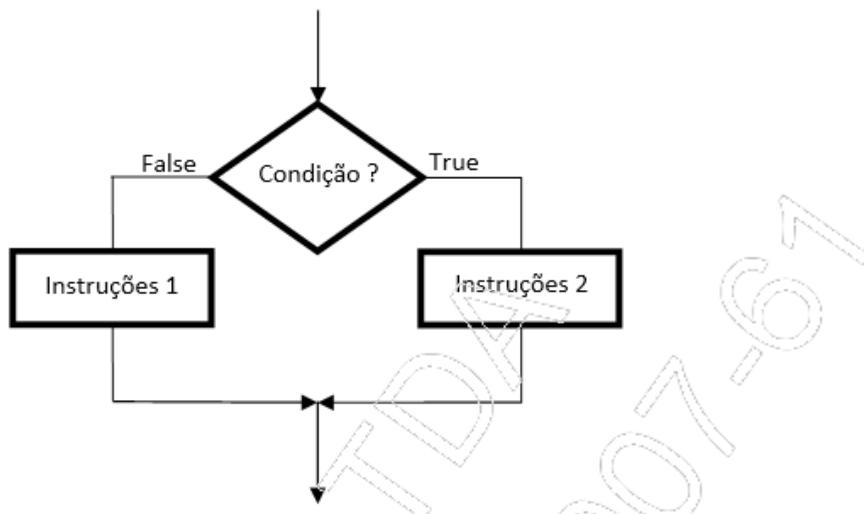
O **teste lógico** precisa estar entre parênteses e pode ser qualquer expressão que retorne **true** ou **false**. Em caso verdadeiro, o bloco de instruções que se encontra entre as chaves (`{ }`) será executado.

## 2.2.2. O comando if / else

Outra possibilidade é incluir um bloco alternativo, a ser executado se a condição testada resultar em **false**. É o caso do comando **if / else**, cuja forma geral é:

```
if(teste lógico)
{
    Instruções para o teste lógico verdadeiro
}
else
{
    Instruções para o teste lógico falso
}
```

O fluxo para este cenário é:



As chaves ({} ) do bloco poderão ser omitidas caso haja apenas uma instrução. Mesmo assim, as boas práticas requerem que sejam incluídas as chaves.

Veja, no código a seguir, alguns exemplos com os comandos **if** e **if / else**:

```
static void Main(string[] args)
{
    //Definir as variáveis
    int a;
    int b;
    //-----
    Console.WriteLine("Exemplo 1");
    a = 20;
    b = 100;

    //Como a condição é verdadeira, o texto será exibido
    if (a < b)
    {
        Console.WriteLine(a + " é menor que " + b);
    }
    Console.WriteLine(new string('-', 30));
    //-----

    Console.WriteLine("Exemplo 2");
    a = 20;
    b = 10;

    //Como a condição é falsa, o texto NÃO será exibido
    if (a < b)
    {
        Console.WriteLine(a + " é menor que " + b);
    }
    Console.WriteLine(new string('-', 30));
    //-----
```

```
Console.WriteLine("Exemplo 3");
a = 20;
b = 10;

//Como a condição é falsa, será exibido o texto do else
if (a < b)
{
    Console.WriteLine(a + " é menor que " + b);
}
else
{
    Console.WriteLine(a + " é maior que " + b);
}
Console.WriteLine(new string('-', 30));
//-----
```

```
Console.WriteLine("Exemplo 4 - com erro");
a = 20;
b = 20;

//Como a condição é falsa, será exibido o texto do else
//que, nesse caso, será um erro
if (a < b)
{
    Console.WriteLine(a + " é menor que " + b);
}
else
{
    Console.WriteLine(a + " é maior que " + b);
}
Console.WriteLine(new string('-', 30));
//-----
```

```
Console.WriteLine("Exemplo 4 - corrigido");
a = 20;
b = 20;

//Para corrigir esta decisão
//precisaremos de mais uma comparação
if (a < b)
{
    Console.WriteLine(a + " é menor que " + b);
}
else if (a > b)
{
    Console.WriteLine(a + " é maior que " + b);
}
else
{
    Console.WriteLine(a + " é igual a " + b);
}
Console.WriteLine(new string('-', 30));

Console.ReadKey();
```

O resultado desse código pode ser visto a seguir:

```
D:\Documentos\Projetos\VS201... Exemplo 1  
20 é menor que 100  
-----  
Exemplo 2  
-----  
Exemplo 3  
20 é maior que 10  
-----  
Exemplo 4 - com erro  
20 é maior que 20  
-----  
Exemplo 4 - corrigido  
20 é igual a 20
```

## 2.2.3. A instrução switch / case

Quando temos diversas instruções **if** em cascata avaliando uma expressão idêntica, podemos usar **switch / case**, desde que sejam atendidas as seguintes exigências:

- A expressão avaliada pelo **switch** tem que ser de um tipo INTEIRO, BOOL, STRING, ENUMERADO, além de classes, estruturas e tipos genéricos, permitindo avaliar o seu tipo. **Os tipos genéricos foram introduzidos no C# 7.0;**
- O resultado avaliado tem que ser exato, ou seja, não podemos usar maior, menor, **&&**, **||** etc.

Veja a sintaxe de **switch**:

```
switch (expressão)
{
    case valor1:
        instruções a executar se a expressão for igual a valor1
        break;
    case valor2:
        instruções a executar se a expressão for igual a valor2
        break;
    case valor3:
        instruções a executar se a expressão for igual a valor3
        break;
    default
        instruções a executar se a expressão for diferente
            dos valores anteriores
        break;
}
```

O funcionamento da sintaxe de **switch** pode ser descrito da seguinte forma:

1. A **expressão** é avaliada;
2. O controle passa para o bloco de código em que **valor** é igual à **expressão**;
3. A execução do código é realizada até que se atinja a instrução **break**;
4. A instrução **switch** é finalizada;
5. O programa prossegue em sua execução a partir da instrução localizada após a chave de fechamento de **switch**.

Apenas nos casos em que não for encontrado um **valor** igual à **expressão**, as instruções do rótulo **default** serão executadas. Se tal rótulo não tiver sido criado, a execução continuará a partir da instrução imediatamente após a chave de fechamento de **switch**.

A seguir, temos um exemplo do uso de **switch / case**:

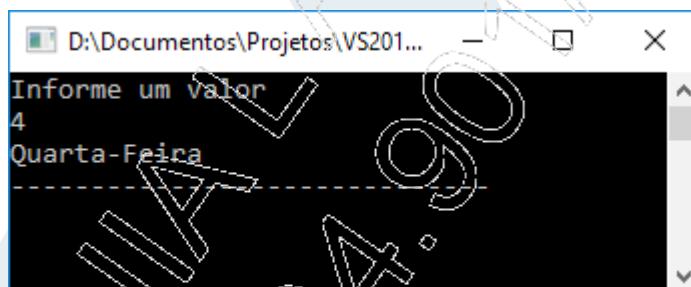
```
static void Main(string[] args)
{
    //Definir as variáveis
    Console.WriteLine("Informe um valor");
    string diaSemana = Console.ReadLine();
    string resultado = "";

    //Verificar a existência de algum valor para a variável
    diaSemana
    if (diaSemana.Trim() == "")  

    {
        return;
    }
    //Utilizar a instrução switch
    switch (diaSemana)
    {
        case "1":
            resultado = "Domingo";
            break;
        case "2":
            resultado = "Segunda-Feira";
            break;
        case "3":
            resultado = "Terça-Feira";
            break;
        case "4":
            resultado = "Quarta-Feira";
            break;
    }
}
```

```
        case "5":  
            resultado = "Quinta-Feira";  
            break;  
        case "6":  
            resultado = "Sexta-Feira";  
            break;  
        case "7":  
            resultado = "Sábado";  
            break;  
        default:  
            resultado = "Informe um valor entre 1 e 7";  
            break;  
    }  
  
    Console.WriteLine(resultado);  
    Console.WriteLine(new string('-', 30));  
  
    Console.ReadKey();  
}
```

O resultado desse código pode ser visto a seguir:



Para executar as mesmas instruções para mais de um valor, deve-se inserir uma lista de rótulos **case** com as instruções localizadas apenas ao final da lista. Se para algum rótulo existir uma ou mais instruções associadas, será gerado um erro pelo compilador, impedindo que a execução prossiga para os próximos rótulos. Portanto, quando uma opção **case** possui o mesmo resultado de outra opção **case**, podemos escrever o código da seguinte forma:

```
static void Main(string[] args)  
{  
    //Definir as variáveis  
    Console.WriteLine("Informe um valor");  
    string diaSemana = Console.ReadLine();  
    string resultado = "";  
  
    //Verificar a existência de algum valor para a variável  
    diaSemana  
    if (diaSemana.Trim() == "")  
    {  
        return;  
    }
```

```
//Utilizar a instrução switch
switch (diaSemana)
{
    case "1":
        resultado = "Domingo";
        break;
    case "2":
    case "3":
    case "4":
    case "5":
    case "6":
        resultado = "Dia útil";
        break;
    case "7":
        resultado = "Sábado";
        break;
    default:
        resultado = "Informe um valor entre 1 e 7";
        break;
}

Console.WriteLine(resultado);
Console.WriteLine(new string('-', 30));

Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:



## 2.2.4. Expressões switch (switch expressions)

O C# 8.0 expandiu a utilização das estruturas **switch**, deixando-as mais limpas e simplificadas. Trata-se das **switch expressions**. Nessas expressões:

- O comando case foi omitido, e o valor analisado é precedido pelo operador seta (`=>`);
  - A palavra default foi substituída pelo símbolo `_`.

# Programando com C#

Vamos analisar alguns exemplos:

```
class Program
{
    enum Periodo
    {
        M, T, N
    }
    static void Main(string[] args)
    {
        Periodo periodo = Periodo.M;
        string descricao;

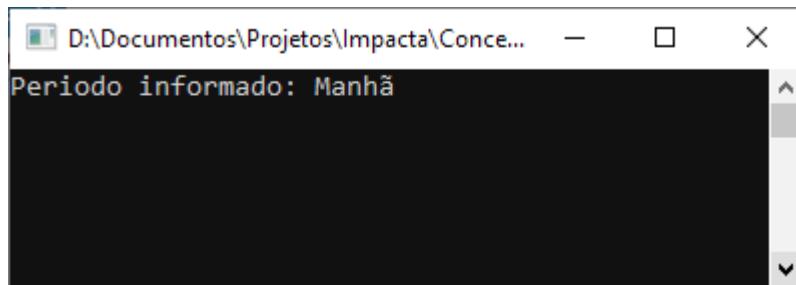
        _ = (periodo switch
        {
            Periodo.M => descricao = "Manhã",
            Periodo.T => descricao = "Tarde",
            Periodo.N => descricao = "Noite",
            _ => descricao = "Período inválido"
        });

        Console.WriteLine("Período informado: " + descricao);
        Console.ReadKey();
    }
}
```

É possível usarmos switch expressions com métodos. Mais detalhes sobre métodos serão apresentados posteriormente.

```
class Program
{
    enum Periodo
    {
        M, T, N
    }
    static string GetPeriodo(Periodo periodo) =>
        periodo switch
    {
        Periodo.M => "Manhã",
        Periodo.T => "Tarde",
        Periodo.N => "Noite",
        _ => "Período inválido"
    };
    static void Main(string[] args)
    {
        Console.WriteLine("Período informado: " + GetPeriodo(Periodo.M));
        Console.ReadKey();
    }
}
```

Em ambos os casos, a resposta será:



## 2.3. Estruturas de repetição

As estruturas de repetição de linguagem em C# são as seguintes:

- **while**;
- **do / while**;
- **for**;
- **foreach**.

A seguir, veremos cada uma delas mais detalhadamente e, também, estudaremos outros comandos utilizados em conjunto com as instruções **while**, **do / while**, **for** e **foreach**.

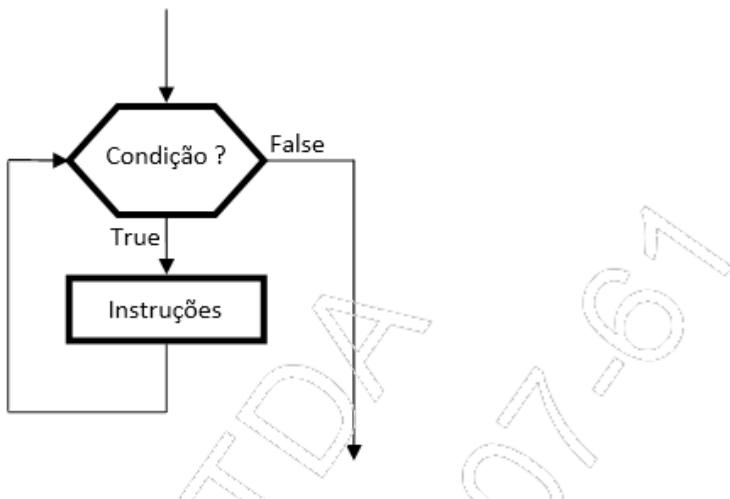
### 2.3.1. Estrutura while (teste no início)

Quando não sabemos quantas vezes um determinado bloco de instruções precisa ser repetido, utilizamos o loop **while**. Com ele, a execução das instruções continuará enquanto uma determinada condição for verdadeira (**true**). A condição a ser analisada para execução do loop de repetição deverá retornar um valor booleano.

Veja a sintaxe do loop **while**:

```
while (teste lógico)
{
    comandos;
}
```

O fluxo para esta estrutura pode ser mostrado na figura a seguir:



No exemplo a seguir, o corpo da estrutura será executado somente se a condição for verdadeira. Assim, a execução se repetirá até que essa condição seja falsa. Veja:

```
static void Main(string[] args)
{
    int num = 1000;

    //Enquanto num for menor ou igual a 5000
    while (num <= 5000)
    {
        Console.WriteLine("Num: " + num);
        num += 1000;
    }

    Console.WriteLine(new string('-', 30));
    Console.ReadKey();
}
```

O resultado é o seguinte:

A screenshot of a terminal window titled "D:\Documentos\Projetos\VS2017\Console...". The window displays the output of the C# program. The text shows the variable "Num" being printed five times, starting at 1000 and increasing by 1000 each time, up to 5000. After the loop, there is a separator line consisting of 30 dashes, followed by a blank line and a cursor at the bottom.

```
Num: 1000
Num: 2000
Num: 3000
Num: 4000
Num: 5000
-----
```

É importante saber que o loop **while** testa, em primeiro lugar, a condição: se for verdadeira, o bloco de instruções será executado; caso a condição seja falsa, o programa passará para a execução da próxima instrução após o loop.

! É fundamental incluir sempre uma expressão booleana que seja avaliada como **false**. Assim, definimos o ponto em que deve ser finalizado o loop e, consequentemente, evitamos a execução contínua do programa.

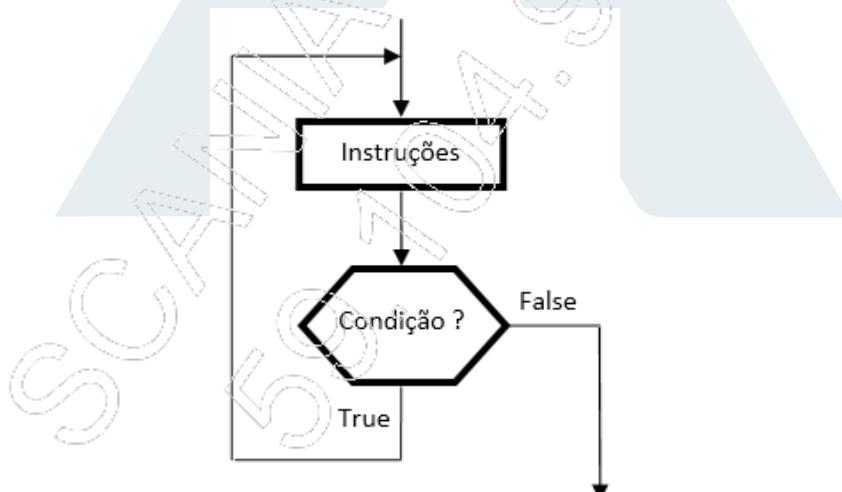
## 2.3.2. Estrutura do / while (teste no final)

A estrutura **do / while** tem basicamente o mesmo funcionamento da instrução **while**, porém, executa todos os comandos ao menos uma única vez, mesmo quando a condição não é verdadeira.

Veja sua sintaxe:

```
do
{
    comandos;
}
while(teste lógico);
```

Fluxo:



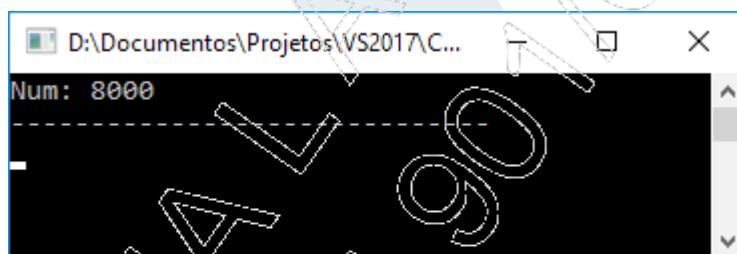
Observe o exemplo a seguir:

```
static void Main(string[] args)
{
    int num = 8000;

    //O bloco de instruções será executado uma vez
    do
    {
        Console.WriteLine("Num: " + num);
        num += 1000;
    } while (num <= 5000); //Enquanto num for menor ou igual a
    5000

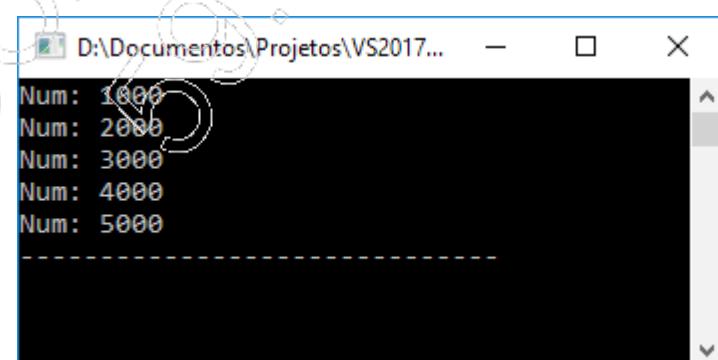
    Console.WriteLine(new string('-', 30));
    Console.ReadKey();
}
```

O resultado é o seguinte:



As linhas de comando abaixo da instrução **do** foram executadas uma vez, já que a variável possui o valor **8000** e, por isso, não atende à condição **while**.

Ao alterarmos o valor dessa variável para **1000**, por exemplo, obtemos o seguinte resultado:



### 2.3.3. Estrutura for

A instrução **for** é usada para fazer um loop que inicia um contador em um valor específico. Ao executar o código que está dentro do loop, o contador é incrementado e, enquanto o limite estabelecido para o contador não for alcançado, o loop será repetido. Utilizamos o loop **for** quando sabemos exatamente quantas vezes um bloco de instruções deve ser repetido.

A declaração simples e mais comum do loop **for** é a seguinte:

```
for(declaração e inicialização de variáveis; teste lógico; expressão  
de iteração)  
{  
    Bloco de instruções;  
}
```

Além do corpo do loop, **for** possui as seguintes partes principais: declaração e inicialização de variáveis, expressão condicional e expressão de iteração.

- **Declaração e inicialização de variáveis**

Nesta primeira parte da instrução **for**, podemos declarar e inicializar uma ou mais variáveis, quando houver. Elas são colocadas entre parênteses, após a palavra-chave **for** e, se houver mais de uma variável do mesmo tipo, elas são separadas por vírgulas, conforme o seguinte exemplo:

```
for (int a = 1, b = 1;
```

É importante lembrarmos que a declaração e a inicialização das variáveis, na instrução **for**, acontecem sempre antes de outros comandos. Elas ocorrem apenas uma vez no loop, sendo que o teste booleano e a expressão de iteração são executados a cada loop do programa.

Veja o exemplo a seguir:

```
static void Main(string[] args)  
{  
    for (int a = 1, b = 1; (a < 300 && b < 300); a *= 3, b *= 2)  
    {  
        Console.WriteLine("Valor a: " + a);  
        Console.WriteLine("Valor b: " + b);  
        Console.WriteLine(new string('-', 30));  
    }  
  
    Console.ReadKey();  
}
```

Em que:

- `int` refere-se ao tipo de variável que está sendo declarada;
- `a = 1` e `b = 1` referem-se às variáveis que estão sendo inicializadas;
- `(a < 300 && b < 300)` são testes booleanos que serão feitos nas variáveis;
- `a *= 3`, `b *= 2` são as iterações das variáveis `a` e `b`.

Veja o resultado da execução desse código:

```
D:\Documentos\Projetos\VS2012\Exercícios\Exercício 01>
Valor a: 1
Valor b: 1
-----
Valor a: 3
Valor b: 2
-----
Valor a: 9
Valor b: 4
-----
Valor a: 27
Valor b: 8
-----
Valor a: 81
Valor b: 16
-----
Valor a: 243
Valor b: 32
```

Podemos deixar de declarar uma das três partes descritas do loop **for**, apesar de essa não ser uma prática recomendada. Nesse caso, o loop será infinito e, não havendo seções de declaração e inicialização no loop **for**, ele agirá como um loop de repetição **while**.

Em relação ao escopo das variáveis que foram declaradas no loop **for**, ele é finalizado juntamente com o loop. Sendo assim, devemos observar o seguinte:

- A variável declarada no loop **for** pode ser utilizada apenas no corpo do próprio loop **for**;
- A variável declarada fora do loop **for**, mas inicializada na instrução **for**, pode ser utilizada tanto dentro quanto fora do loop **for**.

- O comando **break**

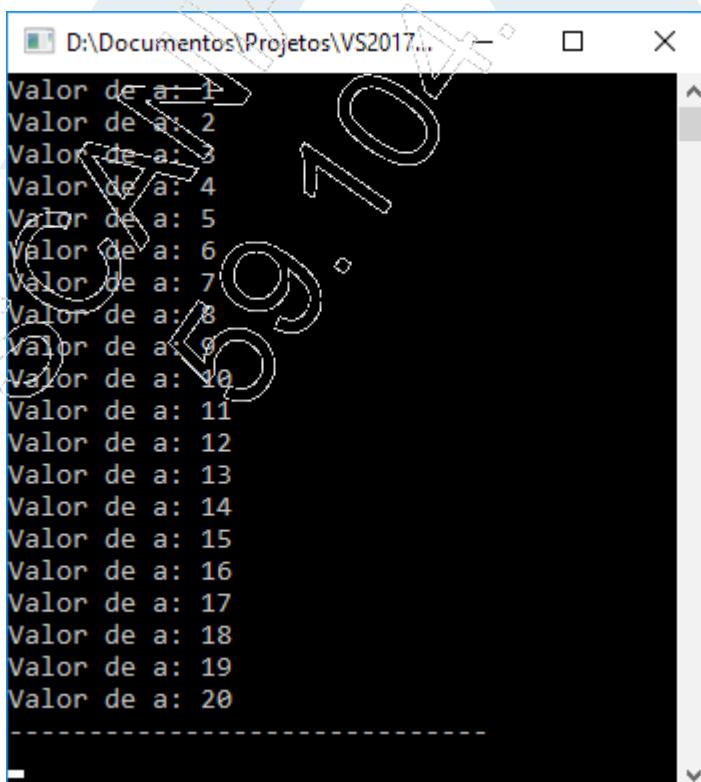
O comando **break** é utilizado para interromper um bloco de instruções, seja ele um loop **while**, **do / while**, **for** ou **foreach**, ou comando **switch / case**, antes que ele seja completado. Feita a interrupção, a próxima instrução após o comando **break** é executada, caso haja alguma. Se existir mais de uma instrução **while**, **do / while**, **for** ou **foreach**, ou **switch / case** aninhadas, o comando **break** terá efeito sobre aquela que, entre elas, estiver no nível mais interno, ou seja, contida em todas as outras. Caso o comando **break** não esteja contido em nenhuma instrução ou comando a que ele se atribua, um erro ocorrerá no momento de compilação.

O exemplo a seguir ilustra a utilização da instrução **break**:

```
static void Main(string[] args)
{
    for (int a = 1; a <= 1000; a++)
    {
        if (a > 20) { break; }
        Console.WriteLine("Valor de a: " + a);
    }

    Console.WriteLine(new string('-', 30));
    Console.ReadKey();
}
```

O resultado será o seguinte:



```
D:\Documentos\Projetos\VS2017...
Valor de a: 1
Valor de a: 2
Valor de a: 3
Valor de a: 4
Valor de a: 5
Valor de a: 6
Valor de a: 7
Valor de a: 8
Valor de a: 9
Valor de a: 10
Valor de a: 11
Valor de a: 12
Valor de a: 13
Valor de a: 14
Valor de a: 15
Valor de a: 16
Valor de a: 17
Valor de a: 18
Valor de a: 19
Valor de a: 20
-----
```

- O comando **continue**

O comando **continue** pode ser usado em estruturas de repetição **while**, **do / while**, **for** ou **foreach**, com a finalidade de voltar à execução da condição do loop de repetição ao qual ele se aplica. Assim como ocorre com o comando **break**, um erro durante a compilação é gerado caso o **continue** não esteja contido em um ambiente em que ele possa ser aplicado, ou seja, um loop de repetição. Outras características que os dois comandos compartilham são o fato de que, para instruções aninhadas, terão efeito sobre a que estiver no nível mais interno, e também a impossibilidade de definir o ponto final do comando em si, já que ele transfere o controle para outra parte do código.

O exemplo a seguir ilustra a utilização da instrução **continue**:

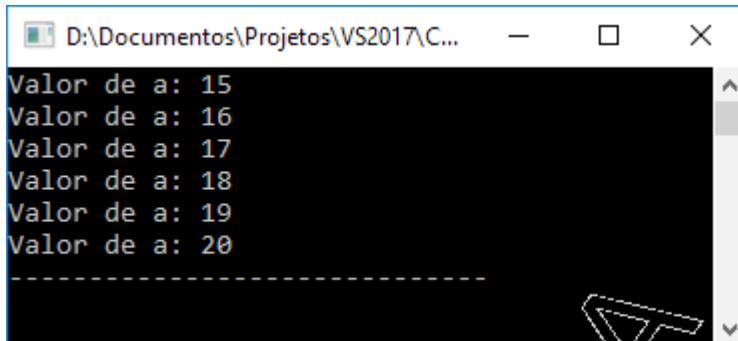
```
while (!EOF)
{
    if (erro)
    {
        continue;
    }
}
```

Quando esse código for executado, será verificado se há um erro. Se a condição for verdadeira, o próximo campo do arquivo será lido até que não seja o final de arquivo (**!EOF**). A instrução **continue** será utilizada para voltar ao loop de repetição e fazer com que os outros campos continuem a ser lidos até o final de arquivo. **EOF** simula uma variável de leitura de um recordset.

O exemplo a seguir mostra outra utilização do comando **continue** com o loop de repetição **for**:

```
static void Main(string[] args)
{
    for (int a = 1; a <= 20; a++)
    {
        if (a == 5 || a == 10) { continue; }
        Console.WriteLine("Valor de a: " + a);
    }
    Console.WriteLine(new string('-', 30));
    Console.ReadKey();
}
```

A compilação e a execução desse código resultarão no seguinte:



```
D:\Documentos\Projetos\VS2017\C...\
```

```
Valor de a: 15
Valor de a: 16
Valor de a: 17
Valor de a: 18
Valor de a: 19
Valor de a: 20
```

Repare que os valores 5 e 10 não foram impressos na lista.

## 2.3.4. Estrutura foreach

Este loop é utilizado para percorrer listas (arrays ou coleções). Ao ser executado, seu iterador percorrerá todos os elementos da lista, permitindo que possamos interagir com suas propriedades e métodos.

 Os assuntos array e coleções serão abordados com detalhes em outro capítulo.

A declaração do loop **foreach** é a seguinte:

```
foreach(declaração da variável dentro da lista)
{
    Bloco de instruções;
}
```

O exemplo a seguir mostra o loop de repetição **foreach**:

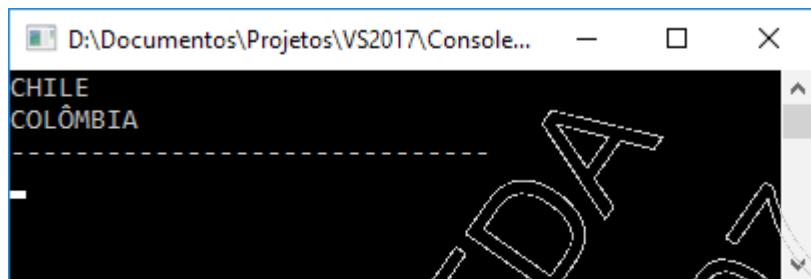
```
static void Main(string[] args)
{
    string[] lista = {"Brasil", "Alemanha", "México",
                      "Chile", "Holanda", "Colômbia",
                      "Argentina", "Uruguai" };

    foreach (string pais in lista)
    {
        //Países que começam com a letra C
        if (pais.ToUpper().StartsWith("C"))
        {
            Console.WriteLine(pais.ToUpper());
        }
    }
    Console.WriteLine(new string(' - ', 30));

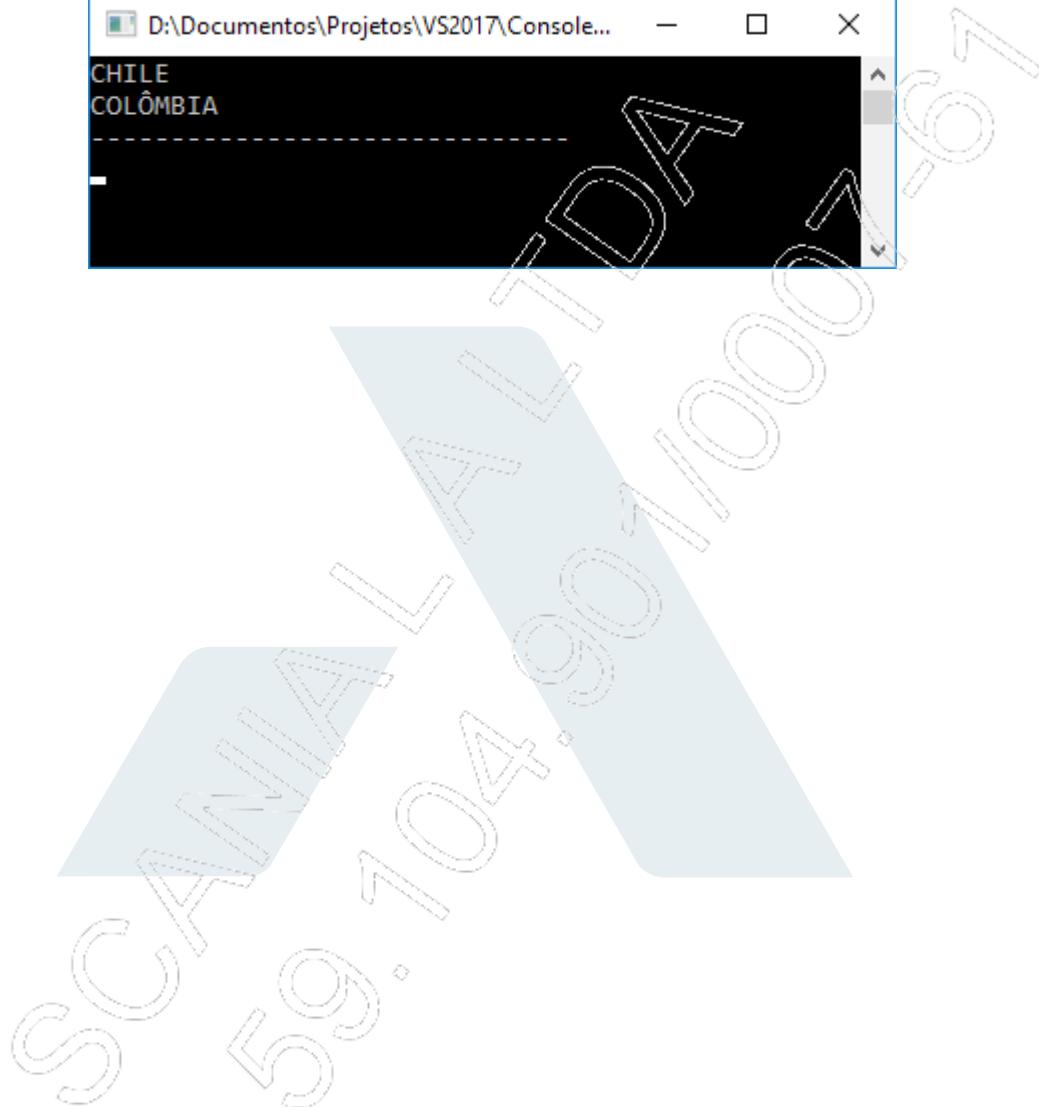
    Console.ReadKey();
}
```

O iterador deve ser do mesmo tipo da lista que será percorrida (no caso do exemplo, `string`).

A compilação e a execução desse código resultarão no seguinte:



```
D:\Documentos\Projetos\VS2017\Console...
CHILE
COLÔMBIA
```



## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Para decidirmos qual ação deverá ser tomada em determinada parte de um programa, solicitamos que este avalie uma expressão. Caso essa expressão seja avaliada como verdadeira, uma sequência de comandos será executada;
- Quando utilizamos uma instrução **if**, apenas expressões com resultados booleanos podem ser avaliadas. E, se desejamos executar comandos nos casos em que a condição é avaliada como falsa, é necessário digitar a cláusula **else** após as instruções da condição verdadeira e, na linha seguinte, inserir o bloco de instruções a serem executadas;
- A instrução **switch** é um modo de simular a utilização de várias instruções **if** e pode somente verificar uma relação de igualdade. É indicada para avaliar uma única expressão para mais de um valor possível, deixando o programa mais legível e tornando sua execução mais eficiente;
- Desde a versão 7 do C#, é permitido o uso de classes genéricas no comando **switch**;
- A versão 8 do C# adicionou o conceito de switch expressions, visando simplificar as estruturas **switch**;
- Quando não sabemos quantas vezes um determinado bloco de instruções precisa ser repetido, utilizamos o loop de repetição **while**. Com ele, a execução das instruções continuará enquanto uma determinada condição for verdadeira;
- A instrução **do / while** tem basicamente o mesmo funcionamento do **while**, porém, executa todos os comandos ao menos uma única vez, mesmo quando a condição não é verdadeira;
- Podemos utilizar o loop de repetição **for** quando sabemos exatamente quantas vezes um bloco de instruções deve ser repetido ou até que um limite preestabelecido seja alcançado;
- Utilizamos o loop de repetição **foreach** quando queremos percorrer os elementos de uma lista e interagir com suas propriedades e métodos;
- Utilizamos **break** para interromper um bloco de instruções antes que ele seja completado;
- Utilizamos **continue** para voltar à condição do loop de repetição ao qual ele se aplica.

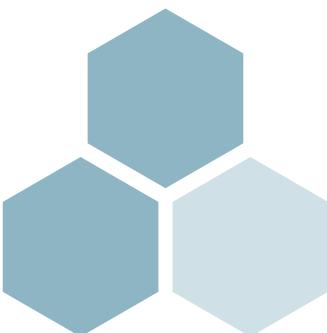




2

# Estruturas de Controle da Linguagem C#

Teste seus conhecimentos



Editora  
**IMPACTA**



**1. Qual a alternativa que completa o código de modo que entre no bloco if somente quando o funcionário ganhar menos que 1000 e tiver código de cargo igual a 3?**

```
double salarioFuncionario = double.Parse(Console.  
ReadLine());  
double codDepFuncionario = double.Parse(Console.  
ReadLine());  
if (_____)  
{  
    salarioFuncionario *= 1.1;  
}
```

- a) salarioFuncionario > 1000 && codDepFuncionario == 3
- b) salarioFuncionario < 1000 && codDepFuncionario = 3
- c) salarioFuncionario < 1000 || codDepFuncionario = 3
- d) salarioFuncionario < 1000 && codDepFuncionario == 3
- e) salarioFuncionario < 1000 || codDepFuncionario == 3

**2. No código a seguir, se o conteúdo da variável salarioFuncionario for 3000, que valor será mostrado no final do código?**

```
double salarioFuncionario = double.Parse(Console.  
ReadLine());  
if (salarioFuncionario <= 1000) salarioFuncionario *=  
1.5;  
else if (salarioFuncionario <= 2000) salarioFuncionario  
*= 1.4;  
else if (salarioFuncionario <= 5000) salarioFuncionario  
*= 1.2;  
else salarioFuncionario *= 1.1;  
Console.WriteLine(salarioFuncionario.ToString());
```

- a) 4500
- b) 4200
- c) 3600
- d) 3300
- e) O código provoca erro de compilação.

**3. Qual a alternativa que contém o texto de label1 no final do processo contido no trecho de código a seguir?**

```
int n = 0;
string texto = "";
while (n <= 10)
{
    texto += n.ToString("00");
    n += 2;
}
Console.WriteLine(texto);
```

- a) Ocorre um loop infinito.
- b) 00020406081012
- c) 020406081012
- d) 000204060810
- e) 0246810

**4. Qual das instruções de repetição adiante será executada enquanto a condição for falsa?**

- a) !while
- b) do / while
- c) for
- d) O C# não possui instrução de repetição executada enquanto a condição for falsa.
- e) Todas as instruções de repetição do C# serão executadas enquanto a condição for falsa.

**5. Qual a alternativa que contém o texto de label1 no final do processo contido no trecho de código a seguir?**

```
string texto = "";
for (int n = 0; n < 10; n += 2)
{
    texto += n.ToString("00");
}

Console.WriteLine(texto);
```

- a) Ocorre um loop infinito.
- b) 0002040608
- c) 020406081012
- d) 00020406081012
- e) 0246810



2

# Estruturas de Controle da Linguagem C#



Mãos à obra!



Editora  
**IMPACTA**



**Observação:** As telas com a criação dos projetos serão omitidas, uma vez que já fazem parte do roteiro de criação de projetos. Quando um novo recurso for apresentado, as telas serão mostradas.

## Laboratório 1

### A – Escrevendo um programa que recebe dados de entrada e retorna um valor

#### Objetivo:

Em um certo clube, o valor da entrada é cobrado de acordo com a idade do convidado. Os valores são os seguintes:

- Pessoas até 17 anos pagam R\$ 30,00;
- Pessoas acima de 17 e até 59 anos pagam R\$ 40,00;
- A partir de 60 anos pagam R\$ 20,00.

Escrever um programa que receba o nome e a idade como valores fornecidos como dado de entrada. Com base nestas informações, apresentar o valor a ser pago como entrada para o clube.

1. Crie um projeto **Console App** chamado **Lab01**, em uma solução (novo solution) com o nome **Capítulo02.Labs**;

2. No método **Main**, escreva uma instrução para solicitar o nome e a idade:

```
namespace Lab01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe seu nome: ");
            string nome = Console.ReadLine();

            Console.Write("Informe sua idade: ");
            int idade = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

3. Declare uma variável para armazenar o valor do ingresso. Usando o comando **if**, atribua o valor para esta variável em função da idade, conforme descrito no objetivo deste laboratório:

```
namespace Lab01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe seu nome: ");
            string nome = Console.ReadLine();

            Console.Write("Informe sua idade: ");
            int idade = Convert.ToInt32(Console.ReadLine());

            double ingresso;
            if(idade <= 17)
            {
                ingresso = 30;
            }
            else if(idade > 17 && idade <= 59)
            {
                ingresso = 40;
            }
            else
            {
                ingresso = 20;
            }

        }
    }
}
```

4. Apresente o resultado:

```
namespace Lab01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe seu nome: ");
            string nome = Console.ReadLine();

            Console.Write("Informe sua idade: ");
            int idade = Convert.ToInt32(Console.ReadLine());

            double ingresso;
            if(idade <= 17)
            {
                ingresso = 30;
            }
            else if(idade > 17 && idade <= 59)
            {
                ingresso = 40;
            }
            else
            {
                ingresso = 20;
            }

            string resposta = $"O convidado {nome} tem {idade} " +
                $"anos, e pagará {ingresso:c}";
            Console.WriteLine(resposta);

            Console.ReadKey();
        }
    }
}
```

## Estruturas de Controle da Linguagem C#

5. Execute o programa:

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\Lab0...
Informe seu nome: Margarida
Informe sua idade: 15
O convidado Margarida tem 15 anos, e pagará R$30,00
```

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\Lab0...
Informe seu nome: Danusa
Informe sua idade: 30
O convidado Danusa tem 30 anos, e pagará R$40,00
```

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\Lab0...
Informe seu nome: Amarildo
Informe sua idade: 65
O convidado Amarildo tem 65 anos, e pagará R$20,00
```

## Laboratório 2

### A – Gerando uma senha numérica

#### Objetivo:

É bastante comum a necessidade de uma aplicação gerar uma senha numérica, seja para uma conta bancária, ou como segunda etapa na verificação de uma conta do Facebook, por exemplo.

O propósito deste programa é solicitar ao usuário o número de dígitos que a senha deverá ter. Com base nesse número, o programa gera os dígitos de forma aleatória, entre 0 e 9, e os acumula em uma variável. As regras a serem seguidas são:

- A senha deve ter entre 4 e 10 dígitos;
- A quantidade de dígitos deve ser par.

Se alguma dessas regras falhar, o programa deve apresentar uma mensagem de erro. Caso contrário, apresentar a senha gerada.

1. Na solução **Capítulo02.Labs**, adicione um novo projeto **Console App** chamado **Lab02**;

2. No método **Main**, solicite ao usuário a quantidade de dígitos. A verificação das regras está mostrada no código:

```
namespace Lab02
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Informe a quantidade de dígitos da senha: ");
            int digitos = Convert.ToInt32(Console.ReadLine());

            if(digitos < 4 || digitos > 10 || digitos % 2 != 0)
            {
                Console.WriteLine($"O valor {digitos} é " +
                    $"inválido de acordo com as regras.");
                Console.ReadKey();
                return;
            }
        }
    }
}
```

3. Usando a estrutura de repetição **for**, determine os dígitos da senha, acumulando-os em uma variável do tipo **string**:

```
namespace Lab02
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Informe a quantidade de dígitos da senha: ");
            int digitos = Convert.ToInt32(Console.ReadLine());

            if(digitos < 4 || digitos > 10 || digitos % 2 != 0)
            {
                Console.WriteLine($"O valor {digitos} é " +
                    $"inválido de acordo com as regras.");
                Console.ReadKey();
                return;
            }

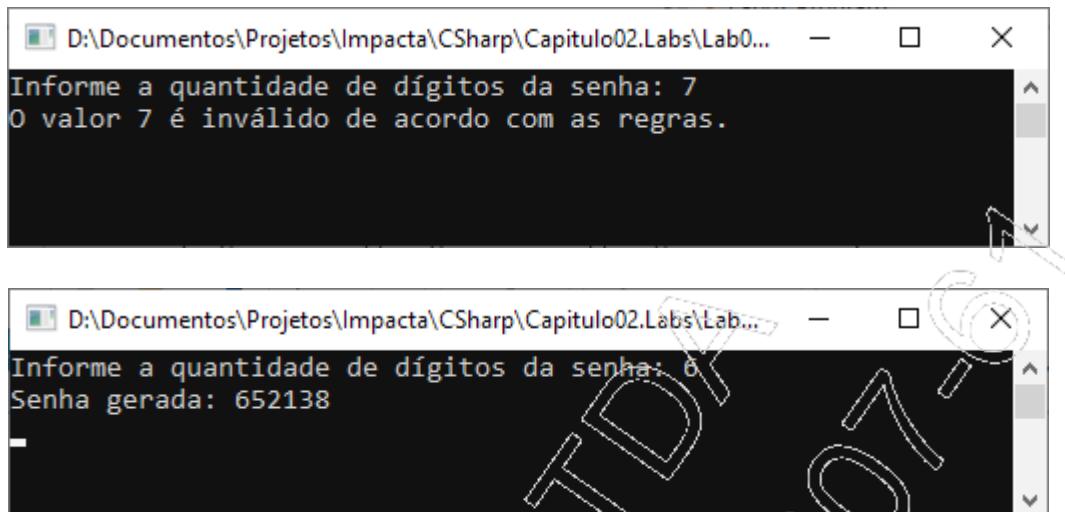
            //variável a ser usada como senha
            string senha = "";

            //objeto a partir do qual os valores aleatórios serão gerados
            var rnd = new Random();

            //estrutura for
            for (int i = 0; i < digitos; i++)
            {
                //valor aleatório entre 0 e 9
                int numero = rnd.Next(0, 9);

                //acumulo do valor em senha
                senha += numero;
            }
            Console.WriteLine($"Senha gerada: {senha}");
            Console.ReadKey();
        }
    }
}
```

4. Teste o programa, tornando este projeto como StartUp:



```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\Lab0... Informe a quantidade de dígitos da senha: 7 O valor 7 é inválido de acordo com as regras.

D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\Lab0... Informe a quantidade de dígitos da senha: 6 Senha gerada: 652138
```

## Laboratório 3

### A – Criando um jogo de adivinhar números

#### Objetivo:

Neste laboratório, faremos um jogo de adivinhar números. O jogo funciona assim:

- O programa produz um valor aleatório inteiro entre 0 e 100;
- Na primeira vez, o programa pede para o usuário fornecer um valor entre 0 e 100;
- Se for fornecido um valor abaixo do valor gerado, o programa deve pedir para fornecer um valor na faixa entre o valor gerado e 100;
- Se for fornecido um valor acima do valor gerado, o programa deve pedir para fornecer um valor na faixa entre 0 e o valor gerado;
- Nas execuções seguintes, o valor mínimo e máximo fornecidos devem levar em conta os dados fornecidos;
- A cada iteração, os valores máximo e mínimo devem ser atualizados;
- Se for fornecido um valor fora da faixa, o programa não deve atualizar os valores máximo e mínimo;
- A cada iteração, a quantidade de tentativas deve ser registrada;
- No final, quando o usuário acertar, o programa deve apresentar o número de tentativas.

1. Na solução **Capitulo02.Labs**, adicione um novo projeto **Console App** chamado **Lab03**;

2. No método **Main**, escreva as instruções para gerar o número aleatório e as variáveis representando os valores máximo e mínimo iniciais, além da variável para armazenar o número de tentativas:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            int maximo = 100, minimo = 0, tentativas = 0;
            int valorGerado = new Random().Next(minimo, maximo);
        }
    }
}
```

3. Em uma estrutura de repetição **while**, peça para o usuário fornecer um valor entre o máximo e mínimo, registrando o número de tentativas:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            int maximo = 100, minimo = 0, tentativas = 0;
            int valorGerado = new Random().Next(minimo, maximo);

            while (true)
            {
                //toda iteração registra uma tentativa
                tentativas++;

                Console.WriteLine($"Informar um valor entre {minimo} e
{maximo}: ");
                int valor = int.Parse(Console.ReadLine());
            }
        }
    }
}
```

4. Verifique se os valores estão fora da faixa, dando continuidade com o comando `continue`:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            int maximo = 100, minimo = 0, tentativas = 0;
            int valorGerado = new Random().Next(minimo, maximo);

            while (true)
            {
                //toda iteração registra uma tentativa
                tentativas++;

                Console.Write($"Informar um valor entre {minimo} e
{maximo}: ");
                int valor = int.Parse(Console.ReadLine());

                if(valor < minimo || valor > maximo)
                {
                    continue;
                }

            }
        }
    }
}
```

5. Se o valor for menor que o valor gerado, atualize a variável **minimo** adequadamente. Se o valor for maior que o valor gerado, atualize a variável **maximo** adequadamente. Se for igual, interrompa a estrutura de repetição:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            int maximo = 100, minimo = 0, tentativas = 0;
            int valorGerado = new Random().Next(minimo, maximo);

            while (true)
            {
                //toda iteração registra uma tentativa
                tentativas++;

                Console.WriteLine($"Informar um valor entre {minimo} e
{maximo}: ");
                int valor = int.Parse(Console.ReadLine());

                if(valor < minimo || valor > maximo)
                {
                    continue;
                }

                if(valor < valorGerado)
                {
                    minimo = valor + 1;
                }
                else if(valor > valorGerado)
                {
                    maximo = valor - 1;
                }
                else
                {
                    break;
                }
            }
        }
    }
}
```

6. Ao acertar, apresente o resultado com a quantidade de tentativas:

```
namespace Lab03
{
    class Program
    {
        static void Main(string[] args)
        {
            int maximo = 100, minimo = 0, tentativas = 0;
            int valorGerado = new Random().Next(minimo, maximo);

            while (true)
            {
                //toda iteração registra uma tentativa
                tentativas++;

                Console.WriteLine($"Informar um valor entre {minimo} e
{maximo}: ");
                int valor = int.Parse(Console.ReadLine());

                if(valor < minimo || valor > maximo)
                {
                    continue;
                }

                if(valor < valorGerado)
                {
                    minimo = valor + 1;
                }
                else if(valor > valorGerado)
                {
                    maximo = valor - 1;
                }
                else
                {
                    break;
                }
            }

            Console.WriteLine($"Você acertou em {tentativas} tentativas.");
            Console.ReadKey();
        }
    }
}
```

## Estruturas de Controle da Linguagem C#

7. Teste seu novo jogo (lembre-se de tornar seu projeto como StartUp):

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\... - X
Informar um valor entre 0 e 100: 50
```

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\... - X
Informar um valor entre 0 e 100: 50
Informar um valor entre 0 e 49: 25
```

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\... - X
Informar um valor entre 0 e 100: 50
Informar um valor entre 0 e 49: 25
Informar um valor entre 0 e 24: 10
```

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\... - X
Informar um valor entre 0 e 100: 50
Informar um valor entre 0 e 49: 25
Informar um valor entre 0 e 24: 10
Informar um valor entre 11 e 24: 18
```

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\... - X
Informar um valor entre 0 e 100: 50
Informar um valor entre 0 e 49: 25
Informar um valor entre 0 e 24: 10
Informar um valor entre 11 e 24: 18
Informar um valor entre 19 e 24: 22
```

```
D:\Documentos\Projetos\Impacta\CSharp\Capitulo02.Labs\... - X
Informar um valor entre 0 e 100: 50
Informar um valor entre 0 e 49: 25
Informar um valor entre 0 e 24: 10
Informar um valor entre 11 e 24: 18
Informar um valor entre 19 e 24: 22
Você acertou em 5 tentativas.
```



# 3

# Orientação a objetos

- Classes;
- Objetos;
- Atributos;
- Modificadores de acesso;
- Namespaces;
- Propriedades;
- A referência this;
- Objetos implícitos e anônimos;
- Enumeradores;
- Estruturas;
- O modificador static;
- Métodos.

## 3.1. Introdução

Essencialmente, todos os elementos de uma aplicação em C# são caracterizados como objetos. Quando falamos em orientação a objetos, estamos nos referindo a um modelo de desenvolvimento em que os componentes se comportam como objetos do mundo real, podendo ser reaproveitados por outras partes da aplicação, ou mesmo por outras aplicações. Mas o conceito não está limitado a esta característica: um objeto é criado a partir de um modelo (um *template*), que é a classe. Na classe, definimos as responsabilidades que seus objetos terão, como valores permitidos para seus atributos, o que é possível fazer com eles, como podemos estendê-los, e assim por diante.

Vamos estudar, neste capítulo, os elementos fundamentais para a elaboração de classes e as consequentes formas de obtenção de objetos.

## 3.2. Classes

Classe é um conceito central na programação orientada a objetos. Com uma classe, representamos uma categoria de elementos do mundo real. Podemos definir, genericamente, uma classe como um modelo que representa um conjunto de elementos (pessoas, objetos, animais etc.) com características comuns.

Quando estamos interessados nos atributos (características) de um elemento, podemos fazer a pergunta:

**Qual a marca do automóvel?**

Analisando a pergunta, não temos uma resposta objetiva, uma vez que não sabemos de qual automóvel especificamente estamos falando. No entanto, sabemos que todos os automóveis possuem uma marca. A definição da marca para um automóvel é realizada em uma classe. Além do atributo marca, é possível especificar o que podemos fazer com o automóvel: estamos falando dos **métodos**.

**! Sendo assim, a classe é uma entidade teórica e não ocupa lugar na memória, é apenas a definição de todas as características que um objeto pertencente a ela pode assumir. A classe é uma abstração de um possível elemento, que pode ainda não existir.**

Além das classes predefinidas no Framework, é usual criarmos nossas próprias classes.

Para definir uma nova classe, usamos a palavra-chave **class**:

```
namespace Conceitos.CSharp.Cap03
{
    class Automovel
    {
    }
}
```

### 3.3. Objetos

Retomando a pergunta realizada anteriormente:

## **Qual a marca do automóvel?**

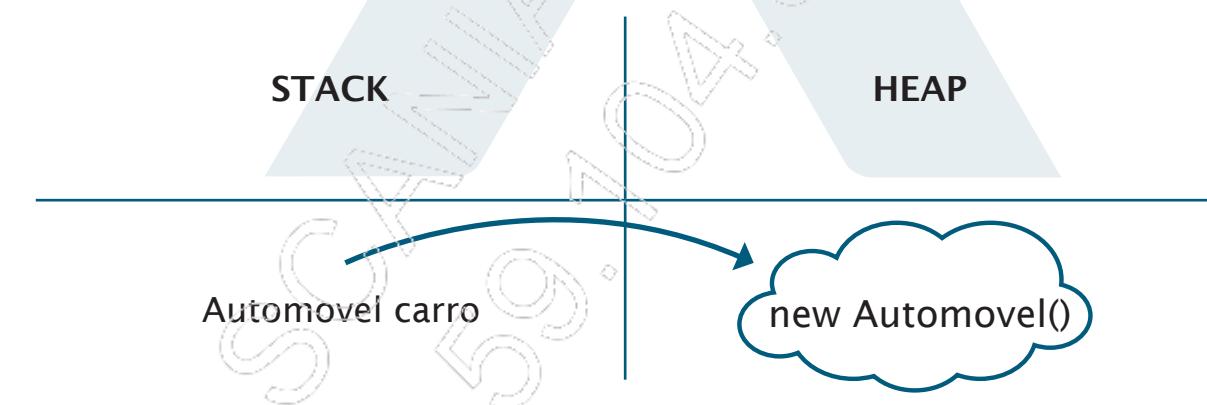
Sabemos que essa pergunta não possui resposta, mas podemos mudá-la para:

**Qual a marca do automóvel estacionado na esquina?**

Nesse caso, estamos nos referindo a um automóvel em particular. Em outras palavras, estamos nos referindo ao **objeto**.

O objeto é uma **instância** da classe. Levando em conta a classe **Automovel** definida anteriormente, sua instanciação (criação do objeto) é realizada da seguinte forma:

```
Automovel carro = new Automovel();
```



A variável **carro** é a **referência** ao objeto e, como toda variável, é definida na memória **STACK**. O objeto, na realidade, não possui nome, mas por ser referenciado pela variável **carro**, dizemos simplificadamente que se trata do objeto carro.

Enquanto a referência por si só não possui valor alocado na memória, o objeto é alocado na memória dinâmica, chamada **HEAP**.



Em outras palavras, quando criamos um objeto a partir de uma determinada classe (instanciação), o objeto criado passa a existir e a ocupar espaço na memória.

### 3.4. Atributos

Os atributos podem ser definidos como características específicas de um objeto. Podemos ampliar um pouco essa definição e compreendê-los também como variáveis ou campos que armazenam valores referentes a diferentes características de um objeto.

Para poder definir e controlar todas as características que os objetos podem ter, a classe as armazena em **propriedades**.

O comportamento de um objeto é a funcionalidade que pode ser aplicada a ele e a maneira como ele responde ao uso. Um método, que corresponde a um procedimento ou função, é o responsável por descrever tal funcionalidade. Embora os métodos sejam correspondentes a procedimentos e funções, eles são capazes de manipular somente os atributos definidos para um objeto.

Para compreendermos classes de maneira adequada, consideraremos a necessidade de criar um objeto do tipo **Automovel**, por exemplo. Para isso, listaremos as características que os automóveis têm em comum:

```
class Automovel
{
    public string Fabricante;
    public string Modelo;
    public string Placa;
    public string Cor;
    public short Ano;
}
```

- **Valores default dos atributos de uma classe**

Podemos determinar os valores de inicialização a qualquer atributo de uma classe. No entanto, caso os atributos de uma classe não possuam valores de inicialização, tais valores serão determinados conforme seu tipo no momento de sua instanciação:

- Se for do tipo **int**, seu valor será **0** (zero);
- Se for do tipo **boolean**, seu valor será **false**;
- Se for do tipo **string**, seu valor será **null**.

Caso os valores de inicialização desejados sejam estabelecidos aos atributos de uma classe, eles serão os valores utilizados no processo de instanciação do objeto dessa mesma classe, por exemplo:

```
class Automovel
{
    //Inicializa com Toyota
    public string Fabricante = "Toyota";
    //Inicializa com Corolla
    public string Modelo = "Corolla";
    //Inicializa com null
    public string Placa;
    //Inicializa com null
    public string Cor;
    //Inicializa com zero
    public short Ano;
}
```

- **Instanciação e atribuição de valores para os atributos do objeto**

Considere a classe **Automovel** novamente:

```
class Automovel
{
    public string Fabricante;
    public string Modelo;
    public string Placa;
    public string Cor;
    public short Ano;
}
```

Sua utilização em uma aplicação seria como no exemplo a seguir:

```
static void Main(string[] args)
{
    //Declarar e instanciar um objeto do tipo Automovel
    Automovel carro = new Automovel();

    //Definir valor aos seus campos/atributos
    carro.Fabricante = "Toyota";
    carro.Modelo = "Corolla";
    carro.Placa = "ppp0808";
    carro.Cor = "Preto";
    carro.Anو = 2013;
```

```
//Exibir os dados na listbox
Console.WriteLine("Fabricante: " + carro.Fabricante);
Console.WriteLine("Modelo: " + carro.Modelo);
Console.WriteLine("Placa: " + carro.Placa);
Console.WriteLine("Cor: " + carro.Cor);
Console.WriteLine("Ano: " + carro.Ano);
Console.WriteLine(new string('-', 30));

Console.ReadKey();
}
```

O resultado é exibido a seguir:



É importante lembrar que, apesar de termos declarado e instanciado uma classe em uma só operação, ela, na verdade, acontece em duas etapas: a declaração na memória **stack** e a instanciação na memória **heap**. Veja:

- **Declaração e instanciação em uma etapa**

```
//Declarar e instanciar um objeto do tipo Automovel
Automovel carro = new Automovel();
```

- **Execução em duas etapas**

```
//Declarar um objeto do tipo Automovel
//Apenas o tipo de dado é escolhido
Automovel carro;

//Instanciar o objeto do tipo Automovel
//Efetivamente o objeto é construído em memória
carro = new Automovel();
```

### 3.5. Modificadores de acesso

Os modificadores de acesso definem a visibilidade de um campo, propriedade, evento e método de uma classe. A seguir, temos uma descrição dos tipos de modificadores de acesso que a linguagem C# oferece:

- **public**

O acesso é livre quando utilizamos esse modificador.

- **private**

Quando utilizamos esse modificador, apenas o tipo que contém o modificador tem o acesso. Por padrão, **private** é a visibilidade definida para métodos e campos da própria classe.

- **protected**

Ao utilizarmos esse modificador, apenas a classe que contém o modificador e os tipos derivados dessa classe possuem acesso.

- **internal**

Esse modificador limita o acesso apenas ao assembly atual (compilação).

- **protected internal**

Quando utilizamos esse modificador, o acesso é limitado ao assembly atual e aos tipos derivados da classe que contém o modificador.

Para que o mecanismo do encapsulamento seja possível, é fundamental que o desenvolvedor tenha a possibilidade de restringir ou garantir o acesso a determinados objetos e seus membros (atributos e métodos) para outros objetos. Isso é possível por meio da utilização de modificadores ou qualificadores de acesso.

Quanto aos nomes que atribuímos aos métodos e campos, consideremos as seguintes convenções:

- Identificadores privados são escritos com a primeira letra minúscula, ou seja, seguindo o sistema conhecido como **camelCase**;
- Identificadores públicos são escritos com a primeira letra maiúscula, ou seja, seguindo o esquema conhecido como **PascalCase**.

É importante considerar, também, as seguintes exceções:

- Nomes de classes devem ter a primeira letra maiúscula;
- Construtores devem coincidir com o nome de sua classe;
- Construtores **private** devem usar letra maiúscula.

Devemos estar conscientes de que declarar dois membros de classe diferindo-os somente por letras maiúsculas e minúsculas é uma prática altamente desaconselhável, pois isso impede o uso da classe em outras linguagens que não são case-sensitive (que não diferem maiúsculas de minúsculas).

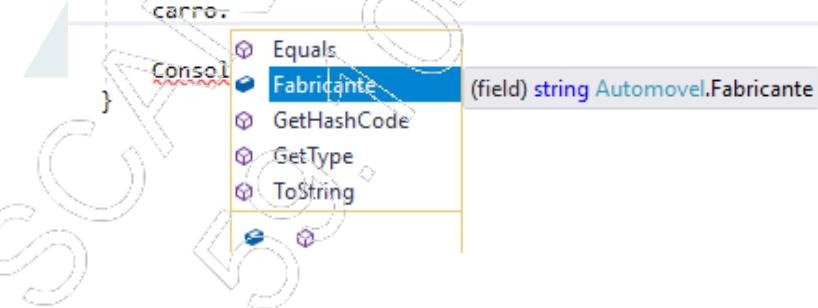
Veja um exemplo na classe Automovel:

```
class Automovel
{
    //Fabricante será visível pelo objeto
    public string Fabricante;

    //Modelo não será visível pelo objeto
    private string Modelo;
}
```

Na aplicação que faz uso da classe, o campo **Modelo** não estará disponível:

```
static void Main(string[] args)
{
    //Declarar e instanciar um objeto do tipo Automovel
    Automovel carro = new Automovel();
```



## 3.6. Namespaces

Conforme uma aplicação cresce, começamos a nos deparar com os seguintes problemas:

- Quanto maior a quantidade de código, mais difícil sua compreensão;
- Com uma grande quantidade de código, costumamos trabalhar com mais nomes (classes, métodos e dados nomeados). Isso pode acabar provocando um conflito entre dois ou mais nomes, e, consequentemente, gerar uma falha de desenvolvimento. Tal conflito é mais provável quando o programa utiliza bibliotecas de terceiros.

A utilização de namespaces em conjunto com as classes é o procedimento recomendado.

O Framework possui diversas classes distribuídas em diferentes namespaces. Assim também podemos fazer com nossas classes.

Como ilustração, temos uma tabela com os principais namespaces utilizados em C#:

Namespace	Descrição
<b>System</b>	Este é o namespace base do .NET. Nele estão contidas classes fundamentais e classes base empregadas para definir valores e para referenciar tipos de dados, interfaces, atributos, manipuladores de evento e exceções.
<b>System.Data</b>	Neste namespace estão contidas as classes constituintes da estrutura ADO.NET.
<b>System.Diagnostics</b>	Neste namespace estão contidas as classes que empregamos para corrigir erros das aplicações e traçar a execução do código.
<b>System.Drawing</b>	Neste namespace estão contidas as classes que empregamos para acessar a funcionalidade dos gráficos básicos do Graphics Device Interface (GDI+).
<b>System.IO</b>	Neste namespace estão contidas as classes que empregamos para ler a partir de streams de dados e arquivos, bem como para escrever neles.
<b>System.Net</b>	Neste namespace estão as classes que disponibilizam uma interface de programação simplificada para diversos protocolos da rede.
<b>System.Security</b>	Neste namespace estão contidas as classes que empregamos como estrutura básica para o sistema de segurança do CLR.
<b>System.Web</b>	Neste namespace estão contidas as classes nas quais estão disponíveis as interfaces que empregamos para estabelecer comunicação com o navegador ou servidor.
<b>System.Windows.Forms</b>	Neste namespace estão contidas as classes que empregamos para criar aplicações que podem aproveitar a interface de usuário do Windows.
<b>System.Xml</b>	Neste namespace estão contidas as classes que oferecem suporte baseado em padrões para o processamento de XML.

- **Uso de namespaces e assemblies**

A instrução **using** pode ser usada para que os itens de um namespace sejam colocados em escopo sem que, no entanto, seja necessário qualificar, no código, os nomes das classes. Isso é possível devido à compilação das classes em assemblies.

Para usar o namespace sem a instrução **using** e, assim, evitar possíveis conflitos de nomes de classes (como já mencionado), podemos proceder da seguinte forma:

```
namespace Conceitos.CSharp.Cap03
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Uso do nome qualificado da classe");
        }
    }
}
```

Observe que a classe **Console** foi usada em conjunto com seu namespace, **System**. Dessa forma, não foi necessário importar esse namespace com a instrução **using**.

No processo de compilação, diversas classes podem fazer parte do mesmo programa, ou seja do mesmo **assembly**.

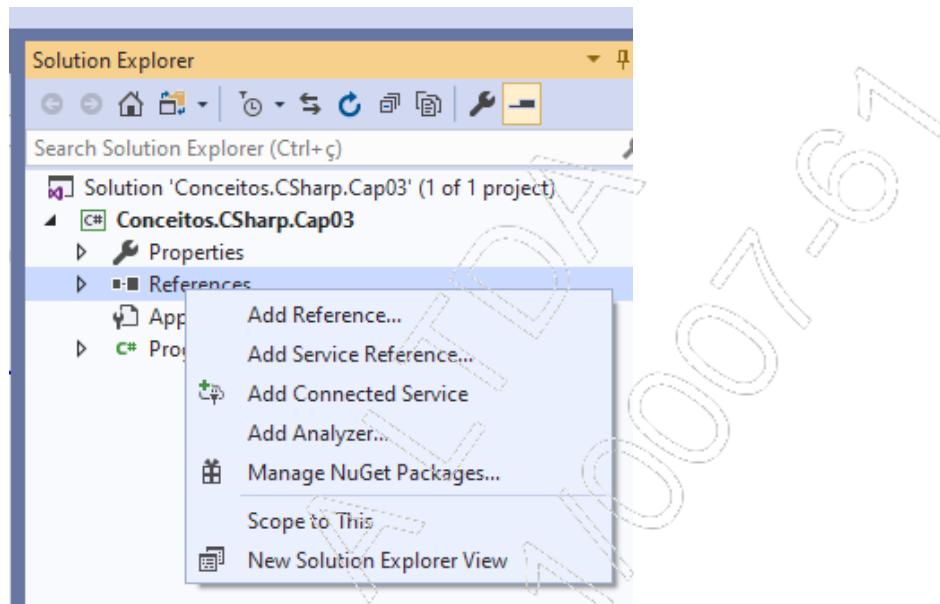
**Assemblies** são blocos de código de aplicativos CLR (Common Language Runtime), reutilizáveis, autodescritíveis e passíveis de versão.

Um único assembly pode conter várias classes e, normalmente, os assemblies são agrupados de acordo com a funcionalidade das classes que estiverem armazenando, como classes para acesso a Web services, para manipulação de bancos de dados, entre outras.

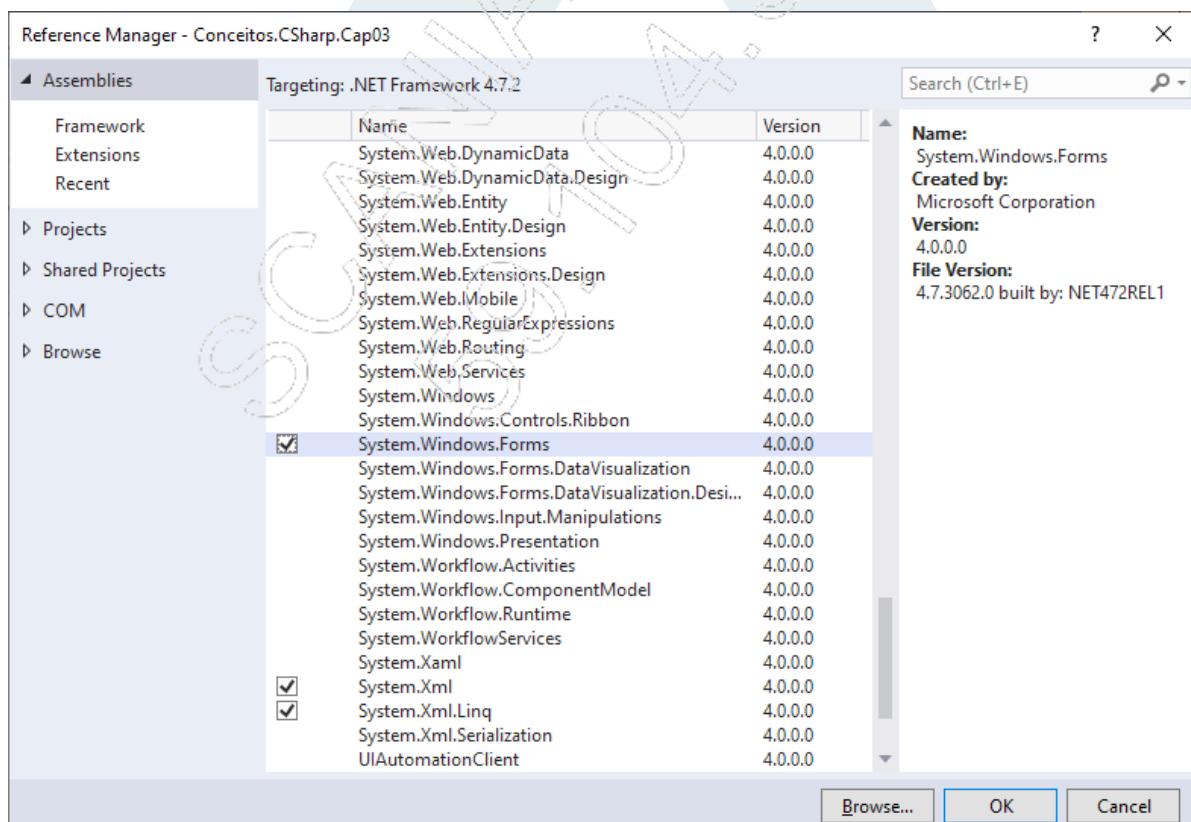
Para utilizar uma classe em um assembly, é necessário adicionar uma referência a ele no projeto em que pretendemos utilizá-la. Depois, basta usar instruções **using** no código para que sejam colocados em escopo os itens do namespace nesse assembly.

Assim como é possível que um único assembly abranja classes para diversos namespaces, um único namespace também pode abranger diversos assemblies.

Quando criamos um aplicativo com o Visual Studio, não precisamos nos preocupar com referências a assemblies, uma vez que o template selecionado incluirá referências aos assemblies apropriados. No entanto, referências a assemblies adicionais devem ser inseridas por nós, bastando, para tanto, clicar com o botão direito do mouse sobre a pasta **References** e, em seguida, clicar sobre a opção **Add Reference**:



A seguir, temos um exemplo de utilização de **using**, em que a classe **MessageBox**, que pertence ao namespace **System.Windows.Forms**, está sendo utilizada:



# Programando com C#

Nosso objetivo é usar a classe **MessageBox**. Um recurso para permitir que o Visual Studio importe o namespace para nós, ou mesmo permitir a utilização da classe totalmente qualificada, é usar as teclas **CTRL + .**:

```
1  namespace Conceitos.CSharp.Cap03
2  {
3      class Program
4      {
5          static void Main(string[] args)
6          {
7              System.Console.WriteLine("Uso do nome qualificado da classe");
8              MessageBox.
9
10     using System.Windows.Forms;
11     System.Windows.Forms.MessageBox
12     Generate property 'Program.MessageBox'
13     Generate field 'Program.MessageBox'
14     Generate read-only field 'Program.MessageBox'
15     Generate local 'MessageBox'
16     Generate parameter 'MessageBox'
```

Um cuidado deve ser tomado nesse procedimento se os termos **using System.Windows.Forms** ou **System.Windows.Forms.MessageBox** não aparecerem na lista. Se isso acontecer, significa que a classe não foi escrita corretamente, ou que seu namespace não está disponível, ou ainda, que o escopo da classe não permite acesso. Neste caso, não prossiga e verifique o que pode estar errado.

Seja como for, uma vez que o namespace esteja disponível, podemos usá-lo de uma das formas:

```
using System.Windows.Forms;

namespace Conceitos.CSharp.Cap03
{
    class Program
    {
        static void Main(string[] args)
        {
            MessageBox.Show("Exemplo de Namespace");
            Console.ReadKey();
        }
    }
}
```

Sem o comando **using**, o código deverá utilizar o nome completo da classe, como mostrado a seguir:

```
class Program
{
    static void Main(string[] args)
    {
        System.Windows.Forms.MessageBox.Show("Exemplo de Namespace");
        Console.ReadKey();
    }
}
```

## 3.7. Propriedades

Uma propriedade é um valor que a classe disponibiliza para a leitura, a gravação ou ambas as ações. Sob o ponto de vista lógico, uma propriedade é semelhante a um campo. No entanto, essa sintaxe semelhante à do campo é convertida pelo compilador, automaticamente, em chamadas para assessores especiais semelhantes aos dos métodos.

A seguir, observe o formato geral de uma declaração de propriedade:

```
ModificadorDeAcesso Tipo NomeDaPropriedade
{
    get
    {
        bloco de código de leitura
    }
    set
    {
        bloco de código de gravação
    }
}
```

Depois de observar o formato anterior, podemos constatar que uma propriedade pode conter dois blocos de código, que iniciam com as palavras-chave **get** e **set**. Descreveremos, adiante, cada um desses blocos:

- **get**: Este bloco contém expressões que são executadas assim que ocorre a leitura da propriedade;
- **set**: Este bloco contém expressões que são executadas assim que a propriedade é escrita.

É importante destacarmos que, do ponto de vista do objeto que vai utilizar uma propriedade, não existe diferença entre uma propriedade ou um campo.

Podemos observar, a seguir, um exemplo na classe **Automovel**:

```
class Automovel
{
    //Campo público Fabricante, será visível pelo objeto
    public string Fabricante;

    //Campo particular
    private string _modelo;

    // Propriedade que encapsula o campo é pública
    public string Modelo
    {
        get { return _modelo; }
        set { _modelo = value; }
    }
}
```

Agora, no formulário que faz uso da classe:

```
static void Main(string[] args)
{
    //Declarar um objeto do tipo Automovel
    Automovel carro = new Automovel();

    //Definir valor das propriedades
    carro.Fabricante = "Toyota";
    carro.Modelo = "Corolla";

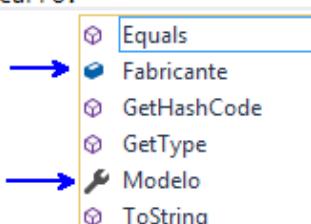
    //Exibir os dados na listBox
    Console.WriteLine("Fabricante: " + carro.Fabricante);
    Console.WriteLine("Modelo: " + carro.Modelo);
    Console.WriteLine(new string('-', 30));

    Console.ReadKey();
}
```

A seguir, veja, no código que faz uso da classe, uma diferença visual entre campo público e propriedade:

```
//Declarar um objeto do tipo Automovel
Automovel carro = new Automovel();
```

```
//Definir valor das propriedades
carro.
```



Vale ressaltar que o tipo de dado lido e escrito pelos assessores **get** e **set** são determinados pelo tipo da propriedade. Além disso, é importante saber que os campos e as propriedades seguem a convenção de nomeação padrão Visual C# (**public** e **private**).

Toda informação pública da classe deve ser exposta em forma de propriedades, não de campos, e com letras maiúsculas. Por outro lado, os campos privados são indicados com letras minúsculas.

**Para criarmos propriedades somente leitura, basta declará-las apenas com o método **get**.**

- **Restrições**

As principais restrições das propriedades estão listadas a seguir:

- Não pode existir nenhum parâmetro para os métodos **get** e **set**;
- Não podemos declarar propriedades **const**;
- As propriedades não podem ser utilizadas como um argumento **ref** ou **out** para um método;
- Não pode existir mais que um método **get** e um método **set** em uma mesma propriedade, nem outros métodos, campos ou propriedades;
- Os valores só podem ser atribuídos por meio de uma propriedade de estrutura ou de classe se a estrutura ou classe tiver sido inicializada.

**Sabendo da existência das propriedades, a partir de agora não mais usaremos variáveis (atributos) com elementos expostos para o programador.**

### 3.7.1. Encapsulamento

O encapsulamento é um mecanismo que permite ocultar os detalhes da implementação interna de uma classe, restringindo o acesso a suas variáveis e métodos. Permite que você utilize uma classe sem conhecer seu mecanismo interno de implementação. Isso otimiza a legibilidade do código, ajudando a minimizar os erros de programação, além de facilitar a ampliação do código com novas atualizações, já que o proprietário de uma classe pode modificá-la internamente sem que o usuário precise saber disso.

Com o uso de propriedades, o encapsulamento fica assegurado. Para exemplificar, considere uma classe **Funcionario**, com as propriedades: **Nome**, **Cargo**, **Salario** e **Irpf**. Por questões didáticas, vamos considerar que o valor da propriedade **Irpf** seja 10% do salário. Nesse caso, não é conveniente expor essa propriedade para o usuário da classe, deixando que seu valor seja determinado no momento da determinação do valor do salário:

```
namespace Conceitos.CSharp.Cap03
{
    public class Funcionario
    {
        private string _nome;

        public string Nome
        {
            get { return _nome; }
            set { _nome = value; }
        }

        private double _salario;

        public double Salario
        {
            get { return _salario; }
            set { _salario = value; }
        }

        public double Irpf
        {
            get { return Salario * 0.1; }
        }
    }
}
```

Graças ao encapsulamento, pudemos vincular a propriedade **Irpf** à propriedade **Salario**, sem o risco de termos valores incoerentes.

### 3.7.2. Propriedades como membros aptos para expressões

A partir do C# 7.0, as propriedades receberam algumas simplificações significativas. As propriedades foram reduzidas a expressões.

```
namespace Conceitos.CSharp.Cap03
{
    public class Funcionario
    {
        private string _nome;

        public string Nome
        {
            get => _nome;
            set => _nome = value;
        }

        private double _salario;

        public double Salario
        {
            get => _salario;
            set => _salario = value;
        }

        public double Irpf
        {
            get => Salario * 0.1;
        }
    }
}
```

### 3.7.3. Propriedades implícitas

Se não houver nenhuma necessidade de validação para uma propriedade em particular, podemos defini-la como implícita. Nesse caso, não há necessidade de definir atributos `private`.

Como exemplo, vamos alterar a classe `Funcionario`:

```
namespace Conceitos.CSharp.Cap03
{
    public class Funcionario
    {
        public string Nome { get; set; }
        public double Salario { get; set; }

        public double Irpf => Salario * 0.1;
    }
}
```

Observe que no caso da propriedade `Irpf`, por ser somente leitura, nem foi necessário o uso do assessor `get`.

## 3.8.A referência this

Quando criamos um objeto, definimos uma variável para referenciá-lo. No caso da classe em si, quando os membros são acessados dentro da própria classe, é através de uma referência, chamada `this`. Essa referência é útil quando necessitamos mencionar os atributos presentes na própria classe, e não através de outras referências, mesmo quando passadas como parâmetro.

Veja o exemplo adiante:

```
public class Funcionario
{
    public string Nome { get; set; }
    public double Salario { get; set; }

    public double Irpf => Salario * 0.1;

    public void Atribuir(string Nome, double Salario)
    {
        this.Nome = Nome;
        this.Salario = Salario;
    }
}
```

Se não fosse pelo uso do `this`, o método atribuiria o valor do parâmetro para ele mesmo, já que os parâmetros e as propriedades da classe possuem o mesmo nome. O uso do `this` permitiu que pudéssemos tornar explícito quem é o parâmetro e quem é a propriedade.

## 3.9. Objetos implícitos e anônimos

Podemos criar instâncias de classes de forma implícita, ao atribuirmos valores para suas propriedades. Considere a seguinte instância da classe `Funcionario`:

```
static void Main(string[] args)
{
    Funcionario funcionario = new Funcionario();
    funcionario.Nome = "Mirosmar";
    funcionario.Salario = 3000;

    //continua...
}
```

Essa mesma instância poderia ter sido criada de **forma implícita**, da seguinte forma:

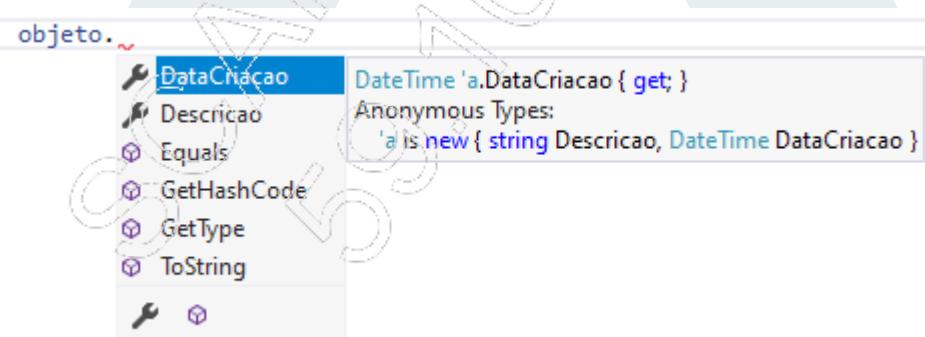
```
static void Main(string[] args)
{
    Funcionario funcionario = new Funcionario()
    {
        Nome = "Mirosmar", Salario = 3000
    };
    //continua...
}
```

A vantagem da forma implícita é que podemos fornecer valores para a instância, independentemente da variável que a referencia. Essa definição ficará mais clara quando estudarmos **Herança e Polimorfismo**.

Objetos anônimos, como o próprio nome indica, são objetos que podem ser criados sem a presença de uma classe. Nesse caso, a variável que o referencia deve ser declarada com **var**.

```
static void Main(string[] args)
{
    var objeto = new
    {
        Descricao = "Objeto Anônimo",
        DataCriacao = DateTime.Now
    };
}
```

As propriedades **Descricao** e **DataCriacao** foram definidas arbitrariamente. Apesar de não pertencerem a classe nenhuma, essas propriedades são acessadas normalmente por meio da variável **objeto**:



No Intellisense da descrição da propriedade, podemos ver a menção ao tipo anônimo.

## 3.10. Enumeradores

A palavra-chave **enum** é usada para declarar um conjunto de constantes. Um enumerador não pode herdar ou ser herdado.

O tipo padrão de um enumerador é um **int**. Por conta disso, os literais de enumeração recebem valores do tipo **int** na sua declaração. Porém, há casos em que não será necessário utilizar toda a faixa de valores disponíveis para um **int**. Uma alternativa é utilizar tipos de dados menores do que o **int**, o que proporciona economia de memória.

Além do **int**, podemos basear uma enumeração nos seguintes tipos inteiros: **short**, **ushort**, **uint**, **long**, **ulong**, **byte** e **sbyte**.

O trecho de código a seguir especifica outro tipo de inteiro primário, **byte**, para o enumerador **EstadosCivis**:

```
enum EstadosCivis : byte
{
    Casado,
    Solteiro,
    Viuvo,
    Separado,
    Outro
}
```

Nesse exemplo, especificamos **byte** como o tipo primário de **EstadosCivis**, sendo que **byte** é um tipo que ocupa menos memória do que o **int**. Os valores de todos os literais de enumeração devem estar compreendidos na faixa de valores do tipo base escolhido (uma enumeração baseada no tipo **byte**, por exemplo, pode ter no máximo 256 literais).

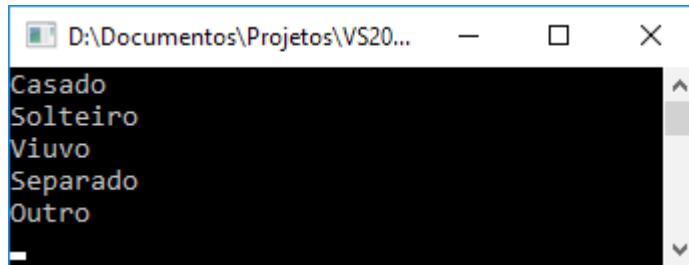
- Iterando valores do enum

Quando estamos trabalhando com o **enum**, podemos iterar seus valores. Para isso, utilizamos loops de repetição com um conjunto de valores **enum**.

Considere o seguinte código:

```
static void Main(string[] args)
{
    for (byte i = 0; i <= 4; i++)
    {
        Console.WriteLine(Enum.GetName(typeof(EstadosCivis), i));
    }
    Console.ReadKey();
}
```

Agora, veja o resultado da iteração:



## 3.11. Estruturas

Algumas vezes, uma classe é usada para armazenar valores para suas propriedades, mas não participam de recursos mais avançados, como herança e polimorfismo. Em casos como esse, podemos economizar a memória heap, melhorando a performance da aplicação.

Para esses dados, podemos usar estruturas em vez de classes, pois são armazenadas na memória stack e não no heap.

Em linguagem C#, os tipos numéricos primitivos **float**, **int** e **long** são apelidos de estruturas. Uma estrutura é um tipo por valor, apesar de poder ter seus próprios métodos, construtores e campos, como uma classe.

É importante dizer que a utilização de diversos operadores comuns em nossos próprios tipos de estrutura, por padrão, não é possível.

Na tabela a seguir, temos a relação dos tipos primitivos utilizados em C#, seus tipos equivalentes e sua definição, entre classe ou estrutura:

Palavra-chave	Tipo equivalente	Classe ou estrutura
object	System.Object	Classe
string	System.String	Classe
bool	System.Boolean	Estrutura
byte	System.Byte	Estrutura
decimal	System.Decimal	Estrutura
double	System.Double	Estrutura
float	System.Single	Estrutura
int	System.Int32	Estrutura
long	System.Int64	Estrutura
sbyte	System.Sbyte	Estrutura
short	System.Int16	Estrutura

Palavra-chave	Tipo equivalente	Classe ou estrutura
uint	System.UInt32	Estrutura
ulong	System.UInt64	Estrutura
ushort	System.UInt16	Estrutura

## 3.11.1. Diferenças entre classes e estruturas

As classes e as estruturas, apesar de semelhantes do ponto de vista sintático, apresentam algumas diferenças. Primeiramente, uma estrutura é um **tipo valor**, ao passo que uma classe é um **tipo referência** (servem para referenciar objetos). Em segundo lugar, a instância de uma estrutura é chamada **value** e fica armazenada na pilha. A instância de uma classe, por sua vez, é chamada **object** e fica armazenada no heap.

Em uma classe, caso não tenhamos criado um construtor próprio, um construtor padrão será criado pelo compilador. Uma estrutura, por sua vez, não aceita a declaração de um construtor padrão (ou seja, um construtor sem parâmetros), pois o próprio compilador sempre cria um construtor. Abordaremos construtores em um capítulo posterior.

Em uma estrutura, não podemos inicializar campos de instância em seu próprio ponto de declaração, ao passo que, nas classes, isso é possível. As estruturas têm como regra inicializar todos os seus campos em todos os seus construtores.

Após ter sido definido, um tipo-estrutura pode ser utilizado como qualquer outro tipo, ou seja, é possível criar variáveis, campos e parâmetros que lhe sejam correspondentes.

Veja um exemplo de estrutura:

```
namespace Conceitos.CSharp.Cap03
{
    struct Endereco
    {
        public string Rua { get; set; }
        public string Bairro { get; set; }
        public string Cidade { get; set; }
        public string UF { get; set; }
        public string CEP { get; set; }

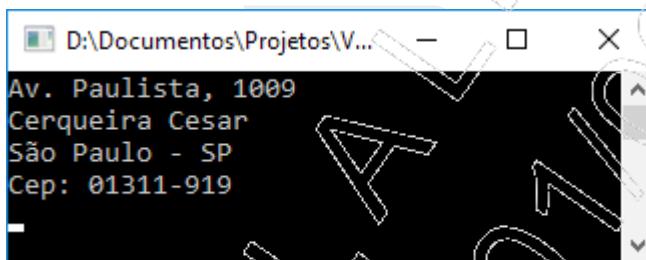
        public string ExibirCompleto()
        {
            return $"{Rua}\n{Bairro}\n{Cidade} - {UF}\nCep: {CEP}";
        }
    }
}
```

Agora, veja o programa que faz uso da estrutura:

```
static void Main(string[] args)
{
    Endereco comercial = new Endereco();
    comercial.Rua = "Av. Paulista, 1009";
    comercial.Bairro = "Cerqueira Cesar";
    comercial.Cidade = "São Paulo";
    comercial.UF = "SP";
    comercial.CEP = "01311-919";
    Console.WriteLine(comercial.ExibirCompleto());

    Console.ReadKey();
}
```

O resultado é o seguinte:



## 3.12. O modificador static

Podemos criar funções e dados que podem ser acessados sem a necessidade de criar uma instância da classe. Para isso, adicionamos a palavra-chave **static** na frente da classe ou membro da classe, gerando, assim, classes estáticas e membros estáticos, os quais descreveremos a seguir.

- **Membros estáticos**

Os membros estáticos são campos, métodos, eventos ou propriedades que podem ser chamados em uma classe mesmo que não tenha sido criada uma instância dessa classe.

Podemos utilizar membros estáticos para os dados compartilhados por toda a aplicação.

Vale destacarmos algumas considerações sobre a utilização de membros estáticos:

- Não podem ser acessados por meio de instâncias criadas para a classe;
- Propriedades e métodos estáticos podem acessar apenas campos estáticos;
- Eventos e campos estáticos possuem apenas uma cópia.

A seguir, veja um exemplo de campo e propriedade estáticos e não estáticos:

- Na classe Automovel

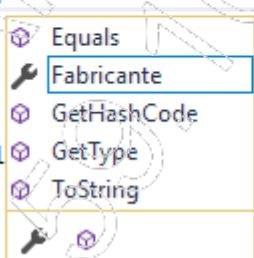
```
class Automovel
{
    //Campo e propriedade não estáticos
    private string _fabricante;
    public string Fabricante
    {
        get { return _fabricante; }
        set { _fabricante = value; }
    }

    //Campo e propriedade estáticos
    private static string _modelo;
    public static string Modelo
    {
        get { return _modelo; }
        set { _modelo = value; }
    }
}
```

- No código que faz uso da classe

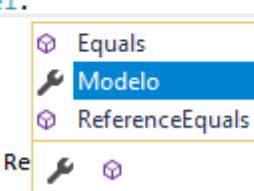
```
static void Main(string[] args)
{
    //Declarar um objeto do tipo Automovel
    Automovel carro = new Automovel();

    //Definir valor das propriedades
    carro.
```



```
static void Main(string[] args)
{
    //Declarar um objeto do tipo Automovel
    Automovel carro = new Automovel();

    //Definir valor das propriedades
    Automovel.
```





Para a criação de um campo estático, podemos usar a palavra-chave **const**. Além de estático, esse campo nunca poderá ter seu valor alterado, uma vez que **const** é a abreviação da palavra constante.

## 3.13. Métodos

Os métodos definem as ações dos objetos. Por meio deles, os objetos podem ter seus atributos modificados, manifestar-se e interagir com outros objetos.

Podemos dizer, também, que um método funciona como a implementação de uma operação para uma classe específica. Os objetos de uma mesma classe compartilham as mesmas operações. Toda operação possui um objeto-alvo e é a classe desse objeto que determina o comportamento da operação.

Todas as instruções em C# são executadas por métodos, assim que o método tenha sido chamado e os argumentos solicitados tenham sido especificados pelo programa.

Em um método, devemos definir os seguintes itens:

- Nome do método;
- Parâmetros recebidos, quando houver;
- Comandos (corpo do método);
- Valor de retorno, quando houver.

A sintaxe de um método é a seguinte:

```
tipo nomeDoMetodo(lista_de_parametros)
{
    corpo do método
}
```

Em que:

- **tipo** refere-se ao tipo de dados válido, retornado pelo método. Esse tipo pode ser qualquer um, incluindo aqueles de classes desenvolvidas pelo programador;
- **nomeDoMetodo** refere-se ao nome do método, que pode ser um identificador legal que não seja utilizado por outros itens do escopo atual;
- **lista\_de\_parametros** refere-se à sequência de pares compostos de um tipo e um identificador. Ambos estão separados por vírgulas.

Em relação aos parâmetros, eles são variáveis, ou seja, recebem o valor dos argumentos que são passados para o método assim que este é chamado. Quando não houver parâmetros para o método, não deve ser inserido conteúdo entre os parênteses ( ).

Quando o método não retornar um valor, ele deve ser declarado como **void**. Mas se o método retornar um valor, este será para a rotina que fez a chamada do método. Para isso, o comando **return** é utilizado, como vemos a seguir:

```
tipo nomeDoMetodo(lista_de_parâmetros)
{
    Corpo do método
    return valor;
}
```

Para declararmos e utilizarmos um método, analisemos o seguinte:

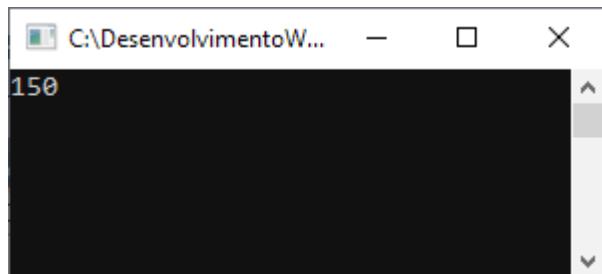
```
namespace Conceitos.CSharp.Cap03
{
    class Program
    {
        //Definição do método
        private static int Multiplicar(int valor1, int valor2)
        {
            //Retorna o resultado da multiplicação
            //entre o valor1 e o valor2
            return valor1 * valor2;
        }

        static void Main(string[] args)
        {
            int n1 = 10;
            int n2 = 15;

            //O retorno do método é atribuído à variável
            int resultado = Multiplicar(n1, n2);

            Console.WriteLine(resultado);
            Console.ReadKey();
        }
    }
}
```

O resultado desse código pode ser visto a seguir:



O método **Multiplicar** possui as seguintes características:

- **private**: É um modificador de acesso que indica que o método só pode ser referenciado dentro da mesma classe em que foi criado;
- **static**: Pelo fato de ser chamado em outro método estático (que é o caso do método **Main()**);
- **int**: Indica que o método retornará um valor inteiro;
- **Multiplicar**: O nome dado ao método;
- **int valor1 e int valor2**: Indicam que o método (**soma**) receberá dois parâmetros do tipo inteiro, que possuem os nomes **valor1** e **valor2**;
- **return valor1 \* valor2**: Indica que o resultado da multiplicação entre os parâmetros **fator1** e **fator2** informados será o valor de retorno do método.

O método **Main()** realiza as seguintes ações:

- Declara uma variável inteira **resultado**;
- Atribui a ela o valor do método **Multiplicar**, com o valor dos parâmetros sendo **n1 = 10** e **n2 = 15**;
- Apresenta o resultado.

No caso da classe **Automovel**, escreveremos o método que exibe todos os seus dados e faremos uma chamada a partir do objeto instanciado no formulário:

- Na classe Automovel

```
class Automovel
{
    private string _fabricante;
    public string Fabricante
    {
        get { return _fabricante; }
        set { _fabricante = value; }
    }

    private string _modelo;
    public string Modelo
    {
        get { return _modelo; }
        set { _modelo = value; }
    }

    //Método ExibirDados()
    public string ExibirDados()
    {
        return "Marca: " + Fabricante +
               "\nModelo: " + Modelo;
    }
}
```

- No código que faz uso da classe

```
static void Main(string[] args)
{
    //Definir o objeto Automovel
    Automovel carro = new Automovel();

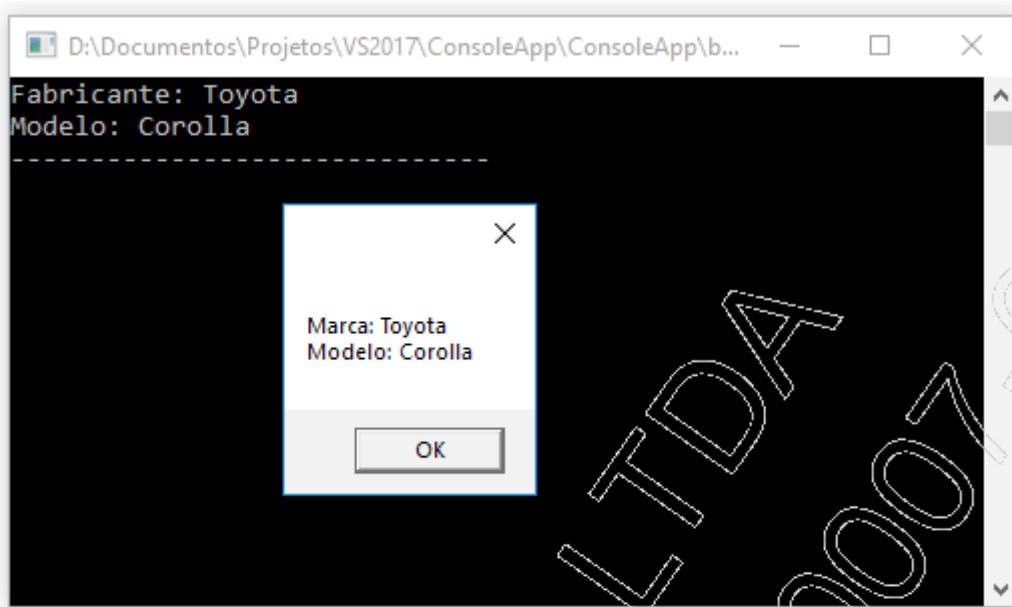
    //Definir valor aos seus campos/atributos
    carro.Fabricante = "Toyota";
    carro.Modelo = "Corolla";

    //Exibir os dados na listBox
    Console.WriteLine("Fabricante: " + carro.Fabricante);
    Console.WriteLine("Modelo: " + carro.Modelo);
    Console.WriteLine(new string('-', 30));

    //Exibir em uma MessageBox o retorno do método ExibirDados()
    System.Windows.Forms.MessageBox.Show(carro.ExibirDados());

    Console.ReadKey();
}
```

Veja, a seguir, o resultado desse código:



## 3.13.1. Sobre carga de métodos

A principal utilidade da sobre carga é a possibilidade de fazer com que uma mesma operação seja executada de formas diferentes. Assim, quando temos implementações de mesmo nome, mas com um número diferente de parâmetros, ou parâmetros de tipos diferentes, podemos sobre carregar um método.

**!** Não é possível declarar dois métodos com o mesmo nome, mesmos parâmetros e tipos de retorno diferentes, ou seja, a sobre carga do tipo de retorno não é permitida.

Veja um exemplo:

- Na classe

```
class Pessoa
{
    public string Nome { get; set; }
    public byte Idade { get; set; }

    public string ExibirDados()
    {
        return string.Format("Nome: {0}\nIdade: {1} anos",
            Nome, Idade.ToString());
    }

    public string ExibirDados(string observacao)
    {
        return string.Format("Nome: {0}\nIdade: {1} anos\nObs.: {2}",
            Nome, Idade.ToString(), observacao);
    }
}
```

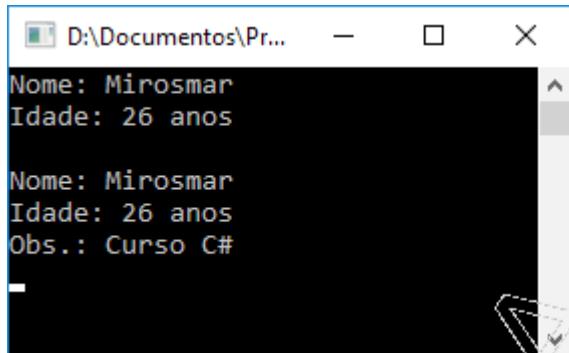
- No código que faz uso da classe

```
static void Main(string[] args)
{
    Pessoa gente = new Pessoa();
    gente.Nome = "Mirosmar";
    gente.Idade = 26;

    Console.WriteLine(gente.ExibirDados());
    Console.WriteLine();
    Console.WriteLine(gente.ExibirDados("Curso C#"));

    Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:



```
D:\Documentos\Pr...
Nome: Mirosmar
Idade: 26 anos

Nome: Mirosmar
Idade: 26 anos
Obs.: Curso C#
```

### 3.13.2. Métodos estáticos

Um método estático é aquele que não faz uso da instância da classe à qual pertence para ser utilizado.

Para acessar um método estático, é necessário digitar o nome da classe e não o nome da instância.

Métodos estáticos podem ser sobre carregados, mas não sobreescritos. Eles pertencem à classe e não a uma instância da classe.

Na classe **Metodos**, temos um método estático para retirar acentos e caracteres especiais. Esse método é de uso comum da aplicação:

```
class Metodos
{
    //Método estático
    public static string RetirarAcentos(string texto)
    {
        string comAcentos =
            "ÀÁÃÃÃÃÃäáâàÃÉÊÈéêééÍÍÍííííöÓÔÔöóôòõÜÚÛüúûùÇç";
        string semAcentos =
            "AAAAAAaaaEEEEEeeeeIIIiIIIiiii00000oooooUUUuuuuCc";

        for (int i = 0; i < comAcentos.Length; i++)
        {
            texto = texto.Replace(comAcentos[i].ToString(),
                                  semAcentos[i].ToString());
        }

        return texto;
    }
}
```

- No código que faz uso da classe

```
static void Main(string[] args)
{
    string texto =
        @"São cabeçalhos, rodapés, construções de gráficos coordenados com
a aparência do documento.";

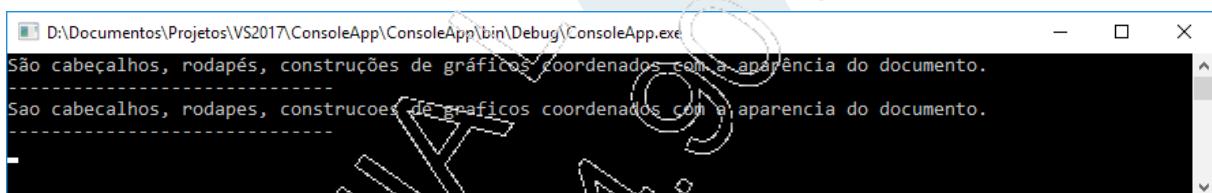
    //Chamada do método estático
    string textoTratado = Metodos.RetirarAcentos(texto);

    Console.WriteLine(texto);
    Console.WriteLine(new string('-', 30));

    Console.WriteLine(textoTratado);
    Console.WriteLine(new string('-', 30));

    Console.ReadKey();
}
```

O resultado desse código é o seguinte:



### 3.13.3. Métodos de extensão

Ao utilizarmos um método de extensão, poderemos ampliar uma classe ou uma estrutura já existente com métodos estáticos adicionais.

Para definir um método de extensão, devemos usar uma classe estática. Para especificar o tipo que será aplicado à classe, ou seja, o tipo que será estendido, basta defini-lo como sendo o primeiro parâmetro para o método precedido da palavra-chave `this`.

Na classe **Metodos**, que deve ser estática, temos um método para retirar acentos e caracteres especiais, que obrigatoriamente deverá ser estático também. Esse método é de uso comum da aplicação e será aplicado ao tipo **string**:

```
static class Metodos
{
    //Método estático
    public static string RetirarAcentos(this string texto)
    {
        string comAcentos =
            "ÄÅÂÂÃääâäÃÉËÈéêëÍÍÌííïÖÓÔÔÖöôòðÜÙÛüûùÇç";
        string semAcentos =
            "AAAAAAaaaaaaEEEEeeeeIIIIIiiii000000oooooUUJuuuuCc";
        for (int i = 0; i < comAcentos.Length; i++)
        {
            texto = texto.Replace(comAcentos[i].ToString(),
                semAcentos[i].ToString());
        }
        return texto;
    }
}
```

- **No código que faz uso da classe**

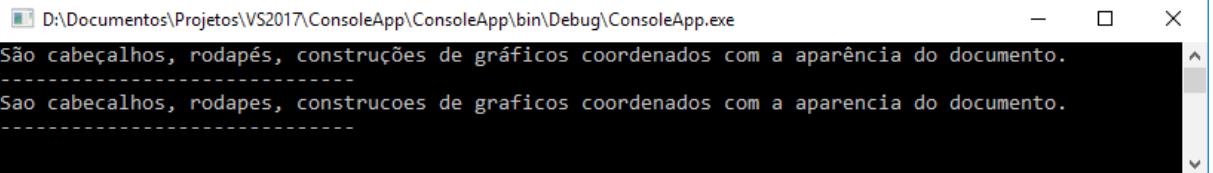
```
static void Main(string[] args)
{
    string texto =
        @"São cabeçalhos, rodapés, construções de gráficos coordenados com
        a aparência do documento.";

    //Chamada do método estático
    string textoTratado = texto.RetirarAcentos();

    Console.WriteLine(texto);
    Console.WriteLine(new string('-', 30));
    Console.WriteLine(textoTratado);
    Console.WriteLine(new string('-', 30));

    Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:



```
D:\Documentos\Projetos\VS2017\ConsoleApp\ConsoleApp\bin\Debug\ConsoleApp.exe
São cabeçalhos, rodapés, construções de gráficos coordenados com a aparência do documento.
-----
Sao cabecalhos, rodapes, construcoes de graficos coordenados com a aparence do documento.
-----
```

## 3.13.4. Parâmetros de métodos

Por executarem tarefas sobre objetos e de forma independente dos objetos (como é o caso dos métodos estáticos), podemos perceber que os métodos são os elementos mais importantes de um programa. Por conta disso, vamos apresentar as principais características dos métodos no que diz respeito aos seus parâmetros.

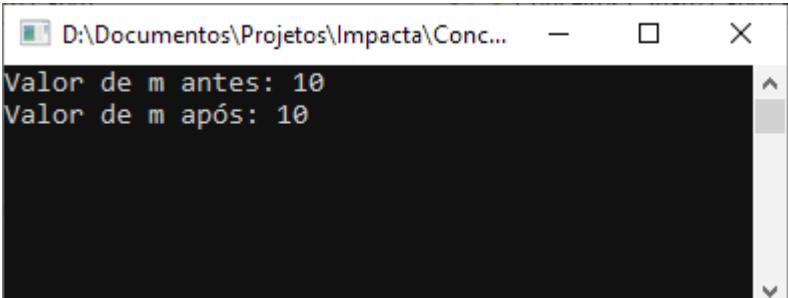
### 3.13.4.1. Passagem por referência - ref

Por padrão, os parâmetros dos métodos são passados por **valor**. Considere o exemplo a seguir:

```
class Program
{
    static void MudarValor(int x)
    {
        x += 100;
    }
    static void Main(string[] args)
    {
        int m = 10;
        Console.WriteLine("Valor de m antes: " + m);
        MudarValor(m);
        Console.WriteLine("Valor de m após: " + m);

        Console.ReadKey();
    }
}
```

Resultado:



```
D:\Documentos\Projetos\Impacta\Conc...
Valor de m antes: 10
Valor de m após: 10
```

O método **MudarValor** altera o valor do parâmetro **x**. Porém, quando chamado, o valor passado como parâmetro durante sua execução (a variável **m**) não sofre alterações. Isso porque **m** foi passado por valor, ou seja, seu valor foi copiado na variável **x** e, portanto, quaisquer alterações feitas em **x** não alteraram o valor original de **m**.

Para que a alteração seja realizada pelo método, o valor passado deve ser compartilhado com o parâmetro, ou seja, a passagem deve ser feita por referência. Isso é conseguido com o uso da palavra reservada **ref** tanto na definição como na chamada ao método.

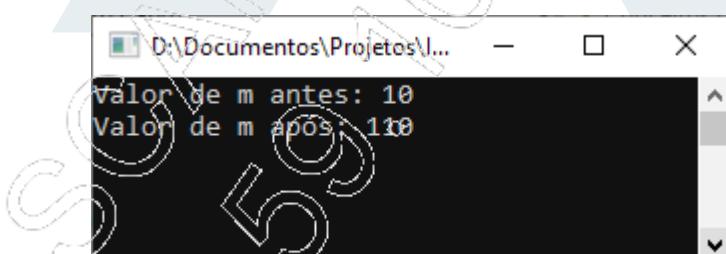
Veja o exemplo:

```
class Program
{
    static void MudarValor(ref int x)
    {
        x += 100;
    }
    static void Main(string[] args)
    {
        int m = 10;
        Console.WriteLine("Valor de m antes: " + m);

        MudarValor(ref m);

        Console.WriteLine("Valor de m após: " + m);
        Console.ReadKey();
    }
}
```

A execução deste programa produzirá o resultado:



## 3.13.4.2. Parâmetro de saída: out

Em certas ocasiões os métodos devem atribuir valores para os parâmetros. É um procedimento inverso à passagem tradicional de parâmetro: definimos uma variável, cujo valor é atribuído pelo próprio método. Situações como essa são usadas nos casos em que o método deve mapear **stored procedures** em bancos de dados, por exemplo, onde os valores são determinados por instruções internas, e seus resultados repassados para as variáveis externas.

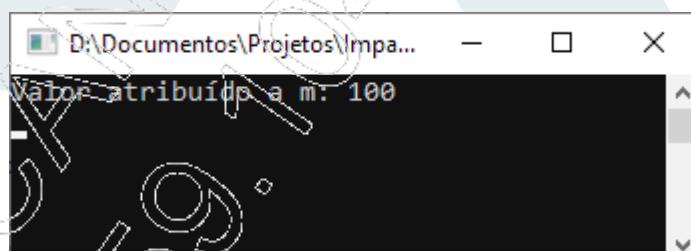
Vamos ver um exemplo de utilização, para ficar mais claro:

```
class Program
{
    static void AtribuirValor(out int x)
    {
        x = 100;
    }

    static void Main(string[] args)
    {
        int m;
        AtribuirValor(out m);
        Console.WriteLine("Valor atribuído a m: " + m);

        Console.ReadKey();
    }
}
```

Veja que o método atribuiu o valor 100 para o parâmetro `x`, que, por sua vez, foi repassado para `m`.



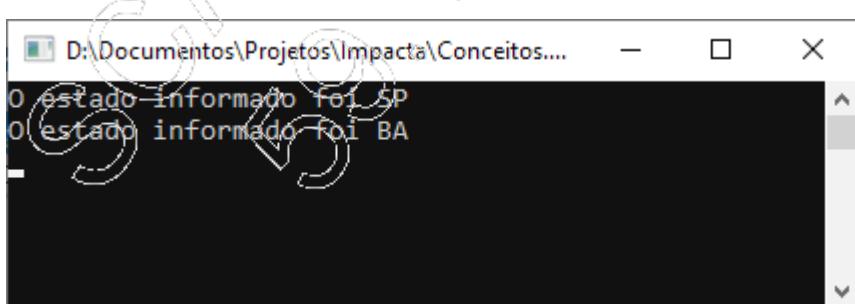
### 3.13.4.3. Parâmetros default e opcionais

Nos casos em que um ou mais parâmetros assumem um determinado valor com frequência, podemos defini-los com valores default. Podemos considerar como exemplo típico o caso de um método que recebe como parâmetro a UF de um país. Suponha que na grande maioria das vezes, o estado considerado seja São Paulo (SP). Para não ter que informar o valor para o parâmetro o tempo todo, podemos considerá-lo como opcional. Vejamos o exemplo a seguir:

```
namespace Conceitos.CSharp.Cap03
{
    enum Estados: sbyte
    {
        SP, MG, RJ, ES, BA
    }
    class Program
    {
        static string ApresentarEstado(Estados estado = Estados.SP)
        {
            return $"O estado informado foi {estado}";
        }
        static void Main(string[] args)
        {
            Console.WriteLine(ApresentarEstado());
            Console.WriteLine(ApresentarEstado(Estados.BA));

            Console.ReadKey();
        }
    }
}
```

O resultado da execução é:



O parâmetro, quando não informado na execução, assume o valor atribuído a ele na definição do método. Caso contrário, o valor predominante é o informado na sua chamada.

Uma observação importante é que podemos ter um método com vários parâmetros default. Se esse for o caso, eles devem ser os últimos e, no caso da omissão de algum valor, esta deve ocorrer do último para o primeiro. Considere o exemplo:

```
class Program
{
    static double CalcularLiquido(double valor, double taxa=0, double
imposto=0)
    {
        double desc1 = valor * taxa / 100;
        double desc2 = valor * imposto / 100;
        double valorLiquido = valor - desc1 - desc2;

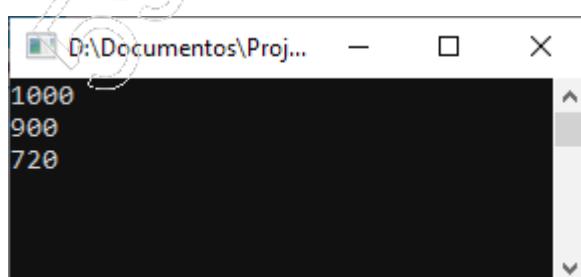
        return valorLiquido;
    }

    static void Main(string[] args)
    {
        double valor = 1000;
        double v1 = CalcularLiquido(valor);           //chamada 1
        double v2 = CalcularLiquido(valor, 10);        //chamada 2
        double v3 = CalcularLiquido(valor, 10, 18);    //chamada 3

        Console.WriteLine(v1);
        Console.WriteLine(v2);
        Console.WriteLine(v3);

        Console.ReadKey();
    }
}
```

No exemplo anterior, a linha comentada como chamada 1 assume os valores default para os parâmetros **taxa** e **imposto**. Na linha comentada como chamada 2, o valor 10 é atribuído para **taxa**, e o último parâmetro assumiu seu valor default. Na última instrução comentada, todos os parâmetros foram fornecidos. Mas o ponto crucial aqui é a ordem de atribuição dos parâmetros na execução do método. Se for passado em qualquer ordem, haverá uma confusão sobre qual parâmetro deve ser informado.



### 3.13.4.4. Parâmetros nomeados

No exemplo anterior, suponha que tenhamos o valor do imposto, mas não temos o valor da taxa (segundo parâmetro). Em casos como esse, podemos considerar a nomeação de parâmetros na chamada. Quando usamos parâmetros nomeados, estamos livres da ordem desses parâmetros. Mas não podemos esquecer que, ao trabalhar com parâmetros nomeados, aqueles que não possuem valores default devem ser informados.

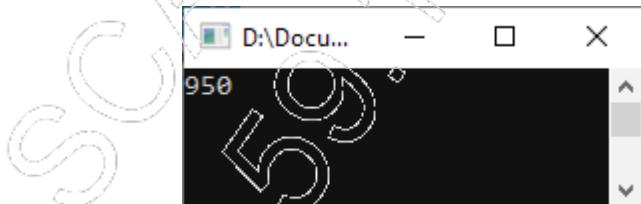
Vamos executar o exemplo anterior, com parâmetros nomeados:

```
class Program
{
    static double CalcularLiquido(double valor, double taxa=0, double
imposto=0)
    {
        double desc1 = valor * taxa / 100;
        double desc2 = valor * imposto / 100;
        double valorLiquido = valor - desc1 - desc2;

        return valorLiquido;
    }
    static void Main(string[] args)
    {
        double valor = 1000;

        double v1 = CalcularLiquido(valor, imposto: 5);
        Console.WriteLine(v1);

        Console.ReadKey();
    }
}
```



Verifique que, pelo fato de o parâmetro **taxa** ter sido omitido, seu valor default foi fornecido, mesmo que em uma ordem diferente, pois fornecemos o valor para o terceiro parâmetro, pulando o segundo.

## 3.13.4.5. Method references

Quando um método possui apenas uma instrução, é possível omitir seu bloco formado pelo par de chaves, usando o operador => para vincular o método ao seu resultado.

Vejamos os exemplos:

```
static double Calcular(double valor) => valor * 10;  
static void Mostrar() => Console.WriteLine("Method References");
```

No primeiro exemplo, o método **Calcular** retorna o valor do parâmetro multiplicado por 10, e no segundo exemplo, o método apresenta um texto na tela. Em ambos os casos, omitimos o bloco do método.

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- No campo computacional, objetos são elementos que representam uma entidade no domínio de interesse do problema analisado;
- Em termos gerais, entendemos por objetos quaisquer elementos da natureza aos quais é possível atribuir comportamentos e características. No entanto, em termos de computação, podemos definir um objeto como sendo uma estrutura de programação na qual estão encapsulados dados e funcionalidade, formando uma unidade possível de ser acessada publicamente por meio de propriedades, métodos e eventos;
- Os objetos possuem atributos, que são as suas características particulares, e métodos, que definem as ações dos objetos. Além disso, podem se comunicar com outros objetos por meio de mensagens;
- Uma classe é um modelo para um conjunto de objetos que possuem os mesmos atributos e métodos. Os objetos são considerados instâncias das classes. Quando trabalhamos com as classes, podemos criar aplicações empregando técnicas de programação orientada a objeto;
- O encapsulamento permite ocultar os detalhes da implementação interna de uma classe, restringindo o acesso a suas variáveis e métodos. Isso permite que tarefas dependentes sejam executadas sem a interferência do usuário, eliminando as possibilidades de erros;
- Com a utilização de namespaces, possíveis conflitos entre nomes são resolvidos. Cria-se um contêiner nomeado para outros identificadores, como classes. Com duas classes de mesmo nome, mas cada uma localizada em um namespace diferente, não haverá conflito entre elas;
- Para acessar os membros dentro da própria classe que os define, podemos usar a referência `this`;
- Podemos criar instâncias de classes implicitamente ou de forma anônima;
- Os métodos são as ações que os objetos desempenham e podem ser considerados como funções. Sempre que um método é chamado, acontece a execução de um código;
- Podemos passar parâmetros por referência para métodos, usando a palavra reservada `ref` tanto na definição como na chamada ao método;
- Podemos também definir métodos que produzem valores a serem atribuídos a variáveis disponíveis no programa, ou seja, executando um processo inverso à passagem de parâmetro. Para essa tarefa, usamos os parâmetros de saída com a palavra `out`;



- Em cenários onde valores de parâmetros seguem determinados padrões, podemos defini-los como parâmetros opcionais. Na passagem de parâmetro, a ordem é relevante: os parâmetros opcionais devem ser os últimos, e a atribuição deve seguir a ordem da definição;
  - No caso de não termos todos os parâmetros disponíveis na ordem desejada, podemos utilizar a nomeação de parâmetros. Nesse caso, a ordem dos parâmetros deixa de ser relevante;
  - Em casos mais simples, onde os métodos apresentam apenas uma instrução, podemos omitir as chaves constituindo o corpo do método, através do recurso conhecido como **method reference**.

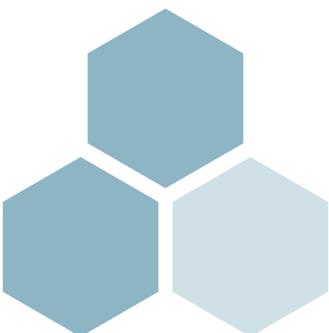




3

# Orientação a objetos

Teste seus conhecimentos



**1. Qual a parte de uma classe que representa as características do objeto?**

- a) Propriedade
- b) Evento
- c) Método
- d) Herança
- e) Nenhuma das alternativas anteriores está correta.

**2. Qual o operador que permite, em um método, acesso aos atributos da superclasse?**

- a) this
- b) base
- c) super
- d) override
- e) Nenhuma das alternativas anteriores está correta.

**3. Qual o nome mais comum para o que chamamos de classe base?**

- a) Superclasse
- b) Subclasse
- c) Polimorfismo
- d) Herança
- e) Sobrecarga

**4. Qual a alternativa que completa adequadamente a frase a seguir?**

Para que possamos fazer \_\_\_\_\_ de um método, é necessário que ele seja \_\_\_\_\_.

- a) override, dynamic
- b) overload, dynamic
- c) overload, virtual
- d) override, virtual
- e) herança, virtual

**5. Qual das alternativas a seguir está correta sobre sobrecarga de métodos?**

- a) Complementa as ações de um método herdado de uma classe ancestral.
- b) Podemos ter dois ou mais métodos com o mesmo nome, mas com parâmetros diferentes.
- c) Podemos ter dois ou mais métodos com o mesmo nome, mas com parâmetros e tipo de retorno diferentes.
- d) Em classes derivadas, podemos ter métodos com o mesmo nome e a mesma assinatura, mas que executam tarefas distintas.
- e) O C# não possui este recurso.





3

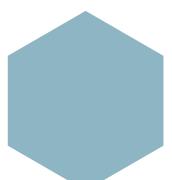
# Orientação a objetos



Mãos à obra!

Editora

**IMPACTA**



Nos laboratórios deste capítulo, escreveremos algumas classes importantes para uma aplicação consumi-la. Essas classes serão aproveitadas e complementadas em laboratórios futuros.

O objetivo é separarmos o código por responsabilidades: os métodos utilitários farão parte de um projeto, as classes a serem instanciadas, a outro projeto, a parte que interage com o usuário em um terceiro projeto e assim por diante. Vamos iniciar os laboratórios.

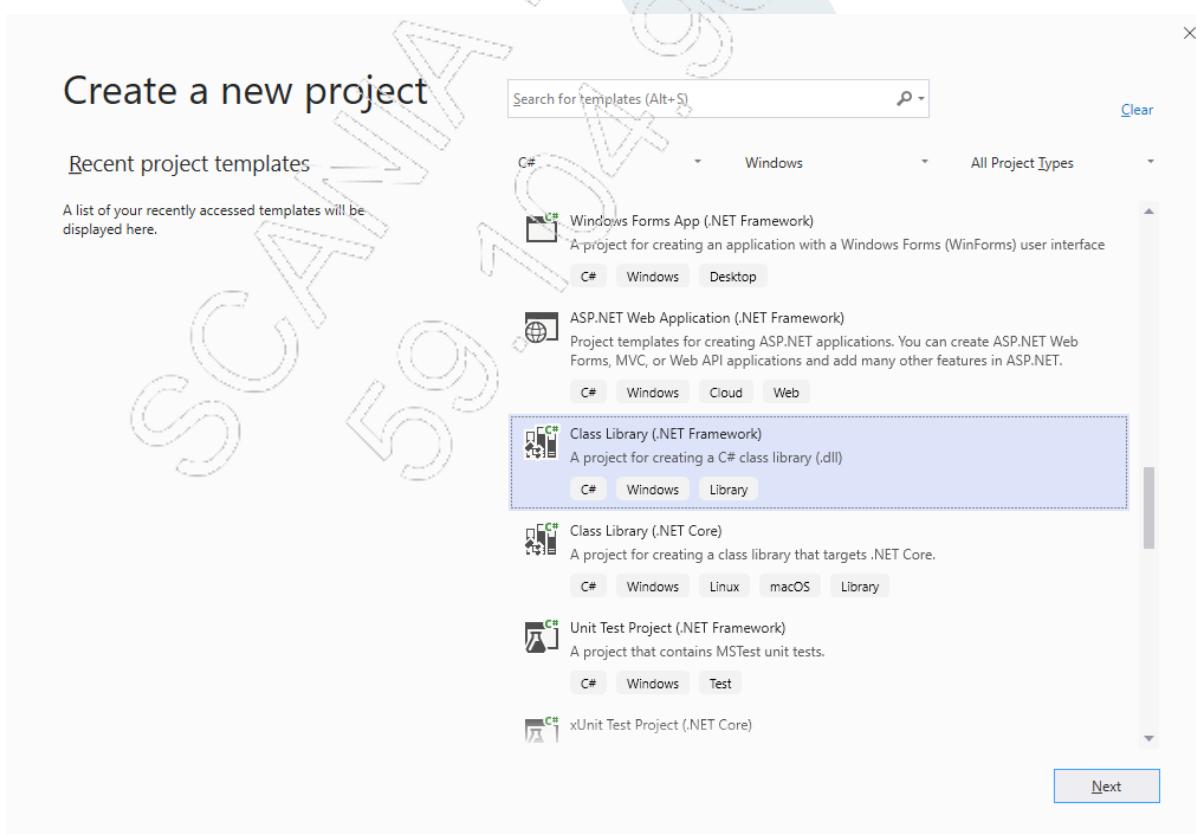
## Laboratório 1

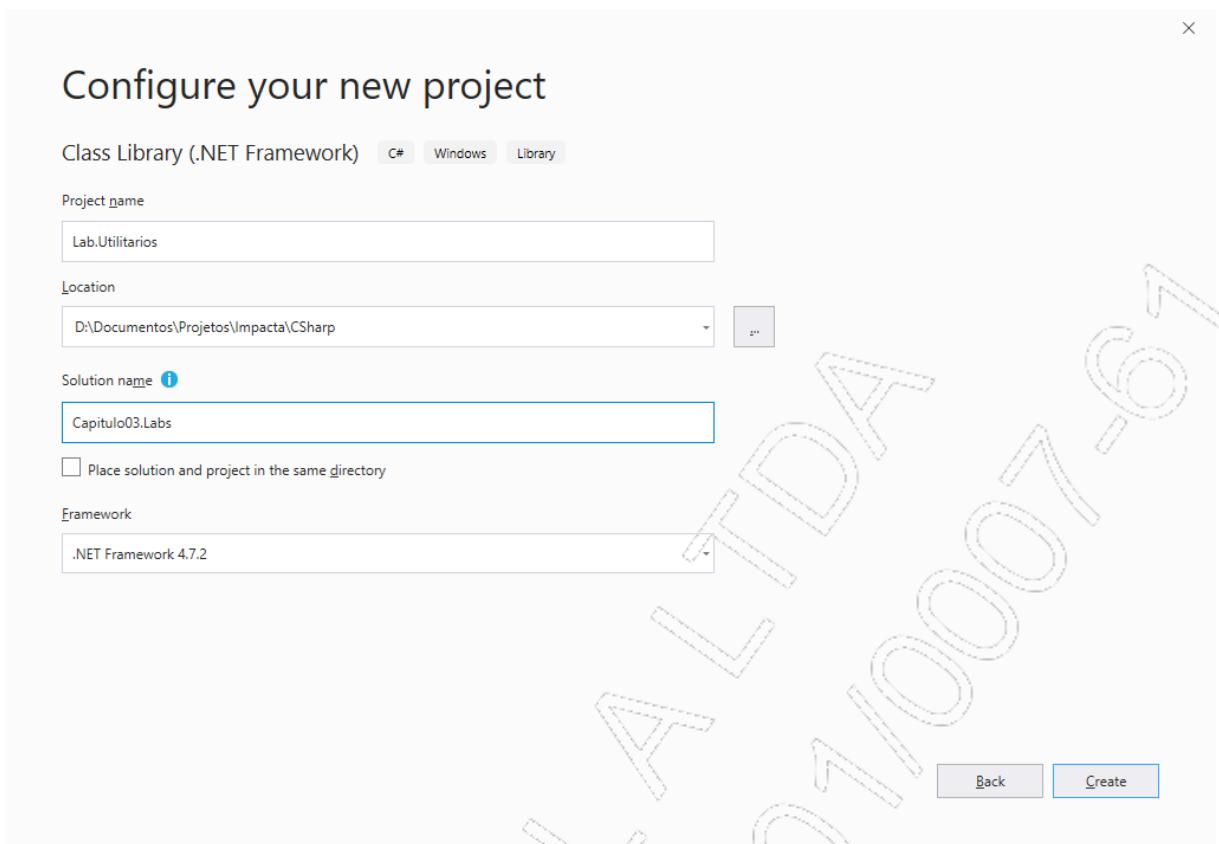
### A - Escrevendo uma biblioteca de classes

#### Objetivo:

Escrever uma biblioteca de classes para definir métodos utilitários e de extensão. Esses métodos serão usados por outras classes e pela aplicação.

1. Defina um novo projeto **Class Library (.NET Framework)** chamado **Lab.Utilitarios** em uma solução chamada **Capítulo03.Labs**:





2. Remova a classe **Class1**, gerada com o projeto;
3. Inclua uma classe estática chamada **MetodosExtensao**:

```
namespace Lab.Utilitarios
{
    public static class MetodosExtensao
    {
    }
}
```

# Programando com C#

4. Adicione o seguinte método de extensão para a classe string, capaz de validar os dígitos de um CPF:

```
namespace Lab.Utilitarios
{
    public static class MetodosExtensao
    {
        public static bool ValidarCPF(this string vrCPF)
        {
            string valor = vrCPF.Replace(".", "");
            valor = valor.Replace("-", "");

            if (valor.Length != 11)
                return false;

            bool igual = true;

            for (int i = 1; i < 11 && igual; i++)
                if (valor[i] != valor[0])
                    igual = false;

            if (igual || valor == "12345678909")
                return false;

            int[] numeros = new int[11];

            for (int i = 0; i < 11; i++)
                numeros[i] = int.Parse(
                    valor[i].ToString());

            int soma = 0;

            for (int i = 0; i < 9; i++)
                soma += (10 - i) * numeros[i];

            int resultado = soma % 11;

            if (resultado == 1 || resultado == 0)
            {
                if (numeros[9] != 0)
                    return false;
            }
            else if (numeros[9] != 11 - resultado)
                return false;

            soma = 0;

            for (int i = 0; i < 10; i++)
                soma += (11 - i) * numeros[i];

            resultado = soma % 11;

            if (resultado == 1 || resultado == 0)
            {
                if (numeros[10] != 0)
                    return false;
            }
            else
                if (numeros[10] != 11 - resultado)
                    return false;
        }

    }
}
```

B – Escrevendo uma biblioteca de classes contendo classes para armazenar valores referentes a uma conta bancária

**Objetivo:**

Serão criadas a principais classes para conter os dados da conta e dos clientes bancários, além dos métodos usados para a execução de operações sobre as contas.

1. Adicione um projeto **Class Library (.NET Framework)** ao solution **Capítulo03.Labs** chamado **Lab.Models**;
2. Neste projeto, adicione as seguintes classes, estruturas e enumerações:

- **Estrutura Endereco**

```
namespace Lab.Models
{
    public struct Endereco
    {
        public string Logradouro { get; set; }
        public int Numero { get; set; }
        public string Cidade { get; set; }
        public string Cep { get; set; }
    }
}
```

- **Enumeração Sexos**

```
namespace Lab.Models
{
    public enum Sexos
    {
        Masculino, Feminino
    }
}
```

- **Enumeração Operacoes**

```
namespace Lab.Models
{
    public enum Operacoes
    {
        Deposito, Saque
    }
}
```

- **Classe ContaCorrente**

```
namespace Lab.Models
{
    public class ContaCorrente
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }
        public double Saldo { get; private set; }

        public Cliente ClienteInfo { get; set; }
    }
}
```

- **Classe Cliente**

```
namespace Lab.Models
{
    public class Cliente
    {
        public string Cpf { get; set; }
        public string Nome { get; set; }
        public int Idade { get; set; }
        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public ContaCorrente Conta { get; set; }
    }
}
```

3. Insira a instrução **static** antes da instrução **class**:

```
namespace Lab_07
{
    class MetodosDeExtensao
    {
    }
}
```

4. Remova a classe **Class1**;

5. Faça as alterações a seguir na classe **Cliente** (aqui usamos uma instrução para lançamento de exceção, que será apresentada em capítulo futuro). Para esta alteração, refcrcie o projeto **Lab.Utilitarios** no projeto atual, e inclua a instrução **using Lab.Utilitarios**:

```
using Lab.Utilitarios;

namespace Lab.Models
{
    public class Cliente
    {
        private string _cpf;
        public string Cpf
        {
            get => _cpf;
            set => _cpf = (value.ValidarCPF() ? value : throw new Exception("CPF inválido"));
        }

        public string Nome { get; set; }
        public int Idade { get; set; }
        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public ContaCorrente Conta { get; set; }
    }
}
```

6. Na classe **ContaCorrente**, adicione um método responsável por efetuar as operações de saque ou depósito. O tipo de operação deve ser passado como parâmetro, assim como o valor da operação. Consideremos o **Depósito** como operação default:

```
namespace Lab.Models
{
    public class ContaCorrente
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }
        public double Saldo { get; private set; }

        public Cliente ClienteInfo { get; set; }

        public void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            switch (operacao)
            {
                case Operacoes.Deposito:
                    this.Saldo += valor;
                    break;
                case Operacoes.Saque:
                    this.Saldo -= valor;
                    break;
            }
        }
    }
}
```

7. Adicione métodos para retornar as informações do cliente, do endereço e da conta corrente. Para isso, faça as seguintes alterações nas classes **Cliente** e **ContaCorrente**, além da estrutura **Endereco**. Vamos adicionar, em cada uma, um método chamado **Exibir**:

```
namespace Lab.Models
{
    public struct Endereco
    {
        public string Logradouro { get; set; }
        public int Numero { get; set; }
        public string Cidade { get; set; }
        public string Cep { get; set; }

        public string Exibir()
        {
            return $"Logradouro: {this.Logradouro}\n" +
                $"Número: {this.Numero}\n" +
                $"Cidade: {this.Cidade}\n" +
                $"CEP: {this.Cep}";
        }
    }

    namespace Lab.Models
    {
        public class Cliente
        {
            private string _cpf;
            public string Cpf
            {
                get => _cpf;
                set => _cpf = (value.ValidarCPF() ? value :
                    throw new Exception("CPF inválido"));
            }

            public string Nome { get; set; }
            public int Idade { get; set; }
            public Sexos Sexo { get; set; }
            public Endereco EnderecoResidencial { get; set; }

            public ContaCorrente Conta { get; set; }

            public string Exibir()
            {
                return $"Cpf: {this.Cpf}\n" +
                    $"Nome: {this.Nome}\n" +
                    $"Idade: {this.Idade}\n" +
                    $"Sexo: {this.Sexo}\n" +
                    $"ENDEREÇO DO CLIENTE:\n" +
                    $"{this.EnderecoResidencial.Exibir()}";
            }
        }
    }
}
```

```
namespace Lab.Models
{
    public class ContaCorrente
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }
        public double Saldo { get; private set; }

        public Cliente ClienteInfo { get; set; }

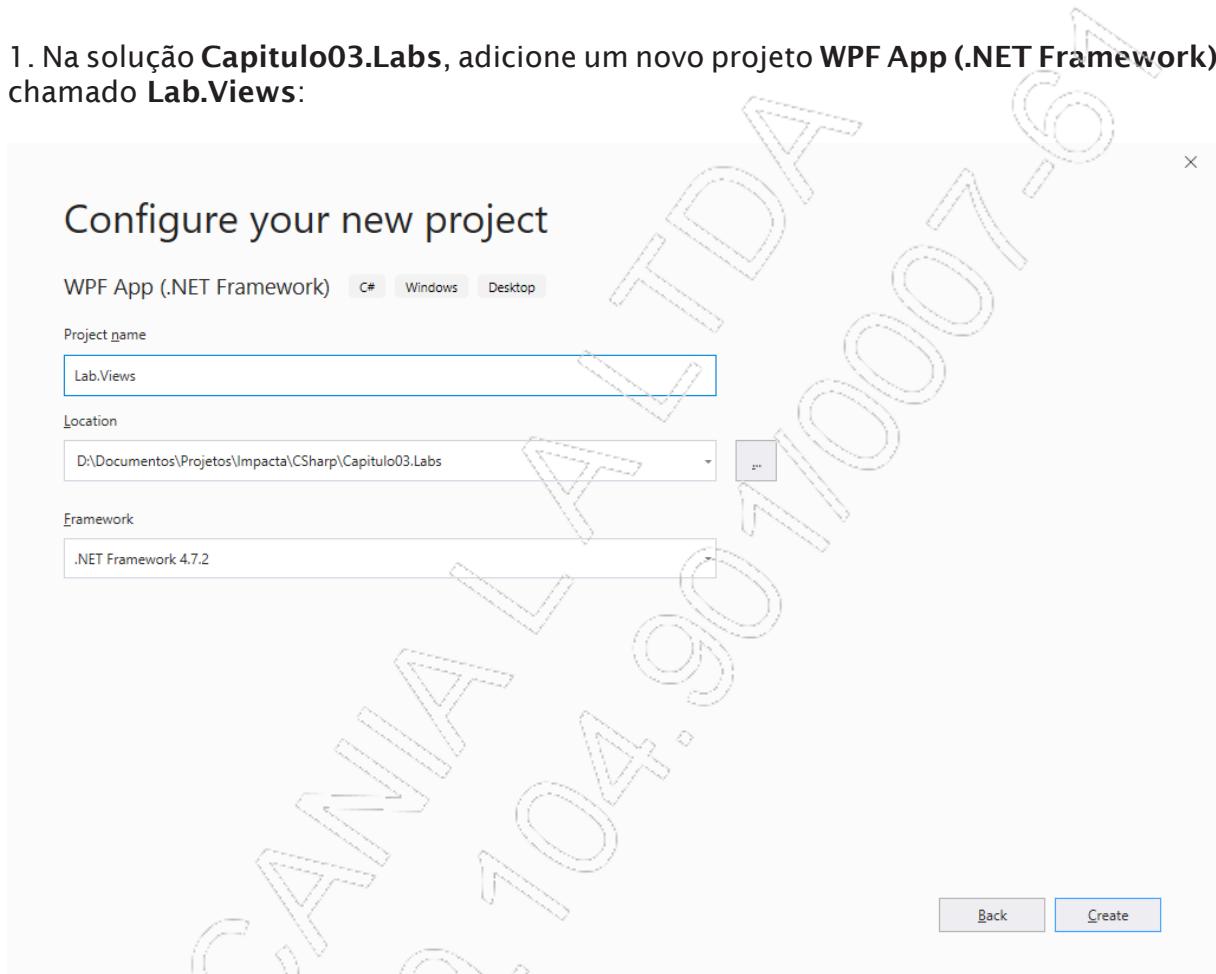
        public void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            switch (operacao)
            {
                case Operacoes.Deposito:
                    this.Saldo += valor;
                    break;
                case Operacoes.Saque:
                    this.Saldo -= valor;
                    break;
            }
        }

        public string Exibir()
        {
            string cliente = this.ClienteInfo != null ?
                this.ClienteInfo.Exibir() + '\n' : "";
            return $"{cliente}" +
                $"Banco: {this.NumeroBanco}\n" +
                $"Agência: {this.NumeroAgencia}\n" +
                $"Conta: {this.NumeroConta}\n" +
                $"Saldo Atual: {this.Saldo}";
        }
    }
}
```

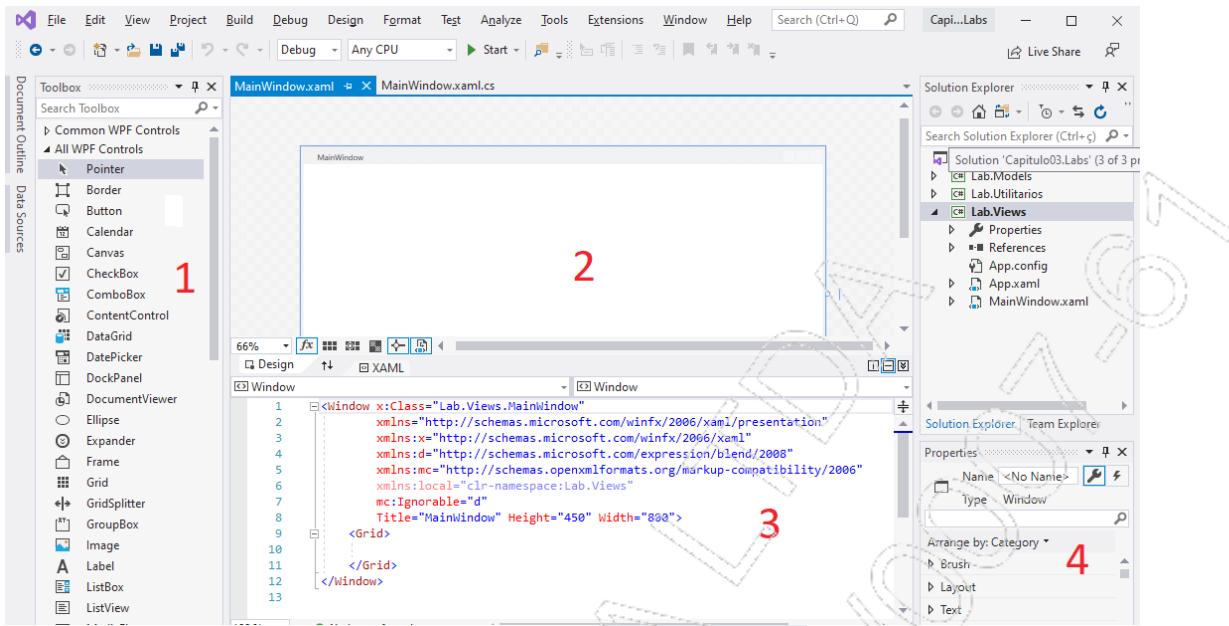
## C – Criando uma aplicação que consuma as classes definidas nos itens A e B

Esta aplicação, diferente dos programas definidos nos capítulos anteriores, terá uma interface gráfica, melhorando assim a experiência do usuário. Assim como as classes definidas neste capítulo, esta aplicação será evoluída ao longo dos capítulos, na medida em que novos conceitos forem estudados.

1. Na solução **Capitulo03.Labs**, adicione um novo projeto **WPF App (.NET Framework)** chamado **Lab.Views**:



O projeto nos fornece, dentre outros, os seguintes elementos disponíveis no Visual Studio:



Na nova janela podemos destacar as partes numeradas:

- **1 – ToolBox**: Local onde se localizam os componentes a serem inseridos na interface gráfica (parte 2);
- **2 – Design**: Parte visual da interface gráfica;
- **3 – XAML (eXtensible Application Markup Language)**: Parte codificada da interface gráfica. Os componentes podem ser inseridos na parte visual ou no XAML, tanto através do **ToolBox** como manualmente;
- **4 – Properties**: Janela contendo as propriedades dos componentes manipulados na interface gráfica. Esta janela apresenta as propriedades do componente selecionado. É possível incluirmos os valores das propriedades diretamente no XAML. A escolha depende da habilidade do programador.

Neste novo tipo de projeto, teremos a oportunidade de criar componentes gráficos em uma interface com o objetivo de definir uma aplicação **WPF (Windows Presentation Foundation)**. Neste tipo de projeto, desenhar os componentes a partir do **ToolBox** (1) ou escrevê-los diretamente no arquivo XAML (3) depende da facilidade que o programador encontrar no momento de estruturar a aplicação.

Neste laboratório, procuraremos usar o código XAML, para termos maior controle sobre a interface, especialmente na hora de definirmos elementos de grade e tabs.

2. Inicialmente, temos o conteúdo do arquivo **MainWindow.xaml** gerado pelo Visual Studio no momento da criação do projeto:

```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

A propriedade **x:Class="Lab.Views.MainWindow"** do elemento **Window** especifica a classe correspondente ao que chamamos de **Code Behind**, ou seja, o local onde escreveremos os eventos a serem executados sobre os componentes da interface gráfica.

Altere o título da janela para "**Aplicação Bancária**". Use, para isso, a propriedade **Title** do elemento **Window**:

```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="Aplicação Bancária" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Todos os elementos do arquivo XAML, como **Window** e **Grid**, nesta fase inicial da aplicação são, na verdade, classes disponíveis no .NET Framework. A classe **Window** está no namespace **System.Windows** e a classe **Grid**, no namespace **System.Windows.Controls**. Basta mover o mouse sobre o elemento para ver em qual namespace sua classe pertence.

3. Envolva o elemento <Grid> com o elemento <DockPanel>:

```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="Aplicação Bancária" Height="450" Width="800">

    <DockPanel>
        <Grid>
            </Grid>
        </DockPanel>
    </Window>
```

Este componente facilita o posicionamento dos componentes em diferentes direções da interface, tornando-as independentes.

4. Dentro do elemento <Grid>, inclua o elemento <TabControl>, com as propriedades:

- **Name:** "TabCadastro"
- **VerticalAlignment:** "Top"

```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="Aplicação Bancária" Height="450" Width="800">

    <DockPanel>
        <Grid>
            <TabControl Name="TabCadastro" VerticalAlignment="Top">
                </TabControl>
        </Grid>
    </DockPanel>
</Window>
```

5. Dentro do elemento <TabControl>, insira três elementos <TabItem>, com os seguintes valores para a propriedade Header:

- (1) Header: "Cadastro de Clientes"
- (2) Header: "Cadastro de Contas"
- (3) Header: "Operações Bancárias"

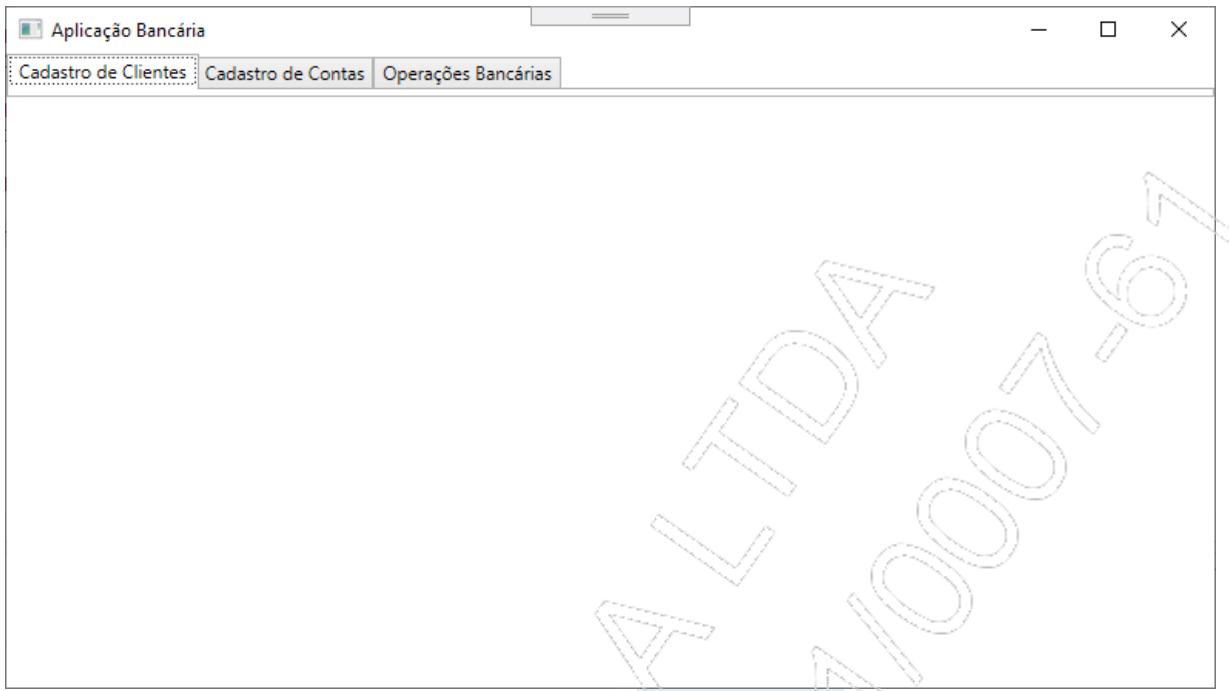
```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="Aplicação Bancária" Height="450" Width="800">

    <DockPanel>
        <Grid>
            <TabControl Name="TabCadastro" VerticalAlignment="Top">

                <TabItem Header="Cadastro de Clientes">
                </TabItem>
                <TabItem Header="Cadastro de Contas">
                </TabItem>
                <TabItem Header="Operações Bancárias">
                </TabItem>

            </TabControl>
        </Grid>
    </DockPanel>
</Window>
```

Veja o resultado destas tabulações na interface gráfica, durante a execução da aplicação até este momento:



6. Vamos agora montar a estrutura para os componentes. No primeiro elemento <TabItem>, inclua um novo elemento <Grid> com a propriedade:



- **Background:** "LightBlue"

Defina duas linhas, uma com 150 pixels e outra com o valor restante disponível (marcado com asterisco (\*)):

```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="Aplicação Bancária" Height="450" Width="800">

    <DockPanel>
        <Grid>
            <TabControl Name="TabCadastro" VerticalAlignment="Top">

                <TabItem Header="Cadastro de Clientes">
                    <Grid Background="LightBlue">
                        <Grid.RowDefinitions>
                            <RowDefinition Height="225" />
                            <RowDefinition Height="*" />
                        </Grid.RowDefinitions>
                    </Grid>
                </TabItem>
                <TabItem Header="Cadastro de Contas">
                    </TabItem>
                <TabItem Header="Operações Bancárias">
                    </TabItem>
            </TabControl>
        </Grid>
    </DockPanel>
</Window>
```

7. Dentro do elemento <Grid> recém inserido, e abaixo de </Grid.RowDefinitions>, adicione na primeira linha (**Grid.Row="0"**) uma estrutura com novas linhas e colunas. Este novo Grid será usado para inserir os elementos para receber os dados básicos do cliente, incluindo o endereço:

```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="Aplicação Bancária" Height="450" Width="800">

    <DockPanel>
        <Grid>
            <TabControl Name="TabCadastro" VerticalAlignment="Top">

                <TabItem Header="Cadastro de Clientes">
                    <Grid Background="LightBlue">
                        <Grid.RowDefinitions>
                            <RowDefinition Height="225" />
                            <RowDefinition Height="*" />
                        </Grid.RowDefinitions>

                        <Grid Grid.Row="0">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="100" />
                                <ColumnDefinition Width="*" />
                            </Grid.ColumnDefinitions>

                            <Grid.RowDefinitions>
                                <RowDefinition Height="25" />
                                <RowDefinition Height="25" />
                            </Grid.RowDefinitions>
                        </Grid>
                    </Grid>
                </TabItem>
                <TabItem Header="Cadastro de Contas">
                </TabItem>
                <TabItem Header="Operações Bancárias">
                </TabItem>
            </TabControl>
        </Grid>
    </DockPanel>
</Window>
```

# Programando com C#

8. Dentro do elemento `<Grid Grid.Row="0">` recém inserido, imediatamente antes do seu fechamento (abaixo de `</Grid.RowDefinitions>`), adicione os elementos de interface gráfica apresentados adiante (estes elementos são compostos por **Label**, **TextBox**, **ComboBox** e **Button**):

Componente	Nome	Texto	Posição (Linha,Coluna)
Label	cpfLabel	CPF:	(0,0)
Label	nomeLabel	Nome:	(1,0)
Label	idadeLabel	Idade:	(2,0)
Label	sexoLabel	Sexo:	(3,0)
Label	ruaLabel	Rua:	(4,0)
Label	numeroLabel	Número:	(5,0)
Label	cidadeLabel	Cidade:	(6,0)
Label	cepLabel	CEP	(7,0)
TextBox	cpfTextBox		(0,1)
TextBox	nomeTextBox		(1,1)
TextBox	idadeTextBox		(2,1)
TextBox	ruaTextBox		(4,1)
TextBox	numeroTextBox		(5,1)
TextBox	cidadeTextBox		(6,1)
TextBox	cepTextBox		(7,1)
ComboBox	sexoComboBox	[Masculino,Feminino]	(3,1)
Button	incluirClienteButton	Incluir Cliente	(8,1)

```
<Window x:Class="Lab.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:Lab.Views"
    mc:Ignorable="d"
    Title="Aplicação Bancária" Height="450" Width="800">

    <DockPanel>
        <Grid>
            <TabControl Name="TabCadastro" VerticalAlignment="Top">
                <TabItem Header="Cadastro de Clientes">
                    <Grid Background="LightBlue">
                        <Grid.RowDefinitions>
                            <RowDefinition Height="225" />
                            <RowDefinition Height="*" />
                        </Grid.RowDefinitions>

                        <Grid Grid.Row="0">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="100" />
                                <ColumnDefinition Width="*" />
                            </Grid.ColumnDefinitions>

                            <Grid.RowDefinitions>
                                <RowDefinition Height="25"/>
                                <RowDefinition Height="25"/>
                            </Grid.RowDefinitions>
                            <!--Componentes Label-->
                            <Label Name="cpfLabel" Content="CPF:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="0" Grid.Column="0"/>

                            <Label Name="nomeLabel" Content="Nome:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="1" Grid.Column="0"/>
                        </Grid>
                    </Grid>
                </TabItem>
            </TabControl>
        </Grid>
    </DockPanel>

```

```
<Label Name="idadeLabel" Content="Idade:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="2" Grid.Column="0"/>  
  
<Label Name="sexoLabel" Content="Sexo:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="3" Grid.Column="0"/>  
  
<Label Name="ruaLabel" Content="Rua:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="4" Grid.Column="0"/>  
  
<Label Name="numeroLabel" Content="Número:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="5" Grid.Column="0"/>  
  
<Label Name="cidadeLabel" Content="Cidade:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="6" Grid.Column="0"/>  
  
<Label Name="cepLabel" Content="CEP:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="7" Grid.Column="0"/>  
  
<!--Componentes TextBox e ComboBox-->  
<TextBox Name="cpfTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="0" Grid.Column="1"/>  
  
<TextBox Name="nomeTextBox" Width="200"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="1" Grid.Column="1"/>  
  
<TextBox Name="idadeTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="2" Grid.Column="1"/>  
  
<ComboBox Grid.Row="3" Grid.Column="1"  
          Name="sexoComboBox"  
          VerticalAlignment="Center"  
          HorizontalAlignment="Left"  
          Width="150">  
  </ComboBox>
```

```
<TextBox Name="ruaTextBox" Width="200"
    VerticalAlignment="Center"
    HorizontalAlignment="Left"
    Grid.Row="4" Grid.Column="1"/>

<TextBox Name="numeroTextBox" Width="100"
    VerticalAlignment="Center"
    HorizontalAlignment="Left"
    Grid.Row="5" Grid.Column="1"/>

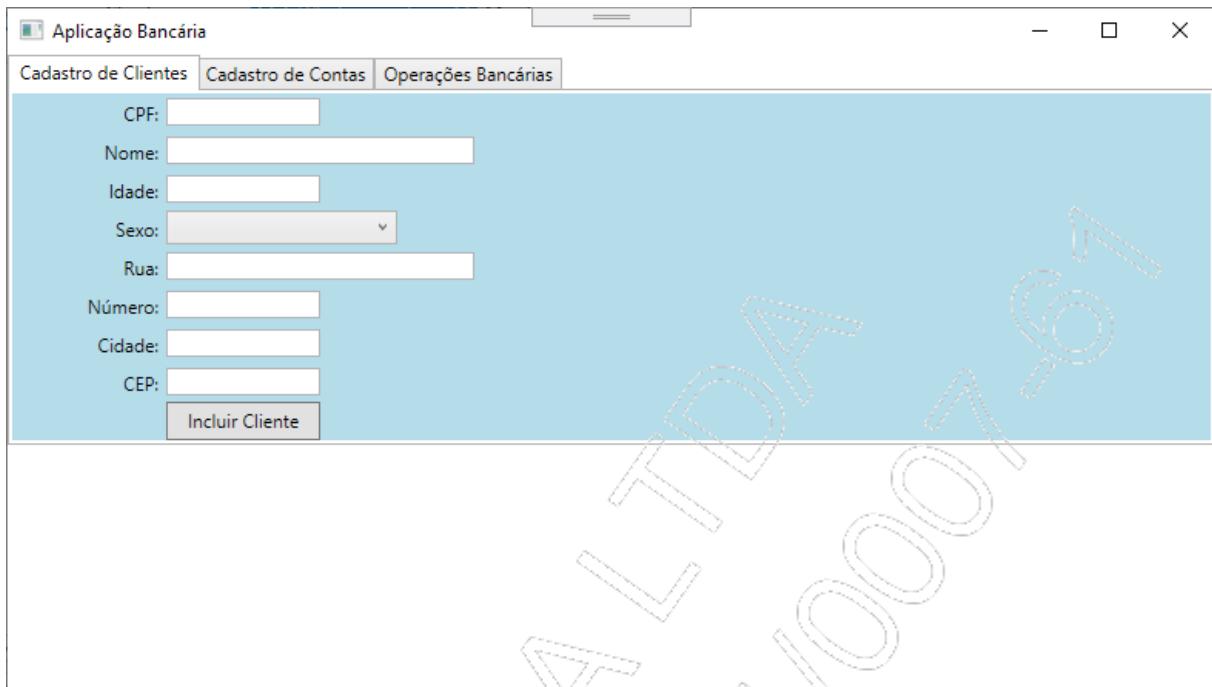
<TextBox Name="cidadeTextBox" Width="100"
    VerticalAlignment="Center"
    HorizontalAlignment="Left"
    Grid.Row="6" Grid.Column="1"/>

<TextBox Name="cepTextBox" Width="100"
    VerticalAlignment="Center"
    HorizontalAlignment="Left"
    Grid.Row="7" Grid.Column="1"/>
<!--Botão para incluir um cliente--&gt;
&lt;Button Grid.Row="8" Grid.Column="1"
    Name="incluirClienteButton"
    Content="Incluir Cliente"
    HorizontalAlignment="Left"
    Width="100"&gt;
    &lt;Button.ToolTip&gt;
        &lt;StackPanel Width="150" Height="20"
            Background="Beige"&gt;
            &lt;TextBlock&gt;
                Permite incluir um cliente
            &lt;/TextBlock&gt;
        &lt;/StackPanel&gt;
    &lt;/Button.ToolTip&gt;
&lt;/Button&gt;
&lt;/Grid&gt;
&lt;/Grid&gt;
&lt;/TabItem&gt;
&lt;TabItem Header="Cadastro de Contas"&gt;

&lt;/TabItem&gt;
&lt;TabItem Header="Operações Bancárias"&gt;

&lt;/TabItem&gt;
&lt;/TabControl&gt;
&lt;/Grid&gt;
&lt;/DockPanel&gt;
&lt;/Window&gt;</pre>
```

9. Execute a aplicação e veja como ficou a interface até este momento:



10. Repita todo o processo para o segundo elemento <TabItem>, aquele que permite incluir uma nova conta. Neste caso, o cliente deverá ser selecionado em um componente ComboBox. Sua codificação será realizada em laboratórios futuros. Fique atento aos elementos no código. Os elementos serão inseridos dentro do elemento <TabItem Header="Cadastro de Contas">:

```
<TabItem Header="Cadastro de Contas">

    <Grid Background="LightGreen">
        <Grid.RowDefinitions>
            <RowDefinition Height="150" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid Grid.Row="0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="25" />
                <RowDefinition Height="25" />
            </Grid.RowDefinitions>
        
```

```
</Grid>
```

```
<!--Componentes Label-->
<Label Name="clienteLabel" Content="Cliente:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="0" Grid.Column="0"/>

<Label Name="bancoLabel" Content="Núm. Banco:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="1" Grid.Column="0"/>

<Label Name="agenciaLabel" Content="Núm.  
Agência:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="2" Grid.Column="0"/>

<Label Name="contaLabel" Content="Núm. Conta:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="3" Grid.Column="0"/>

<Label Name="tipoLabel" Content="Tipo:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="4" Grid.Column="0"/>

<!--Componentes TextBox e ComboBox-->
<ComboBox Grid.Row="0" Grid.Column="1"  
          Name="clienteComboBox"  
          VerticalAlignment="Center"  
          HorizontalAlignment="Left"  
          Width="150">  
</ComboBox>

<TextBox Name="bancoTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="1" Grid.Column="1"/>

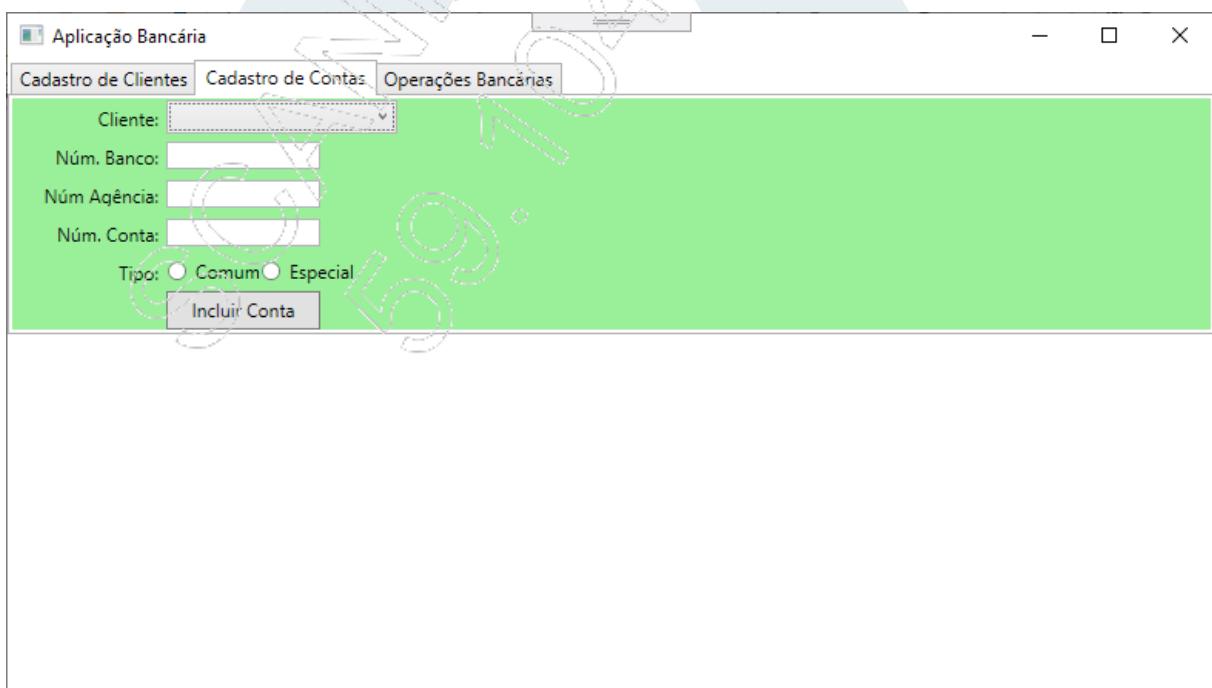
<TextBox Name="agenciaTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="2" Grid.Column="1"/>

<TextBox Name="contaTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="3" Grid.Column="1"/>
```

```
<StackPanel Grid.Row="4" Grid.Column="1"
            Orientation="Horizontal"
            VerticalAlignment="Center">
    <RadioButton GroupName="tipo"
    Content="Comum" />
    <RadioButton GroupName="tipo"
    Content="Especial"/>
</StackPanel>

<!--Botão para incluir uma conta-->
<Button Grid.Row="5" Grid.Column="1"
        Name="incluirContaButton"
        Content="Incluir Conta"
        HorizontalAlignment="Left"
        Width="100">
    <Button.ToolTip>
        <StackPanel Width="150" Height="20"
                    Background="Beige">
            <TextBlock>
                Permite incluir uma conta
            </TextBlock>
        </StackPanel>
    </Button.ToolTip>
</Button>
</Grid>
</Grid>
</TabItem>
```

11. Teste esta nova aba:



## D – Definindo eventos na interface gráfica para apresentar os dados do cliente para o usuário

1. Defina o código para o evento **Click** do botão **incluirClienteButton**, de forma a exibir os dados do cliente quando seus dados forem fornecidos na interface gráfica correspondente. Para isso, dê um duplo clique no botão de comandos, na parte referente ao design da interface. Esse mesmo efeito poderia ter sido obtido manualmente no XAML, chamando o evento **Click** no botão:

```
<Button Grid.Row="8" Grid.Column="1"
        Name="incluirClienteButton"
        Content="Incluir Cliente"
        HorizontalAlignment="Left"
        Width="100"
        Click="incluirClienteButton_Click">
    <Button.ToolTip>
        <StackPanel Width="150" Height="20"
                    Background="Beige">
            <TextBlock>
                Permite incluir um cliente
            </TextBlock>
        </StackPanel>
    </Button.ToolTip>
</Button>
```

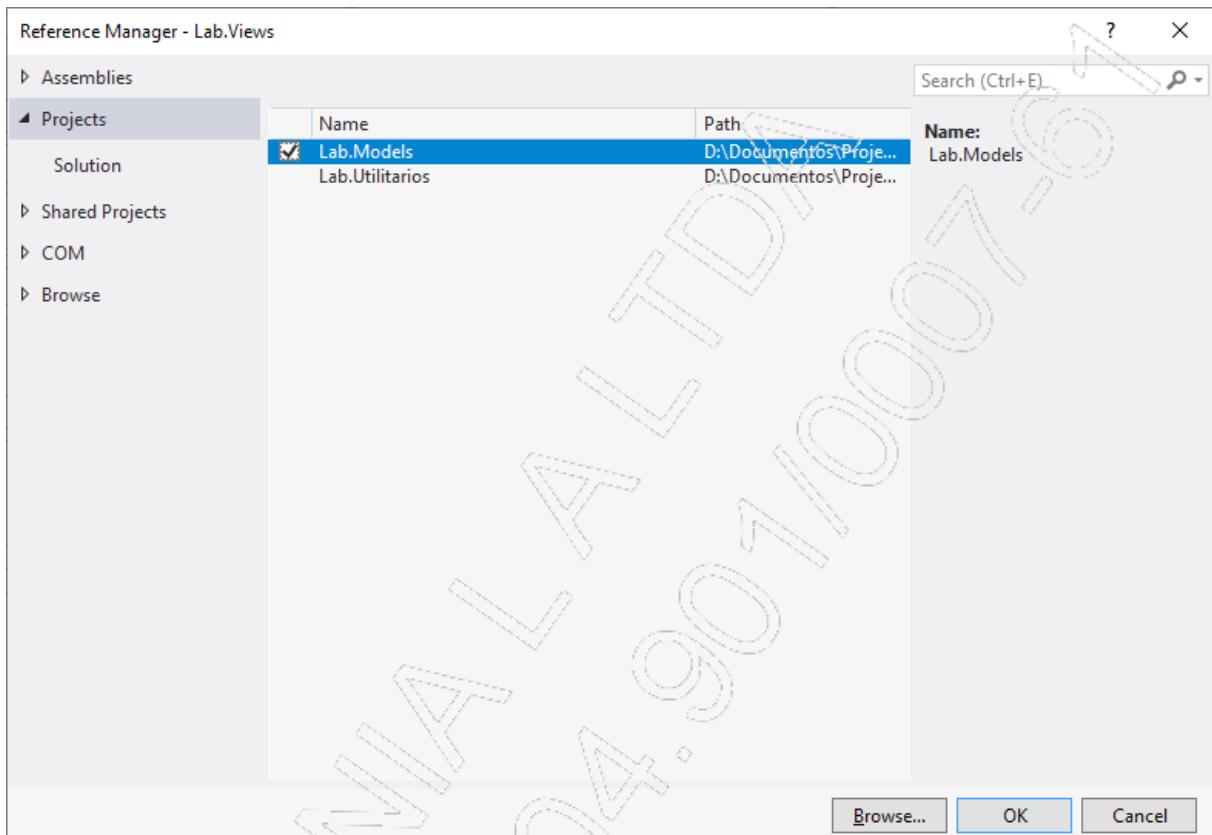
O método a ser executado pelo evento **Click** está na classe **MainWindow**, presente no arquivo **MainWindow.xaml.cs**:

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void incluirClienteButton_Click(object sender,
RoutedEventArgs e)
        {
        }
    }
}
```

## Programando com C#

2. Neste método, obtenha as informações dos campos de entrada da interface gráfica, atribua-las adequadamente a propriedades da estrutura **Endereco** e da classe **Cliente**. Em seguida, apresente seus novos dados para o usuário em uma caixa de mensagens, através da execução do método **Exibir** que escrevemos anteriormente. Para usar as classes definidas no projeto **Lab.Models** na nossa aplicação, este deve ser referenciado. Portanto, no projeto **Lab.Views**, adicione uma referência ao projeto **Lab.Models**:



Após essa referência, podemos usar suas classes:

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

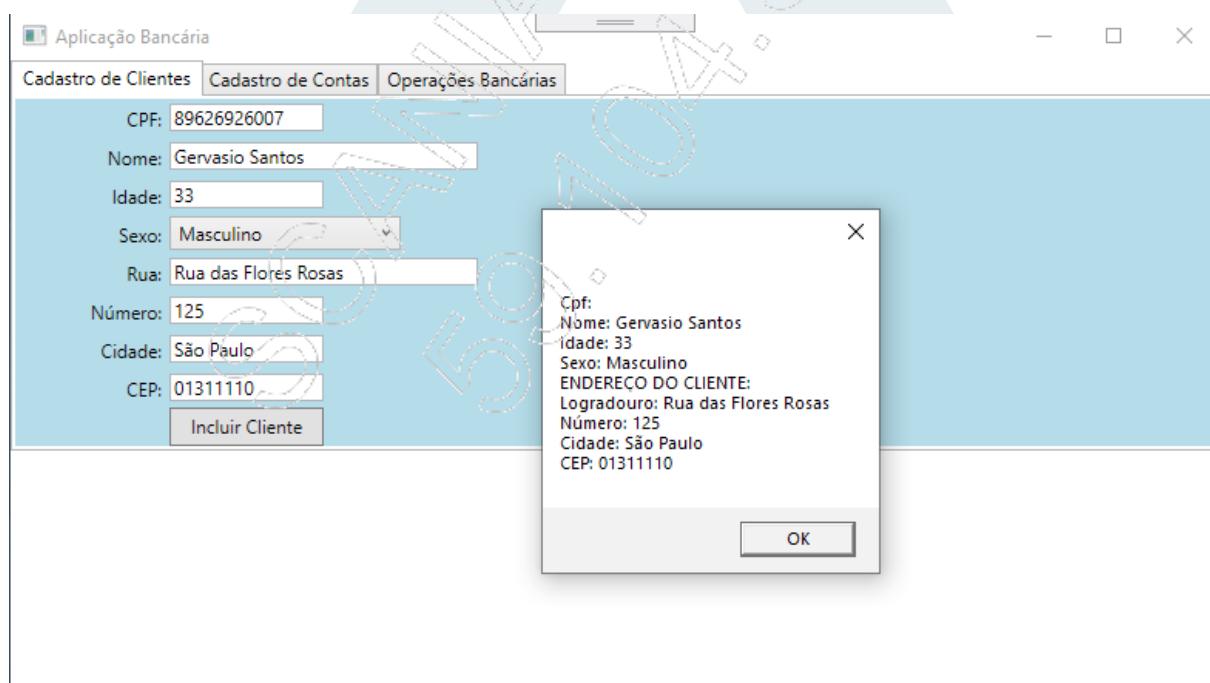
            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
        }
    }
}
```

```
        private void incluirClienteButton_Click(object sender,
RoutedEventArgs e)
{
    //obtendo os dados do endereço
    Endereco endereco = new Endereco()
    {
        Logradouro = ruaTextBox.Text,
        Numero = int.Parse(numeroTextBox.Text),
        Cidade = cidadeTextBox.Text,
        Cep = cepTextBox.Text
    };

    //obtendo os dados do cliente
    Cliente cliente = new Cliente()
    {
        Nome = nomeTextBox.Text,
        Idade = int.Parse(idadeTextBox.Text),
        Sexo = (Sexos)sexoComboBox.SelectedItem,
        EnderecoResidencial = endereco
    };

    MessageBox.Show(cliente.Exibir());
}
}
```

### 3. Execute a aplicação:

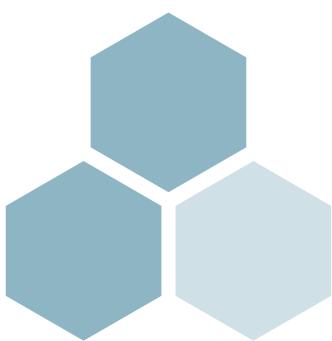




# 4

# Construtores e destrutores

- ◆ Construtor default;
- ◆ Regras sobre construtores;
- ◆ Construtores personalizados (com parâmetros);
- ◆ Sobrecarga de construtores;
- ◆ Destrutores.



## 4.1. Introdução

Na vida real, um objeto é criado sob certas circunstâncias. Por exemplo, quando uma camisa é fabricada, ela já é entregue pronta para o usuário consumi-la. Não faria sentido se a camisa fosse comprada com a matéria prima bruta, senão o produto (objeto) seria inviabilizado. O mesmo ocorre com os objetos na linguagem C#. Como objeto, ele deve ser criado (instanciado) com condições adequadas, de acordo com seu objetivo. Os construtores são usados para essa finalidade.

Estudaremos, neste capítulo, os critérios para que um objeto, ao ser criado, contenha as condições apropriadas para ser utilizado.

## 4.2. Construtor default

Um construtor é um método executado automaticamente toda vez que instanciamos uma classe, através do operador **new**.

No capítulo sobre Orientação a Objetos, nós chegamos a usar construtores diversas vezes. Agora, chegou a hora de detalhá-los.

Considere a classe **Pessoa**:

```
public class Pessoa
{
    public string Nome { get; set; }
    public int Idade { get; set; }
    public double Peso { get; set; }
    public double Altura { get; set; }

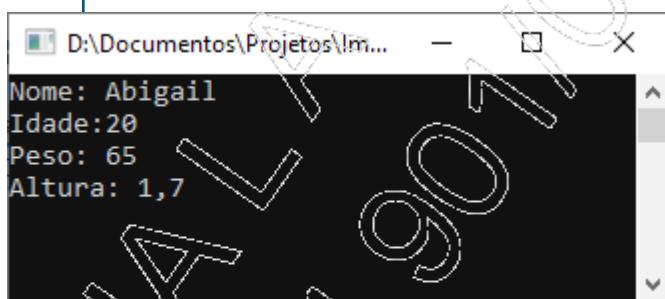
    public string GetPessoa()
    {
        return $"Nome: {this.Nome}\nIdade:{this.Idade}" +
            $"\\nPeso: {this.Peso}\\nAltura: {this.Altura}";
    }
}
```

Na aplicação, ao instanciar a classe, temos:

```
class Program
{
    static void Main(string[] args)
    {
        Pessoa pessoa = new Pessoa();
        pessoa.Nome = "Abigail";
        pessoa.Idade = 20;
        pessoa.Peso = 65;
        pessoa.Altura = 1.70;

        Console.WriteLine(pessoa.GetPessoa());
        Console.ReadKey();
    }
}
```

Sua execução produz o seguinte resultado:



No código anterior, o construtor é o seguinte método destacado:

```
Pessoa pessoa = new Pessoa();
```

Observe que é o nome da classe seguido de parênteses, o que caracteriza um método. E esse método não possui parâmetros, e nada foi escrito na classe a respeito de construtores. Isso significa que estamos usando o **construtor default**, ou seja, aquele fornecido pela própria linguagem C#.

Isso nos leva às regras do construtor.

## 4.3. Regras sobre construtores

As seguintes regras são aplicáveis a um construtor:

1. Seu nome é sempre igual ao da classe.
2. Toda classe tem um construtor, necessariamente.
3. Se nenhum construtor for fornecido explicitamente, o compilador insere o construtor default.
4. A partir do momento em que incluirmos pelo menos um construtor na classe, o compilador passa a utilizá-lo, e o construtor default é ignorado.
5. Assim como qualquer outro método, podemos ter construtores sobrecarregados.
6. Ele não pode retornar um valor, incluindo **void**.
7. O único momento em que o construtor é executado é na criação do objeto, através do operador **new**.

Outras regras sobre construtores serão apresentadas no capítulo sobre Herança e Polimorfismo.

## 4.4. Construtores personalizados (com parâmetros)

O principal papel de um construtor é definir o estado inicial do objeto. Sendo assim, quando o construtor default é utilizado, as propriedades também assumem valores default, de acordo com seu tipo.

A seguir, temos uma tabela em que estão listados os valores padrão dos tipos de valor retornados por construtores padrão:

Tipo de valor	Valor padrão
<b>bool</b>	false
<b>byte</b>	0
<b>char</b>	\0
<b>decimal</b>	0.0M
<b>double</b>	0.0D
<b>enum</b>	Valor resultante da expressão (E)0, em que E é o identificador de <b>enum</b> .
<b>float</b>	0.0F
<b>int</b>	0
<b>long</b>	0L



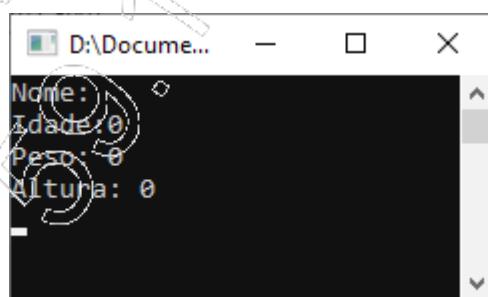
Tipo de valor	Valor padrão
<b>sbyte</b>	0
<b>short</b>	0
<b>struct</b>	Valor resultante quando todos os campos de tipo por valor são configurados para seu valor padrão e todos os campos de tipo por referência são configurados para <b>null</b> .
<b>uint</b>	0
<b>ulong</b>	0
<b>ushort</b>	0

Analisando o código apresentado no início deste capítulo, podemos perceber que não tem nada que nos obrigue a fornecer valores para as propriedades. Sendo assim, se tivermos o código na aplicação...

```
class Program
{
    static void Main(string[] args)
    {
        Pessoa pessoa = new Pessoa();
        Console.WriteLine(pessoa.GetPessoa());

        Console.ReadKey();
    }
}
```

...teríamos como resultado:



Apesar de o programa ter funcionado perfeitamente, os valores das propriedades não fazem sentido para o objeto analisado. Não existem pessoas com peso e/ou altura zero. Isso significaria que tal pessoa não existe!

Nesse caso, devemos interferir na forma como o objeto é criado, já que, como já dito, não somos obrigados a fornecer valores para as propriedades no programa.

Interferimos na criação do objeto, definindo nossos próprios construtores e, nesse caso, fornecendo parâmetros (pelo menos os parâmetros necessários). A classe **Pessoa** ficará da seguinte forma:

```
public class Pessoa
{
    public Pessoa(string nome, int idade, double peso, double altura)
    {
        this.Nome = nome;
        this.Idade = idade;
        this.Peso = peso;
        this.Altura = altura;
    }

    public string Nome { get; set; }
    public int Idade { get; set; }
    public double Peso { get; set; }
    public double Altura { get; set; }

    public string GetPessoa()
    {
        return $"Nome: {this.Nome}\nIdade:{this.Idade}" +
            $"\\nPeso: {this.Peso}\\nAltura: {this.Altura}";
    }
}
```

Seguimos os padrões estabelecidos pelas regras dos construtores. Com a inclusão deste construtor, o programa deixa de funcionar:

```
class Program
{
    static void Main(string[] args)
    {
        Pessoa pessoa = new Pessoa();
        Console.WriteLine(pessoa.GetPessoa());
        Console.ReadKey();
    }
}
```



O construtor default foi ignorado e, a partir de agora, a única forma de instanciarmos a classe **Pessoa** é através do construtor que adicionamos na classe:

```
class Program
{
    static void Main(string[] args)
    {
        Pessoa pessoa = new Pessoa("Abigail", 20, 65, 1.7);
        Console.WriteLine(pessoa.GetPessoa());

        Console.ReadKey();
    }
}
```

## 4.5. Sobrecarga de construtores

Existem situações em que nem todas as propriedades de uma classe são requeridas. No caso da classe **Pessoa**, apenas o peso e a altura são críticos. Outras propriedades, como é o caso da idade, podem ser omitidas.

Em casos como esse, podemos especificar mais de um construtor, ou seja, podemos sobrecarregar os construtores. Veja a nova classe **Pessoa**:

```
public class Pessoa
{
    public Pessoa(double peso, double altura)
    {
        this.Peso = peso;
        this.Altura = altura;
    }

    public Pessoa(string nome, double peso, double altura)
    {
        this.Nome = nome;
        this.Peso = peso;
        this.Altura = altura;
    }

    public Pessoa(int idade, double peso, double altura)
    {
        this.Idade = idade;
        this.Peso = peso;
        this.Altura = altura;
    }

    public Pessoa(string nome, int idade, double peso, double altura)
    {
        this.Nome = nome;
        this.Idade = idade;
        this.Peso = peso;
        this.Altura = altura;
    }
}
```

```
public string Nome { get; set; }
public int Idade { get; set; }
public double Peso { get; set; }
public double Altura { get; set; }

public string GetPessoa()
{
    return $"Nome: {this.Nome}\nIdade:{this.Idade}" +
        $"\\nPeso: {this.Peso}\\nAltura: {this.Altura}";
}
```

Agora temos diversas opções de construtores, mas todas elas requerem o peso e a altura que, como já mencionado, são as propriedades críticas para essa classe.

Mesmo com a sobrecarga de construtores, tivemos uma repetição de código. O ideal é que um construtor se beneficie do código do outro, ou seja, um construtor pode chamar o outro, de forma a aproveitar o código. A forma de fazermos isso é usando a palavra **this** como método, da seguinte forma:

```
public class Pessoa
{
    public Pessoa(double peso, double altura)
    {
        this.Peso = peso;
        this.Altura = altura;
    }

    public Pessoa(string nome, double peso, double altura)
        : this(peso, altura)
    {
        this.Nome = nome;
    }

    public Pessoa(int idade, double peso, double altura)
        : this(peso, altura)
    {
        this.Idade = idade;
    }

    public Pessoa(string nome, int idade, double peso, double altura)
        : this(idade, peso, altura)
    {
        this.Nome = nome;
    }
}
```



```
public string Nome { get; set; }
public int Idade { get; set; }
public double Peso { get; set; }
public double Altura { get; set; }

public string GetPessoa()
{
    return $"Nome: {this.Nome}\nIdade:{this.Idade}" +
        $"\\nPeso: {this.Peso}\\nAltura: {this.Altura}";
}
```

Um construtor padrão também pode ser criado por meio da adição de um método público, com o mesmo nome da classe e sem retornar um valor.

Assim como métodos e campos, os construtores podem ser públicos (se inserirmos a palavra-chave **public**) ou privados (se não inserirmos palavra-chave alguma), o que determina se o construtor poderá ser utilizado fora da classe ou não. Além disso, também podem ser definidos como **protected**, **internal** ou **protected internal**.

Além dos construtores de classes, temos o construtor para tipos de estruturas (**struct**), o qual é criado automaticamente pelo compilador quando a estrutura é instanciada com o operador **new**. Sua função é inicializar os campos na estrutura para os valores padrão.

Estruturas não possuem construtores padrão explícitos, já que seus construtores são criados automaticamente.

Assim como os construtores das classes, os construtores das estruturas podem ter parâmetros, se chamados por meio de uma instrução **new** ou de uma instrução base. Além disso, construtores de estruturas também podem ser sobre carregados.

Já a palavra-chave **static** faz com que o construtor seja declarado como estático. Esse tipo de construtor é chamado sempre que um campo estático é acessado. Sua principal função é inicializar membros de classe estática.

## 4.6. Destrutores

Da mesma forma que usamos construtores para criar objetos, usamos os destrutores para liberar recursos que porventura tenham sido consumidos pelo objeto durante seu ciclo de vida.

Os destrutores possuem mais limitações que os construtores:

- Não podem ser definidos em estruturas;
- Existem somente em classes;
- Não podem ser sobre carregados;
- Não podem receber modificadores de acesso;
- Não recebem parâmetros;
- São chamados automaticamente quando um objeto é descartado.

Para exemplificar, vamos novamente considerar a classe **Pessoa**:

```
public class Pessoa
{
    //construtores omitidos por brevidade

    ~Pessoa()
    {
        Console.WriteLine("Objeto sendo finalizado");
        Console.ReadKey();
    }

    public string Nome { get; set; }
    public int Idade { get; set; }
    public double Peso { get; set; }
    public double Altura { get; set; }

    public string GetPessoa()
    {
        return $"Nome: {this.Nome}\nIdade:{this.Idade}" +
               "\nPeso: {this.Peso}\nAltura: {this.Altura}";
    }
}
```

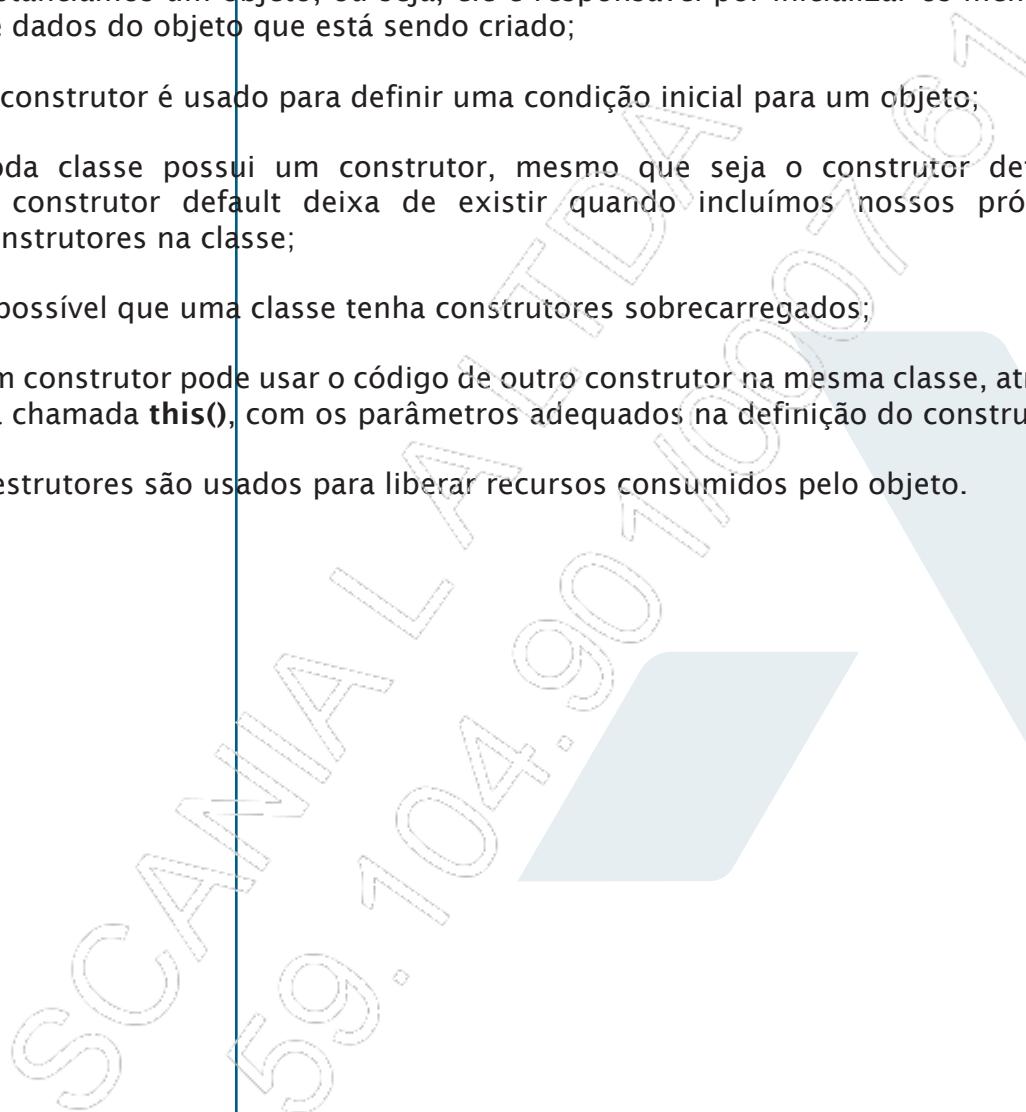
Nesse caso, o destrutor é chamado quando o programa que instancia o objeto é finalizado.



# Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Um construtor é um método executado automaticamente toda vez que instanciamos um objeto, ou seja, ele é responsável por inicializar os membros de dados do objeto que está sendo criado;
- O construtor é usado para definir uma condição inicial para um objeto;
- Toda classe possui um construtor, mesmo que seja o construtor default. O construtor default deixa de existir quando incluímos nossos próprios construtores na classe;
- É possível que uma classe tenha construtores sobrecarregados;
- Um construtor pode usar o código de outro construtor na mesma classe, através da chamada `this()`, com os parâmetros adequados na definição do construtor;
- Destruidores são usados para liberar recursos consumidos pelo objeto.



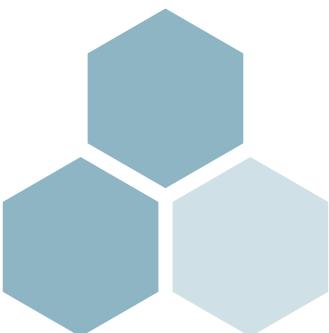
SCANALAL ALTA  
59.704.907/0007-67



4

# Construtores e destrutores

Teste seus conhecimentos



**1. O método construtor é executado exclusivamente por qual operador?**

- a) override
- b) this
- c) new
- d) base
- e) virtual

**2. Qual das alternativas é correta com relação a construtores?**

- a) Uma classe pode ter diversos construtores, desde que tenham nomes diferentes.
- b) Uma classe pode ter apenas um construtor, e não pode ter parâmetros.
- c) Uma classe pode ter vários construtores, mas todos devem ser declarados como void.
- d) Uma classe pode ter vários construtores, e todos devem ter o mesmo nome da classe.
- e) Uma classe pode ter vários construtores, desde que o construtor padrão seja void.

**3. Qual a alternativa que completa adequadamente a frase a seguir?**

O método \_\_\_\_\_ é responsável pela criação e inicialização de um objeto de uma determinada classe.

- a) override
- b) create
- c) Init
- d) construtor
- e) overload

## 4. Marque a alternativa ERRADA:

- a) A partir de um construtor, podemos chamar métodos estáticos.
- b) Em um construtor podemos declarar variáveis locais.
- c) Os construtores podem ter vários parâmetros.
- d) Podemos chamar um construtor em qualquer parte do programa, desde que usemos o nome da classe como um método.
- e) A definição do construtor não considera o tipo de retorno.

## 5. Quando não escrevemos explicitamente um construtor na classe:

- a) O programa não compila.
- b) O compilador inclui um construtor default.
- c) A classe inclui um construtor com o número de parâmetros correspondente ao número de atributos da classe.
- d) A classe permanece sem construtor e, dessa forma, nenhum objeto pode ser criado.
- e) Não podemos definir nossos próprios construtores na classe.





4

# Construtores e destrutores



Mãos à obra!



Editora  
**IMPACTA**



A partir deste capítulo, aproveitaremos as classes desenvolvidas no Capítulo 3. Criaremos um novo solution, copiaremos os projetos mencionados a partir de capítulos anteriores e complementaremos cada projeto com os temas do capítulo atual.

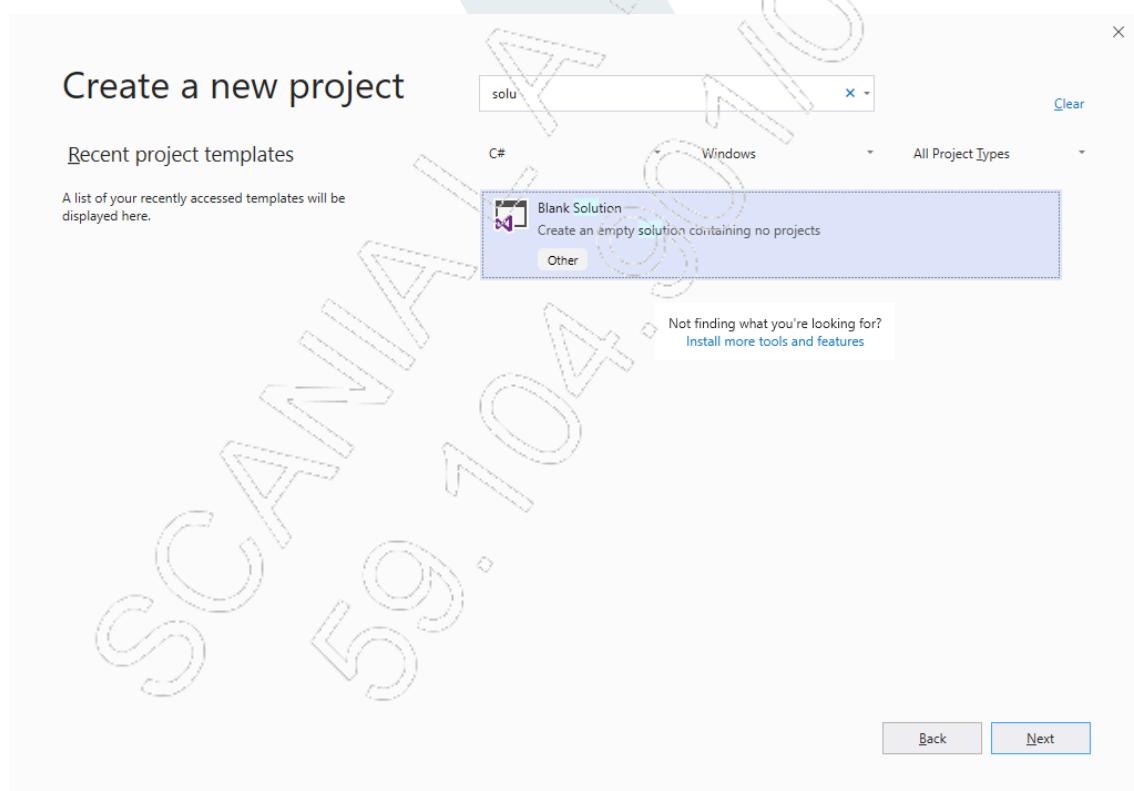
## Laboratório 1

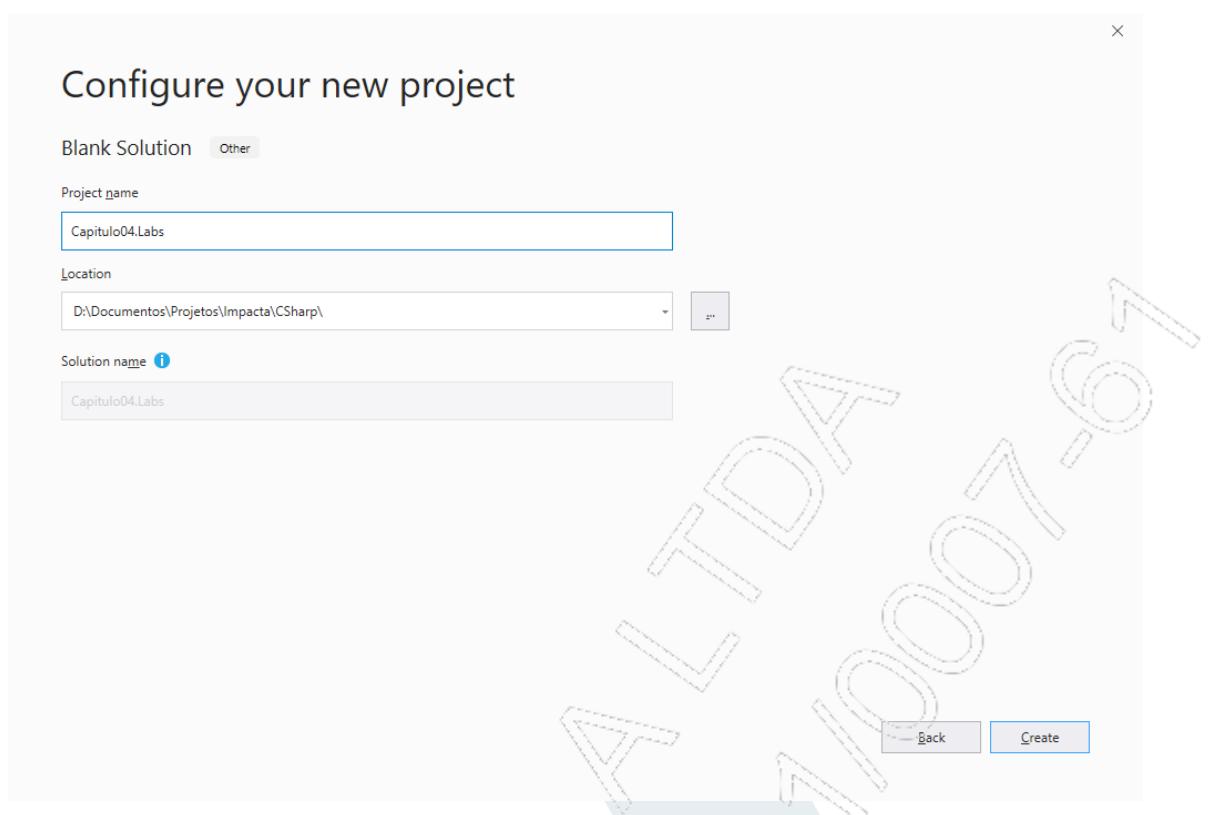
### A – Adicionando construtores personalizados

#### Objetivos:

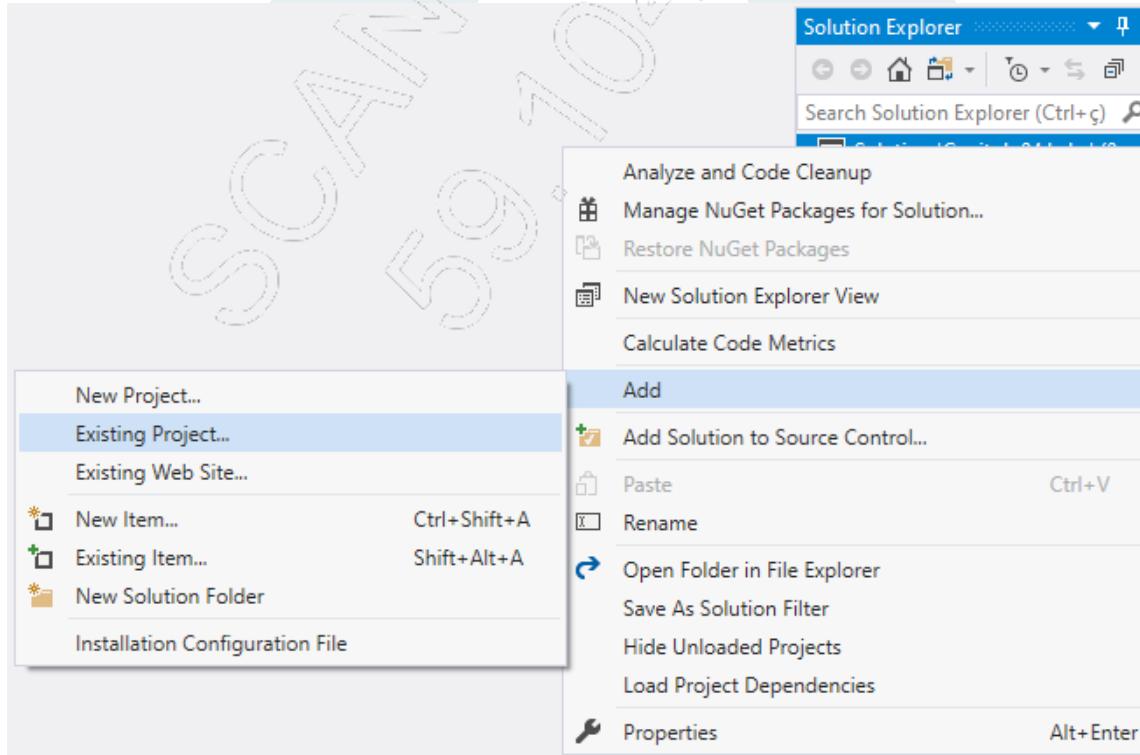
Criar um novo solution, copiar os projetos do capítulo anterior e atualizar suas informações. No nosso caso, adicionaremos construtores personalizados nas classes **Cliente**, **ContaCorrente** e na estrutura **Endereco**.

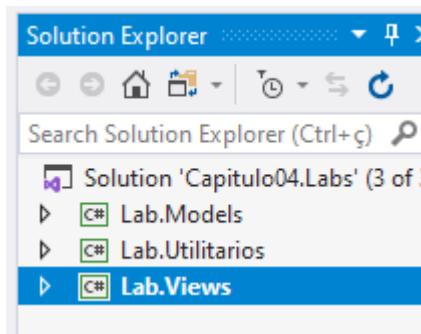
1. Crie um solution vazio chamado **Capitulo04.Labs**:





2. Na pasta correspondente ao novo solution criado, copie os projetos **Lab.Models**, **Lab.Utilitarios** e **Lab.Views**, desenvolvidos no Capítulo 3;
3. Após a cópia, adicione os três projetos no solution criado no item 1, **Capítulo04.Labs**;





4. Altere a estrutura **Endereco**, adicionando um construtor para receber valores para todas as propriedades:

```
namespace Lab.Models
{
    public struct Endereco
    {
        public string Logradouro { get; set; }
        public int Numero { get; set; }
        public string Cidade { get; set; }
        public string Cep { get; set; }

        public Endereco(string Logradouro, int Numero, string Cidade,
string Cep)
        {
            this.Logradouro = Logradouro;
            this.Numero = Numero;
            this.Cidade = Cidade;
            this.Cep = Cep;
        }

        public string Exibir()
        {
            return $"Logradouro: {this.Logradouro}\n" +
                $"Número: {this.Numero}\n" +
                $"Cidade: {this.Cidade}\n" +
                $"CEP: {this.Cep}";
        }
    }
}
```

5. Altere a classe **Cliente**, adicionando os construtores:

```
namespace Lab.Models
{
    public class Cliente
    {
        private string _cpf;
        public string Cpf
        {
            get => _cpf;
            set => _cpf = (value.ValidarCPF() ? value :
                throw new Exception("CPF inválido"));
        }

        public string Nome { get; set; }
        public int Idade { get; set; }
        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public ContaCorrente Conta { get; set; }

        public Cliente(string Cpf, string Nome, Sexos Sexo)
        {
            this.Cpf = Cpf;
            this.Nome = Nome;
            this.Sexo = Sexo;
        }

        public Cliente(string Cpf, string Nome, Sexos Sexo, Endereco
endereco)
            : this(Cpf, Nome, Sexo)
        {
            this.EnderecoResidencial = endereco;
        }

        public string Exibir()
        {
            return $"Cpf: {this.Cpf}\n" +
                $"Nome: {this.Nome}\n" +
                $"Idade: {this.Idade}\n" +
                $"Sexo: {this.Sexo}\n" +
                $"ENDEREÇO DO CLIENTE:\n" +
                $"{this.EnderecoResidencial.Exibir()}";
        }
    }
}
```

# Programando com C#

6. Altere a classe **ContaCorrente**, adicionando os construtores indicados:

```
namespace Lab.Models
{
    public class ContaCorrente
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }
        public double Saldo { get; private set; }

        public Cliente ClienteInfo { get; set; }

        public ContaCorrente(int Banco, string Agencia, string Conta)
        {
            this.NumeroBanco = Banco;
            this.NumeroAgencia = Agencia;
            this.NumeroConta = Conta;
        }

        public Cliente(string Cpf, string Nome, Sexos Sexo, int idade)
            : this(Cpf, Nome, Sexo)
        {
            this.Idade = Idade;
        }

        public ContaCorrente(int Banco, string Agencia, string Conta,
                            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public void EfetuarOperacao(double valor,
                                    Operacoes operacao = Operacoes.Deposito)
        {
            switch (operacao)
            {
                case Operacoes.Deposito:
                    this.Saldo += valor;
                    break;
                case Operacoes.Saque:
                    this.Saldo -= valor;
                    break;
            }
        }

        public string Exibir()
        {
            string cliente = this.ClienteInfo != null ?
                this.ClienteInfo.Exibir() + '\n' : "";
            return $"{cliente}" +
                $"Banco: {this.NumeroBanco}\n" +
                $"Agência: {this.NumeroAgencia}\n" +
                $"Conta: {this.NumeroConta}\n" +
                $"Saldo Atual: {this.Saldo}";
        }
    }
}
```

7. Adicione o destrutor na classe Cliente:

```
namespace Lab.Models
{
    public class Cliente
    {
        private string _cpf;
        public string Cpf
        {
            get => _cpf;
            set => _cpf = (value.ValidarCPF() ? value :
                throw new Exception("CPF inválido"));
        }

        public string Nome { get; set; }
        public int Idade { get; set; }
        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public ContaCorrente Conta { get; set; }

        public Cliente(string Cpf, string Nome, Sexos Sexo)
        {
            this.Cpf = Cpf;
            this.Nome = Nome;
            this.Sexo = Sexo;
        }

        public Cliente(string Cpf, string Nome, Sexos Sexo, int idade)
            : this(Cpf, Nome, Sexo)
        {
            this.Idade = idade;
        }

        public Cliente(string Cpf, string Nome, Sexos Sexo, Endereco endereco)
            : this(Cpf, Nome, Sexo)
        {
            this.EnderecoResidencial = endereco;
        }

        ~Cliente()
        {
            if(this.Conta != null)
            {
                this.Conta = null;
            }
        }

        public string Exibir()
        {
            return $"Cpf: {this.Cpf}\n" +
                $"Nome: {this.Nome}\n" +
                $"Idade: {this.Idade}\n" +
                $"Sexo: {this.Sexo}\n" +
                $"ENDEREÇO DO CLIENTE:\n" +
                $"{this.EnderecoResidencial.Exibir()}";
        }
    }
}
```

8. Atualize o evento Click do botão de comando `incluirClienteButton`, em `MainWindow.xaml.cs`:

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
        }

        private void incluirClienteButton_Click(object sender,
RoutedEventArgs e)
        {
            //obtendo os dados do endereço
            Endereco endereco = new Endereco(
                ruaTextBox.Text,
                int.Parse(numeroTextBox.Text),
                cidadeTextBox.Text,
                cepTextBox.Text);

            //obtendo os dados do cliente
            Cliente cliente = new Cliente(
                cpfTextBox.Text,
                nomeTextBox.Text,
                (Sexos)sexoComboBox.SelectedItem,
                int.Parse(idadeTextBox.Text));

            cliente.EnderecoResidencial = endereco;

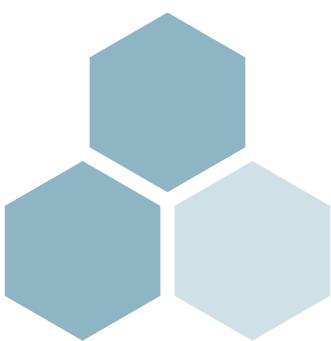
            MessageBox.Show(cliente.Exibir());
        }
    }
}
```

9. Execute a aplicação.

# 5

# Herança e polimorfismo

- ◆ Herança de classes;
- ◆ Acesso à superclasse com o operador base;
- ◆ Polimorfismo;
- ◆ Tipos de classes;
- ◆ Chamada a métodos polimórficos.



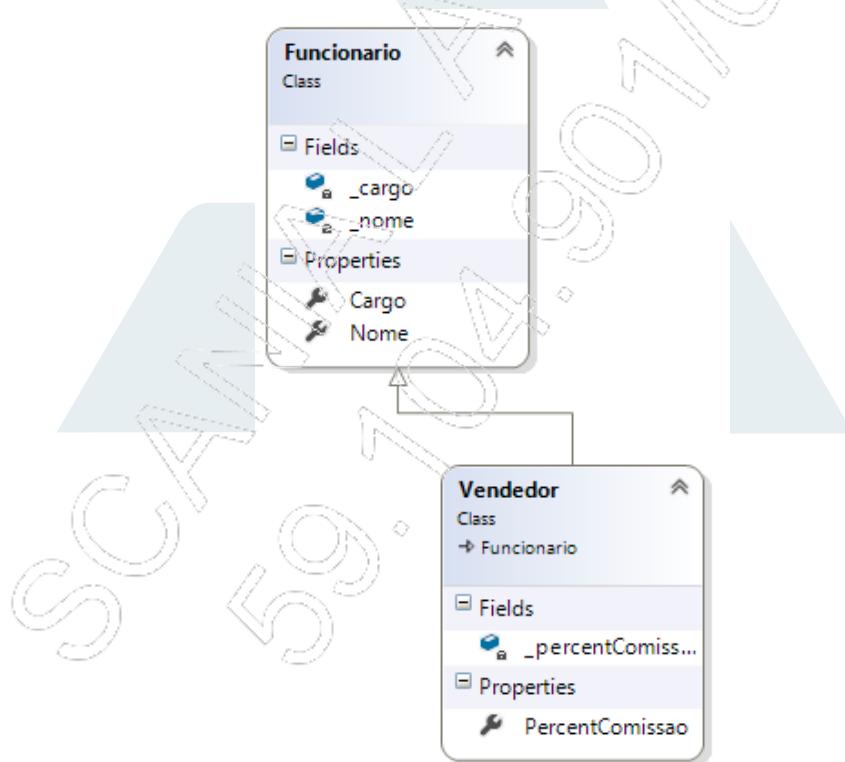
## 5.1. Introdução

O mecanismo de herança é um dos maiores diferenciais entre a programação orientada a objetos e outros tipos de programação, como a programação estruturada. Por meio da herança, uma classe herda os atributos e métodos de outra. A classe que herda as características se chama classe filha (ou subclasse ou classe derivada) e a classe que fornece as características se chama classe pai (ou superclasse ou classe base).

O mecanismo de herança pressupõe uma estrutura hierárquica de classes. Quando temos uma hierarquia de classes planejada de forma adequada, temos a base para que um código possa ser reutilizado, estendido, modificado, o que permite poupar esforço e tempo no desenvolvimento.

## 5.2. Herança de classes

A herança possibilita às classes compartilharem métodos e propriedades, dentre outros membros de classe entre si. Para que haja herança, deve haver um relacionamento entre as classes:



- **Classe base ou superclasse:** A classe que concede as características à outra classe;
- **Classe derivada ou subclasse:** A classe que herda as características da classe superclasse.

Na herança, a subclasse é uma implementação mais específica da superclasse. Ela define, portanto, suas características particulares.

Para que algum membro da superclasse fique visível apenas para suas subclasses, podemos usar a palavra-chave **protected**. Assim, apenas as classes derivadas terão acesso a esses membros, enquanto as classes que não fizerem parte da hierarquia não poderão acessá-los.

A seguinte sintaxe é utilizada na criação de uma herança:

```
class ClasseDerivada : ClasseBase  
{  
    Implementação ...  
}
```

O exemplo a seguir ilustra a criação efetiva de uma herança. Considere a classe **Funcionario**:

```
class Funcionario  
{  
    private string _nome;  
    public string Nome  
    {  
        get => _nome;  
        set => _nome = value;  
    }  
  
    private string _cargo;  
    public string Cargo  
    {  
        get => _cargo;  
        set => _cargo = value;  
    }  
}
```

Vamos considerar agora uma classe chamada **Vendedor**. Entendemos que cada vendedor é um funcionário, mas nem todo funcionário é um vendedor. Essa relação nos obriga a criar uma extensão da classe **Funcionario** a fim de que ela contenha os dados específicos dos vendedores.

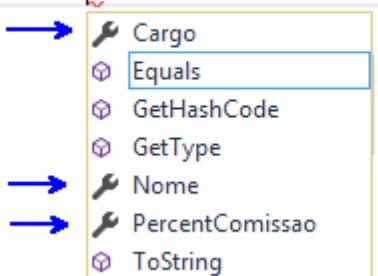
Desse modo, a subclasse **Vendedor** herda os membros de **Funcionario**:

```
//Definição da herança  
class Vendedor : Funcionario  
{  
    public double PercentComissao { get; set; }  
}
```

O resultado desse código é o seguinte:

```
Vendedor vend1 = new Vendedor();
```

```
vend1.
```



No C#, não é definido o conceito de herança múltipla, ou seja, uma classe não pode ser subclasse de duas ou mais superclasses simultaneamente.

## 5.3. Acesso à superclasse com o operador base

Assim como a referência **this** aponta para a própria classe, a referência **base** aponta para a superclasse. Apesar de não ser um procedimento recomendado, podemos manter atributos de nomes iguais na superclasse e na subclasse. Esse procedimento é chamado de **hiding**. Quando o utilizamos, devemos adicionar a instrução **new** à frente do atributo da subclasse. Caso contrário, o compilador irá enviar um alerta sobre o uso de **hiding**.

Veja um exemplo de código:

```
class Automovel
{
    public string Fabricante = "GM";
}

class AutomovelLuxo : Automovel
{
    public new string Fabricante = "Mercedez-Benz";

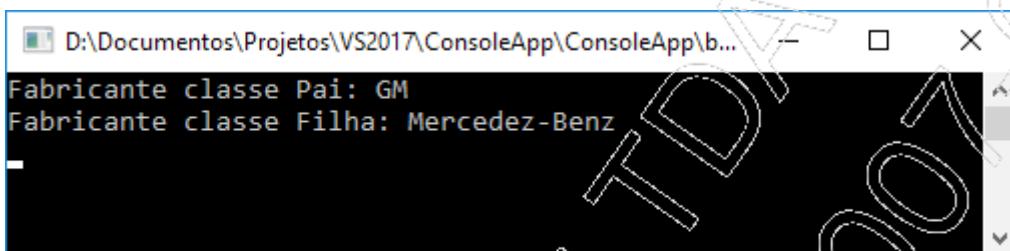
    public string ExibirExemplo()
    {
        return
            "Fabricante superclasse: " + base.Fabricante +
            "\nFabricante subclasse: " + this.Fabricante;
    }
}
```

Agora, no código:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    AutomovelLuxo carro = new AutomovelLuxo();

    exemploLabel.Text = carro.ExibirExemplo();
}
```

O resultado desse código pode ser visto a seguir:



## 5.4. Polimorfismo

O polimorfismo é um mecanismo que confere a um mesmo objeto a capacidade de se comportar de diferentes maneiras, dependendo do contexto em que ele esteja sendo empregado.

Assim, se um objeto A for criado a partir de uma classe X, ele se comportará de uma determinada maneira. Se esse mesmo objeto A for criado a partir de uma classe Y, ele se comportará de uma maneira diferente. E quando for necessário, o mesmo objeto A poderá converter-se em X ou Y, conforme a necessidade do desenvolvedor, para atender a uma demanda específica.

- **Implementação do polimorfismo**

Polimorfismo ocorre quando classes derivadas de uma única classe base são capazes de invocar métodos que, embora apresentem a mesma assinatura, comportam-se de forma diferente para cada uma das classes derivadas.

É importante destacar que a assinatura de um método diz respeito somente ao seu nome e aos tipos e números de seus parâmetros, aspectos esses que, caso sejam os mesmos em métodos diferentes, levarão esses métodos a terem a mesma assinatura, ainda que seus tipos de retorno possam ser diferentes.

Como exemplo de polimorfismo, considere uma classe chamada **Cliente** e outra chamada **Funcionario** que têm como base uma classe chamada **Pessoa** com um método chamado **EnviarEmail()**.

Se esse método (definido na classe base) se comportar de maneira diferente, dependendo do fato de ter sido chamado a partir de uma instância de **Cliente** ou a partir de uma instância de **Funcionario**, será considerado um método polimórfico, ou seja, um método de várias formas. Esse princípio é chamado polimorfismo.

- **Palavras-chave virtual e override**

Com a utilização da palavra-chave **virtual** aplicada a métodos, propriedades, eventos e indexadores, determinamos que estes membros permitem serem sobreescritos em uma subclasse. Por meio da palavra-chave **override**, determinamos que, na classe derivada, um membro virtual da classe base está sendo sobreescrito.

Agora que já descrevemos a utilidade das palavras-chave **virtual** e **override**, devemos ter em mente que ambas complementam uma à outra.

É importante saber, ainda, que a propagação da palavra-chave **virtual** ocorre para seus descendentes. Um método ou propriedade virtual pode ser sobreescrito em descendentes e até mesmo em uma classe derivada.

Existem algumas regras às quais devemos obedecer quando trabalhamos com **virtual** e **override**:

- Devemos nos certificar de que ambos os métodos têm o mesmo acesso, lembrando que um método privado não pode ser declarado com uma dessas palavras-chave;
- Quanto às assinaturas de método, as duas devem ser idênticas (mesmo nome e mesmos tipos e números de parâmetros);
- A redefinição só pode ser feita em métodos virtuais, o que é definido na classe base;
- É necessário que a classe derivada declare o método usando a palavra-chave **override** para que a redefinição do método da classe base de fato ocorra;
- Não devemos declarar explicitamente um método **override** com a palavra-chave **virtual**; esse tipo de método já é implicitamente virtual, sendo permitida sua redefinição em uma classe derivada posterior.

Veja o exemplo adiante:

- Na classe base, o método **ExibirDados()** retorna o nome e o preço do produto:

```
class Produto
{
    public string Nome { get; set; }
    public decimal Preco { get; set; }

    //Concede a permissão de sobreescrita
    public virtual string ExibirDados()
    {
        return string.Format(
            "Nome: {0}\nPreço: {1}",
            Nome.ToUpper(), Preco.ToString("C2"));
    }
}
```

- Na classe derivada, o método **ExibirDados()** retorna o nome, o preço e o número de páginas do livro:

```
class Livro: Produto
{
    public short NPaginas { get; set; }

    //Sobreescriva o método
    public override string ExibirDados()
    {
        return base.ExibirDados() +
            "\nNº de Páginas: " + NPaginas.ToString();
    }
}
```

- No código:

```
static void Main(string[] args)
{
    Produto prod = new Produto();
    prod.Nome = "Jatobá";
    prod.Preco = 6.47m;

    Console.WriteLine("PRODUTO\n" + prod.ExibirDados());

    //-----

    Livro gibi = new Livro();
    gibi.Nome = "Soldado Invernal - Marcha da Amargura";
    gibi.Preco = 19.90m;
    gibi.NPaginas = 128;

    Console.WriteLine("LIVRO\n" + gibi.ExibirDados());

    Console.ReadKey();
}
```

Agora, veja o resultado desse código:



```
D:\Documentos\Projetos\VS2017\ConsoleApp\ConsoleApp\bin\Debug>
PRODUTO
Nome: JATOBÁ
Preço: R$ 6,47
LIVRO
Nome: SOLDADO INVERNAL - MARCHA DA AMARGURA
Preço: R$ 19,90
Nº de Páginas: 128
```

## 5.5. Tipos de classes

Podemos utilizar algumas palavras que mudam o comportamento padrão de uma classe. Neste tópico, serão demonstrados os modificadores **abstract**, **sealed** e **static**.

### 5.5.1. Classes e membros abstratos

**Classes abstratas** não podem ser instanciadas. São classes usadas exclusivamente como superclasses. Em regra, as subclasses deverão sobrescrever os métodos da superclasse.

A funcionalidade dos **membros abstratos** que são herdados pelas subclasses depende do seu objetivo específico, uma vez que métodos abstratos não possuem definição.

Os métodos abstratos estão presentes somente em classes abstratas, e são aqueles que não possuem implementação. A sintaxe desse tipo de método é a seguinte:

```
abstract tipo_retorno identificador;
```

No exemplo a seguir, faremos com que a classe **Figura** seja abstrata e, como ela será a classe base, outras duas classes, **Retangulo** e **Circulo**, serão suas subclasses:

```
public abstract class Figura
{
    public string Exibir()
    {
        string nomeClasse = this.GetType().Name;
        double area = this.CacularArea();

        return $"Classe: {nomeClasse}\nÁrea : {area}";
    }
    public abstract double CalcularArea();
}
```

A palavra **abstract** como prefixo da declaração da classe **Figura** a define como abstrata; contudo, quando tal palavra é utilizada como prefixo da declaração do método **CalcularArea()**, ela indica que esse é um método abstrato e, portanto, não possui uma implementação.

O exemplo a seguir mostra como o método **CalcularArea()** é implementado nas classes **Retangulo** e **Circulo**.

- **Classe Retangulo**

```
public class Retangulo : Figura
{
    public double Base { get; set; }
    public double Altura { get; set; }

    public override double CalcularArea()
    {
        return this.Base * this.Altura;
    }
}
```

Todos os métodos abstratos declarados na superclasse deverão necessariamente ser implementados na subclasse utilizando a sobrescrita deste método. A sobrescrita é feita por meio da palavra-chave **override**, que apenas pode ser utilizada em métodos e atributos virtuais. Um método abstrato é implicitamente **virtual**.

- **Classe Circulo**

```
public class Circulo : Figura
{
    public double Raio { get; set; }
    public override double CalcularArea()
    {
        return Math.PI * Math.Pow(this.Raio, 2);
    }
}
```

### 5.6. Chamada a métodos polimórficos

Na aplicação, podemos ter variáveis da superclasse e objetos da subclasse. Neste caso, os métodos sobrescritos serão chamados a partir das variáveis. Veja o exemplo de aplicação:

```
Figura figura = new Retangulo();
```

O problema desse exemplo é que não podemos chamar nenhuma propriedade específica da classe **Retangulo**, pois a variável é do tipo **Figura**. A solução para este problema é uma das duas adiante:

- Criar o objeto de forma implícita;
- Definir um construtor na classe.

Vamos definir um construtor com parâmetros nas classes **Retangulo** e **Círculo**:

- **Retangulo com construtor**

```
public class Retangulo : Figura
{
    public Retangulo() { }
    public Retangulo(double Base, double Altura)
    {
        this.Base = Base;
        this.Altura = Altura;
    }
    public double Base { get; set; }
    public double Altura { get; set; }

    public override double CalcularArea()
    {
        return this.Base * this.Altura;
    }
}
```

- Círculo com construtor

```
public class Circulo : Figura
{
    public Circulo() { }
    public Circulo(double Raio)
    {
        this.Raio = Raio;
    }

    public double Raio { get; set; }
    public override double CalcularArea()
    {
        return Math.PI * Math.Pow(this.Raio, 2);
    }
}
```

Na aplicação:

```
class Program
{
    static void Main(string[] args)
    {
        Figura fig1 = new Retangulo(2,3);

        Figura fig2 = new Retangulo()
        {
            Base = 2,
            Altura = 3
        };

        Figura fig3 = new Circulo(5);

        Figura fig4 = new Circulo()
        {
            Raio = 5
        };

        Console.WriteLine(fig1.Exibir());
        Console.WriteLine("-----");

        Console.WriteLine(fig2.Exibir());
        Console.WriteLine("-----");

        Console.WriteLine(fig3.Exibir());
        Console.WriteLine("-----");

        Console.WriteLine(fig4.Exibir());
        Console.WriteLine("-----");

        Console.ReadKey();
    }
}
```

O resultado é:

```
D:\Documentos\Projetos\Impacta\Conceitos.C...
Classe: Retangulo
Área : 6
-----
Classe: Retangulo
Área : 6
-----
Classe: Circulo
Área : 78,5398163397448
-----
Classe: Circulo
Área : 78,5398163397448
```

O problema surge quando necessitamos alterar alguma propriedade. Por exemplo: como podemos alterar o valor do **Raio** a partir da variável **fig4**? Como **fig4** é uma variável do tipo **Figura**, não temos acesso a nenhum membro específico da subclasse.

Neste caso, é necessário criar uma referência do tipo da subclasse, e fazer com que ela refencie a mesma instância, aplicando o typecast:

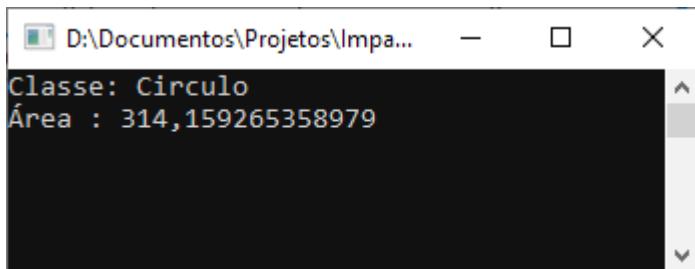
```
class Program
{
    static void Main(string[] args)
    {
        Figura fig4 = new Circulo()
        {
            Raio = 5
        };

        //Alterando o valor do raio
        Circulo circulo = (Circulo)fig4;
        circulo.Raio = 10;

        Console.WriteLine(fig4.Exibir());

        Console.ReadKey();
    }
}
```

Resultado:



## 5.6.1. Classe sealed

O modificador **sealed** define que uma classe não pode ser estendida. De forma geral, elas representam o oposto das classes abstratas.

Além dos aspectos já mencionados referentes às classes **sealed**, devemos considerar alguns outros, conforme apresentado adiante:

- Os membros (propriedades e/ou métodos) **sealed** não podem ser sobreescritos;
- O modificador **sealed** deve ser combinado com o modificador **override**. Apesar de não poder ser aplicado a membros estáticos, o modificador **sealed** pode ser aplicado a propriedades, métodos, indexadores e eventos.

Considerando que não temos uma especialização da classe **Círculo**, ela pode ser **sealed**:

```
public sealed class Circulo : Figura
{
    public double Raio { get; set; }
    public override double CalcularArea()
    {
        return Math.PI * Math.Pow(this.Raio, 2);
    }
}
```

## 5.6.2. Classe estática

As classes **static** funcionam como recipientes de campos e métodos utilitários. Uma classe marcada como **static** só pode ter membros estáticos. Seu uso é indicado nas situações em que desejamos evitar a associação de métodos a um objeto específico. Podemos utilizá-las no caso de não haver comportamento ou dados na classe que dependam da identidade do objeto.

Uma das principais características das classes estáticas é que elas são seladas, ou seja, não podem ser herdadas.

Quando o programa ou o namespace que possui a classe estática é carregado, o CLR (Common Language Runtime) do .NET Framework carrega automaticamente as classes estáticas.

O trabalho de criar uma classe estática é muito semelhante à criação de uma classe que possua somente membros estáticos. Não podemos utilizar a palavra-chave **new** para criar instâncias de uma classe estática.

Além de não poderem ser instanciadas, as classes estáticas não podem ter **Instance Constructors**. Embora possamos declarar um construtor estático para atribuir valores iniciais ou aplicar algum estado estático, a existência de construtores em classes estáticas não pode ocorrer.

**!** O compilador, além de não permitir a criação de instâncias de classes estáticas, verifica se membros de instância não foram acrescentados por acidente.

Veja o seguinte exemplo:

```
static class ClasseEstatica
{
    //Membros estáticos
    public static void Metodo()
    {
        //Implementação ...
    }
}
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- As classes podem ser estendidas. As novas classes obtidas a partir da extensão são chamadas de **subclasses**. As classes originais, em relação às subclasses, são as **superclasses**;
- Métodos podem ser executados de formas diferentes nas subclasses. Para isso, temos que sobrescrever os métodos na subclasse. O processo de sobreescrita de métodos é chamado de **polimorfismo**;
- A partir de um método, o acesso aos membros da superclasse é realizado com o operador **base**;
- Quando o objeto de uma subclasse é criado, primeiro é instanciada a superclasse e, a partir dela, a subclasse. Isso significa que o construtor da superclasse é executado antes do construtor da subclasse;
- Entendemos por **classes abstratas** as classes a partir das quais não é possível realizar qualquer tipo de instância. São classes feitas especialmente para serem modelos para suas classes derivadas. As classes derivadas, em regra, deverão sobreescrita os métodos para realizar a implementação deles. As classes derivadas das classes abstratas são conhecidas como **classes concretas**;
- O modificador **sealed** define que uma classe não pode ser uma classe base, ou seja, não pode ter herdeiras. Uma classe **sealed** está pronta para uso e deve ser empregada no programa. Devido especialmente às suas características, as classes **sealed** podem ser consideradas o oposto das classes abstratas.

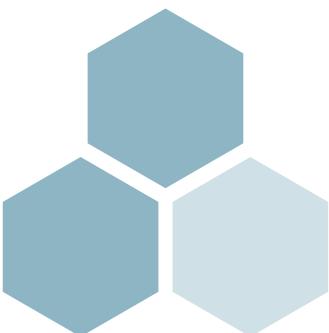




5

# Herança e polimorfismo

• Teste seus conhecimentos



**1. Para que um método seja sobreescrito na subclasse, o método na superclasse deve ser declarado como:**

- a) override
- b) static
- c) virtual
- d) void
- e) dynamic

**2. Para que uma classe admita apenas métodos estáticos, ela deve ser declarada como:**

- a) virtual
- b) public
- c) abstract
- d) static
- e) override

**3. Considere o seguinte método:**

```
public abstract double Calcular();
```

**Para que ele seja compilado sem erros, a classe que o define:**

- a) Deve ser pública.
- b) Deve ser abstrata.
- c) Deve ser uma interface.
- d) Deve ser estática.
- e) Não podemos ter métodos abstratos.

**4. Uma classe declarada como sealed tem como principal característica:**

- a) Ter todos os seus métodos sem valor de retorno.
- b) Admitir herança múltipla.
- c) Não admitir herança.
- d) Não admitir propriedades.
- e) Admitir herança somente de métodos.

**5. Considere uma classe com um método abstrato. O que acontece quando estendemos esta classe?**

- a) Devemos ter um método também abstrato na subclasse.
- b) Devemos ter propriedades abstratas na subclasse.
- c) Devemos sobrescrever o método abstrato na subclasse.
- d) Devemos redefinir este método como virtual.
- e) Nada precisa ser feito.





5

# Herança e poliformismo



Mãos à obra!

Editora

**IMPACTA**



## Laboratório 1

### A – Utilizando recursos de herança e polimorfismo

#### Objetivos:

Definir um novo solution, aproveitando os projetos do Capítulo 4, de modo a incluir os recursos da herança e do polimorfismo.

1. Crie um novo solution, vazio, chamado **Capitulo05.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models** e **Lab.Views**, do Capítulo 4, para a pasta do solution **Capitulo05.Labs**;
3. Adicione esses projetos ao novo solution;
4. No projeto **Lab.Models**, adicione a classe abstrata **Conta**. Transfira as propriedades **NumeroBanco**, **NumeroAgencia** e **NumeroConta** para essa classe. Inclua um método abstrato chamado **MostrarExtrato**. Inclua também um construtor para receber os valores dessas três propriedades como parâmetro:

```
namespace Lab.Models
{
    public abstract class Conta
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }

        public Conta(int Banco, string Agencia, string Conta)
        {
            this.NumeroBanco = Banco;
            this.NumeroAgencia = Agencia;
            this.NumeroConta = Conta;
        }

        public abstract string MostrarExtrato();
    }
}
```

5. Realize as alterações na classe **ContaCorrente**, de modo a torná-la subclasse de **Conta**. Mantenha somente as propriedades **Saldo** e **ClienteInfo**. Atualize também os construtores da nova versão da classe. Além disso, torne os métodos **EfetuarOperacao** e **Exibir** como **virtual**:

```
namespace Lab.Models
{
    public class ContaCorrente : Conta
    {
        public double Saldo { get; private set; }
        public Cliente ClienteInfo { get; set; }

        public ContaCorrente(int Banco, string Agencia, string Conta):
            base(Banco, Agencia, Conta)
        { }

        public ContaCorrente(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public virtual void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            switch (operacao)
            {
                case Operacoes.Deposito:
                    this.Saldo += valor;
                    break;
                case Operacoes.Saque:
                    this.Saldo -= valor;
                    break;
            }
        }

        public virtual string Exibir()
        {
            string cliente = this.ClienteInfo != null ?
                this.ClienteInfo.Exibir() + '\n' : "";
            return $"{cliente}" +
                $"Banco: {this.NumeroBanco}\n" +
                $"Agência: {this.NumeroAgencia}\n" +
                $"Conta: {this.NumeroConta}\n" +
                $"Saldo Atual: {this.Saldo}";
        }

        public override string MostrarExtrato()
        {
            throw new NotImplementedException();
        }
    }
}
```

6. Adicione uma nova classe chamada **ContaEspecial**, subclasse de **ContaCorrente**. Essa nova classe deve incluir uma nova propriedade chamada **Limite**, representando o limite de crédito da conta:

```
namespace Lab.Models
{
    public class ContaEspecial : ContaCorrente
    {
        public double Limite { get; set; }

        public ContaEspecial(int Banco, string Agencia, string Conta) :
            base(Banco, Agencia, Conta)
        {
        }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente, double Limite)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
            this.Limite = Limite;
        }

        //métodos sobreescritos
        public override void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            base.EfetuarOperacao(valor, operacao);
        }

        public override string Exibir()
        {
            return base.Exibir();
        }

        public override string MostrarExtrato()
        {
            return base.MostrarExtrato();
        }
    }
}
```

7. Altere o modificador de acesso da propriedade **Saldo**, na classe **ContaCorrente**, para **protected**, pois ela deverá ser acessada também na nova subclasse. Altere também o método **EfetuarOperacao** na classe **ContaCorrente**, de modo a permitir saque apenas se houver saldo disponível:

```
namespace Lab.Models
{
    public class ContaCorrente : Conta
    {
        public double Saldo { get; protected set; }
        public Cliente ClienteInfo { get; set; }

        public ContaCorrente(int Banco, string Agencia, string Conta):
            base(Banco, Agencia, Conta)
        { }

        public ContaCorrente(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public virtual void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            switch (operacao)
            {
                case Operacoes.Deposito:
                    this.Saldo += valor;
                    break;
                case Operacoes.Saque:
                    if(valor <= this.Saldo)
                    {
                        this.Saldo -= valor;
                    }
                    break;
            }
        }

        public virtual string Exibir()
        {
            string cliente = this.ClienteInfo != null ?
                this.ClienteInfo.Exibir() + '\n' : "";

            return $"{cliente}" +
                $"Banco: {this.NumeroBanco}\n" +
                $"Agência: {this.NumeroAgencia}\n" +
                $"Conta: {this.NumeroConta}\n" +
                $"Saldo Atual: {this.Saldo}";
        }

        public override string MostrarExtrato()
        {
            throw new NotImplementedException();
        }
    }
}
```

## 8. Na classe ContaEspecial:

- Sobrescreva o método **Exibir**, de modo a incluir o limite;
- Sobrescreva o método **EfetuarOperacao** de modo a considerar o limite na verificação do saldo.

O método **MostrarExtrato** será implementado em capítulos posteriores.

```
namespace Lab.Models
{
    public class ContaEspecial : ContaCorrente
    {
        public double Limite { get; set; }

        public ContaEspecial(int Banco, string Agencia, string Conta) :
            base(Banco, Agencia, Conta)
        { }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente, double Limite)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
            this.Limite = Limite;
        }

        //métodos sobrescritos
        public override void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            switch (operacao)
            {
                case Operacoes.Deposito:
                    this.Saldo += valor;
                    break;
                case Operacoes.Saque:
                    if (valor <= (this.Saldo + this.Limite))
                    {
                        this.Saldo -= valor;
                    }
                    break;
            }
        }
    }
}
```

```
public override string Exibir()
{
    return $"{base.Exibir()}\n" +
        $"Limite: {this.Limite}\n" +
        $"Salto Disponível: {this.Saldo + this.Limite}";
}

public override string MostrarExtrato()
{
    return base.MostrarExtrato();
}
}
```

9. No projeto **Lab.Views**, na interface referente ao cadastro de contas (arquivo **MainWindow.xaml**), adicione um campo para informar o limite;

```
<TabItem Header="Cadastro de Contas">

    <Grid Background="LightGreen">
        <Grid.RowDefinitions>
            <RowDefinition Height="175" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid Grid.Row="0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
            </Grid.RowDefinitions>
            <!--Componentes Label-->
            <Label Name="clienteLabel" Content="Cliente:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="0" Grid.Column="0"/>

            <Label Name="bancoLabel" Content="Núm. Banco:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="1" Grid.Column="0"/>
        
```

```
Agência:"
```

```
<Label Name="agenciaLabel" Content="Núm  
VerticalAlignment="Center"  
HorizontalAlignment="Right"  
Grid.Row="2" Grid.Column="0"/>  
  
<Label Name="contaLabel" Content="Núm. Conta:"  
VerticalAlignment="Center"  
HorizontalAlignment="Right"  
Grid.Row="3" Grid.Column="0"/>  
  
<Label Name="tipoLabel" Content="Tipo:"  
VerticalAlignment="Center"  
HorizontalAlignment="Right"  
Grid.Row="4" Grid.Column="0"/>  
  
<Label Name="limiteLabel" Content="Limite:"  
VerticalAlignment="Center"  
HorizontalAlignment="Right"  
Grid.Row="6" Grid.Column="0"/>  
  
<TextBox Name="limiteTextBox" Width="100"  
VerticalAlignment="Center"  
HorizontalAlignment="Left"  
Grid.Row="6" Grid.Column="1"/>  
  
<!--Componentes TextBox e ComboBox-->  
<ComboBox Grid.Row="0" Grid.Column="1"  
Name="clienteComboBox"  
VerticalAlignment="Center"  
HorizontalAlignment="Left"  
Width="150">  
</ComboBox>  
  
<TextBox Name="bancoTextBox" Width="100"  
VerticalAlignment="Center"  
HorizontalAlignment="Left"  
Grid.Row="1" Grid.Column="1"/>  
  
<TextBox Name="agenciaTextBox" Width="100"  
VerticalAlignment="Center"  
HorizontalAlignment="Left"  
Grid.Row="2" Grid.Column="1"/>  
  
<TextBox Name="contaTextBox" Width="100"  
VerticalAlignment="Center"  
HorizontalAlignment="Left"  
Grid.Row="3" Grid.Column="1"/>  
  
<StackPanel Grid.Row="4" Grid.Column="1"  
Orientation="Horizontal"  
VerticalAlignment="Center">  
<RadioButton GroupName="tipo"  
Content="Comum"  
IsChecked="True"  
Name="comumRadioButton" />
```

```

<RadioButton GroupName="tipo"
Content="Especial"
Name="especialRadioButton" />
</StackPanel>

<!--Botão para incluir uma conta-->
<Button Grid.Row="5" Grid.Column="1"
Name="incluirContaButton"
Content="Incluir Conta"
HorizontalAlignment="Left"
Width="100">
<Button.ToolTip>
<StackPanel Width="150" Height="20"
Background="Beige">
<TextBlock>
    Permite incluir uma conta
</TextBlock>
</StackPanel>
</Button.ToolTip>
</Button>
</Grid>
</Grid>
</TabItem>

```

10. No elemento **especialRadioButton**, adicione o evento **Checked**. Ao atribuir um valor, selecione a opção "New Event Handler":

```

<RadioButton GroupName="tipo"
Content="Especial"
Name="especialRadioButton"
Checked=""/>
</StackPanel>
<TextBox Name="limiteTextBox" ...>

```

Essa opção criará o método a ser executado pelo evento **Checked** na classe **MainWindow**.

```

<RadioButton GroupName="tipo"
Content="Especial"
Name="especialRadioButton"
Checked="especialRadioButton_Checked"/>

```

```

private void especialRadioButton_Checked(object sender, RoutedEventArgs
e)
{
}

```

11. No componente **comumRadioButton**, execute o mesmo processo, mas não inclua um novo evento, e sim, aproveite o mesmo método gerado anteriormente:

```
<RadioButton GroupName="tipo"
             Content="Comum"
             IsChecked="True"
             Name="comumRadioButton"
             Checked="" />
<RadioButton GroupName="tipo"
             Content="Especial"
             Name="especialRadioButton"
             Checked="especialRadioButton_Checked"/>
```

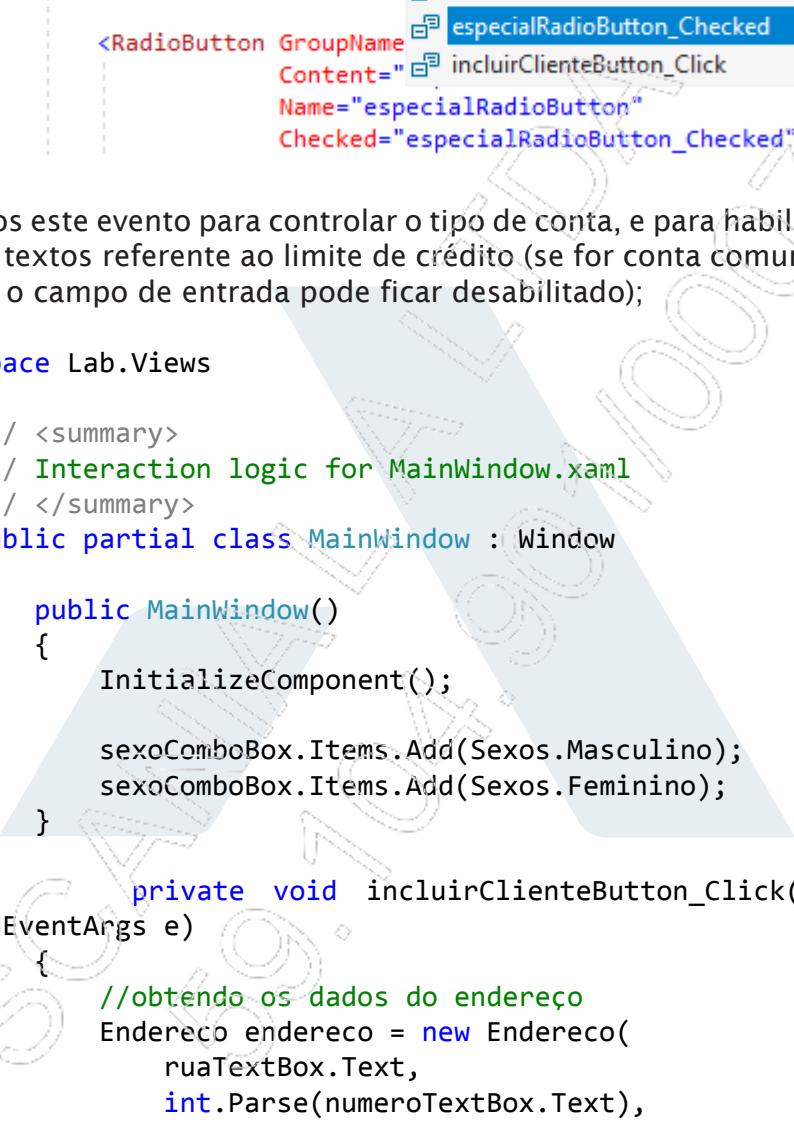


12. Usaremos este evento para controlar o tipo de conta, e para habilitar ou desabilitar o campo de textos referente ao limite de crédito (se for conta comum, não terá limite e, portanto, o campo de entrada pode ficar desabilitado);

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
        }

        private void incluirClienteButton_Click(object sender,
RoutedEventArgs e)
        {
            //obtendo os dados do endereço
            Endereco endereco = new Endereco(
                ruaTextBox.Text,
                int.Parse(numeroTextBox.Text),
                cidadeTextBox.Text,
                cepTextBox.Text);
        }
    }
}
```



```
//obtendo os dados do cliente
Cliente cliente = new Cliente(
    cpfTextBox.Text,
    nomeTextBox.Text,
    (Sexos)sexoComboBox.SelectedItem,
    int.Parse(idadeTextBox.Text));

cliente.EnderecoResidencial = endereco;
MessageBox.Show(cliente.Exibir());
}

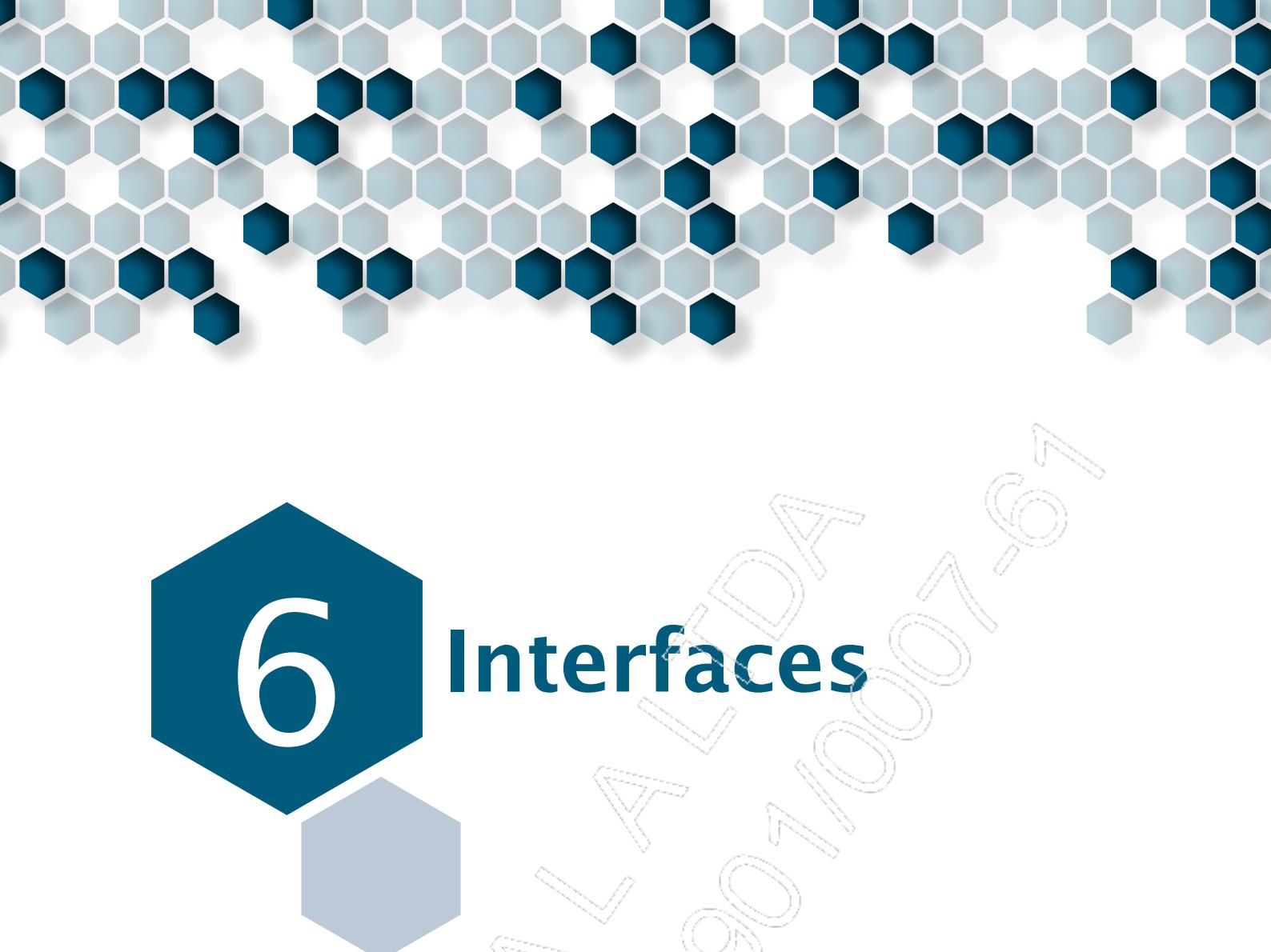
private bool VerificarEspecial { get; set; }

private void especialRadioButton_CheckedChanged(object sender,
    RoutedEventArgs e)
{
    var radio = sender as RadioButton;
    VerificarEspecial = (radio == especialRadioButton);

    limiteLabel.IsEnabled = VerificarEspecial;
    limiteTextBox.IsEnabled = VerificarEspecial;
}
}
```

13. Teste esta nova funcionalidade.

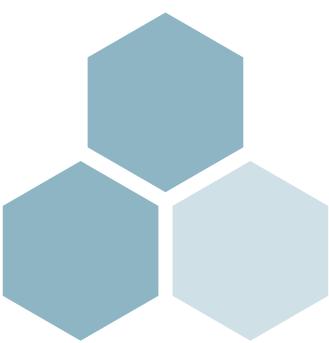




# 6

## Interfaces

- Interfaces;
- Valores default para membros de interfaces.



## 6.1. Introdução

Interfaces são tipos especiais de classes, que funcionam como modelos de implementação. Elas são puramente abstratas, e não há possibilidade de implementarmos algum método ou propriedade.

Apresentaremos, neste capítulo, a definição e a implementação de interfaces.

## 6.2. Interfaces

As classes ajudam a definir um objeto e seu comportamento. As interfaces, por sua vez, auxiliam na definição dessas classes. Elas são formadas pela declaração de um ou mais métodos, os quais obrigatoriamente não possuem corpo, ou seja, são abstratos. Quando implementada, uma interface garante que seus métodos sejam implementados na classe.

Assim como uma classe abstrata, uma interface não pode ser instanciada.

As restrições impostas pela interface são:

- Não permitem definir campos;
- Não permitem definir construtores nem destrutores;
- Não permitem aninhar nenhum tipo;
- Não podem ser herdadas de uma estrutura ou classe – apenas de outra interface;
- Não podem conter membros estáticos;
- Os membros da interface são, automaticamente, públicos e não podem incluir modificadores de acesso.

Em uma mesma classe, podem ser implementadas uma ou mais interfaces, separadas por vírgulas. Devemos considerar, ainda, que não é permitido declarar variáveis em uma interface, por ser esta uma implementação do atributo de um objeto.

Para declarar uma interface, utiliza-se a palavra-chave **interface**. Além disso, define-se, por convenção, que ao declarar o nome de uma interface, ele seja iniciado com a letra **i** em caixa alta (**I**).

Vamos considerar a classe **Figura** do capítulo **Herança e Polimorfismo**, só que, desta vez, em forma de interface, chamada **IFigura**. As classes **Retangulo** e **Circulo** implementarão esta interface:

- Interface IFigura

```
namespace Conceitos.CSharp.Cap06
{
    public interface IFigura
    {
        double CalcularArea();
    }
}
```

- Classes Retangulo e Circulo

```
namespace Conceitos.CSharp.Cap06
{
    public class Retangulo : IFigura
    {
        public double Base { get; set; }
        public double Altura { get; set; }
        public double CalcularArea()
        {
            return this.Base * this.Altura;
        }
    }
}

namespace Conceitos.CSharp.Cap06
{
    public class Circulo : IFigura
    {
        public double Raio { get; set; }
        public double CalcularArea()
        {
            return Math.PI * Math.Pow(this.Raio, 2);
        }
    }
}
```

Observe que a implementação do método da interface não utiliza a palavra **override**, como acontece na sobreescrita de métodos abstratos ou virtuais.

A interface pode ser usada para declarar variáveis, como ocorre com classes:

```
namespace Conceitos.CSharp.Cap06
{
    class Program
    {
        static void Main(string[] args)
        {
            IFigura f1 = new Retangulo()
            {
                Base = 20,
                Altura = 10
            };

            IFigura f2 = new Circulo()
            {
                Raio = 2
            };

            //continua...
        }
    }
}
```

Uma classe pode implementar várias interfaces, mas herdar apenas uma classe. Nesse caso, a herança da classe deve vir primeiro e, separando por vírgula, implementamos as interfaces. Para ilustrar, consideraremos a classe **Cliente**, a interface **IDocumento** e as implementações **DocumentoPJ** e **DocumentoPF**:

- **Interface IDocumento**

```
namespace Conceitos.CSharp.Cap06
{
    public interface IDocumento
    {
        string Numero { get; set; }

        string MostrarDocumento();
    }
}
```

- Classes DocumentoPF e DocumentoPJ

```
namespace Conceitos.CSharp.Cap06
{
    public class DocumentoPF : IDocumento
    {
        private string _numero;
        public string Numero
        {
            get => _numero;
            set => _numero = (value.Length == 11 ? value :
throw new Exception("Documento inválido"));
        }

        public string MostrarDocumento()
        {
            return "Número do CPF: " + this.Numero;
        }
    }
}

namespace Conceitos.CSharp.Cap06
{
    public class DocumentoPJ: IDocumento
    {
        private string _numero;
        public string Numero
        {
            get => _numero;
            set => _numero = (value.Length == 14 ? value :
throw new Exception("Documento inválido"));
        }

        public string MostrarDocumento()
        {
            return "Número do CNPJ: " + this.Numero;
        }
    }
}
```

- **Classe Cliente**

```
namespace Conceitos.CSharp.Cap06
{
    public class Cliente
    {
        public IDocumento Documento { get; set; }
        public string NomeCliente { get; set; }
    }
}
```

Para a classe **DocumentoPF**, se o número do documento não tiver 11 dígitos (indicando um CPF), a propriedade lança uma exceção. As exceções serão estudadas em capítulos posteriores. Analogamente, a classe **DocumentoPJ** deve ter um documento com 14 dígitos, indicando um CNPJ. A classe **Cliente** possui uma propriedade do tipo **IDocumento**, indicando que o cliente pode ser uma pessoa física ou pessoa jurídica.

Uma utilização da classe **Cliente** é mostrada no programa a seguir:

```
namespace Conceitos.CSharp.Cap06
{
    class Program
    {
        static void Main(string[] args)
        {
            Cliente pf = new Cliente()
            {
                Documento = new DocumentoPF()
                {
                    Numero = "12345678901"
                },
                NomeCliente = "Pessoa Física"
            };

            Cliente pj = new Cliente()
            {
                Documento = new DocumentoPJ()
                {
                    Numero = "12345678901234"
                },
                NomeCliente = "Pessoa Jurídica"
            };

            //continua...
        }
    }
}
```

## 6.3. Valores default para membros de interfaces

A partir da versão 8 do C#, é permitido atribuir valores default para propriedades e métodos de interfaces. Não se trata de implementações de métodos, e sim da criação de estruturas padrão para todas as implementações dos métodos.

Considere a interface:

```
namespace Conceitos.CSharp.Cap06
{
    public interface ITrabalhador
    {
        string Nome { get; set; }
        bool PagtoPorHora { get => true; }

        string MostrarDados()
        {
            return $"Nome: {Nome}, Recebe por hora? " + (PagtoPorHora
? "Sim" : "Não");
        }
    }
}
```

### Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

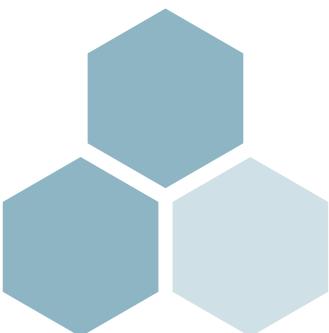
- Uma interface auxilia na definição das classes e é formada pela declaração de um ou mais métodos, que não possuem corpo. Quando implementada, garante que os métodos especificados nela pertencerão a uma classe determinada;
- Desde a versão 8 do C#, é possível definirmos valores default para membros de uma interface. Esses valores serão propagados por todas as classes que implementarem a interface.



6

# Interfaces

Teste seus conhecimentos



## 1. Um método default na interface:

- a) É executado apenas por uma instância da interface.
- b) É executado por todas as classes que implementam a interface.
- c) Deve ser sobrescrito.
- d) Deve ser declarado como abstract.
- e) Deve ter propriedades default.

## 2. Para definir um método abstrato em uma interface, qual a sintaxe correta?

- a) public abstract void Metodo();
- b) public abstract void Metodo() {};
- c) void Metodo();
- d) abstract void Metodo();
- e) abstract void Metodo() {};

## 3. Quantas interfaces podem ser implementadas por uma classe?

- a) Apenas uma.
- b) No máximo, três.
- c) Várias.
- d) Uma interface e várias classes.
- e) Duas, desde que elas tenham relação de herança.

**4. Marque a alternativa correta:**

- a) Uma interface pode implementar outra interface.
- b) Uma interface pode estender outra interface.
- c) Uma interface pode implementar uma classe.
- d) Uma classe pode estender várias classes.
- e) Uma interface pode estender várias classes.

**5. Na terminologia de orientação a objetos, uma interface serve para:**

- a) Definir eventos.
- b) Implementar métodos.
- c) Estabelecer um contrato de implementação.
- d) Estabelecer a herança múltipla.
- e) Nenhuma das alternativas anteriores está correta.





6

# Interfaces

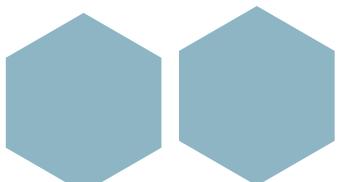


Mãos à obra!

SCALDA  
50.  
94.  
95.  
96.  
97.  
10007-67  
ALTA  
MÁXIMA



Editora  
**IMPACTA**



## Laboratório 1

### A – Implementando uma interface

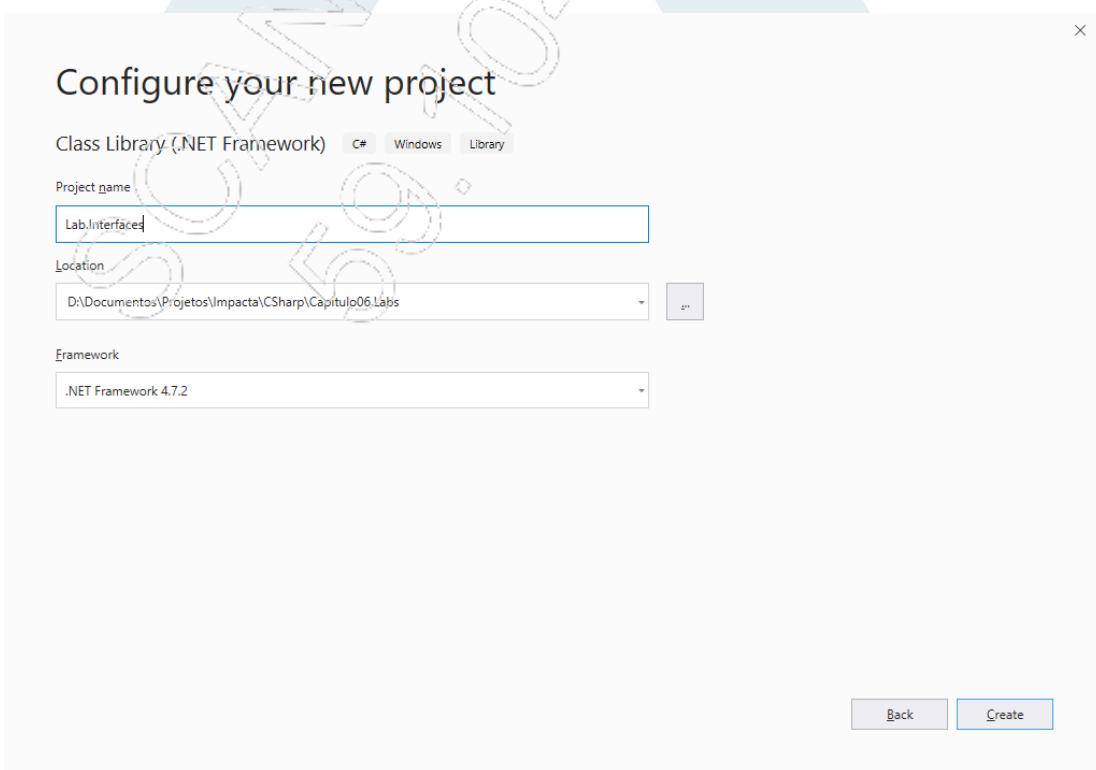
#### Objetivos:

Neste laboratório, modificaremos a natureza da classe **Cliente**. Vamos torná-la abstrata e, além disso, vamos remover a propriedade **Cpf**.

Em seguida, criaremos duas subclasses chamadas **ClientePF** (para Pessoa Física) e **ClientePJ** (para Pessoa Jurídica). A propriedade **Cpf** será definida na classe **ClientePF**. Na classe **ClientePJ**, definiremos a propriedade **Cnpj**. A especificação do documento será feita por meio da implementação de uma interface.

Faremos algumas alterações também na interface gráfica. A decisão pela instância de **ClientePF** ou **ClientePJ** será feita tomando como base o número de caracteres fornecidos no campo de entrada correspondente ao **Cpf**, que será alterado para **Documento**.

1. Crie um novo solution, vazio, chamado **Capitulo06.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models** e **Lab.Views** do Capítulo 5 na pasta do solution **Capitulo06.Labs**;
3. Adicione esses projetos ao novo solution;
4. No novo solution, adicione um novo projeto **Class Library** chamado **Lab.Interfaces**. Em seguida, exclua a classe **Class1**;



5. No projeto **Lab.Interfaces**, adicione a interface **IDocumento**:

```
namespace Lab.Interfaces
{
    public interface IDocumento
    {
        string NumeroDocumento { get; set; }

        string MostrarDocumento();
    }
}
```

6. Remova a propriedade **Cpf** da classe **Cliente** e torne-a abstrata. Observe que os construtores sofreram alterações e o método **Exibir** se tornou virtual:

```
namespace Lab.Models
{
    public abstract class Cliente
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public ContaCorrente Conta { get; set; }

        public Cliente(string Nome, Sexos Sexo)
        {
            this.Nome = Nome;
            this.Sexo = Sexo;
        }

        public Cliente(string Nome, Sexos Sexo, int idade)
            : this(Nome, Sexo)
        {
            this.Idade = Idade;
        }

        public Cliente(string Nome, Sexos Sexo, Endereco endereco)
            : this(Nome, Sexo)
        {

            this.EnderecoResidencial = endereco;
        }
    }
}
```

```
~Cliente()
{
    if(this.Conta != null)
    {
        this.Conta = null;
    }
}

public string Exibir()
{
    return $"Nome: {this.Nome}\n" +
        $"Idade: {this.Idade}\n" +
        $"Sexo: {this.Sexo}\n" +
        $"ENDEREÇO DO CLIENTE:\n" +
        $"{this.EnderecoResidencial.Exibir()}";
}
```

7. Adicione a classe **ClientePF** no projeto **Lab.Models**. Essa classe deve estender a classe **Cliente** e implementar a interface **IDocumento** (lembre-se de referenciar o novo projeto **Lab.Interfaces**). Essa classe deve definir construtores para acessar os construtores da superclasse e sobrescrever o método virtual **Exibir** e os membros da interface:

```
namespace Lab.Models
{
    public class ClientePF : Cliente, IDocumento
    {
        //propriedade implementada da interface
        private string _cpf;
        public string NumeroDocumento
        {
            get => _cpf;
            set => _cpf = (value.ValidarCPF() ? value :
                throw new Exception("CPF Inválido"));

        }
        //método implementado da interface
        public string MostrarDocumento()
        {
            return $"CPF do cliente: {NumeroDocumento}";
        }

        //construtores
        public ClientePF(string Cpf, string Nome, Sexos Sexo)
            : base(Nome, Sexo)
        {
            this.NumeroDocumento = Cpf;
        }
    }
}
```

```

public ClientePF(string Cpf, string Nome, Sexos Sexo,
    int Idade, Endereco endereco)
    : base(Nome, Sexo, Idade)
{
    this.NumeroDocumento = Cpf;
    base.EnderecoResidencial = endereco;
}

public override string Exibir()
{
    return $"{MostrarDocumento()}\n" +
        $"{base.Exibir()}";
}
}
}

```

8. Analogamente, adicione a classe **ClientePJ** no projeto **Lab.Models**. Essa classe também deve estender a classe **Cliente** e implementar a interface **IDocumento**:

```

namespace Lab.Models
{
    public class ClientePJ : Cliente, IDocumento
    {
        //propriedade implementada da interface
        private string _cnpj;
        public string NumeroDocumento
        {
            get => _cnpj;
            set => _cnpj = (value.Length == 14 ? value :
                throw new Exception("CNPJ Inválido"));
        }

        //método implementado da interface
        public string MostrarDocumento()
        {
            return $"CNPJ do cliente: {NumeroDocumento}";
        }

        //construtores
        public ClientePJ(string Cnpj, string Nome, Sexos Sexo)
            : base(Nome, Sexo)
        {
            this.NumeroDocumento = Cnpj;
        }
    }
}

```

```
public ClientePJ(string Cnpj, string Nome, Sexos Sexo,
    int Idade, Endereco endereco)
    : base(Nome, Sexo, Idade)
{
    this.NumeroDocumento = Cnpj;
    base.EnderecoResidencial = endereco;
}

public override string Exibir()
{
    return $"{MostrarDocumento()}\n" +
        $"{base.Exibir()}";
}
}
```

9. No projeto **Lab.Views**, em **MainWindow.xaml**, altere o nome dos componentes **Label** e **TextBox**, da interface correspondente ao cadastro de Clientes, de **Cpf** para **Documento**:

```
<TabItem Header="Cadastro de Clientes">
    <Grid Background="LightBlue">
        <Grid.RowDefinitions>
            <RowDefinition Height="225" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid Grid.Row="0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
            </Grid.RowDefinitions>
```

```

    <!--Componentes Label-->
<Label Name="documentoLabel" Content="Documento:" 
      VerticalAlignment="Center"
      HorizontalAlignment="Right"
      Grid.Row="0" Grid.Column="0"/>

    <!--demais elementos omitidos -->

    <!--Componentes TextBox e ComboBox-->
<TextBox Name="documentoTextBox" Width="100"
      VerticalAlignment="Center"
      HorizontalAlignment="Left"
      Grid.Row="0" Grid.Column="1"/>

    <!--demais elementos omitidos -->

        </Grid>
    </Grid>
</TabItem>

```

10. Para decidir pela instância da classe **ClientePF** ou **ClientePJ**, usaremos a quantidade de caracteres: 11 para **Cpf** (Pessoa Física) ou 14 para **Cnpj** (Pessoa Jurídica). Vamos promover algumas alterações no evento **click** do botão de comandos que permite "criar" um novo cliente - o botão **incluirClienteButton**:

```

namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
        }

        private void incluirClienteButton_Click(object sender,
RoutedEventArgs e)
        {
            //obtendo os dados do endereço
            Endereco endereco = new Endereco(
                ruaTextBox.Text,
                int.Parse(numeroTextBox.Text),
                cidadeTextBox.Text,
                cepTextBox.Text);
        }
    }
}

```

```
//obtendo os dados do cliente
int digitos = documentoTextBox.Text.Length;
Cliente cliente;

if(digitos == 11)
{
    cliente = new ClientePF(
        documentoTextBox.Text,
        nomeTextBox.Text,
        (Sexos)sexoComboBox.SelectedItem,
        int.Parse(idadeTextBox.Text),
        endereco);
}
else if(digitos == 14)
{
    cliente = new ClientePJ(
        documentoTextBox.Text,
        nomeTextBox.Text,
        (Sexos)sexoComboBox.SelectedItem,
        int.Parse(idadeTextBox.Text),
        endereco);
}
else
{
    throw new Exception("Documento inválido");
}

MessageBox.Show(cliente.Exibir());
}

private bool VerificarEspecial { get; set; }

private void especialRadioButton_Checked(object sender,
RoutedEventArgs e)
{
    var radio = sender as RadioButton;
    VerificarEspecial = (radio == especialRadioButton);

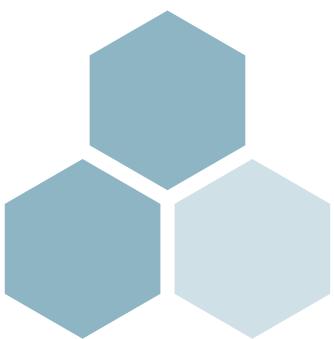
    limiteLabel.IsEnabled = VerificarEspecial;
    limiteTextBox.IsEnabled = VerificarEspecial;
}
}
```

11. Teste a aplicação.

# 7

## Genéricos

- Classes genéricas;
- Métodos genéricos;
- Genéricos e herança;
- Interfaces genéricas;
- Restrições ao uso dos genéricos;
- Covariância e contravariância.



## 7.1. Introdução

Existem muitas situações em que um conjunto de classes realiza tarefas similares, porém com tipos de elementos diferentes. Torna-se mais viável, em vez de termos uma classe para cada tipo, termos uma classe funcionando como um template e, no momento da utilização, serem fornecidos os tipos a serem manipulados. Esses tipos especiais de classes são conhecidos como classes genéricas. Além de classes, podemos ter estruturas, interfaces e outros modelos de dados como genéricos.

Vamos apresentar, na sequência, os procedimentos necessários para definirmos elementos genéricos. Nos exemplos a serem apresentados, deixaremos a maior parte dos detalhes de implementação de lado, pois nosso foco é no conceito de genéricos.

## 7.2. Classes genéricas

Os genéricos permitem parametrizar classes, estruturas, interfaces, delegates e métodos. Eles são úteis quando pretendemos criar um conjunto de classes que são similares na sua funcionalidade, mas envolvendo tipos diferentes. Para ilustrar, consideremos a classe **Pilha**, definida a seguir:

```
public class Pilha
{
    private ArrayList elementos = new ArrayList();

    public object Top
    {
        get => elementos[elementos.Count - 1];
    }

    public void Push(object item)
    {
        elementos.Add(item);
    }

    public object Pop()
    {
        object aux = null;
        if(elementos.Count == 0)
        {
            return aux;
        }
        aux = Top;
        elementos.RemoveAt(elementos.Count - 1);
        return aux;
    }
}
```

Vemos que, apesar de não ser um exemplo original (já temos classes no Framework que se comportam como pilhas), ele foi usado para introduzir o conceito de genéricos.

O código apresentado permite utilizar a classe com qualquer tipo de elemento. Isto deve-se ao fato da classe utilizar o tipo **object** (que é a classe base de todos os elementos definidos em .NET).

O fato de a classe poder manipular qualquer tipo de dado não significa que seja uma classe genérica, pois falta um ingrediente fundamental: a **tipificação** dos elementos.

É possível que uma classe, de fato, receba qualquer tipo de dado, desde que este dado seja fortemente tipado. É justamente para atingir esse objetivo que temos as classes genéricas. Vamos reescrever a classe **Pilha**, mas como classe genérica, e com o nome **PilhaGenerica**:

```
public class PilhaGenerica<T>
{
    private List<T> elementos = new List<T>();

    public T Top {
        get => elementos[elementos.Count - 1];
    }

    public void Push(T item)
    {
        elementos.Add(item);
    }

    public T Pop()
    {
        T aux = default(T);
        if (elementos.Count == 0)
        {
            return aux;
        }
        aux = Top;
        elementos.RemoveAt(elementos.Count - 1);
        return aux;
    }
}
```

Nessa nova classe, temos os seguintes pontos a serem considerados:

- A classe define um elemento de parametrização, que aqui chamamos de **T**;
- Substituímos a classe **ArrayList** pela classe **List** que, por sua vez, também é genérica;
- Em todas as ocorrências de **object** na classe **Pilha**, usamos o parâmetro de tipo **T** em seu lugar;
- Como **T** pode ser qualquer tipo, incluindo classes ou estruturas, definimos o valor padrão de **T** no método **Pop()** como **default(T)**. Não usamos o valor null porque este valor só pode ser atribuído a classes, ou tipos valor **Nullable**.

Com essas considerações, podemos agora definir instâncias da classe **PilhaGenerica**:

```
class Program
{
    static void Main(string[] args)
    {
        PilhaGenerica<string> pilhaString = new PilhaGenerica<string>();
        pilhaString.Push("Osvaldo");
        pilhaString.Push("Jesse");
        pilhaString.Push("Abigail");

        Console.WriteLine("Elemento no topo (pilhaString): " +
pilhaString.Pop());

        PilhaGenerica<int> pilhaInt = new PilhaGenerica<int>();
        pilhaInt.Push(122);
        pilhaInt.Push(2956);
        pilhaInt.Push(365);

        Console.WriteLine("Elemento no topo (pilhaInt): " + pilhaInt.
Pop());

        Console.ReadKey();
    }
}
```

A parametrização, ou a tipagem, ocorre na declaração e na instância da classe. Observe que, apesar de genérica, a classe se torna de um tipo em particular para cada instância definida.

## 7.3. Métodos genéricos

Além das classes, é possível parametrizar outros tipos de elementos, como métodos.

Mostraremos como podemos parametrizar um método que não pertence a uma classe genérica.

```
public static string MetodoGenerico<T>(T elemento)
{
    return elemento.ToString();
}
```

## 7.4. Genéricos e herança

O conceito de genéricos se estende a herança, levando-se em conta:

- Se a superclasse é genérica;
- Se a subclasse é genérica;
- Se ambas são genéricas.

Além das classes, a sobrescrita dos métodos considera a parametrização na subclasse e o parâmetro de tipo na superclasse.

Vamos entender esses conceitos por meio dos exemplos adiante.

### 7.4.1. Sobrescrita de métodos genéricos

Neste exemplo, temos uma superclasse genérica e abstrata chamada **ClasseBase**, com um método abstrato recebendo um parâmetro do tipo **T**, e outro método não abstrato retornando **T**:

```
public abstract class ClasseBase<T>
{
    public virtual T Metodo()
    {
        return default(T);
    }

    public abstract void Metodo2(T item);
}
```

A subclasse chamada **SubClasse** estende **ClasseBase**, fornecendo um tipo para a superclasse:

```
public class SubClasse : ClasseBase<string>
{
    public override string Metodo()
    {
        return base.Metodo();
    }

    public override void Metodo2(string item)
    {
        throw new NotImplementedException();
    }
}
```

Observe o retorno de **Metodo()**, e o parâmetro de **Metodo2()**. São do tipo usado na parametrização.

### 7.4.2. Subclasse genérica

Se, na subclasse, o tipo não for especificado na superclasse, durante o processo de herança, isso significa que a subclasse também será genérica. Veja o exemplo a seguir, na classe **SubClasse2**:

```
public class SubClasse2<T> : ClasseBase<T>
{
    public override T Metodo()
    {
        return base.Metodo();
    }

    public override void Metodo2(T item)
    {
        throw new NotImplementedException();
    }
}
```

A classe **SubClasse2** está repassando o parâmetro de tipo para a superclasse. Quando instanciada, **SubClasse2** deverá receber seu tipo adequadamente.

### 7.5. Interfaces genéricas

Assim como classes, interfaces também podem ser genéricas. Como interfaces são implementadas, valem as mesmas regras aplicadas na herança de tipos genéricos.

Considere a interface **IDataAccess**, simulando um cenário de acesso a dados:

```
public interface IDataAccess<T, K>
{
    int Incluir(T elemento);
    T Buscar(K chave);
}
```

Usamos esse exemplo para ilustrar também que podemos ter mais de um elemento de parametrização. No caso, o tipo **K** é usado como chave de busca, nessa simulação.

Na implementação dessa interface, torna-se necessário fornecer os tipos para **T** e para **K**. A classe **DataAccess** adiante apresenta essa implementação, considerando a existência de uma classe chamada **Cliente**:

- **Classe Cliente**

```
public class Cliente
{
    public int Código { get; set; }
    public string Descrição { get; set; }
}
```

- **Classe DataAccess**

```
public class DataAccess : IDataAccess<Cliente, int>
{
    public Cliente Buscar(int chave)
    {
        throw new NotImplementedException();
    }

    public bool Incluir(Cliente elemento)
    {
        throw new NotImplementedException();
    }
}
```

## 7.6. Restrições ao uso dos genéricos

Devido à forma como funcionam os genéricos, torna-se necessário definir determinadas restrições de forma que seja possível efetuar a compilação de uma classe deste tipo.

A necessidade de restrições fica clara no exemplo a seguir, onde temos a interface ITest com o método Executar().

- **Interface ITest**

```
public interface ITest
{
    void Executar();
}
```

- **Classe Test**

```
public class Test<T>
{
    public void Método(T item)
    {
        item.Executar(); //ERRO: Não sabemos se T implementa Executar()
    }
}
```

O único problema do exemplo anterior reside na chamada ao método `Executar()`. Uma vez que o parâmetro `T` pode ser substituído por qualquer tipo, não é possível ao compilador efetuar a validação do método sem a utilização de uma restrição.

Ao declararmos um genérico, podemos definir um conjunto de restrições que são aplicadas aos parâmetros genéricos. Assim, com base no exemplo anterior, o uso de uma restrição leva a:

```
public class Test<T> where T: ITest
{
    public void Metodo(T item)
    {
        item.Executar();
    }
}
```

Apesar de `T` ser genérico, está condicionado à implementação de `ITest`. Ou seja, podemos informar quaisquer tipos para `T`, desde que implementem `ITest`. Assim, garantiremos a presença do método `Executar()` em uma variável do tipo `T`.

## 7.6.1. Tabela de restrições

Na tabela a seguir, apresentamos as restrições que podem ser usadas na criação de genéricos:

Restrição	Descrição
<code>where T: class</code>	<code>T</code> deve ser um tipo referência, ou seja, uma classe.
<code>where T: struct</code>	<code>T</code> deve ser uma estrutura, o que significa que necessariamente não pode ser nula (estruturas não podem ser nulas) e deve conter o construtor padrão.
<code>where T: notnull</code>	<code>T</code> deve ser qualquer tipo explicitamente definido como não nulo (not Nullable).
<code>where T: unmanaged</code>	<code>T</code> deve ser um tipo não gerenciável não nulo. Essa restrição implica no tipo <code>struct</code> e não pode ser combinada com a restrição <code>struct nem new()</code> .
<code>where T: new()</code>	<code>T</code> deve possuir um construtor público sem parâmetros.
<code>where T: &lt;nome da superclasse&gt;</code>	<code>T</code> deve ser do tipo, ou subtipo, da classe mencionada.
<code>where T: &lt;nome da interface&gt;</code>	<code>T</code> deve ser uma classe que implementa a interface mencionada, que por sua vez também pode ser genérica.
<code>where T: U</code>	<code>T</code> deve ser uma subclasse do parâmetro de tipo <code>U</code> .

## 7.7. Covariância e contravariância

Covariância e contravariância são termos que se referem à capacidade de usar um tipo mais específico ou um tipo menos específico do que o especificado originalmente para o parâmetro de tipo.

Os parâmetros de tipo genéricos oferecem suporte a covariância e contravariância para fornecer maior flexibilidade na atribuição e no uso de tipos genéricos.

### 7.7.1. Covariância

Permite um tipo mais específico que o original. Vamos analisar a interface `IEnumerable<T>`, presente no framework:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

Elá é a interface implementada por praticamente todas as coleções. Considerando a classe **Figura** e sua subclasse, **Retangulo**, podemos realizar as declarações:

```
class Program
{
    static void Main(string[] args)
    {
        IEnumerable<Figura> figuras = new List<Retangulo>();
        //continua o programa
    }
}
```

Perceba que a instância usou na parametrização um subtipo em relação à parametrização na declaração da variável.

### 7.7.2. Contravariância

Da mesma forma que podemos usar um subtipo na parametrização quando instanciamos uma classe genérica, podemos ter o oposto: a instância possuir um supertipo. Trata-se de tipos genéricos **contravariantes**. Analise a interface **IForma** e a classe **Forma**, definidas a seguir:

- **IForma**

```
public interface IForma<in T>
{
    void Mostra(T item);
}
```

- **Forma**

```
public class Forma<T> : IForma<T>
{
    public void Mostra(T item)
    {
        Console.WriteLine(item);
    }
}
```

Na aplicação da classe, poderemos ter:

```
class Program
{
    static void Main(string[] args)
    {
        IForma<Retangulo> retangulo = new Forma<Figura>();
        //continua o programa
    }
}
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Os genéricos definem elementos que possuem comportamentos similares, porém manipulando dados de diferentes tipos. Na definição de uma variável e uma instância da classe, devemos fornecer um parâmetro de tipo, tornando a classe fortemente tipada para a variável em questão;
- Métodos sobrescritos também podem se beneficiar dos genéricos. Quando um tipo é especificado na implementação da classe, os parâmetros e retornos de métodos assumem aquele tipo;
- É possível aplicarmos restrições aos tipos genéricos, fazendo com que eles passem a fazer parte de uma categoria predeterminada;
- Os tipos genéricos podem ser covariantes ou contravariantes, quando desejamos usar como parâmetro de tipo algo mais específico ou mais geral, respectivamente, sobre o tipo original.

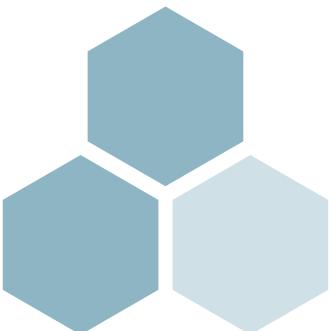




7

# Genéricos

Teste seus conhecimentos



## 1. Considere a classe:

```
public class Generica<T> where T : new() { }
```

## O que podemos dizer sobre o parâmetro de tipo T?

- a) Deve ser uma classe abstrata.
- b) Deve ter um construtor sem parâmetros.
- c) Deve ser criado com o operador new.
- d) Deve ter um método chamado new().
- e) Deve ser uma classe abstrata.

## 2. Para garantir que uma classe genérica tenha como parâmetro de tipo T algo que implemente a interface IEnumerable<T>, como a classe deve ser declarada?

- a) public class Generica<T> where T : IEnumerable<T> { }
- b) public class Generica<T> where T : IEnumerable { }
- c) public class Generica<T> where IEnumerable<T> : T { }
- d) public class Generica<T> where T is IEnumerable { }
- e) public class Generica<T> where T = IEnumerable<T> { }

## 3. A interface declarada como:

```
public interface ITipo<out T> { }
```

## possui o parâmetro de tipo:

- a) Restrito
- b) Contravariante
- c) Covariante
- d) Invariante
- e) Variante

**4. Uma classe não genérica pode estender uma classe genérica, desde que:**

- a) O tipo seja fornecido na superclasse.
- b) O tipo seja informado no momento de instanciarmos a classe.
- c) O tipo seja mantido como genérico.
- d) A subclasse tenha métodos genéricos.
- e) A subclasse tenha apenas um parâmetro de tipo.

**5. Uma classe definida como Generica<T,U> possui o parâmetro de tipo U como sendo uma extensão de T. Como a classe pode ser definida?**

- a) public class Generica<T, U> where U = T {}
- b) public class Generica<T, U> where U : T {}
- c) public class Generica<T, U> where T = U {}
- d) public class Generica<T, U> where T : U {}
- e) public class Generica<T, U> where U, T {}





7

# Genéricos

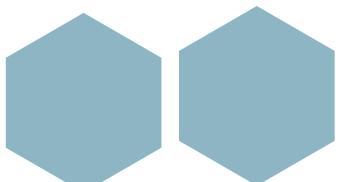


Mãos à obra!

SCAMVIA ALTA  
50° 104° 10007-67



Editora  
**IMPACTA**



## Laboratório 1

### A – Escrevendo classes genéricas

#### Objetivos:

Neste laboratório, escreveremos algumas classes genéricas que serão usadas de fato nos capítulos relacionados a manipulação de arquivos e acesso a banco de dados.

1. Crie um novo solution, vazio, chamado **Capítulo07.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models**, **Lab.Views** e **Lab.Interfaces** do Capítulo 6, na pasta do solution **Capítulo07.Labs**;
3. Adicione esses projetos ao novo solution;
4. Adicione dois novos projetos **Class Library (.NET Framework)** chamados **Lab.Arquivos** e **Lab.Dados** no solution **Capítulo07.Labs**. Remova as classes **Class1** dos dois projetos;
5. No projeto **Lab.Arquivos**, adicione uma referência ao projeto **Lab.Models**. Adicione também uma classe chamada **AcessoArquivo**. Essa classe servirá para conter métodos responsáveis por gerar os dados de uma conta corrente no arquivo, como o extrato. Sendo assim, adicione um método genérico chamado **GerarExtrato**:

```
using Lab.Models;

namespace Lab.Arquivos
{
    public class AcessoArquivo<T>
    {
        public static void GerarExtrato<T>(T conta) where T: Conta
        {
            //implementação em breve
        }
    }
}
```

6. No projeto **Lab.Dados**, adicione uma referência ao projeto **Lab.Models**. Inclua uma classe genérica chamada **Dao**, abstrata, contendo os métodos abstratos responsáveis pelas operações comuns no banco de dados:

```
namespace Lab.Dados
{
    public abstract class Dao<T, K> where T: class
    {
        public abstract int Incluir(T item);
        public abstract T Buscar(K chave);
        public abstract IEnumerable<T> Listar(K chave = default(K));
        public abstract int Remover(T item);
    }
}
```

Observe que o método **Listar** recebe um parâmetro opcional. Como o tipo desse parâmetro é genérico, atribuímos o comando **default(T)** para que seja assumido o valor default de acordo com o tipo do parâmetro de tipo **K**.

7. Defina, no projeto **Lab.Dados**, as subclasses de **Dao**: **ClientsDao** e **ContasDao**. Essas classes considerarão as classes **Cliente** e **Conta**, respectivamente, como tipos parametrizados:

- **Classe ClientsDao**

```
using Lab.Models;
using System;
using System.Collections.Generic;

namespace Lab.Dados
{
    public class ClientsDao : Dao<Cliente, string>
    {
        public override Cliente Buscar(string chave)
        {
            throw new NotImplementedException();
        }

        public override int Incluir(Cliente item)
        {
            throw new NotImplementedException();
        }

        public override IEnumerable<Cliente> Listar(string chave = null)
        {
            throw new NotImplementedException();
        }

        public override int Remover(Cliente item)
        {
            throw new NotImplementedException();
        }
    }
}
```

- **Classe ContasDao**

```
using Lab.Models;
using System;
using System.Collections.Generic;

namespace Lab.Dados
{
    public class ContasDao : Dao<Conta, string>
    {
        public override Conta Buscar(string chave)
        {
            throw new NotImplementedException();
        }

        public override int Incluir(Conta item)
        {
            throw new NotImplementedException();
        }

        public override IEnumerable<Conta> Listar(string chave = null)
        {
            throw new NotImplementedException();
        }

        public override int Remover(Conta item)
        {
            throw new NotImplementedException();
        }
    }
}
```

 Outras classes serão adicionadas à medida que for necessário, de acordo com o tópico a ser estudado.

# 8

# Coleções

- Arrays;
- Coleções;
- A classe ArrayList;
- A classe Stack;
- A classe Queue;
- A classe Hashtable;
- A classe List<T>;
- As classes HashSet<T> e SortedSet<T>;
- A interface IEnumerable<T>.

## 8.1. Introdução

Em qualquer aplicação é bastante comum o uso de coleções. Elas servem para armazenar conjuntos de dados, como números, textos, objetos ou mesmo outras coleções.

Dependendo do objetivo, os dados podem ser armazenados em conjuntos de formas diferentes. Existem coleções que não admitem elementos repetidos, por exemplo, outras que não permitem que, uma vez armazenados, mantenham a posição do elemento, e temos ainda aquelas cujos elementos podem ser colocados em ordem.

A finalidade deste capítulo é apresentar as principais coleções e as formas de acessá-las.

## 8.2. Arrays

Chamamos de **array** uma sequência não ordenada de elementos do mesmo tipo. Eles podem conter uma ou múltiplas dimensões, e seus valores são identificados ou referenciados por um índice inteiro. O valor do índice inicia em zero.

Um array pode ter elementos de qualquer tipo, inclusive outros arrays. Um array de arrays também é chamado de **jagged array**. Seus elementos são tipos-referência e seus valores iniciais são **null**.

### 8.2.1. Construção e instanciação de arrays

Para construir um array, é necessário especificar o tipo de dado dos elementos, seguido por um par de colchetes e pelo nome da variável, conforme a seguinte sintaxe:

```
<tipo>[] <identificador>;
```

Podemos, também, criar uma instância de um array, utilizando, para isso, o operador **new**, seguido pelo nome do tipo de elemento e pelo tamanho do array entre colchetes, como vemos na sintaxe a seguir:

```
<tipo>[] <identificador> = new <tipo>[nº ocorrências];
```

- **Exemplo 1: Array unidimensional**

```
int[] numeros = new int[10];  
  
    numeros[0] = 18;  
    numeros[1] = 27;  
    numeros[2] = 10;  
    numeros[3] = 0;  
    numeros[4] = 45;  
    numeros[5] = 22;  
    numeros[6] = 34;  
    numeros[7] = 56;  
    numeros[8] = 76;  
    numeros[9] = 5;
```

0	1	2	3	4	5	6	7	8	9
18	27	10	0	45	22	34	56	76	5

O primeiro elemento de um array está sempre na posição ZERO.

- **Exemplo 2: Array bidimensional**

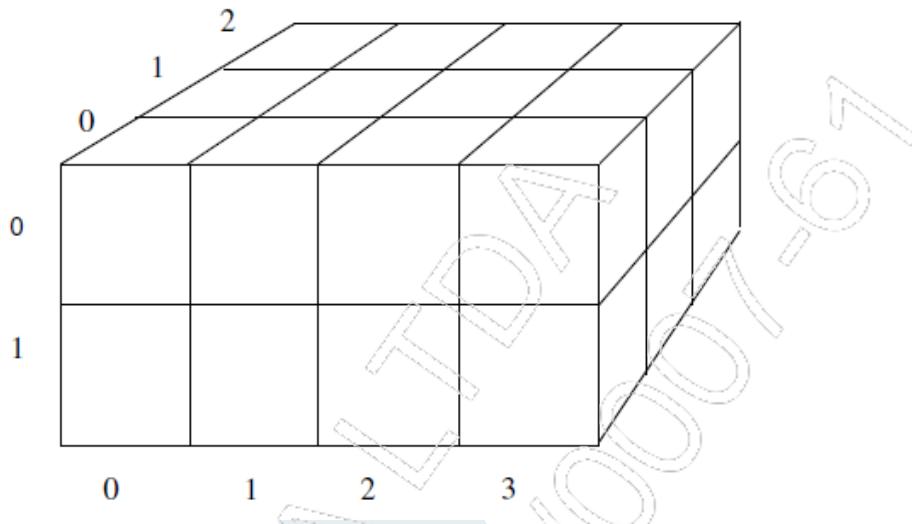
```
int[,] numeros = { { 17, 22, 12, 44, 34 },  
                  { 23, 33, 12, 31, 45},  
                  { 2, 45, 35, 1, 22} };
```

	0	1	2	3	4
0	17	22	12	44	34
1	23	33	12	31	45
2	2	45	35	1	22



- ### • Exemplo 3: Array tridimensional

```
int[, ,] numeros = new int[2, 4, 3];
```



- **Conhecendo o tamanho de um array**

Para saber o tamanho de um array, utilizamos sua propriedade **Length**. Por meio de tal propriedade, é possível iterar por todos os elementos de um array utilizando uma instrução **for**. Na instrução **for**, a iteração é sempre feita do índice **0** para o índice **Length - 1**, nos arrays de uma única dimensão.

Há, ainda, a instrução **foreach**, que permite iterar por todos os elementos de um array diretamente. Ela declara uma variável que recebe automaticamente o valor de cada elemento do array. A instrução **foreach** indica a intenção do código de maneira direta e torna desnecessário utilizar toda a estrutura **for**.

Por essas características, a instrução `foreach` é a maneira preferida para iterar um array. Ainda assim, há algumas situações em que a instrução `for` é mais adequada:

- Para iterar apenas por parte de um array;
  - Para iterar de trás para frente;
  - Quando é necessário que o corpo do loop saiba não apenas o valor, mas o índice do elemento;
  - Quando houver intenção de alterar os elementos do array, já que a variável de iteração de **foreach** é uma cópia somente para leitura de cada um dos elementos que compõem o array.

Veja um exemplo utilizando **for**:

```
static void Main(string[] args)
{
    int[] numeros = { 0, 50, 100, 200, 250, 200, 450, 500 };

    for (int i = 0; i <= numeros.Length - 1; i++)
    {
        Console.WriteLine("numeros[" + i + "] = " + numeros[i]);
    }

    Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:

```
D:\Documentos\Projetos\...
numeros[0] = 0
numeros[1] = 50
numeros[2] = 100
numeros[3] = 200
numeros[4] = 250
numeros[5] = 200
numeros[6] = 450
numeros[7] = 500
```

É possível, também, percorrer os elementos do array por meio de um loop **foreach**:

```
static void Main(string[] args)
{
    int[] numeros = { 0, 50, 100, 200, 250, 200, 450, 500 };

    foreach (int i in numeros)
    {
        Console.WriteLine("numeros[" + i + "] = " + numeros[i]);
    }

    Console.ReadKey();
}
```

- **Arrays com várias dimensões**

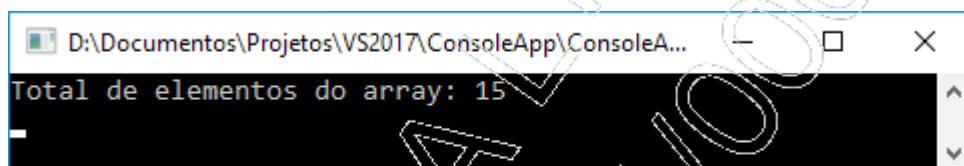
Em um array com várias dimensões, a propriedade **Length** retorna a quantidade total de elementos do array:

```
static void Main(string[] args)
{
    //Definir um array bidimensional
    int[,] numeros = new int[3, 5];

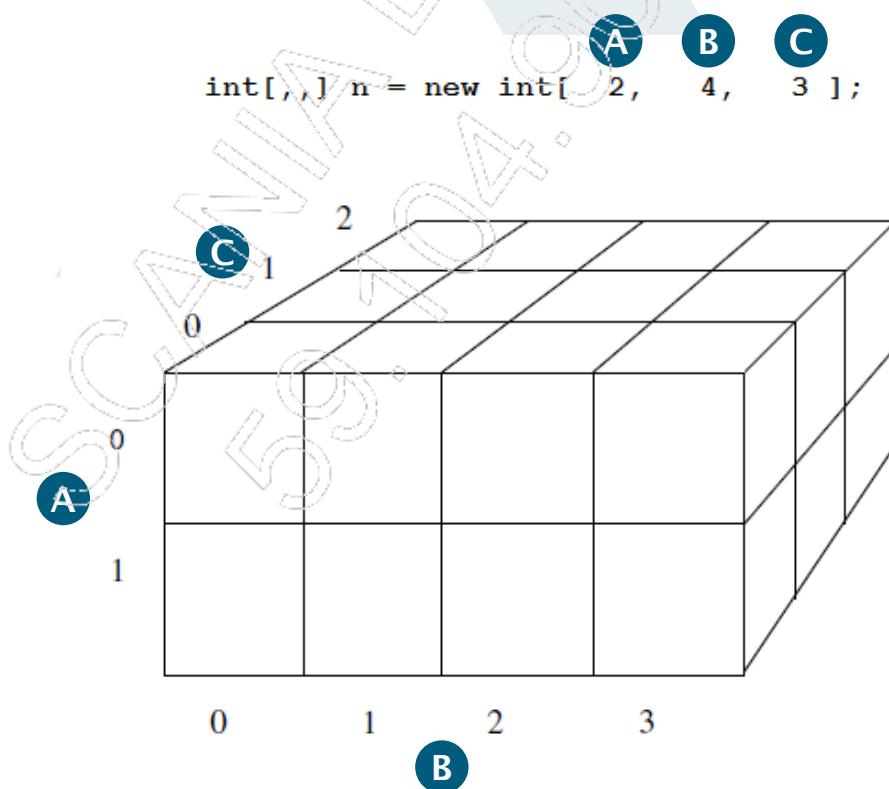
    //Length retornará a quantidade total de elementos do array
    Console.WriteLine("Total de elementos do array: " + numeros.Length);

    Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:



Neste caso, para sabermos o tamanho de cada dimensão, vamos usar o método **GetLength(dimensão)**:



- A – Primeira dimensão ou dimensão 0: `numeros.GetLength(0)` retorna 2;
- B – Segunda dimensão ou dimensão 1: `numeros.GetLength(1)` retorna 4;
- C – Terceira dimensão ou dimensão 2: `numeros.GetLength(2)` retorna 3.

Há alguns pontos que merecem ser destacados com relação ao uso de arrays:

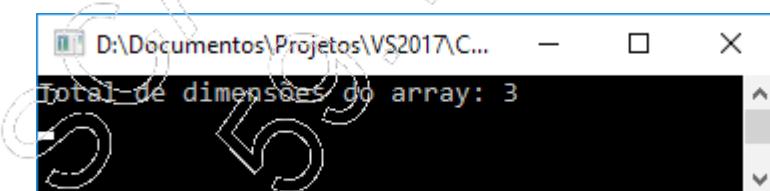
- Em qualquer dimensão de um array, o primeiro elemento tem índice 0;
- Todos os elementos de um array serão do mesmo tipo;
- As dimensões de um array são determinadas em tempo de compilação. Não é possível aumentar ou diminuir o tamanho de um array durante a execução da aplicação;
- Para saber o total de dimensões de um array, deve-se utilizar a propriedade `Rank`:

```
static void Main(string[] args)
{
    //Definir um array tridimensional
    int[, ,] numeros = new int[2, 3, 4];

    Console.WriteLine("Total de dimensões do array: " + numeros.
Rank);

    Console.ReadKey();
}
```

Veja, a seguir, o resultado desse código:



## 8.2.2. Passando um array como parâmetro

Um array pode ser passado e recebido como parâmetro. Tal característica nos permite escrever métodos que podem receber, como parâmetros, um número qualquer de argumentos, de qualquer tipo. Isso pode ser útil em algumas situações, como aquelas em que houver a necessidade de um método que determine o valor mínimo em um conjunto de valores passados como parâmetros.

Veja um exemplo:

```
private static decimal CalcularMedia(decimal[] numeros)
{
    //Definir a variável soma
    decimal soma = 0;

    //Iterar o array numeros e acumular seus valores
    //na variável soma
    for (int i = 0; i <= numeros.GetLength(0) - 1; i++)
    {
        soma += numeros[i];
    }
    //Retornar o total acumulado dividido pelo
    //total de posições do array
    return soma / numeros.Length;
}
```

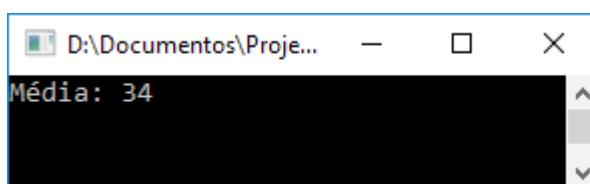
O código da utilização desse método é o seguinte:

```
static void Main(string[] args)
{
    //Definir um array
    decimal[] valores = { 15, 20, 25, 30, 80 };

    //Exibir o resultado
    Console.WriteLine("Média: " + CalcularMedia(valores));

    Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:



Desse modo, evitamos o uso de sobrecargas (utilizadas para declarar dois ou mais métodos, no mesmo escopo, com nomes iguais). Isso torna necessário, porém, escrever um código a mais para preencher o array passado como parâmetro.

É importante destacar, ainda, que quaisquer alterações realizadas nos valores do método que receber um array como parâmetro serão refletidas no array original, já que os arrays são passados por referência.

- **Palavra-chave params**

A palavra-chave **params** funciona como um modificador dos parâmetros de array. Por meio dela, é possível definir um parâmetro de método que utiliza qualquer número de argumentos.

Veja um exemplo:

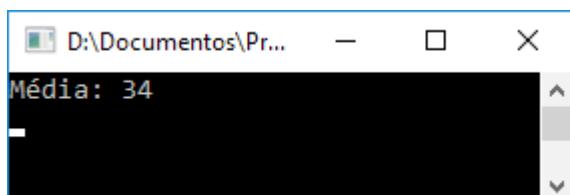
```
private static decimal CalcularMedia(params decimal[] numeros)
{
    //Definir a variável soma
    decimal soma = 0;

    //Iterar o array numeros e acumular seus valores
    //na variável soma
    for (int i = 0; i <= numeros.GetLength(0) - 1; i++)
    {
        soma += numeros[i];
    }
    //Retornar o total acumulado dividido pelo
    //total de posições do array
    return soma / numeros.Length;
}
```

O código da utilização desse método é o seguinte:

```
static void Main(string[] args)
{
    //Exibir o resultado
    Console.WriteLine("Média: " + CalcularMedia(15, 20, 25, 30,
80));
    Console.ReadKey();
}
```

O resultado será o mesmo do exemplo anterior:



É necessário ressaltar alguns aspectos sobre a utilização da palavra-chave **params** na declaração de um método:

- Não é possível utilizar **params** em arrays multidimensionais;
- Só é possível utilizar uma palavra **params** por declaração;
- Após a palavra **params**, não podemos acrescentar outros parâmetros;
- Um método não **params** sempre terá prioridade sobre um método **params**.

- **Params object[]**

Além da possibilidade de passar uma quantidade diversa de argumentos a um método, o C# permite, também, a diversidade de tipos de argumentos. Para isso, é utilizada a classe base **object**. Baseando-se no fato de que o compilador pode gerar um código que transforma tipos-valor em objetos, torna-se possível utilizar um array de parâmetros do tipo **object** para declarar um método que aceite argumentos de qualquer tipo.

A seguir, podemos visualizar um exemplo:

```
private static void ExibirParametros(params object[] valores)
{
    foreach (object i in valores)
    {
        Console.WriteLine("Valor: " + i);
        Console.WriteLine("Tipo: " + i.GetType().ToString());
        Console.WriteLine(new string('-', 30));
    }
}
```

O código da utilização desse método é o seguinte:

```
static void Main(string[] args)
{
    //Exibir o resultado
    ExibirParametros("jatobá", 12.4m, 'k', 23.1f);

    Console.ReadKey();
}
```

Veja, a seguir, o resultado desse código:

```
D:\Documentos\Projetos\... - X
Valor: jatobá
Tipo: System.String
-----
Valor: 12,4
Tipo: System.Decimal
-----
Valor: k
Tipo: System.Char
-----
Valor: 23,1
Tipo: System.Single
```

## 8.3. Coleções

Coleções são estruturas oferecidas pelo .NET Framework para coletar elementos de qualquer tipo. As classes **Collection** podem ser utilizadas no lugar dos arrays, os quais também recolhem elementos, mas apresentam algumas limitações.

O tipo de elemento de uma classe **Collection** é um objeto. Por essa razão, os elementos pertencentes a uma classe **Collection** básica são aceitos como objetos, bem como retornados e mantidos como tais.

Ao introduzirmos um valor em uma coleção, este sofre a operação de **boxing** (se for um tipo-valor, é transformado em um tipo-referência). Porém, devemos aplicar a operação **unboxing** (transformar em referência) ao valor, se, posteriormente, quisermos acessar seus dados.

No caso de um array de objetos, podemos adicionar valores de qualquer tipo nele. Se o tipo do valor for um inteiro, teremos o seguinte:

- O valor é submetido automaticamente ao **boxing**;
- A cópia desse valor inteiro introduzido é referenciada pelo elemento do array, o qual é uma referência de objeto.

As classes **Collection** são mantidas no namespace **System.Collections** e em **System.Collections.Generics**.

### 8.3.1. Diferenças entre coleções e arrays

A tabela adiante descreve as principais diferenças existentes entre arrays e coleções:

Característica	Coleções	Arrays
<b>Redimensionamento</b>	Têm a capacidade de se redimensionar dinamicamente conforme necessário.	Não podem aumentar ou diminuir de tamanho. O array possui tamanho fixo.
<b>Estrutura de dados</b>	Os itens dentro de um loop são disponibilizados como somente leitura.	Não há modo de criar arrays somente leitura. Eles são uma estrutura de dados de leitura e gravação.

### 8.4. A classe ArrayList

Antes de verificarmos a funcionalidade da classe **ArrayList**, devemos considerar alguns aspectos acerca dos arrays comuns, pois sua utilização possui algumas implicações:

- Ao remover um elemento do array, há a necessidade de copiar e mover para cima todos os elementos seguintes, o que resulta em duas cópias do último elemento;
- Para redimensionar um array, é necessário criar um novo array, copiar os elementos (se o array for menor, alguns elementos não serão incluídos) e atualizar as referências do array;
- Para incluir um elemento no array, é preciso mover os elementos para baixo, a fim de liberar um slot, o que faz com que o último elemento do array seja perdido.

Como podemos perceber, os arrays, apesar de úteis, apresentam algumas restrições em sua utilização. Para casos em que o número de elementos varia, a melhor solução é o uso de coleções. Uma das classes do .NET disponíveis para gerenciar uma coleção é a classe **ArrayList**, localizada dentro do namespace **System.Collections**:

```
using System.Collections;
```

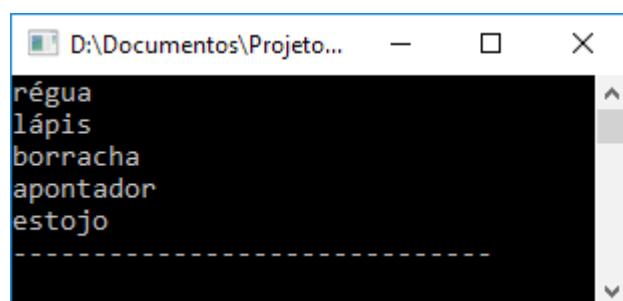
Para instanciarmos um **ArrayList**, utilizamos o código a seguir:

```
static void Main(string[] args)
{
    //Definir o ArrayList
    ArrayList lista = new ArrayList();

    // ArrayList é um array de objetos que tem seus próprios
    // métodos para:
    // inserir, adicionar, procurar, ordenar etc.
    /*
     * Métodos Importantes:
     *      Add(objeto)
     *      Insert(posição, objeto)
     *      IndexOf(objeto)
     *      Remove(objeto)
     *      RemoveAt(posição)
     *      Sort()
     */
    //Adicionando itens ao arraylist
    lista.Add("régua");
    lista.Add("lápis");
    lista.Add("borracha");
    lista.Add("apontador");
    lista.Add("estojo");

    //Exibindo os dados do arraylist na listbox
    for (int i = 0; i < lista.Count; i++)
    {
        Console.WriteLine(lista[i]);
    }
    Console.WriteLine(new string('-', 30));
    Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:



```
D:\Documentos\Projeto...
régua
lápis
borracha
apontador
estojo
```

O método **Insert(posição, item)** insere um novo item na posição selecionada:

```
//Método INSERT
lista.Insert(2, "caderno");
for (int i = 0; i < lista.Count; i++)
{
    Console.WriteLine(lista[i]);
}
Console.WriteLine(new string('-', 30));
```

Por meio do método **Remove(objeto)**, podemos remover um elemento de um **ArrayList** pelo seu valor:

```
//Método REMOVE
lista.Remove("apontador");
for (int i = 0; i < lista.Count; i++)
{
    Console.WriteLine(lista[i]);
}
Console.WriteLine(new string('-', 30));
```

Por meio do método **RemoveAt(posição)**, podemos remover um elemento de um **ArrayList** que está na posição indicada:

```
//Método REMOVEAT
lista.RemoveAt(1);
for (int i = 0; i < lista.Count; i++)
{
    Console.WriteLine(lista[i]);
}
Console.WriteLine(new string('-', 30));
```

O método **IndexOf(objeto)** procura o objeto nos itens do **ArrayList** retornando a posição onde ele foi encontrado ou -1, caso não exista:

```
//Método INDEXOF
int posicao = lista.IndexOf("borracha");

if (posicao >= 0)
{
    Console.WriteLine(
        "borracha está na posição: " + posicao);
}
else
{
    Console.WriteLine("Item não localizado");
}
Console.WriteLine(new string('-', 30));
```

Para ordenar os itens da lista, existe um método chamado **Sort()**, que faz uma ordenação padrão em ordem ascendente:

```
//Método SORT
lista.Sort();
for (int i = 0; i < lista.Count; i++)
{
    Console.WriteLine(lista[i]);
}
Console.WriteLine(new string('-', 30));
```

Para alterar a forma da ordenação para descendente, por exemplo, precisamos implementar uma interface chamada **IComparer**. Ela possui um único método, chamado **Compare**, que recebe dois elementos da lista e deve retornar um número inteiro indicando em que ordem eles deverão ficar.

## 8.5. A classe Stack

Esta classe permite a inclusão de elementos usando o conceito de pilha. Ela usa o sistema LIFO (last-in first-out). Ou seja, o último a entrar é o primeiro a sair. Esta classe não possui indexador, ou seja, não é possível acessar seus elementos pela posição onde estão.

Os principais métodos são os seguintes:

- **Push**: Coloca um item na pilha;
- **Pop**: Retira o último item da pilha e retorna uma referência a esse objeto;
- **Peek**: Retorna o objeto que está no topo da pilha.

Veja um exemplo:

```
static void Main(string[] args)
{
    // é uma lista de objetos do tipo "pilha de pratos"
    // último que entra é o primeiro que sai
    // LIFO (Last-in first-out)
    // não possui indexador
    /*
     * Adicionar elemento no topo da pilha
     *         pilha.Push( objeto );
     *
     * Retirar elemento do topo da pilha
     *         pilha.Pop();
     *
     * Consultar o elemento no topo da pilha
     *         pilha.Peek();
    */
}
```

```
//Definir o objeto Stack
Stack pilha = new Stack();

//Adicionando itens à pilha
pilha.Push("régua");
pilha.Push("lápis");
pilha.Push("borracha");
pilha.Push("apontador");
pilha.Push("estojos");

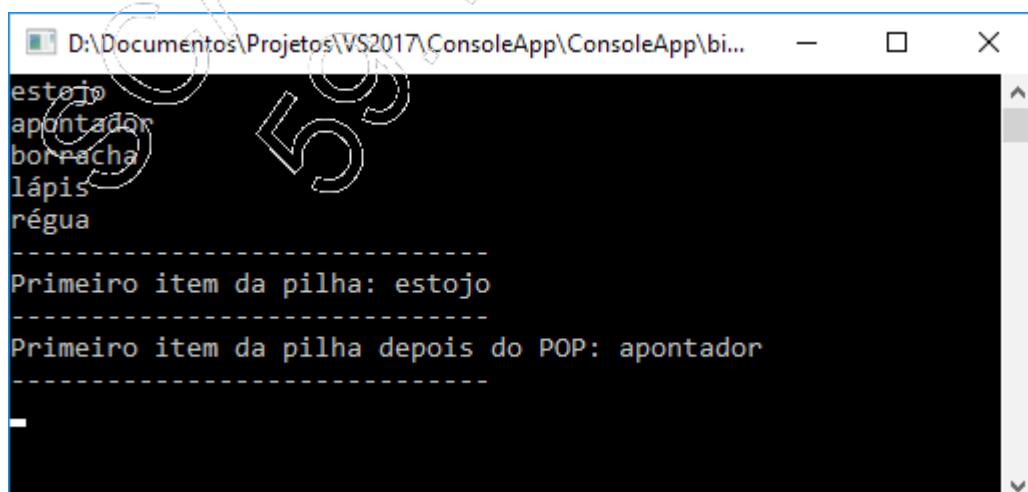
//Exibindo os dados da pilha na listbox
foreach (object i in pilha)
{
    Console.WriteLine(i);
}
Console.WriteLine(new string('-', 30));

//Exibindo o primeiro item da pilha
Console.WriteLine(
    "Primeiro item da pilha: " + pilha.Peek());
Console.WriteLine(new string('-', 30));

//Removendo o primeiro item da pilha
pilha.Pop();

//Exibindo o primeiro item da pilha depois do POP
Console.WriteLine(
    "Primeiro item da pilha depois do POP: " + pilha.Peek());
Console.WriteLine(new string('-', 30));
Console.ReadKey();
}
```

O resultado desse código será o seguinte:



```
D:\Documentos\Projetos\VS2017\ConsoleApp\ConsoleApp\b...
estejo
apontador
borracha
lápis
régua
-----
Primeiro item da pilha: estojo
-----
Primeiro item da pilha depois do POP: apontador
-----
```

Para retirar o elemento que está no topo da pilha e exibi-lo, utilizamos o código a seguir:

```
static void Main(string[] args)
{
    //Definir o objeto Stack
    Stack pilha = new Stack();

    //Adicionando itens à pilha
    pilha.Push("réguas");
    pilha.Push("lápis");
    pilha.Push("borracha");
    pilha.Push("apontador");
    pilha.Push("estojo");

    //Exibindo os dados da pilha na listbox
    foreach (object i in pilha)
    {
        Console.WriteLine(i);
    }
    Console.WriteLine(new string('-', 30));

    // Se tentarmos retirar um elemento da pilha e ela
    // estiver vazia, ocorrerá um erro
    if (pilha.Count > 0)
    {
        // Retira o primeiro elemento da pilha e retorna
        // com o seu conteúdo
        Console.WriteLine("Item removido da pilha: " + pilha.
Pop());
        Console.WriteLine(new string('-', 30));

        //Exibindo os dados da pilha na listbox sem o item removido
        foreach (object i in pilha)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine(new string('-', 30));
    }
    Console.ReadKey();
}
```

## 8.6.A classe Queue

Nesta classe, o elemento é inserido na parte de trás da fila. Esta operação é chamada de **enfileiramento**. Este mesmo elemento sai a partir da frente da fila, operação chamada de **desenfileiramento**. Esse procedimento de entrada e saída recebe o nome de FIFO (first-in first-out), ou seja, o primeiro a entrar é o primeiro a sair. Os principais métodos dessa classe, que também não possui indexador, são os seguintes:

- **Enqueue**: Coloca um item na fila;
- **Dequeue**: Retira o primeiro item da fila e retorna uma referência;
- **Peek**: Retorna o primeiro item.

Veja um exemplo:

```
static void Main(string[] args)
{
    // é uma lista de objetos do tipo "fila"
    // primeiro que entra é o primeiro que sai
    // FIFO (FIRST-IN FIRST-OUT)
    // não possui indexador
    /*
        * Adiciona elemento no final da fila
        *     fila.Enqueue( elemento );
        *
        * Remove elemento do início da fila
        *     fila.Dequeue();
        *
        * Retorna com o primeiro da fila
        *     fila.Peek()
        */
    //Definir o objeto Queue
    Queue fila = new Queue();

    //Adicionando itens à pilha
    fila.Enqueue("régua");
    fila.Enqueue("lápis");
    fila.Enqueue("borracha");
    fila.Enqueue("apontador");
    fila.Enqueue("estojo");

    //Exibindo os dados da fila na listbox
    foreach (object i in fila)
    {
        Console.WriteLine(i);
    }
    Console.WriteLine(new string('-', 30));
}
```

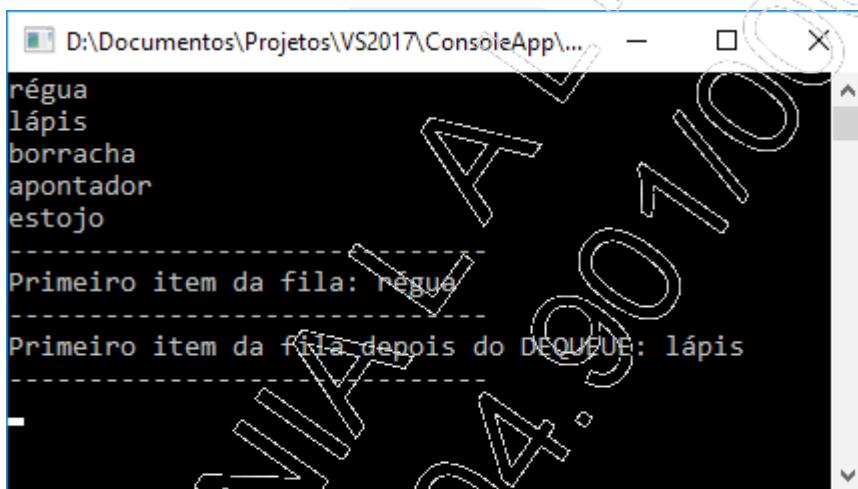
```
//Exibindo o primeiro item da fila
Console.WriteLine("Primeiro item da fila: " + fila.Peek());
Console.WriteLine(new string('-', 30));

//Removendo o primeiro item da fila
fila.Dequeue();

//Exibindo o primeiro item da fila depois do DEQUEUE
Console.WriteLine("Primeiro item da fila depois do DEQUEUE:
" + fila.Peek());
Console.WriteLine(new string('-', 30));

Console.ReadKey();
}
```

O resultado desse código pode ser visto a seguir:



Para retirar o elemento que está no início da fila e exibi-lo, utilizamos o código a seguir:

```
static void Main(string[] args)
{
    //Definir o objeto Queue
    Queue fila = new Queue();

    //Adicionando itens à fila
    fila.Enqueue("régua");
    fila.Enqueue("lápis");
    fila.Enqueue("borracha");
    fila.Enqueue("apontador");
    fila.Enqueue("estojo");
```

```
//Exibindo os dados da fila na listbox
foreach (object i in fila)
{
    Console.WriteLine(i);
}
Console.WriteLine(new string('-', 30));

// se tentarmos retirar um elemento da fila e ela
// estiver vazia, ocorrerá um erro
if (fila.Count > 0)
{
    // retira o primeiro elemento da fila e retorna
    // o seu conteúdo
    Console.WriteLine("Item removido da fila: " + fila.
Dequeue());
    Console.WriteLine(new string('-', 30));

    //Exibindo os dados da fila na listbox sem o item removido
    foreach (object i in fila)
    {
        Console.WriteLine(i);
    }
    Console.WriteLine(new string('-', 30));
}
Console.ReadKey();
}
```

As classes **Stack** e **Queue** também apresentam versões genéricas:

```
Stack<T>
Queue<T>
```

Nesse caso, somente os tipos especificados podem ser armazenados na coleção.

## 8.7. A classe **Hashtable**

Esta classe permite mapear dois itens diferentes, em que o primeiro representa uma chave e o segundo, o valor associado àquela chave.

Ao receber um par valor/chave, a classe **Hashtable** busca automaticamente a chave que pertence ao valor. Então, o valor que está associado a uma chave específica poderá ser obtido.

Esta classe aceita a utilização da instrução **foreach** para o trabalho de iteração dessa mesma classe. Quando isso é feito, uma classe **DictionaryEntry** é retornada. Por meio dela, podemos utilizar as propriedades **Value** e **Key** a fim de acessar os elementos de valor e de chave em qualquer um dos dois arrays de objeto mantidos internamente por **Hashtable**.

Há um método chamado **ContainsKey** cuja função é identificar a existência de uma determinada chave na classe **Hashtable**. A tentativa de inserir uma chave duplicada gera uma exceção.

Os principais métodos são os seguintes:

- **Add**: Adiciona um item. Deve-se passar uma chave e um valor. Tanto a chave como o valor podem ser objetos de qualquer tipo;
- **ContainsKey**: Retorna **true** se a **Hashtable** contiver uma chave específica;
- **ContainsValue**: Retorna **true** se a **Hashtable** contiver um valor específico.

A principal propriedade é **Item**, propriedade padrão que retorna um elemento da coleção do tipo **DictionaryEntry**.

A classe **DictionaryEntry** tem duas propriedades importantes:

- **Key**: A chave para localização do item na coleção;
- **Value**: O valor do item.

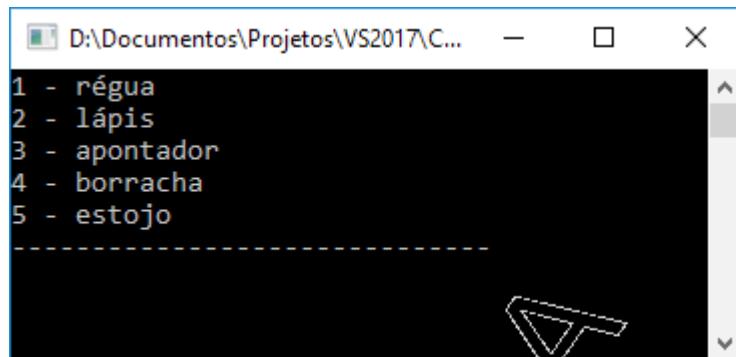
Veja um exemplo:

```
static void Main(string[] args)
{
    //Definir o hashtable
    Hashtable dePara = new Hashtable();

    //Adicionando itens no hashtable
    dePara.Add("1", "régua");
    dePara.Add("2", "lápis");
    dePara.Add("3", "apontador");
    dePara.Add("4", "borracha");
    dePara.Add("5", "estojos");

    //Exibindo os dados do hashtable na listbox
    foreach (DictionaryEntry i in dePara)
    {
        Console.WriteLine(i.Key + " - " + i.Value);
    }
    Console.WriteLine(new string('-', 30));
    Console.ReadKey();
}
```

Veja, a seguir, o resultado desse código:



```
D:\Documentos\Projetos\VS2017\C... - X
1 - régua
2 - lápis
3 - apontador
4 - borracha
5 - estojo
```

## 8.8. A classe List<T>

Uma **List<T>** é muito semelhante a um **ArrayList**, mas com tipo de dado definido. Qualquer tipo de dado pode fazer parte da lista.

Veja um exemplo:

```
static void Main(string[] args)
{
    /*
     * List: É uma coleção na qual podemos definir o tipo
     * de dado dos seus elementos
    */

    //Definir um list de inteiros
    List<int> listaInteiros = new List<int>();
    listaInteiros.Add(10);
    listaInteiros.Add(5);
    listaInteiros.Add(2);
    listaInteiros.Add(8);
    listaInteiros.Add(7);
    listaInteiros.Add(3);
    ///Só aceita tipo Int32
    //listaInteiros.Add("texto");

    foreach (int i in listaInteiros)
    {
        Console.WriteLine(i);
    }
    Console.WriteLine(new string('-', 30));
    Console.ReadKey();
}
```

## 8.8.1. Inicializadores de List

Com uma sintaxe semelhante à que utilizamos para arrays, é possível inicializar uma coleção no momento de sua declaração.

Veja o exemplo a seguir:

```
List<int> lst = new List<int> { 4,8,2,19,28,33,10 };

List<DateTime> lstData = new List<DateTime> {
    new DateTime(2013, 2, 25),
    new DateTime(2013, 3, 30),
    new DateTime(2013, 4, 15),
    new DateTime(2013, 5, 10) };
```

## 8.9. As classes HashSet<T> e SortedSet<T>

Estas classes constituem um grupo especial de coleções que, diferentemente de `List<T>`, não permitem elementos duplicados. Particularmente, a classe `SortedSet<T>` insere elementos de forma *exclusive*, porém esses elementos entram de forma ordenada na coleção. Vamos considerar os exemplos a seguir:

```
class Program
{
    static void Main(string[] args)
    {
        HashSet<string> cursos = new HashSet<string>()
        {
            "Java", "PHP", "Node.js", "Java", "SQL Server"
        };

        Console.WriteLine("Quant. Elementos em cursos: " + cursos.
Count);

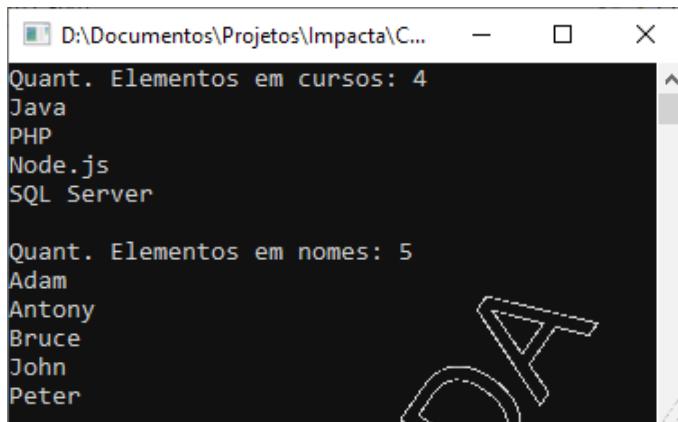
        foreach (var item in cursos)
        {
            Console.WriteLine(item);
        }

        SortedSet<string> nomes = new SortedSet<string>()
        {
            "Peter", "John", "Bruce", "Adam", "John", "Antony"
        };

        Console.WriteLine("\nQuant. Elementos em nomes: " + nomes.
Count);

        foreach (var item in nomes)
        {
            Console.WriteLine(item);
        }
        Console.ReadKey();
    }
}
```

O resultado da execução:



```
D:\Documentos\Projetos\Impacta\C#\Exercicios> Quant. Elementos em cursos: 4
Java
PHP
Node.js
SQL Server

Quant. Elementos em nomes: 5
Adam
Antony
Bruce
John
Peter
```

## 8.10.A interface **IEnumerable<T>**

Praticamente todas as coleções do framework implementam esta interface. As classes **ArrayList**, **Stack** e **Queue** não genéricas implementam a versão não genérica da interface, **IEnumerable**.

A interface **IEnumerable** possui três características básicas:

- Não permite alterações nas coleções, ou seja, funciona apenas como leitura;
- Não fornece nenhuma informação sobre a coleção, além da necessária para percorrê-la;
- Não permite acesso aleatório às posições. Somente conseguimos percorrê-la sequencialmente, ou voltar ao início.

A implementação desta interface está demonstrada adiante:

```
namespace System.Collections.Generic
{
    public interface IEnumerable<out T> : IEnumerable
    {
        IEnumerator<T> GetEnumerator();
    }
}
```

e sua versão não genérica:

```
namespace System.Collections
{
    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }
}
```

Observe que ela possui apenas um método, **GetEnumerator()**, que, por sua vez, é do tipo **IEnumerator<T>** (ou **IEnumerator**, no caso não genérico).

A interface **IEnumerator**, por sua vez, possui a seguinte implementação:

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

E a versão genérica:

```
namespace System.Collections.Generic
{
    public interface IEnumerator<out T> : IDisposable, IEnumerator
    {
        T Current { get; }
    }
}
```

O **IEnumerator** é o que permite a iteração pelos elementos da coleção. Quando executamos uma iteração com **foreach**, por exemplo, o método **GetEnumerator**. é chamado e cada passo do loop chama o método **MoveNext**.

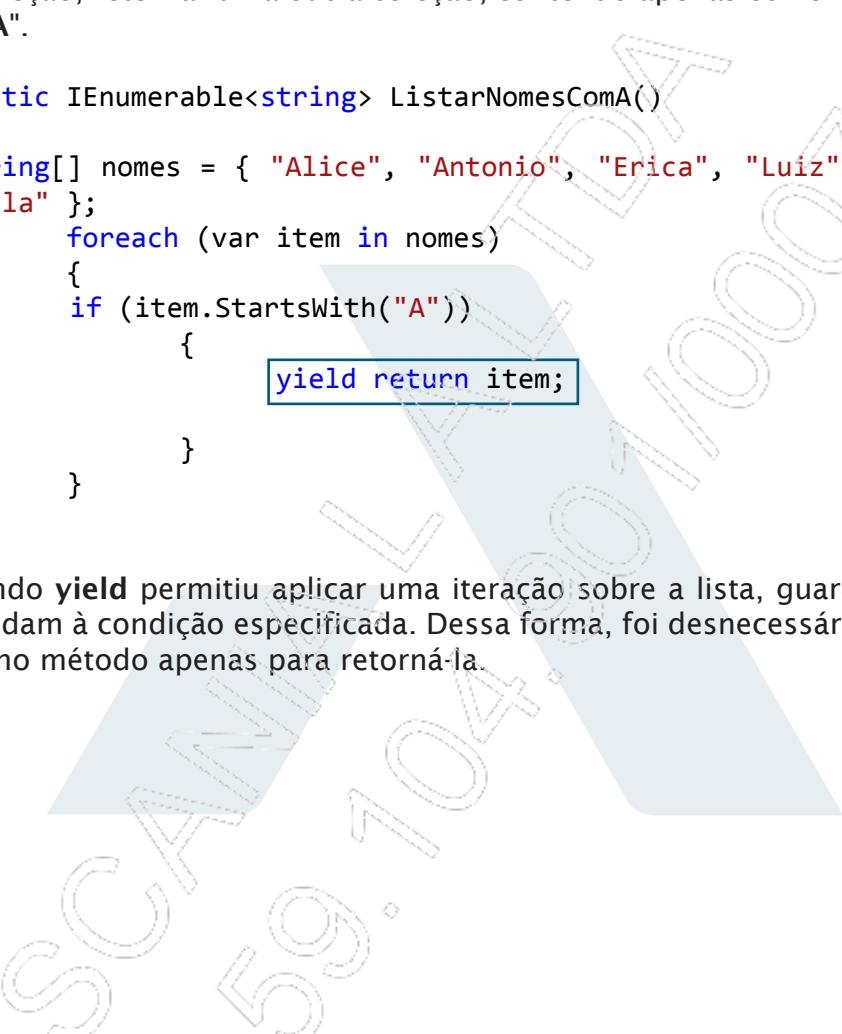
### 8.10.1. A palavra reservada `yield`

Quando usamos a palavra reservada `yield` em uma instrução, indicamos que o membro em que ela é exibida é um iterador (um `IEnumerator`). O uso de `yield` elimina a necessidade de uma classe adicional durante a implementação de `IEnumerable` e `IEnumerator`.

Vamos considerar um exemplo: suponha uma coleção de nomes. Desejamos, a partir dessa coleção, retornar uma outra coleção, contendo apenas os nomes iniciados com a letra "A".

```
static IEnumerable<string> ListarNomesComA()
{
    string[] nomes = { "Alice", "Antonio", "Erica", "Luiz", "Paulo", "Ana
    Paula" };
    foreach (var item in nomes)
    {
        if (item.StartsWith("A"))
        {
            yield return item;
        }
    }
}
```

O comando `yield` permitiu aplicar uma iteração sobre a lista, guardando os valores que atendam à condição especificada. Dessa forma, foi desnecessário criar uma nova coleção no método apenas para retorná-la.



## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Um array é uma sequência não ordenada de elementos do mesmo tipo. Ele, que é um tipo especial de variável, contém múltiplas dimensões, e seus valores são identificados por um índice;
- Os arrays podem ter várias dimensões de dados, o que os torna perfeitamente adequados para armazenar conjuntos de informações relacionadas;
- A maneira preferida para iterar por um array é utilizando a instrução **foreach**;
- Um array pode ser passado e recebido como parâmetro, o que é útil quando for necessário um método que determine o valor mínimo em um conjunto de valores passados como parâmetros;
- O .NET Framework oferece estruturas chamadas coleções, que servem para coletar elementos de qualquer tipo. As classes **Collection** podem ser utilizadas no lugar dos arrays, os quais também recolhem elementos, mas apresentam algumas limitações;
- A classe **ArrayList** é uma das classes do .NET disponíveis para gerenciar uma coleção. Ela está localizada dentro do namespace **System.Collections**;
- A classe **Stack** permite a inclusão de elementos que são colocados em uma pilha. Seus principais métodos são **Push**, **Pop** e **Peek**;
- Na classe **Queue**, ocorre a operação de enfileiramento, isto é, o elemento é inserido na parte de trás da fila. Os principais métodos dessa classe são **Enqueue**, **Dequeue** e **Peek**;
- A classe **Hashtable**, cuja ação é denominada array associativo, mapeia para um elemento de um tipo diferente do inteiro (**int**);
- A interface **IEnumerable<T>** é implementada por praticamente todas as coleções. **ArrayList** e as versões não genéricas de **Stack** e **Queue** implementam a versão não genérica **IEnumerable**;
- A palavra reservada **yield** produz um iterador (um **IEnumerator**), simplificando o retorno de coleções a partir de outra coleção.

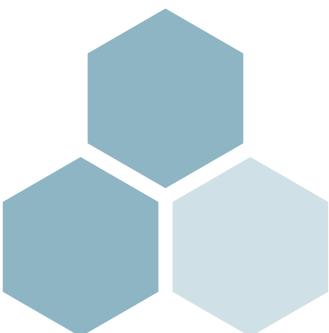




8

# Coleções

Teste seus conhecimentos



Editora  
**IMPACTA**



## 1. Quais valores serão apresentados na tela, respectivamente?

```
int[] numeros = new int[8];  
  
for (int i = 0; i < numeros.Length; i++)  
{  
    numeros[i] = 3 * i + 1;  
}  
Console.WriteLine(numeros[6].ToString());  
Console.WriteLine(numeros[3].ToString());  
Console.WriteLine(numeros[2].ToString());
```

- a) 22, 13, 10
- b) 12, 32, 34
- c) 20, 11, 8
- d) 0,0,0
- e) 19, 10, 7

## 2. Em quais posições da matriz estão os elementos F, H e J?

```
string[,] letras = new string[3,4];
```

A	B	C	D
E	F	G	H
I	J	K	L

- a) letras[2,2] - letras[2,4] - letras[3,2]
- b) letras[1,1] - letras[1,3] - letras[2,1]
- c) letras[3,3] - letras[3,5] - letras[4,3]
- d) letras[1,1] - letras[3,1] - letras[1,2]
- e) letras[1,1] - letras[1,3] - letras[1,2]

**3. Qual o método que adiciona um elemento no final (último) de um objeto do tipo Stack?**

- a) Pop()
- b) Enqueue()
- c) Add
- d) Push()
- e) Não há como adicionar um elemento no final de um Stack.

**4. Qual a alternativa que completa adequadamente a frase a seguir?**

Para ordenar um objeto List de modo diferente do padrão, precisamos implementar uma interface chamada \_\_\_\_\_ e o seu método \_\_\_\_\_.

- a) Comparer, Comparer
- b) Compare, IComparer
- c) IComparer, Sort()
- d) IComparable, CompareTo()
- e) Não há como ordenar um objeto List.

**5. O comando yield deve ser usado em métodos cujo tipo de retorno seja declarado como:**

- a) IComparable<T>
- b) Queue
- c) IEnumerable<T>
- d) IDisposable
- e) List<T>





8

# Coleções



Mãos à obra!

SCAMANIA  
50.  
ALTA  
10007-67



Editora  
**IMPACTA**



## Laboratório 1

### A – Registrando movimentação do usuário e apresentando o extrato

#### Objetivos:

A partir deste laboratório, teremos mais recursos para avançarmos na nossa aplicação. Teremos a oportunidade de registrar a movimentação do usuário e apresentar o extrato. Várias alterações serão realizadas nos projetos existentes, que estamos atualizando capítulo a capítulo.

1. Crie um novo solution, vazio, chamado **Capítulo08.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models**, **Lab.Views**, **Lab.Interfaces**, **Lab.Arquivos** e **Lab.Dados** do Capítulo 7, na pasta do solution **Capítulo08.Labs**;
3. Adicione esses projetos ao novo solution;
4. No projeto **Lab.Models**, adicione uma classe chamada **Movimento**. Essa classe servirá para registrar cada movimento (saque ou depósito). Definiremos a sobrescrita do método **ToString** de forma que cada objeto dessa classe represente uma linha do extrato:

```
namespace Lab.Models
{
    public class Movimento
    {
        public DateTime Data { get; set; }
        public string Historico { get; set; }
        public Operacoes Operacao { get; set; }
        public double Valor { get; set; }

        public override string ToString()
        {
            return string.Format("{0:dd/MM/yyyy} {1,-10} {2,10:N2}",
                Data, Historico, Valor);
        }
}
```

5. Altere a classe **Conta** de modo a adicionar uma nova propriedade, representando uma coleção de objetos **Movimento**, a instância dessa coleção no construtor e um método chamado **RegistrarMovimento** para adicionar a operação realizada. Neste ponto, é conveniente também transferirmos a propriedade **ClienteInfo** da classe **ContaCorrente** para a classe **Conta**. Isso nos poupará da necessidade de realizar operações de typecast todo momento que desejarmos acessar o cliente a partir da conta. Vamos também sobrepor o método **ToString** para representar a agência e a conta, separadas por uma barra, e incluir o método abstrato **EfetuarOperacao**. Isso mesmo! O método **EfetuarOperacao** será abstrato nessa classe, e passará a ser sobreescrito nas classes **ContaCorrente** e **ContaEspecial**:

```
namespace Lab.Models
{
    public abstract class Conta
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }

        public List<Movimento> Movimentos { get; set; }

        public Cliente ClienteInfo { get; set; }

        public Conta(int Banco, string Agencia, string Conta)
        {
            if(this.Movimentos == null)
            {
                this.Movimentos = new List<Movimento>();
            }

            this.NumeroBanco = Banco;
            this.NumeroAgencia = Agencia;
            this.NumeroConta = Conta;
        }

        protected void RegistrarMovimento(double valor, Operacoes
operacao)
        {
            this.Movimentos.Add(new Movimento()
            {
                Data = DateTime.Now,
                Historico = operacao == Operacoes.Saque ? "SAQUE" :
"DEPÓSITO",
                Operacao = operacao,
                Valor = valor
            });
        }

        public abstract string MostrarExtrato();

        public abstract void EfetuarOperacao(double valor,
Operacoes operacao = Operacoes.Deposito);

        public override string ToString()
        {
            return this.NumeroAgencia + "/" + this.NumeroConta;
        }
    }
}
```

6. Altere a classe **Cliente** de modo a substituir a propriedade **Conta** por uma coleção de contas. Isso significa que a partir deste momento, um cliente poderá possuir várias contas. A propriedade passará a se chamar **Contas**. Adicionaremos também a sobreescrita ao método **ToString** para viabilizar a apresentação do objeto em um componente **ComboBox**:

```
namespace Lab.Models
{
    public abstract class Cliente
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public List<Conta> Contas { get; set; }

        public Cliente(string Nome, Sexos Sexo)
        {
            if(this.Contas == null)
            {
                this.Contas = new List<Conta>();
            }

            this.Nome = Nome;
            this.Sexo = Sexo;
        }

        public Cliente(string Nome, Sexos Sexo, int idade)
            : this(Nome, Sexo)
        {
            this.Idade = Idade;
        }

        public Cliente(string Nome, Sexos Sexo, Endereco endereco)
            : this(Nome, Sexo)
        {
            this.EnderecoResidencial = endereco;
        }

        ~Cliente()
        {
            if(this.Contas != null)
            {
                this.Contas = null;
            }
        }
    }
}
```

```

    public virtual string Exibir()
    {
        return $"Nome: {this.Nome}\n" +
            $"Idade: {this.Idade}\n" +
            $"Sexo: {this.Sexo}\n" +
            $"ENDEREÇO DO CLIENTE:\n" +
            $"{this.EnderecoResidencial.Exibir()}";
    }

    public override string ToString()
    {
        return this.Nome;
    }
}

```

7. Altere o método **EfetuarOperacao** das classes **ContaCorrente** e **ContaEspecial**, de forma a torná-lo sobreescrito na classe **ContaCorrente**, inclua a chamada ao método **RegistrarMovimento** da classe **Conta** e implemente o método **MostrarExtrato** também nestas duas classes:

- **Classe ContaCorrente**

```

namespace Lab.Models
{
    public class ContaCorrente : Conta
    {
        public double Saldo { get; protected set; }
        //public Cliente ClienteInfo { get; set; }

        public ContaCorrente(int Banco, string Agencia, string Conta):
            base(Banco, Agencia, Conta)
        { }

        public ContaCorrente(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public override void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            switch (operacao)
            {
                case Operacoes.Deposito:
                    this.Saldo += valor;
                    break;
                case Operacoes.Saque:
                    if(valor <= this.Saldo)
                    {

```

```
        this.Saldo -= valor;
    }
    break;
}
base.RegistrarMovimento(valor, operacao);
}

public virtual string Exibir()
{
    string cliente = this.ClienteInfo != null ?
        this.ClienteInfo.Exibir() + '\n' : "";
    return $"{cliente}" +
        $"Banco: {this.NumeroBanco}\n" +
        $"Agência: {this.NumeroAgencia}\n" +
        $"Conta: {this.NumeroConta}\n" +
        $"Saldo Atual: {this.Saldo}";
}

public override string MostrarExtrato()
{
    StringBuilder builder = new StringBuilder();
    builder.Append($"Cliente: {ClienteInfo.Nome}\n")
        .Append($"Banco: {NumeroBanco}\n")
        .Append($"Agência: {NumeroAgencia}\n")
        .Append($"Conta: {NumeroConta}\n")
        .Append(new string('-', 35) + '\n');

    if(this.Movimentos.Count() == 0)
    {
        builder.Append("Nenhum movimento registrado para esta\n");
        conta");
    }
    else
    {
        foreach(var item in this.Movimentos)
        {
            builder.Append($"{item}\n");
        }
        builder.Append(new string('-', 35) + '\n');
        builder.Append($"Saldo: {this.Saldo:c}");
    }
    return builder.ToString();
}
}
```

- Classe ContaEspecial

```
namespace Lab.Models
{
    public class ContaEspecial : ContaCorrente
    {
        public double Limite { get; set; }

        public ContaEspecial(int Banco, string Agencia, string Conta) :
            base(Banco, Agencia, Conta)
        {}

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente, double Limite)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
            this.Limite = Limite;
        }

        //métodos sobrescritos
        public override void EfetuarOperação(double valor,
            Operações operação = Operações.Depósito)
        {
            switch (operação)
            {
                case Operações.Depósito:
                    this.Saldo += valor;
                    break;
                case Operações.Saque:
                    if (valor <= (this.Saldo + this.Limite))
                    {
                        this.Saldo -= valor;
                    }
                    break;
            }
            base.RegistrarMovimento(valor, operação);
        }

        public override string Exibir()
        {
            return $"{base.Exibir()}\n" +
                $"Limite: {this.Limite}\n" +
                $"Saldo Disponível: {this.Saldo + this.Limite}";
        }
    }
}
```

```
        public override string MostrarExtrato()
    {
        return new StringBuilder(base.MostrarExtrato())
            .Append($"\\nLímite: {this.Límite:c}")
            .Append($"\\nSaldo Disponível: {this.Límite + this.
Saldo:c}")
            .ToString();
    }
}
```

8. No projeto **Lab.Models**, adicione uma pasta chamada **Utils**. Nessa pasta, adicione uma classe chamada **Metodos**. Nessa classe, adicione:

- Uma coleção estática de Clientes;
- Uma coleção estática de Contas;
- Um método estático para adicionar um cliente à coleção de Clientes;
- Um método estático para adicionar uma conta à coleção de Contas;
- Um método estático para retornar a coleção de Clientes;
- Um método estático para retornar a coleção de Contas.

```
namespace Lab.Models.Utils
{
    public class Metodos
    {
        private static HashSet<Cliente> clientes = new HashSet<Cliente>();
        private static HashSet<Conta> contas = new HashSet<Conta>();

        public static void AdicionarCliente(Cliente cliente)
        {
            clientes.Add(cliente);
        }

        public static void AdicionarConta(Conta conta)
        {
            contas.Add(conta);
        }

        public static IEnumerable<Cliente> ListarClientes()
        {
            return clientes;
        }

        public static IEnumerable<Conta> ListarContas()
        {
            return contas;
        }
    }
}
```

## Laboratório 2

### A – Realizando operações de saque e depósito

#### Objetivos:

Nesta fase da aplicação, nós vamos alterar a interface gráfica para incluir um recurso que permita realizar operações de saque e depósito. Além disso, vamos atualizar o evento do botão **incluirClienteButton** de forma a apresentar a lista de clientes incluídos, em vez de apenas mostrar seus dados na tela. Implementaremos o código no botão **incluirContaButton**, que até o momento ficou sem codificação.

1. No arquivo **MainWindow.xaml**, implemente a interface gráfica correspondente à terceira aba intitulada "Operações Bancárias". Vamos incluir um recurso para:

- Selecionar uma conta em um controle **ComboBox**;
- Selecionar a operação: saque ou depósito;
- Fornecer o valor;
- Executar a operação, através de um botão de comandos.

Vamos também incluir uma caixa de textos com visualização de várias linhas, e um botão de comandos para exibir o extrato da conta selecionada, quando clicado:

```
<TabItem Header="Operações Bancárias">

    <Grid Background="LightSalmon">
        <Grid.RowDefinitions>
            <RowDefinition Height="100" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid Grid.Row="0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
            </Grid.RowDefinitions>

            <!--Componentes Label-->
            <Label Name="numeroContaLabel" Content="Conta:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="0" Grid.Column="0"/>

            <Label Name="operacaoLabel" Content="Operação:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="1" Grid.Column="0"/>
        </Grid>
    </Grid>
</TabItem>
```

```
<Label Name="valorLabel" Content="Valor:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="2" Grid.Column="0"/>  
  
<!--Componentes TextBox e ComboBox-->  
<ComboBox Grid.Row="0" Grid.Column="1"  
          Name="numeroContaComboBox"  
          VerticalAlignment="Center"  
          HorizontalAlignment="Left"  
          Width="150">  
</ComboBox>  
  
<ComboBox Grid.Row="1" Grid.Column="1"  
          Name="operacaoComboBox"  
          VerticalAlignment="Center"  
          HorizontalAlignment="Left"  
          Width="150">  
</ComboBox>  
  
<TextBox Name="valorTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="2" Grid.Column="1"/>  
  
<!--Botões para incluir operação e mostrar  
extrato-->  
<StackPanel Grid.Row="3" Grid.Column="1"  
           Orientation="Horizontal"  
           VerticalAlignment="Center">  
  
    <Button Name="executarButton"  
           Content="Executar"  
           VerticalAlignment="Center"  
           Width="100" />  
    <Button Name="extratoButton"  
           Content="Mostrar Extrato"  
           VerticalAlignment="Center"  
           Width="150" />  
  
</StackPanel>  
</Grid>  
  
<Grid Grid.Row="1">  
    <TextBox Name="extratoTextBox"  
            Height="250"  
            TextWrapping="Wrap"  
            AcceptsReturn="True"  
            VerticalScrollBarVisibility="Auto"  
            FontFamily="Courier New"  
            FontSize="16" />  
</Grid>  
  
</Grid>  
  
</TabItem>
```

2. Altere o evento click do botão **incluirClienteButton** de modo a adicionar o cliente na lista de clientes, por meio do método **AdicionarCliente** da classe **Metodos**, e vincule a lista de clientes ao componente **clienteComboBox**. O vínculo será realizado através do método **ListarClientes**, em que o valor apresentado é o nome do cliente:

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
            sexoComboBox.SelectedIndex = 0;

            operacaoComboBox.Items.Add(Operacoes.Deposito);
            operacaoComboBox.Items.Add(Operacoes.Saque);
            operacaoComboBox.SelectedIndex = 0;
        }

        private void incluirClienteButton_Click(object sender,
            RoutedEventArgs e)
        {
            //obtendo os dados do endereço
            Endereco endereco = new Endereco(
                ruaTextBox.Text,
                int.Parse(numeroTextBox.Text),
                cidadeTextBox.Text,
                cepTextBox.Text);

            //obtendo os dados do cliente
            int digitos = documentoTextBox.Text.Length;
            Cliente cliente;

            if(digitos == 11)
            {
                cliente = new ClientePF(
                    documentoTextBox.Text,
                    nomeTextBox.Text,
                    (Sexos)sexoComboBox.SelectedItem,
                    int.Parse(idadeTextBox.Text),
                    endereco);
            }
            else if(digitos == 14)
            {
                cliente = new ClientePJ(
                    documentoTextBox.Text,
                    nomeTextBox.Text,
                    (Sexos)sexoComboBox.SelectedItem,
                    int.Parse(idadeTextBox.Text),
                    endereco);
            }
        }
    }
}
```

```
        }
    else
    {
        throw new Exception("Documento inválido");
    }

    //adicionando o novo cliente na lista
    Metodos.AdicionarCliente(cliente);

    //vinculando a lista de clientes ao
    //componente ComboBox
    clienteComboBox.ItemsSource = Metodos.ListarClientes();
    MessageBox.Show("Cliente incluído com sucesso!");

}

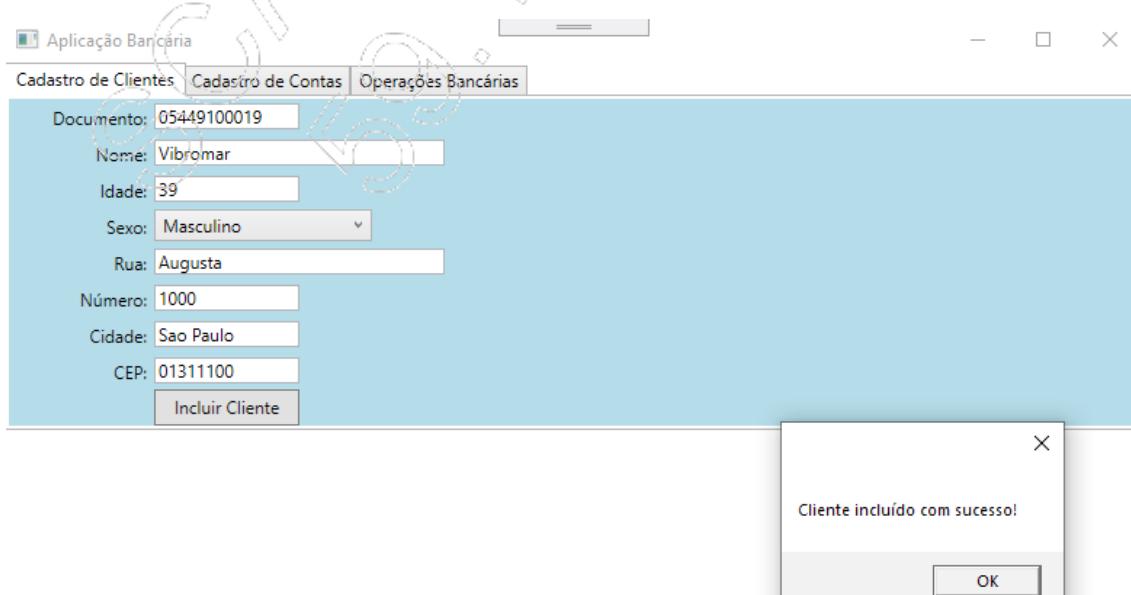
//MessageBox.Show(cliente.Exibir());
}

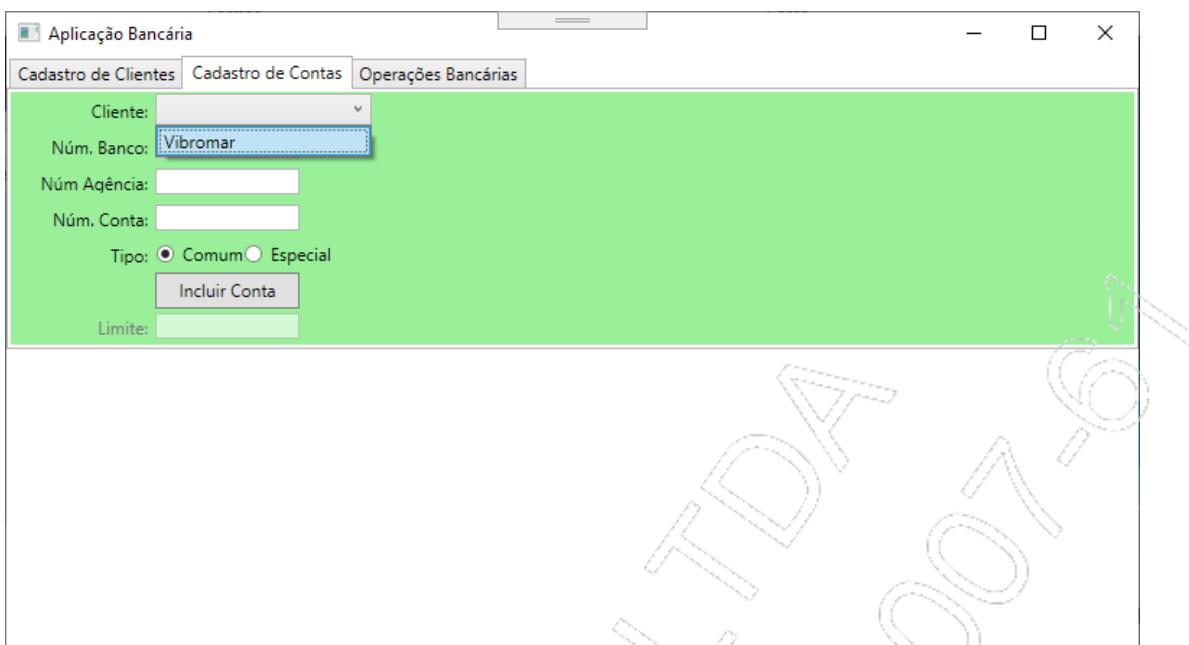
private bool VerificarEspecial { get; set; }

private void especialRadioButton_Checked(object sender,
RoutedEventArgs e)
{
    var radio = sender as RadioButton;
    VerificarEspecial = (radio == especialRadioButton);

    limiteLabel.IsEnabled = VerificarEspecial;
    limiteTextBox.IsEnabled = VerificarEspecial;
}
}
```

Até o momento, a execução da aplicação está permitindo a inclusão do cliente e sua adição no combobox, para posterior inclusão da conta. Vamos ver um exemplo de execução:





3. Escreva o código para o evento **incluirContaButton**. Para isso, dê um duplo-clique neste componente na interface gráfica, ou gere um novo método a partir do código XAML. O código para este evento deverá criar uma conta corrente simples ou especial, dependendo da escolha do usuário. Usaremos a propriedade **VerificarEspecial** para especificar a instância da conta: **ContaCorrente** ou **ContaEspecial** (lembre-se que a classe **Conta** é abstrata);

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
        }

        private void incluirClienteButton_Click(object sender,
            RoutedEventArgs e)
        {
            //obtendo os dados do endereço
            Endereco endereco = new Endereco(
                ruaTextBox.Text,
                int.Parse(numeroTextBox.Text),
                cidadeTextBox.Text,
                cepTextBox.Text);
        }
    }
}
```

```
//obtendo os dados do cliente
int digitos = documentoTextBox.Text.Length;
Cliente cliente;

if(digitos == 11)
{
    cliente = new ClientePF(
        documentoTextBox.Text,
        nomeTextBox.Text,
        (Sexos)sexoComboBox.SelectedItem,
        int.Parse(idadeTextBox.Text),
        endereco);
}
else if(digitos == 14)
{
    cliente = new ClientePJ(
        documentoTextBox.Text,
        nomeTextBox.Text,
        (Sexos)sexoComboBox.SelectedItem,
        int.Parse(idadeTextBox.Text),
        endereco);
}
else
{
    throw new Exception("Documento inválido");
}

//adicionando o novo cliente na lista
Metodos.AdicionarCliente(cliente);

//vinculando a lista de clientes ao
//componente ComboBox
clienteComboBox.ItemsSource = Metodos.ListarClientes();
MessageBox.Show("Cliente incluído com sucesso!");

//MessageBox.Show(cliente.Exibir());
}

private bool VerificarEspecial { get; set; }

private void especialRadioButton_Checked(object sender,
    RoutedEventArgs e)
{
    var radio = sender as RadioButton;
    VerificarEspecial = (radio == especialRadioButton);

    limiteLabel.IsEnabled = VerificarEspecial;
    limiteTextBox.IsEnabled = VerificarEspecial;
}
```

```
    private void incluirContaButton_Click(object sender, RoutedEventArgs
e)
{
    Conta conta;
    var cliente = (Cliente)clienteComboBox.SelectedItem;

    int banco = int.Parse(bancoTextBox.Text);
    string agencia = agenciaTextBox.Text;
    string numConta = contaTextBox.Text;

    if (VerificarEspecial)
    {
        conta = new ContaEspecial(banco, agencia, numConta);
        ((ContaEspecial)conta).Limite = double.Parse(limiteTextBox.
Text);
    }
    else
    {
        conta = new ContaCorrente(banco, agencia, numConta);
    }
    //vinculamos o cliente selecionado à nova conta
    conta.ClienteInfo = cliente;

    //adicionamos a nova conta à lista de contas do
    //cliente selecionado
    cliente.Contas.Add(conta);

    //incluimos a nova conta à lista global de contas
    Metodos.AdicionarConta(conta);

    //vinculamos a lista global de contas ao
    //componente ComboBox
    numeroContaComboBox.ItemsSource = Metodos.ListarContas();

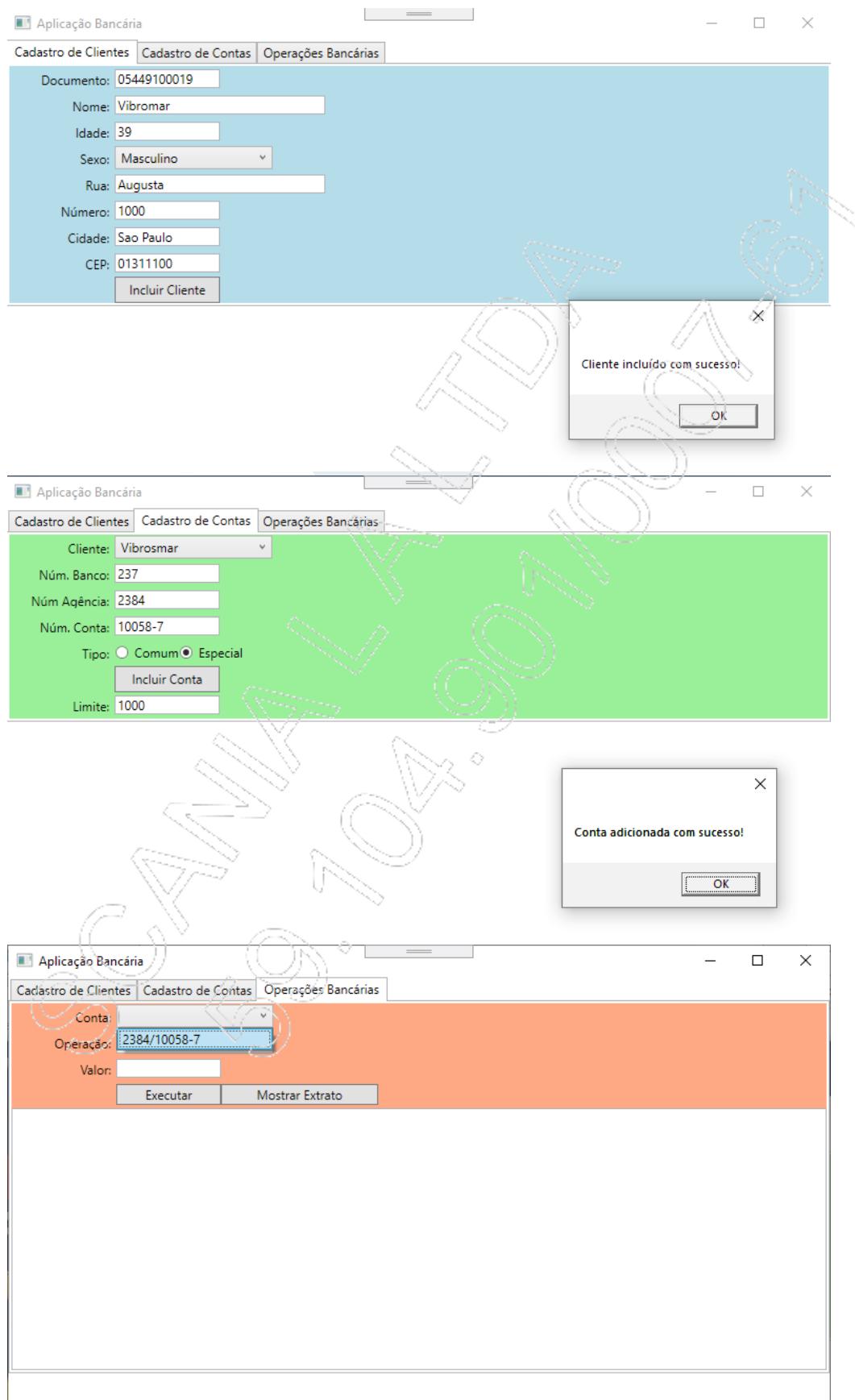
    MessageBox.Show("Conta adicionada com sucesso!");
}
}
```

Até este momento, já podemos:

- adicionar um cliente;
- selecionar o cliente e lhe atribuir uma ou mais contas;
- selecionar as contas adicionadas.

# Programando com C#

Vamos ver o processo de execuções:



4. Escreva o código para os botões **executarButton** e **extratoButton**. O evento click do botão **executarButton** colherá a conta selecionada no componente **ComboBox**, a operação (também no componente **ComboBox** correspondente) e o valor da operação. Executaremos o método **ExecutarOperacao** para atualizar o saldo e registrar, para cada conta, o movimento executado. O evento click do botão **extrato** permitirá visualizar as operações realizadas pela conta selecionada. Para gerar os dois eventos, clique duas vezes sobre cada botão:

- **Botão executarButton**

```
private void executarButton_Click(object sender,
    RoutedEventArgs e)
{
    //obtendo a conta selecionada
    var conta = (Conta)numeroContaComboBox.SelectedItem;

    //obtendo a operação
    var operacao = (Operacoes)operacaoComboBox.SelectedItem;

    //obtendo o valor da operação
    double valor = double.Parse(valorTextBox.Text);

    //executar a operação
    conta.EfetuarOperacao(valor, operacao);

    MessageBox.Show("Operação realizada com sucesso!");
    valorTextBox.Clear();
}
```

- **Botão extratoButton**

```
private void extratoButton_Click(object sender,
    RoutedEventArgs e)
{
    var conta = (Conta)numeroContaComboBox.SelectedItem;

    extratoTextBox.Text = conta.MostrarExtrato();
}
```

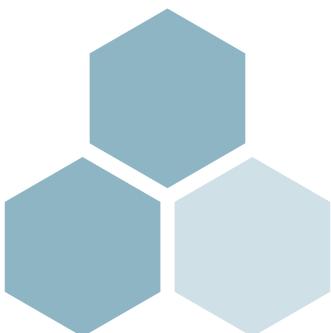
5. Teste a aplicação.



# 9

# Tratamento de exceções

- Tipos de erros;
- O bloco try ... catch;
- A classe Exception e suas subclasses;
- O comando throw;
- O bloco finally;
- O bloco using.



## 9.1. Introdução

O conceito de exceções está associado a erros decorrentes da execução da aplicação, normalmente causados por entradas inválidas, acesso a arquivos que não estão mais disponíveis, entre muitas outras razões. Exceções são diferentes de erros de compilação, que se relacionam com erros na estrutura do código.

O propósito deste capítulo é apresentar os principais tipos de erros e como lidar com as exceções, quando elas surgirem.

## 9.2. Tipos de erros

Na linguagem C#, contamos com um mecanismo sofisticado que nos auxilia a produzir códigos de manipulação de exceções organizados e eficientes.

Há três tipos de erros:

- Erros de lógica;
- Erros de compilação;
- Erros de execução.

Veja, a seguir, a descrição de cada um deles.

### 9.2.1. Erro de lógica

Mais comumente chamado de erro humano, este erro ocorre quando o programador escreve um código que gera resultados inesperados durante a execução. Em virtude da dificuldade de estimar como foram ocasionados, os erros lógicos são mais difíceis de corrigir.

Podemos ver um exemplo desse tipo de erro a seguir:

```
static void Main(string[] args)
{
    int n1 = 10, n2 = 15;
    //O objetivo é somar dois números, mas com o operador de
    //subtração
    int resultado = n1 - n2;

    Console.WriteLine(
        $"{n1.ToString()} + {n2.ToString()} = {resultado.
        ToString()}");
}

Console.ReadKey();
}
```

Veja o resultado do código anterior:

```
D:\Documentos\Projetos\VS...
10 + 15 = -5
```

## 9.2.2. Erro de compilação

Também conhecido como erro de sintaxe, este erro está diretamente relacionado ao código. É o tipo de erro mais comum, pois ocorre ao escrever o código, impedindo que o projeto seja compilado. Ele é facilmente identificado e corrigido.

O exemplo a seguir ilustra este tipo de erro:

```
Program.cs* 13
14
15 int n1 = 10, n2 = 15;
16
17 //O objetivo é somar dois números, mas com o operador de subtração
18 int resultado = n1 - n2;
19
20 Console.WriteLine(
21     $"{n1.ToString()} + {n2.ToString()} = {resultado.ToString()}");
22 }
```

Error List

Code	Description
<b>CS1002</b>	; expected

## 9.2.3. Erro de execução

Este tipo de erro, por sua vez, é aquele que gera uma interrupção inesperada do programa enquanto este é executado. Geralmente, ocorre ao se tentar executar uma operação inválida. Em todos os casos, o programa é encerrado durante o seu carregamento.

Considere o seguinte exemplo de erro de execução:

```
static void Main(string[] args)
{
    int n1 = 10, n2 = 0;

    int resultado = n1 / n2; ✖
    Console.WriteLine(
        $"{n1.ToString()} / {n2.ToString()} = {resultado.ToString()}");
    Console.ReadKey();
}
```



Podemos ver que o código define uma variável **int** de nome **n1**, cujo valor será a divisão de **10** por **n2**, cujo valor foi definido como **0**. Dessa divisão, ocorrerá um erro em tempo de execução. Neste caso, ocorrerá um erro de execução, ou o lançamento de uma exceção. Quando não tratada, uma exceção produz a interrupção do programa.

É possível, ainda, evitar que o Visual C# interrompa a execução quando uma exceção é lançada, por meio de um código específico para manipular exceções, **try ... catch**.

## 9.3. O bloco **try ... catch**

O bloco **try...catch** é usado para capturar exceções que ocorrem durante a execução de um programa. No bloco **try**, colocamos o código desejado, e no bloco **catch**, o código a ser executado no caso de alguma exceção ocorrer.

O bloco **catch** deve ser escrito imediatamente após o bloco **try**. Podemos ter vários blocos **catch**, cada um responsável por uma exceção específica. Dessa forma, quando ocorrer um erro dentro do bloco **try**, uma exceção é lançada e o controle é transferido diretamente para a primeira rotina de tratamento **catch** correspondente.

**!** O código dentro de um bloco **try** é chamado de região protegida.

A sintaxe do bloco **catch** é similar à sintaxe de um método:

```
try
{
    //código protegido
}
catch [(tipo_exceção variável)]
{
    //código alternativo
}
```

O parâmetro no bloco **catch** é opcional.

Em alguns casos, o bloco **try** pode lançar uma exceção que não possua **catch** correspondente. Se o bloco **try** for parte de um método, este encerrará e a execução retornará ao método chamador.

Neste caso, há duas possibilidades:

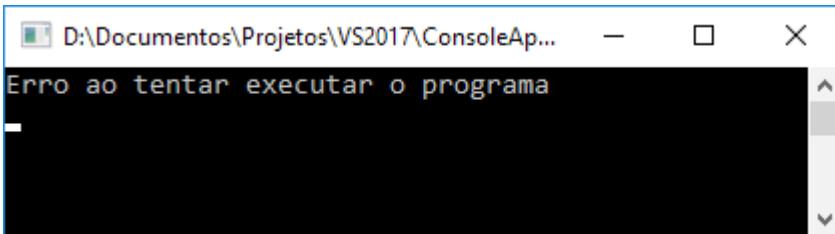
- Se esse método chamador utilizar um bloco **try**, o runtime tentará localizar e executar a rotina de tratamento **catch** correspondente, após o bloco **try**;
- Se esse método chamador não utilizar um bloco **try** ou não houver **catch** correspondente, ele encerrará e a execução retornará ao seu chamador.

A seguir, temos um exemplo de captura e manipulação de exceção por meio de um bloco **try ... catch**:

```
static void Main(string[] args)
{
    try
    {
        int n1 = 10, n2 = 0;
        int resultado = n1 / n2;

        Console.WriteLine(
            $"{n1.ToString()} / {n2.ToString()} = {resultado.
ToString()}");
    }
    catch
    {
        Console.WriteLine("Erro ao tentar executar o programa");
    }
    Console.ReadKey();
}
```

Vemos o resultado desse código a seguir:

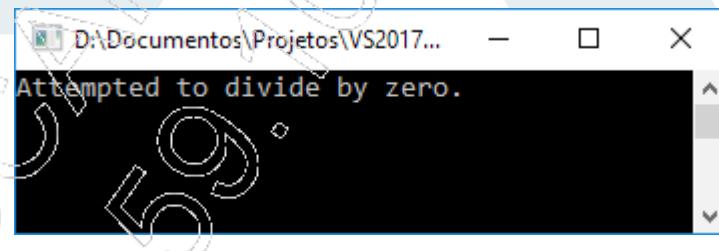


Podemos, também, capturar o erro e uma instância da classe **DivideByZeroException**:

```
static void Main(string[] args)
{
    try
    {
        int n1 = 10, n2 = 0;
        int resultado = n1 / n2;

        Console.WriteLine(
            $"{n1.ToString()} / {n2.ToString()} = {resultado.
ToString()}");
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadKey();
}
```

Vemos o resultado desse código a seguir:



Quando usamos mais de um bloco **catch**, podemos capturar diversos tipos de exceções diferentes. Para isso, devemos apenas cumprir alguns requisitos:

- Eles devem ser colocados depois do bloco **try**;
- Não deve haver instrução ou bloco entre os blocos **catch**, ou seja, eles devem ser colocados imediatamente um após o outro;
- Os blocos **catch** devem seguir uma ordem, ou seja, o **catch** não específico deve ser o último.

Diversas exceções são definidas pelo Framework, de modo que qualquer programa desenvolvido estará apto a lançar a maioria delas. Essas exceções estão organizadas em hierarquias de herança. **FormatException** e **OverflowException**, por exemplo, pertencem à classe **System.Exception**. Dessa forma, pode haver um manipulador que capture as exceções não individualmente, ou seja, que capture múltiplas exceções pertencentes a uma classe. Nesse contexto, a classe mais genérica é chamada **Exception**.

Quando ocorre uma exceção, o primeiro manipulador correspondente a ela será executado para tratá-la, e todos os outros serão ignorados. Dessa forma, se um manipulador para a classe **Exception** estiver posicionado antes de um **FormatException**, um erro de compilação será apresentado.

```
static void Main(string[] args)
{
    try
    {
        int n1 = 10, n2 = 0;

        int resultado = n1 / n2;
        Console.WriteLine(
            $"{n1.ToString()} / {n2.ToString()} = {resultado.ToString()}");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Ocorreu um erro geral");
    }
    catch(DivideByZeroException ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadKey();
}
```

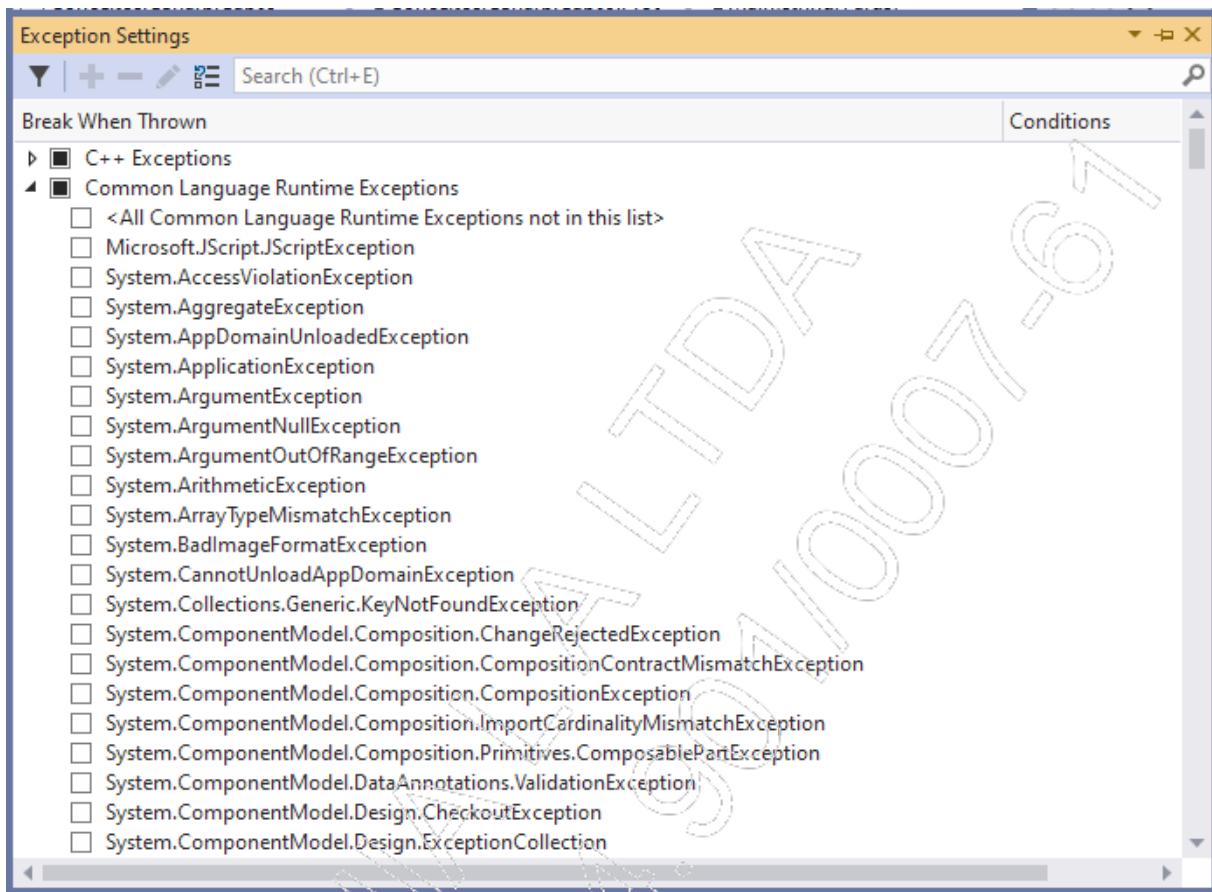
Isso porque o **Exception**, por ser mais genérico (superclasse das exceções), captura qualquer exceção, e a mais específica nunca será processada.

## 9.4. A classe **Exception** e suas subclasses

Existem muitas classes de exceção no .NET Framework. Cada namespace contém as classes de exceção relacionadas com informações relevantes às classes do mesmo namespace. Por exemplo, a classe **SqlException** encontra-se no namespace **System.Data.SqlClient** e contém informações a respeito dos erros gerados quando o banco de dados SQL Server é utilizado.

# Programando com C#

Para visualizarmos todas as exceções do .NET Framework, devemos acessar o menu **Debug / Windows** e acionar o item **Exception Settings**, para que a janela exibida a seguir seja aberta:



Todas as classes **Exception** são subclasses da classe base **System.Exception**. Algumas classes comuns são as seguintes:

- **System.FormatException**: Utilizada para detectar formatos inválidos em conversões de dados;
- **System.DivideByZeroException**: Quando ocorre uma divisão por zero;
- **System.IndexOutOfRangeException**: Ocorre quando um índice de um array está fora dos limites;
- **System.InvalidCastException**: Ocorre quando uma conversão direta (**cast**) não é possível;
- **System.OverflowException**: Um número muito grande ou muito pequeno foi atribuído a uma variável que não o comporta;
- **System.IO.FileNotFoundException**: Arquivo não encontrado;

- **System.Data.SqlClient.SqlException**: Ocorre quando um problema é detectado em uma operação no SQL Server;
- **System.Data.OleDb.OleDbException**: Ocorre quando um problema é detectado em uma operação com banco de dados conectado por meio do componente OLE DB.

#### 9.4.1. Propriedades da classe Exception

A principal propriedade da classe **Exception** é **Message**, que informa uma descrição do erro ocorrido. Cada classe derivada da classe **Exception** tem suas próprias propriedades. A seguir, um pequeno resumo das principais propriedades e métodos da classe **Exception**:

- **Message**: Uma descrição do erro;
- **Source**: O nome da aplicação ou do objeto que causou o erro;
- **Stack Trace**: A pilha de chamadas onde estava o programa no momento em que ocorreu o erro;
- **TargetSite**: O método que disparou o erro;
- **HelpLink**: Um link para a ajuda sobre esse erro;
- **InnerException**: Uma instância da classe **Exception** que gerou a exceção atual;
- **Data**: Coleção com informações adicionais sobre o erro;
- **ToString()**: Método que retorna todos os detalhes do erro.

#### 9.5. O comando throw

Quando usamos o bloco **try ... catch**, protegemos e capturamos as exceções lançadas por algum método ou propriedade. As exceções são lançadas através do comando **throw**.

Essa instrução permite que determinadas exceções que podem ser geradas em métodos ou situações específicas sejam tratadas. A palavra reservada **throw** dispara um erro criado pelo programa em tempo de execução.

Quando o comando **throw** é usado, a exceção produzida deve ser colocada dentro de um bloco **try**.

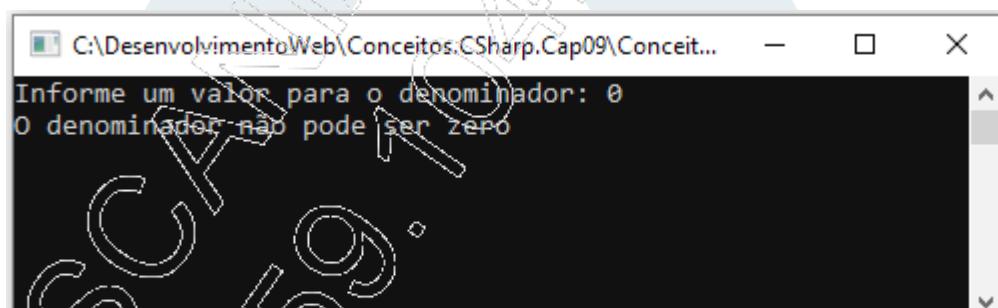
Veja um exemplo:

```
static void Main(string[] args)
{
    try
    {
        int n1 = 10;
        Console.Write("Informe um valor para o denominador: ");

        int n2 = int.Parse(Console.ReadLine());

        if(n2 == 0)
        {
            throw new Exception("O denominador não pode ser zero");
        }
        int resultado = n1 / n2;
        Console.WriteLine(
            $"{n1.ToString()} / {n2.ToString()} = {resultado.
ToString()}");
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        Console.ReadKey();
    }
```

Vemos o resultado desse código a seguir:



## 9.6. O bloco finally

Quando uma exceção é lançada, existe a possibilidade de o código ser interrompido e uma instrução necessária, escrita após o bloco **catch**, nunca ser processada.

Na linguagem C#, contamos com a instrução **finally** para garantir que uma instrução seja sempre executada, independentemente de terem sido ou não lançadas exceções. Basta que a instrução seja escrita em um bloco **finally**, imediatamente após um bloco **try**, ou imediatamente após a última rotina de tratamento **catch**.

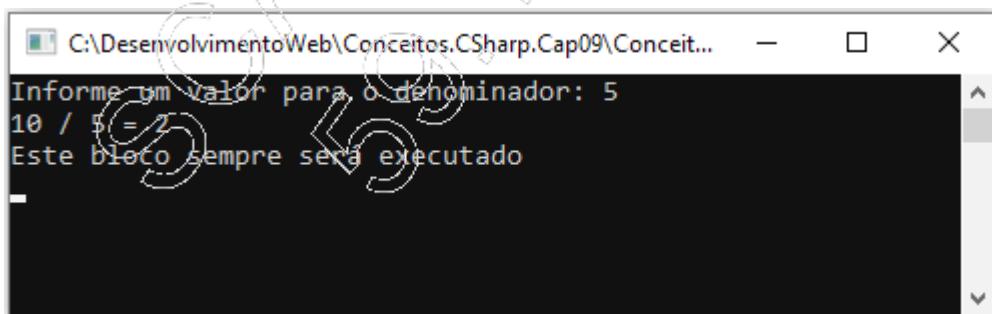
O código a seguir exemplifica o uso de um bloco **finally**:

```
static void Main(string[] args)
{
    try
    {
        int n1 = 10;
        Console.Write("Informe um valor para o denominador: ");

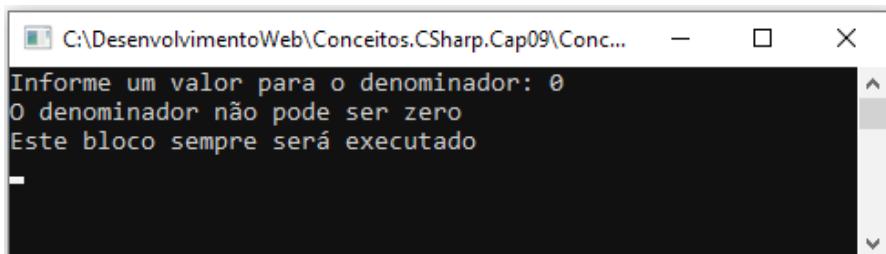
        int n2 = int.Parse(Console.ReadLine());

        if(n2 == 0)
        {
            throw new Exception("O denominador não pode ser zero");
        }
        int resultado = n1 / n2;
        Console.WriteLine(
            $"{n1.ToString()} / {n2.ToString()} = {resultado.
ToString()}");
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            Console.WriteLine("Este bloco sempre será executado");
        }
    Console.ReadKey();
}
```

- Se o valor informado for diferente de zero



- Se o valor informado for igual a zero



## 9.7. O bloco using

Sabemos que quando usamos o bloco **finally**, seu conteúdo é executado independentemente do que ocorra nos blocos **try** ou **catch**. Sendo assim, na prática, sua principal finalidade é liberar recursos utilizados ao longo do código, como por exemplo:

- Fechar um arquivo aberto para uso do código;
- Fechar uma conexão com o banco de dados;
- Remover arquivos porventura criados para uso particular do código.

Existem ainda muitos outros exemplos que podem ser destacados, mas os citados já dão uma ideia da sua finalidade.

Ao trabalharmos com tratamento de exceções, podemos omitir o bloco **catch**, e usar apenas o bloco **try ... finally**. Em cenários como este, podemos usar o bloco **using**. Ele representa um recurso otimizado de gerarmos instâncias. Sua forma geral é:

```
using(instancia)
{
    //código
}
```

O uso do bloco **using** previne o que chamamos de *memory leak*. Em outras palavras, existem objetos, especialmente aqueles relacionados a acesso a dados, que, quando usados diversas vezes, acabam ocupando memória de forma desnecessária, e a finalização do bloco **using** faz o papel de otimizar a memória, tornando a aplicação mais escalável. Veja como podemos utilizá-lo:

```
static void Main(string[] args)
{
    using(var conexao = new SqlConnection())
    {
        //conteúdo do bloco
    }
    //restante do código
    Console.ReadKey();
}
```

Neste exemplo estamos criando uma instância da classe `SqlConnection`, responsável por manter uma conexão com um banco de dados. Dentro do bloco nós utilizamos a variável criada dentro do bloco.

Mas não é qualquer classe que pode ser instanciada em um bloco `using`. Ela deve implementar a interface `IDisposable`.

### 9.7.1.A interface `IDisposable`

Esta interface é usada em ambientes onde se faz necessária a liberação de recursos. Toda classe que aloca memória com recursos não gerenciados deve implementar essa interface. Os recursos a serem liberados são implementados no seu método `Dispose`.

Diversas classes do framework implementam esta interface. Um dos motivos de implementá-la é que teremos um canal para liberar os recursos não gerenciados. Outro motivo é que a implementação do método `Dispose` fará com que o objeto fique mais visível para que o Garbage Collector libere a memória alocada por ele.

### Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

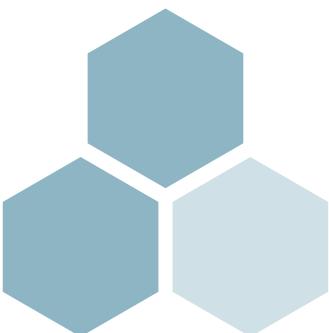
- Para lidar com erros, o C# possui um mecanismo de manipulação de exceções. Ele não evita a ocorrência de erros, mas facilita sua descoberta e solução;
- Podem ser encontrados três tipos de erros: erros de lógica, de compilação e de execução (também conhecidos como exceções);
- Com a instrução **try**, podemos capturar uma exceção durante a execução de um código; por meio da instrução **catch**, a exceção é manipulada; e a instrução **finally** garante que uma instrução seja sempre executada, tenham ou não sido lançadas exceções;
- Utilizamos a instrução **throw** para lançar exceções. Estas, por sua vez, são organizadas em uma hierarquia de classes, das quais a classe base é **System.Exception**;
- O bloco **using** é útil quando desejamos otimizar a memória e evitar o que é chamado de *memory leak*. Para usá-lo, os objetos a serem criados devem ser de classes que implementem a interface **IDisposable**.



9

## Tratamento de exceções

Teste seus conhecimentos



## 1. Para que utilizamos a instrução throw?

- a) Para lançar breakpoints.
- b) Para lançar exceções.
- c) Para lançar instruções finally.
- d) Para lançar instruções try.
- e) Para lançar instruções catch.

## 2. O que faz o mecanismo de manipulação de exceções do C#?

- a) Ele evita que erros aconteçam.
- b) Ele facilita a descoberta e solução de erros considerados simples.
- c) Ele documenta os erros para futuras análises, evitando que aconteçam novamente.
- d) Ele não evita a ocorrência de erros graves.
- e) Ele não evita a ocorrência de erros, mas facilita sua descoberta e solução.

## 3. O que fazem, respectivamente, as instruções try e catch?

- a) Captura uma exceção durante a execução; manipula a exceção capturada.
- b) Manipula uma exceção; captura uma exceção.
- c) Captura e manipula uma exceção; sempre será executada.
- d) Captura e lança uma nova exceção; sempre será executada.
- e) Executa a instrução finally; manipula a exceção capturada.

**4. Quais são os três tipos de erros que podem existir no C#?**

- a) Lógica, aritmética, sintaxe.
- b) Sintaxe, semântica, álgebra.
- c) Lógica, compilação, semântica.
- d) Lógica, compilação, execução.
- e) Compilação, execução, aritmética.

**5. Que instrução sempre será executada no tratamento de erros?**

- a) throw
- b) exception
- c) catch
- d) watch
- e) finally





9

# Tratamento de exceções



Mãos à obra!



## Laboratório 1

### A – Lançando exceções e utilizando o bloco try...catch

#### Objetivos:

Chegou o momento de tornar a aplicação mais segura. Vamos fazer com que as propriedades mais vulneráveis lancem exceções. Além disso, nos eventos dos botões que incluímos na interface gráfica, adicionaremos o código necessário para proteger a aplicação através do bloco `try...catch`.

1. Crie um novo solution, vazio, chamado **Capítulo09.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models**, **Lab.Views**, **Lab.Interfaces**, **Lab.Arquivos** e **Lab.Dados** do Capítulo 8, na pasta do solution **Capítulo09.Labs**;
3. Adicione esses projetos ao novo solution;
4. No projeto **Lab.Models**, abra a classe **Cliente**. Vamos validar a propriedade **Idade** – esta não pode ser negativa. Altere a classe para contemplar esta alteração:

```
namespace Lab.Models
{
    public abstract class Cliente
    {
        public string Nome { get; set; }

        private int _idade;
        public int Idade
        {
            get => _idade;
            set => _idade = (value >= 0 ? value :
                throw new InvalidOperationException("A idade não pode
ser negativa!"));
        }

        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public List<Conta> Contas { get; set; }

        public Cliente(string Nome, Sexos Sexo)
        {
            if(this.Contas == null)
            {
                this.Contas = new List<Conta>();
            }
        }
    }
}
```

```
        this.Nome = Nome;
        this.Sexo = Sexo;
    }

    public Cliente(string Nome, Sexos Sexo, int idade)
        : this(Nome, Sexo)
    {
        this.Idade = Idade;
    }

    public Cliente(string Nome, Sexos Sexo, Endereco endereco)
        : this(Nome, Sexo)
    {

        this.EnderecoResidencial = endereco;
    }

    ~Cliente()
    {
        if(this.Contas != null)
        {
            this.Contas = null;
        }
    }

    public virtual string Exibir()
    {
        return $"Nome: {this.Nome}\n" +
               $"Idade: {this.Idade}\n" +
               $"Sexo: {this.Sexo}\n" +
               $"ENDEREÇO DO CLIENTE:\n" +
               $"{this.EnderecoResidencial.Exibir()}";
    }

    public override string ToString()
    {
        return this.Nome;
    }
}
```

5. No mesmo projeto, abra as classes **Conta**, **ContaCorrente** e **ContaEspecial**:

- **Conta**: no método **RegistrarMovimento**, o parâmetro valor deve ser maior que zero;
- **ContaCorrente**: no método **EfetuarOperacao**, o parâmetro valor deve ser maior que zero. Além disso, para que seja possível efetuar o saque, o saldo deve ser maior ou igual ao valor do saque;
- **ContaEspecial**: no método **EfetuarOperacao** valem as mesmas regras deste método na classe **ContaCorrente**. A diferença é que deve haver saldo suficiente, incluindo o limite de crédito. Além disso, o valor da propriedade **Limite** deve ser maior que zero.

#### • Classe Conta

```
namespace Lab.Models
{
    public abstract class Conta
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }

        public List<Movimento> Movimentos { get; set; }

        public Cliente ClienteInfo { get; set; }

        public Conta(int Banco, string Agencia, string Conta)
        {
            if(this.Movimentos == null)
            {
                this.Movimentos = new List<Movimento>();
            }

            this.NumeroBanco = Banco;
            this.NumeroAgencia = Agencia;
            this.NumeroConta = Conta;
        }

        protected void RegistrarMovimento(double valor, Operacoes
operacao)
        {
            if (valor <= 0)
            {
                throw new
                    ArgumentException("O valor deve ser positivo");
            }
        }
    }
}
```

```

        this.Movimentos.Add(new Movimento()
    {
        Data = DateTime.Now,
        Historico = operacao == Operacoes.Saque ? "SAQUE" :
        "DEPÓSITO",
        Operacao = operacao,
        Valor = valor
    });
}

public abstract string MostrarExtrato();

public abstract void EfetuarOperacao(double valor,
    Operacoes operacao = Operacoes.Deposito);

public override string ToString()
{
    return this.NumeroAgencia + "/" + this.NumeroConta;
}
}
}

```

- **Classe ContaCorrente**

```

namespace Lab.Models
{
    public class ContaCorrente : Conta
    {
        public double Saldo { get; protected set; }
        //public Cliente ClienteInfo { get; set; }

        public ContaCorrente(int Banco, string Agencia, string Conta):
            base(Banco, Agencia, Conta)
        { }

        public ContaCorrente(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public override void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            if(valor <= 0)
            {
                throw new
                    ArgumentException("O valor deve ser positivo");
            }
        }
    }
}

```

```
switch (operacao)
{
    case Operacoes.Deposito:
        this.Saldo += valor;
        break;
    case Operacoes.Saque:
        if(valor > this.Saldo)
        {
            throw new
                InvalidOperationException("Saldo insuficiente");
        }
        this.Saldo -= valor;
        break;
    }
    base.RegistrarMovimento(valor, operacao);
}

public virtual string Exibir()
{
    string cliente = this.ClienteInfo != null ?
        this.ClienteInfo.Exibir() + '\n' : "";

    return $"{cliente}" +
        $"Banco: {this.NumeroBanco}\n" +
        $"Agência: {this.NumeroAgencia}\n" +
        $"Conta: {this.NumeroConta}\n" +
        $"Saldo Atual: {this.Saldo}";
}

public override string MostrarExtrato()
{
    StringBuilder builder = new StringBuilder();
    builder.Append($"Cliente: {ClienteInfo.Nome}\n")
        .Append($"Banco: {NumeroBanco}\n")
        .Append($"Agência: {NumeroAgencia}\n")
        .Append($"Conta: {NumeroConta}\n")
        .Append(new string('-', 35) + '\n');

    if(this.Movimentos.Count() == 0)
    {
        builder.Append("Nenhum movimento registrado para esta
conta");
    }
    else
    {
        foreach (var item in this.Movimentos)
        {
            builder.Append($"{item}\n");
        }
    }
    builder.Append(new string('-', 35) + '\n');
    builder.Append($"Saldo: {this.Saldo:c}");

    return builder.ToString();
}
}
```

- Classe ContaEspecial

```
namespace Lab.Models
{
    public class ContaEspecial : ContaCorrente
    {
        private double _limite;
        public double Limite
        {
            get => _limite;
            set => _limite = (value <= 0 ? value :
                throw new
                    InvalidOperationException("O Limite deve ser
positivo"));
        }

        public ContaEspecial(int Banco, string Agencia, string Conta) :
            base(Banco, Agencia, Conta)
        { }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente, double Limite)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
            this.Limite = Limite;
        }
    }
}
```

```
//métodos sobrescritos
public override void EfetuarOperacao(double valor,
    Operacoes operacao = Operacoes.Deposito)
{
    if (valor <= 0)
    {
        throw new
            ArgumentException("O valor deve ser positivo");
    }
    switch (operacao)
    {
        case Operacoes.Deposito:
            this.Saldo += valor;
            break;
        case Operacoes.Saque:
            if (valor > (this.Saldo + this.Limite))
            {
                throw new
                    InvalidOperationException("Saldo insuficiente");
            }
            this.Saldo -= valor;
            break;
    }
    base.RegistrarMovimento(valor, operacao);
}

public override string Exibir()
{
    return $"{base.Exibir()}\n" +
        $"Limite: {this.Limite}\n" +
        $"Salto Disponível: {this.Saldo + this.Limite}";
}

public override string MostrarExtrato()
{
    return new StringBuilder(base.MostrarExtrato())
        .Append($"\nLimite: {this.Limite}")
        .Append($"\nSaldo Disponível: {this.Limite + this.
Saldo:c}")
        .ToString();
}
}
```

6. Inclua um bloco protegido (**try...catch**) em todos os eventos, no arquivo **MainWindow.xaml.cs**:

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
            sexoComboBox.SelectedIndex = 0;

            operacaoComboBox.Items.Add(Operacoes.Deposito);
            operacaoComboBox.Items.Add(Operacoes.Saque);
            operacaoComboBox.SelectedIndex = 0;
        }

        private void incluirClienteButton_Click(object sender,
            RoutedEventArgs e)
        {
            try
            {
                //obtendo os dados do endereço
                Endereco endereco = new Endereco(
                    ruaTextBox.Text,
                    int.Parse(numeroTextBox.Text),
                    cidadeTextBox.Text,
                    cepTextBox.Text);

                //obtendo os dados do cliente
                int digitos = documentoTextBox.Text.Length;
                Cliente cliente;

                if (digitos == 11)
                {
                    cliente = new ClientePF(
                        documentoTextBox.Text,
                        nomeTextBox.Text,
                        (Sexos)sexoComboBox.SelectedItem,
                        int.Parse(idadeTextBox.Text),
                        endereco);
                }
                else if (digitos == 14)
                {
                    cliente = new ClientePJ(
                        documentoTextBox.Text,
                        nomeTextBox.Text,
                        (Sexos)sexoComboBox.SelectedItem,
                        int.Parse(idadeTextBox.Text),
                        endereco);
                }
                else
                {
                    throw new Exception("Documento inválido");
                }
            }
        }
    }
}
```

```
//adicionando o novo cliente na lista  
Metodos.AdicionarCliente(cliente);  
  
//vinculando a lista de clientes ao  
//componente ComboBox  
clienteComboBox.ItemsSource = Metodos.ListarClientes();  
MessageBox.Show("Cliente incluído com sucesso!");  
}  
catch (Exception ex)  
{  
    MessageBox.Show(ex.Message,  
        "Erro",  
        MessageBoxButton.OK,  
        MessageBoxIcon.Error);  
}  
  
//MessageBox.Show(cliente.Exibir());  
}  
  
private bool VerificarEspecial { get; set; }  
  
private void especialRadioButton_Checked(object sender, RoutedEventArgs e)  
{  
    try  
    {  
        var radio = sender as RadioButton;  
        VerificarEspecial = (radio == especialRadioButton);  
  
        limiteLabel.IsEnabled = VerificarEspecial;  
        limiteTextBox.IsEnabled = VerificarEspecial;  
    }  
    catch (Exception ex)  
    {  
        MessageBox.Show(ex.Message,  
            "Erro",  
            MessageBoxButton.OK,  
            MessageBoxIcon.Error);  
    }  
}  
  
private void incluirContaButton_Click(object sender, RoutedEventArgs e)  
{  
    try  
    {  
        Conta conta;  
        var cliente = (Cliente)clienteComboBox.SelectedItem;  
  
        int banco = int.Parse(bancoTextBox.Text);  
        string agencia = agenciaTextBox.Text;  
        string numConta = contaTextBox.Text;  
  
        if (VerificarEspecial)  
        {  
            conta = new ContaEspecial(banco, agencia, numConta);  
            ((ContaEspecial)conta).Limite =  
                double.Parse(limiteTextBox.Text);  
        }  
        else  
        {  
            conta = new ContaCorrente(banco, agencia, numConta);  
        }  
    }  
}
```

```
//vinculamos o cliente selecionado à nova conta  
conta.ClienteInfo = cliente;  
  
//adicionamos a nova conta à lista de contas do  
//cliente selecionado  
cliente.Contas.Add(conta);  
  
//incluimos a nova conta à lista global de contas  
Metodos.AdicionarConta(conta);  
  
//vinculamos a lista global de contas ao  
//componente ComboBox  
numeroContaComboBox.ItemsSource = Metodos.ListarContas();  
  
MessageBox.Show("Conta adicionada com sucesso!");  
}  
catch (Exception ex)  
{  
    MessageBox.Show(ex.Message,  
        "Erro",  
        MessageBoxButton.OK,  
        MessageBoxImage.Error);  
}  
  
private void executarButton_Click(object sender,  
    RoutedEventArgs e)  
{  
    try  
    {  
        //obtendo a conta selecionada  
        var conta = (Conta)numeroContaComboBox.SelectedItem;  
  
        //obtendo a operação  
        var operacao = (Operacoes)operacaoComboBox.SelectedItem;  
  
        //obtendo o valor da operação  
        double valor = double.Parse(valorTextBox.Text);  
    }  
}
```

```
//executar a operação
conta.EfetuarOperacao(valor, operacao);

MessageBox.Show("Operação realizada com sucesso!");
valorTextBox.Clear();
}

catch (Exception ex)
{
    MessageBox.Show(ex.Message,
        "Erro",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}

private void extratoButton_Click(object sender,
    RoutedEventArgs e)
{
    try
    {
        var conta = (Conta)numeroContaComboBox.SelectedItem;

        extratoTextBox.Text = conta.MostrarExtrato();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message,
            "Erro",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
    }
}
}
```

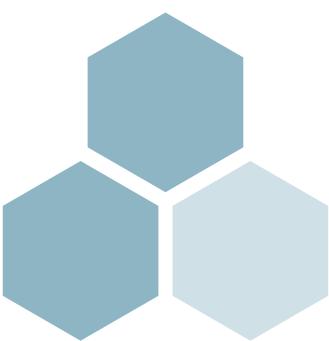
7. Teste a aplicação, incluindo algumas informações inválidas.



# 10

## Delegates e expressões lambda

- ◆ Declaração de delegates;
- ◆ Expressões lambda;
- ◆ Delegates genéricos;
- ◆ Delegates predefinidos no framework.



### 10.1. Introdução

Da mesma forma que temos as classes que geram referências a objetos, temos os delegates que produzem referências a funções ou a operações.

Suponha que alguém pergunte: se desejarmos escrever um método que recebe como parâmetros dois inteiros e retorna um inteiro, quantas possibilidades temos? Observe que não estamos preocupados com uma aplicação específica, e sim com a sintaxe. Existe uma infinidade de métodos que atendem a este requisito funcional, mas sabemos que se tornaria inviável escrevermos todos os métodos para atender a todas as possibilidades.

Muitas vezes, a operação é decidida pelo programador de acordo com a necessidade. É esse conceito que será estudado neste capítulo.

### 10.2. Declaração de delegates

Quando declaramos um delegate, estamos definindo um novo tipo de dado, capaz de referenciar para uma operação com uma assinatura semelhante.

Considere a declaração:

```
public delegate void Mensagem();
```

Definimos um delegate chamado **Mensagem**, capaz de referenciar qualquer método cuja assinatura seja compatível com sua definição, ou seja, sem parâmetros e sem retorno. Sendo assim, este delegate é compatível com os seguintes métodos:

```
static void ExibirMensagem01()
{
    Console.WriteLine("Primeira Mensagem");
}

static void ExibirMensagem02()
{
    Console.WriteLine("Segunda Mensagem");
}
```

Podemos vincular esses métodos ao delegate de duas formas, que veremos a seguir.

### 10.2.1. Vinculando o delegate com o operador new

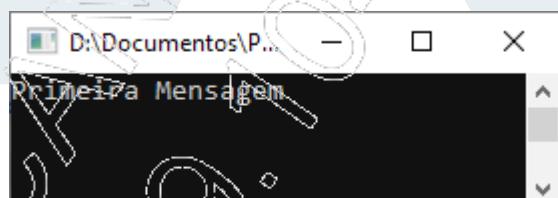
O exemplo adiante mostra o uso do delegate referenciando o método `ExibirMensagem01`. A execução deste método pode ser realizada de forma indireta por meio do delegate:

```
namespace Conceitos.CSharp.Cap10
{
    public delegate void Mensagem();

    class Program
    {
        static void ExibirMensagem01()
        {
            Console.WriteLine("Primeira Mensagem");
        }

        static void Main(string[] args)
        {
            Mensagem msg1 = new Mensagem(ExibirMensagem01);
            msg1();
            Console.ReadKey();
        }
    }
}
```

A execução produz o mesmo resultado que seria produzido com a chamada direta ao método `ExibirMensagem01`:



## 10.2.2. Vinculando o delegate com o operador +=

Usando o operador `+=`, podemos vincular dois ou mais métodos, e a chamada ao delegate produz a chamada sequencial aos métodos que o delegate referencia.

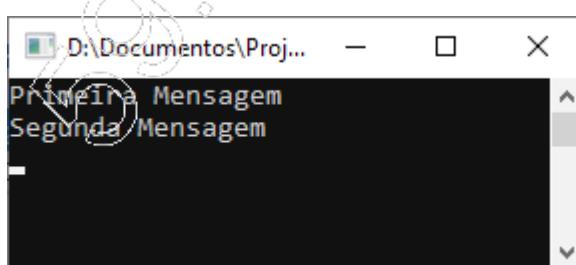
```
namespace Conceitos.CSharp.Cap10
{
    public delegate void Mensagem();

    class Program
    {
        static void ExibirMensagem01()
        {
            Console.WriteLine("Primeira Mensagem");
        }

        static void ExibirMensagem02()
        {
            Console.WriteLine("Segunda Mensagem");
        }

        static void Main(string[] args)
        {
            Mensagem msg1 = null;
            msg1 += ExibirMensagem01;
            msg1 += ExibirMensagem02;
            msg1();
        }
    }
}
```

Resultado:



## 10.3. Expressões lambda

Vamos entender as expressões lambda através de uma sequência de exemplos.

Vamos adicionar o seguinte delegate no namespace, logo abaixo de **Mensagem**:

```
public delegate double Calcular(double x, double y);
```

Desenvolveremos a expressão lambda a partir de um método. Se fosse um método tradicional, teríamos o seguinte exemplo:

```
static double Calcular(double x, double y)
{
    return x + y;
}
```

Observe que consideramos como operação a soma dos dois parâmetros, mas poderia ser qualquer outra.

Como a proposta do delegate é referenciar apenas a operação, vamos fazer alguns ajustes:

1. Não importa o nome do método. Então ele pode ser omitido:

```
static double (double x, double y)
{
    return x + y;
}
```

2. Sabemos que o retorno é double e que os parâmetros necessariamente são do tipo double. Eles também podem ser omitidos:

```
( x, y)
{
    return x + y;
}
```

3. Como existe apenas uma instrução, o comando **return** pode ser omitido:

```
( x, y) { x + y; }
```

4. Havendo apenas uma instrução, as chaves podem ser omitidas, desde que o comando **return** também seja:

```
( x, y) x + y;
```

5. Para justificar o resultado anterior, usamos o operador `=>` entre os dois termos:

```
( x, y ) => x + y;
```

Esta é nossa expressão lambda! Para usá-la, podemos escrever no método `Main()`:

```
Calcular calc = (x, y) => x + y; //expressão lambda
double resultado = calc(12, 15);
Console.WriteLine("Resultado: " + resultado);
```

De forma geral, uma expressão lambda possui a sintaxe:

**(parâmetros) => resultado;**

Cuja leitura pode ser feita da seguinte forma:

*"Uma lista de parâmetros PRODUZ o resultado desejado."*

Verifique que a expressão lambda deve combinar com a definição do delegate.

Como outro exemplo, vamos ver como seria uma expressão lambda para o delegate `Mensagem`:

```
class Program
{
    static void Main(string[] args)
    {
        Mensagem msg1 = () => Console.WriteLine("Uso de expressão
lambda");
        msg1();

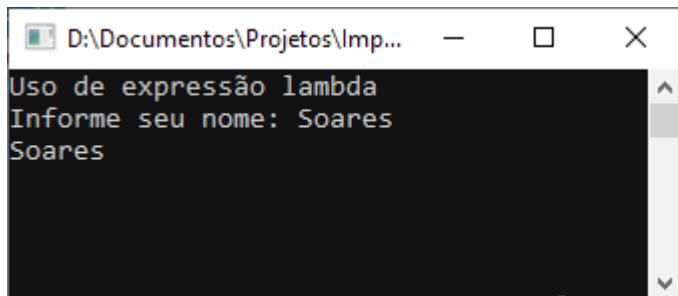
        Mensagem msg2 = () =>
        {
            Console.Write("Informe seu nome: ");
            string s = Console.ReadLine();

            Console.WriteLine(s);
        };
        msg2();

        Console.ReadKey();
    }
}
```

Verifique que se tivermos mais de uma instrução, o conteúdo após a seta (`=>`) é idêntico a um método tradicional, ou seja, não podemos omitir as chaves e, consequentemente, mantemos o comando `return`, se houver.

Resultado da execução:



O uso da expressão lambda dispensa o uso de métodos a serem vinculados.

**!** O propósito da expressão lambda é definir operações cuja sintaxe depende da utilização.

## 10.4. Delegates genéricos

Assim como as classes genéricas, temos também os delegates genéricos. As regras aplicadas aos genéricos também se aplicam aos delegates, quando forem genéricos. Veja o exemplo:

```
namespace Conceitos.CSharp.Cap10
{
    public delegate bool Buscar<T>(T item);

    class Program
    {
        static void Main(string[] args)
        {
            string[] cidades = { "Lorena", "Saquarema", "Paraty",
"Salvador" };
            Buscar<string> busca1 = p => p.Contains("o");

            foreach (var item in cidades)
            {
                if (busca1(item))
                {
                    Console.WriteLine(item);
                }
            }

            Console.ReadKey();
        }
    }
}
```

O delegate retorna um valor **booleano**. Na aplicação, nós o utilizamos em um comando **if**. O valor passado como parâmetro foi compatível com a parametrização que usamos na declaração da variável **busca1**.

## 10.5. Delegates predefinidos no framework

O framework .NET possui diversos delegates previamente definidos, e aplicados em classes do próprio framework. Os mais importantes são:

- **Predicate**
- **Action**
- **Func**

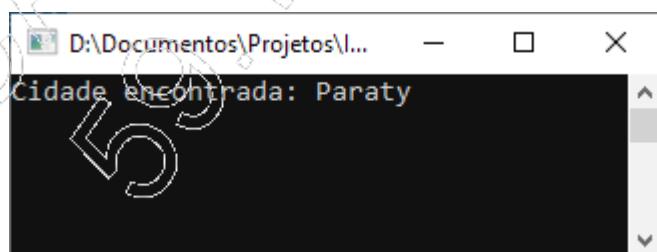
Como exemplo, considere a classe **List**, e o método **Find**:

```
class Program
{
    static void Main(string[] args)
    {
        List<string> cidades = new List<string>()
        {
            "Lorena", "Saquarema", "Paraty", "Salvador"
        };

        var cidade = cidades.Find(p => p.StartsWith("P"));
        Console.WriteLine("Cidade encontrada: " + cidade);

        Console.ReadKey();
    }
}
```

Resultado da execução:



O método **Find** recebe um delegate tipo **Predicate** como parâmetro. A definição do delegate e do método **Find** são dadas a seguir:

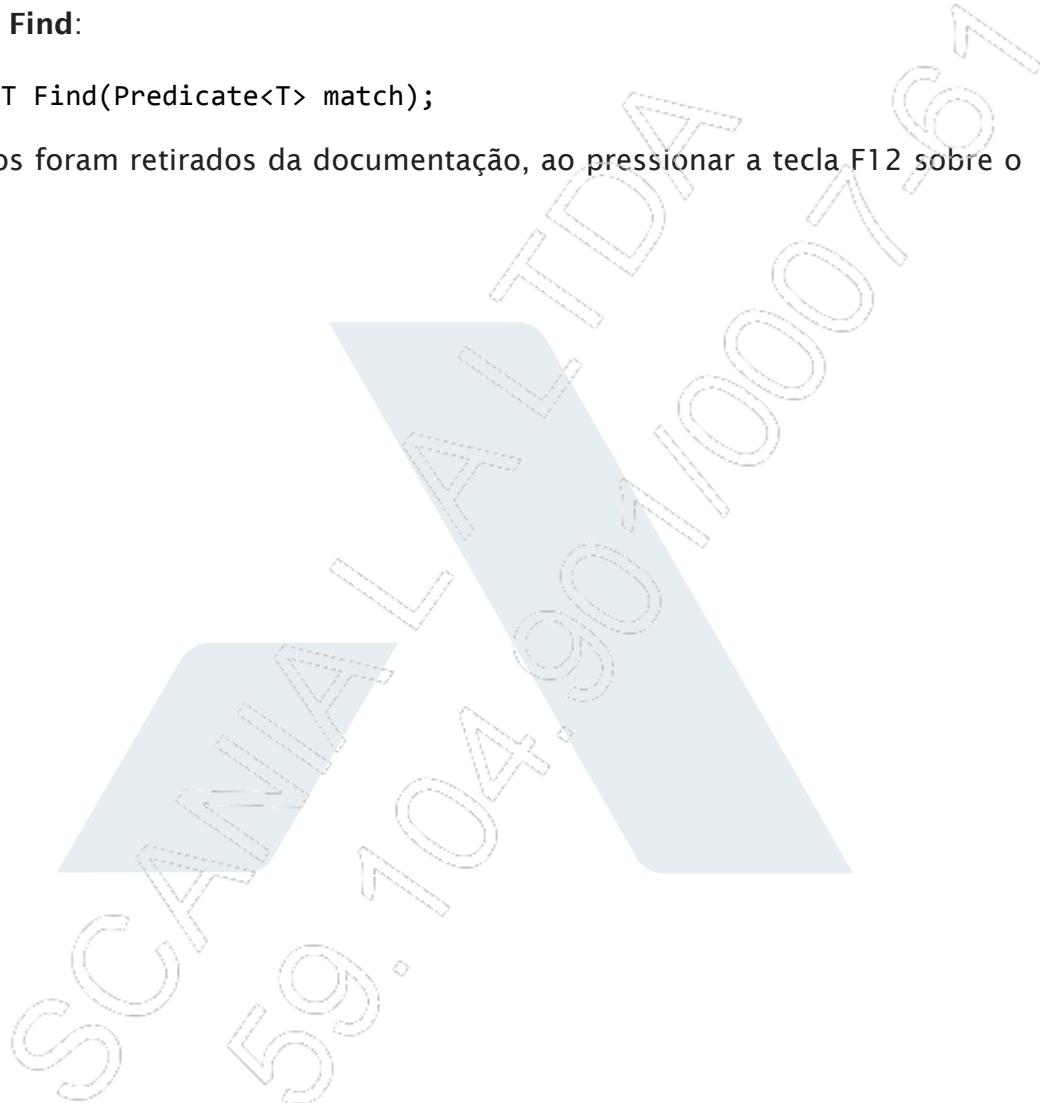
- delegate **Predicate**:

```
namespace System
{
    public delegate bool Predicate<in T>(T obj);
}
```

- Método **Find**:

```
public T Find(Predicate<T> match);
```

Esses métodos foram retirados da documentação, ao pressionar a tecla F12 sobre o recurso.



## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

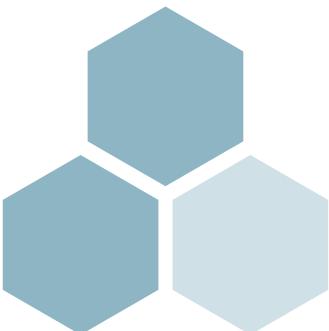
- Delegates são referências para métodos ou, mais precisamente, para operações. As operações referenciadas devem possuir a mesma sintaxe (parâmetros e tipo de retorno) definida no delegate;
- As expressões lambda permitem definir operações dinamicamente, sem a necessidade de existirem métodos previamente definidos;
- Podem existir delegates genéricos, e estes seguem as mesmas regras dos genéricos tradicionais. A variável definida com base no delegate é que definirá os tipos adequados para os parâmetros de tipos definidos no delegate;
- Existem delegates definidos no framework e prontos para uso. Os mais importantes são: **Predicate**, **Action** e **Func**.



10

# Delegates e expressões lambda

Teste seus conhecimentos



**1. Um delegate é usado para definir:**

- a) Novos métodos.
- b) Referências a operadores.
- c) Referências a classes.
- d) Referências a métodos.
- e) Referências a variáveis.

**2. Expressões lambda são usadas para:**

- a) Definir novos métodos.
- b) Definir novas operações.
- c) Definir novas classes.
- d) Definir novas estruturas.
- e) Definir novas interfaces.

**3. Qual das alternativas não representa um delegate predefinido no framework?**

- a) Func
- b) Action
- c) Predicate
- d) Expression
- e) EventHandler

**4. O método Find da classe List recebe como parâmetro o delegate:**

- a) Func
- b) Action
- c) Predicate
- d) Expression
- e) EventHandler

**5. O delegate Predicate possui como valor de retorno:**

- a) O tipo genérico TResult.
- b) O tipo bool.
- c) Não retorna nada.
- d) O tipo inteiro.
- e) Um array.



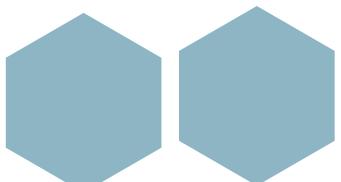


10

# Delegates e expressões lambda



Mãos à obra!



## Laboratório 1

### A – Utilizando expressões lambda

#### Objetivos:

Vamos adicionar um recurso para pesquisa de clientes, como meio de inserir o conceito sobre expressões lambda.

1. Crie um novo solution, vazio, chamado **Capítulo10.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models**, **Lab.Views**, **Lab.Interfaces**, **Lab.Arquivos** e **Lab.Dados** do Capítulo 9, na pasta do solution **Capítulo10.Labs**;
3. Adicione esses projetos ao novo solution;
4. No projeto **Lab.Models**, adicione uma pasta chamada **Delegates**;
5. Nesta nova classe, adicione uma classe chamada **ListaElementos**. Esta classe é genérica, e possuirá dois métodos que receberão um delegate como parâmetro – **Predicate**. Um método, **Buscar**, retornará um objeto que esteja de acordo com a condição informada como parâmetro; O segundo, **Listar**, irá apresentar todos os objetos que satisfaçam a condição informada:

```
namespace Lab.Models.Delegates
{
    public class ListaElementos<T>
    {
        private T[] elementos;

        public void CriarElementos(T[] elementos)
        {
            this.elementos = elementos;
        }

        public T Buscar(Predicate<T> verifica)
        {
            foreach (var item in elementos)
            {
                if (verifica(item))
                {
                    return item;
                }
            }
            return default(T);
        }
    }
}
```

```

public IEnumerable<T> Listar(Predicate<T> verifica)
{
    foreach (var item in elementos)
    {
        if (verifica(item))
        {
            yield return item;
        }
    }
}

```

## Laboratório 2

### A – Alterando a interface gráfica

**Objetivos:**

Alterar a interface gráfica, especificamente a aba onde é realizado o cadastro de clientes, e adicionar componentes para realizar buscas por clientes, usando os métodos definidos na classe **ListaElementos**.

1. Abra o arquivo **MainWindow.xaml**, no projeto **Lab.Views**. Neste arquivo, adicione os componentes destacados, na aba referente ao cadastro de clientes:

```

<TabItem Header="Cadastro de Clientes">
    <Grid Background="LightBlue">
        <Grid.RowDefinitions>
            <RowDefinition Height="225" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid Grid.Row="0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
            </Grid.RowDefinitions>
        
```

```
<!--Componentes Label-->
<Label Name="documentoLabel" Content="Documento:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="0" Grid.Column="0"/>

<Label Name="nomeLabel" Content="Nome:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="1" Grid.Column="0"/>

<Label Name="idadeLabel" Content="Idade:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="2" Grid.Column="0"/>

<Label Name="sexoLabel" Content="Sexo:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="3" Grid.Column="0"/>

<Label Name="ruaLabel" Content="Rua:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="4" Grid.Column="0"/>

<Label Name="numeroLabel" Content="Número:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="5" Grid.Column="0"/>

<Label Name="cidadeLabel" Content="Cidade:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="6" Grid.Column="0"/>

<Label Name="cepLabel" Content="CEP:"  
      VerticalAlignment="Center"  
      HorizontalAlignment="Right"  
      Grid.Row="7" Grid.Column="0"/>

<!--Componentes TextBox e ComboBox-->
<TextBox Name="documentoTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="0" Grid.Column="1"/>

<TextBox Name="nomeTextBox" Width="200"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="1" Grid.Column="1"/>

<TextBox Name="idadeTextBox" Width="100"  
        VerticalAlignment="Center"  
        HorizontalAlignment="Left"  
        Grid.Row="2" Grid.Column="1"/>
```

```
<ComboBox Grid.Row="3" Grid.Column="1"
          Name="sexoComboBox"
          VerticalAlignment="Center"
          HorizontalAlignment="Left"
          Width="150">
</ComboBox>

<TextBox Name="ruaTextBox" Width="200"
         VerticalAlignment="Center"
         HorizontalAlignment="Left"
         Grid.Row="4" Grid.Column="1"/>

<TextBox Name="numeroTextBox" Width="100"
         VerticalAlignment="Center"
         HorizontalAlignment="Left"
         Grid.Row="5" Grid.Column="1"/>

<TextBox Name="cidadeTextBox" Width="100"
         VerticalAlignment="Center"
         HorizontalAlignment="Left"
         Grid.Row="6" Grid.Column="1"/>

<TextBox Name="cepTextBox" Width="100"
         VerticalAlignment="Center"
         HorizontalAlignment="Left"
         Grid.Row="7" Grid.Column="1"/>

<!--Botão para incluir um cliente-->
<Button Grid.Row="8" Grid.Column="1"
        Name="incluirClienteButton"
        Content="Incluir Cliente"
        HorizontalAlignment="Left"
        Width="100"
        Click="incluirClienteButton_Click">
<Button.ToolTip>
<StackPanel Width="150" Height="20"
            Background="Beige">
<TextBlock>
    Permite incluir um cliente
</TextBlock>
</StackPanel>
</Button.ToolTip>
</Button>
</Grid>
```

```
<!--Componente usado para pesquisas por clientes-->
<Grid Grid.Row="1" Background="LightGray">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="25" />
        <RowDefinition Height="100" />
    </Grid.RowDefinitions>

    <Label Name="buscaLabel" Content="Buscar"
        VerticalAlignment="Center"
        HorizontalAlignment="Right"
        Grid.Row="0" Grid.Column="0" />

    <StackPanel Grid.Row="0" Grid.Column="1"
        Orientation="Horizontal">
        <TextBox Name="buscaTextBox" Width="120"
            VerticalAlignment="Center"
            HorizontalAlignment="Left" />
        <Button
            Name="buscarButton"
            Content="Buscar"
            HorizontalAlignment="Left"
            Width="100" Click="buscarButton_Click" />
        <Button
            Name="listarButton"
            Content="Listar"
            HorizontalAlignment="Left"
            Width="100" Click="listarButton_Click" />
    </StackPanel>

    <ListBox Name="resultadoListBox"
        Grid.Row="1" Grid.Column="1" />
</Grid>
</TabItem>
```

2. Escreva o código para o evento click do botão **buscarButton**:

```
e) private void buscarButton_Click(object sender, RoutedEventArgs e)
{
    ListaElementos<Cliente> lista = new ListaElementos<Cliente>();
    lista.CriarElementos(Metodos.ListarClientes().ToArray());

    var item = lista.Buscar(p => p.Nome.Contains(buscaTextBox.
Text));

    resultadoListBox.Items.Clear();
    resultadoListBox.Items.Add(item);

}
```

3. Escreva o código para o evento click do botão **listarButton**:

```
e) private void listarButton_Click(object sender, RoutedEventArgs e)
{
    ListaElementos<Cliente> lista = new ListaElementos<Cliente>();
    lista.CriarElementos(Metodos.ListarClientes().ToArray());

    var itens = lista.Listar(p => p.Nome.Contains(buscaTextBox.
Text));

    resultadoListBox.Items.Clear();
    resultadoListBox.ItemsSource = itens;
}
```

4. Para testar a aplicação, adicione alguns clientes. Em seguida, forneça parte do nome de um ou mais clientes no novo campo de entrada, e clique nos dois botões para constatar o resultado.

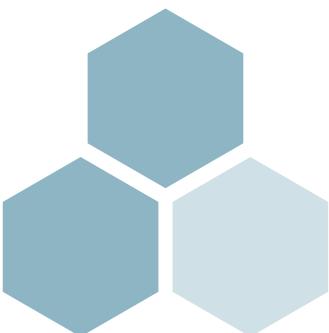




11

# Programação assíncrona

- Criação de tarefas - Classe Task;
- Aguardando pelo término de uma tarefa;
- Valor de retorno de uma Task;
- Operações assíncronas.



### 11.1. Introdução

Muitas aplicações executam tarefas de forma concomitante. Por exemplo, em uma interface gráfica, enquanto um usuário trabalha com os dados (consultando produtos, número de telefone de algum cliente etc.), uma taxa pode ser atualizada e exibida no topo da tela. Essas tarefas executadas em conjunto são bem sucedidas se ocorrerem de forma independente uma da outra, ou seja, de forma assíncrona.

O Framework .NET oferece diversos recursos poderosos para lidar com esse comportamento assíncrono. O objetivo deste capítulo é apresentar um conjunto de ferramentas que permitem realizar essas tarefas.

### 11.2. Criação de tarefas – Classe Task

No Framework .NET existe uma vertente conhecida como **Task Parallel Library**. Como o próprio nome sugere, a classe **Task** representa o núcleo do trabalho envolvendo tarefas.

A classe **Task** permite definir múltiplas tarefas concorrentemente, cada uma executando em um processo diferente (também conhecido como **Thread**).

Existem diversos cenários onde podemos usar a classe **Task**. Para exemplificar, considere o exemplo:

```
namespace Conceitos.CSharp.Cap11
{
    class Program
    {
        static void Main(string[] args)
        {
            Task t1 = new Task(new Action(MostrarHora));
            t1.Start();
            Console.ReadKey();
        }

        private static void MostrarHora()
        {
            Console.WriteLine($"Hora atual: {DateTime.Now:t}");
        }
    }
}
```

A classe **Task** possui diversas versões sobrecarregadas de construtores, mas todas recebem como parâmetro um delegate **Action**. No nosso exemplo, passamos o método **MostrarHora** para o delegate. O método **Start** da classe **Task** iniciou a execução dessa nova tarefa e, apesar de ser transparente para nós, ela ocorreu de forma assíncrona.

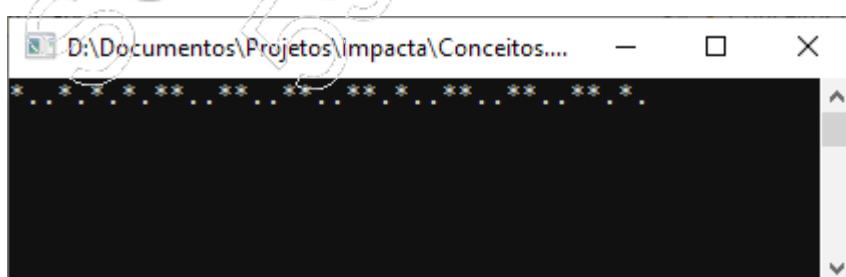
Para ficar mais claro, vamos considerar agora o seguinte exemplo:

```
namespace Conceitos.CSharp.Cap11
{
    class Program
    {
        static void Main(string[] args)
        {
            Task t2 = new Task(new Action(MostrarPontos));
            t2.Start();
            for (int i = 0; i < 20; i++)
            {
                Console.Write("*");
                Thread.Sleep(500);
            }

            Console.ReadKey();
        }

        private static void MostrarPontos()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write(".");
                Thread.Sleep(500);
            }
        }
    }
}
```

Nesse exemplo, criamos uma **Task** para executar um método responsável por mostrar vinte pontos na tela, em um intervalo de 500 milissegundos (esse intervalo é dado pela instrução **Thread.Sleep(500)**). Assim que iniciamos a execução da tarefa, escrevemos um código semelhante, só que mostrando vinte vezes o caractere asterisco. O restante do método **Main** e a tarefa referenciada por **t2** passam a executar independentemente, de forma assíncrona. Veja um resultado:



Como o parâmetro do construtor é um delegate, é possível passarmos uma expressão lambda:

```
namespace Conceitos.CSharp.Cap11
{
    class Program
    {
        static void Main(string[] args)
        {
            Task t1 = new Task(() => Console.WriteLine($"Hora atual:
{DateTime.Now:t}"));
            t1.Start();

            Console.ReadKey();
        }
    }
}
```

## 11.3. Aguardando pelo término de uma tarefa

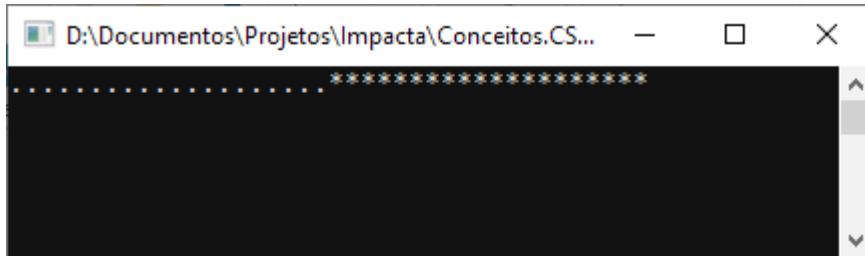
Mesmo quando duas ou mais tarefas são executadas assincronamente, pode haver a necessidade de uma tarefa ser finalizada para que outra possa prosseguir. Podemos executar esse procedimento através do método `Wait` da classe `Task`. Vamos ver o exemplo da apresentação de pontos e asterisco, para ilustrar:

```
namespace Conceitos.CSharp.Cap11
{
    class Program
    {
        static void Main(string[] args)
        {
            Task t3 = Task.Run(() =>
            {
                for (int i = 0; i < 20; i++)
                {
                    Console.Write(".");
                    Thread.Sleep(500);
                }
            });
            t3.Wait();

            for (int i = 0; i < 20; i++)
            {
                Console.Write("*");
                Thread.Sleep(500);
            }

            Console.ReadKey();
        }
    }
}
```

Nesse exemplo, o restante do código do método **Main** continua sua execução após a finalização da tarefa referenciada por **t3**. O resultado é mostrado adiante:



Temos ainda os métodos **WaitAll** e **WaitAny**, quando mais de uma tarefa deve ser executada concorrentemente. Veja o exemplo:

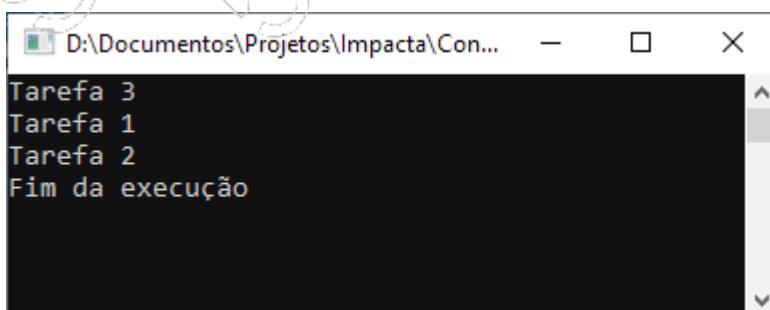
```
namespace Conceitos.CSharp.Cap11
{
    class Program
    {
        static void Main(string[] args)
        {
            Task[] tasks = new Task[3]
            {
                Task.Run( () => Console.WriteLine("Tarefa 1")),
                Task.Run( () => Console.WriteLine("Tarefa 2")),
                Task.Run( () => Console.WriteLine("Tarefa 3"))
            };

            Task.WaitAll(tasks);

            Console.WriteLine("Fim da execução");

            Console.ReadKey();
        }
    }
}
```

Resultado:



Observe que as tarefas em si não guardam uma ordem, como havíamos codificado, mas seu conjunto é aguardado para que o programa dê continuidade à sua execução.

## 11.4. Valor de retorno de uma Task

Na maioria dos cenários, tarefas retornam valores quando executadas. Os valores retornados são também chamados **results**. Para obtermos os valores de retorno de uma **Task**, usamos a versão genérica **Task<TResult>**.

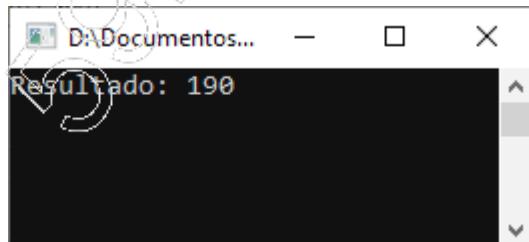
Ao trabalharmos com essa versão genérica, especificamos o tipo de retorno como parâmetro de tipo da **Task** a ser executada. Essa versão expõe a propriedade **Result**, que armazena o valor a ser retornado. O exemplo a seguir ilustra esse processo:

```
namespace Conceitos.CSharp.Cap11
{
    class Program
    {
        static void Main(string[] args)
        {
            Task<int> t4 = Task.Run<int>(() =>
            {
                int cont = 0;
                for (int i = 0; i < 20; i++)
                {
                    cont += i;
                }
                return cont;
            });

            Console.WriteLine("Resultado: " + t4.Result);
        }

        static void Main(string[] args)
        {
            Console.ReadKey();
        }
    }
}
```

Resultado:



Uma observação importante é que a propriedade **Result** é obtida somente quando a tarefa é finalizada. Isso equivale a usar o método **Wait**.

A versão genérica da classe **Task<TResult>** utiliza como parâmetro do método **Run** o delegate **Func**, já que este retorna um valor.

## 11.5. Operações assíncronas

Como pudemos estudar anteriormente, uma operação assíncrona é aquela executada em uma thread separada. Essas operações estão intimamente ligadas às tarefas.

### 11.5.1. `async` e `await`

Desde a versão 5 do C#, tarefas assíncronas se tornaram relativamente simples com o uso dos comandos `async` e `await`.

Usamos o comando `async` para definir métodos assíncronos, e o comando `await` para aguardar uma tarefa ser finalizada.

Geralmente, usamos métodos assíncronos (`async`) em operações que por sua própria natureza podem demorar para produzir um resultado. É o caso de métodos que fazem acesso a banco de dados ou Webservices.

Considere o método a seguir:

```
static async Task<IEnumerable<string>> ListarNomes()
{
    var nomes = new List<string>()
    {
        "Gabi", "Monica", "Junior", "Peter"
    };

    foreach (var item in nomes)
    {
        if (item.Contains("a"))
        {
            Console.WriteLine(item);
            await Task.Delay(200);
        }
    }
    return nomes;
}
```

Temos as seguinte considerações:

- Tecnicamente, o método retorna uma `Task<IEnumerable<string>>`, em vez de um `IEnumerable<string>`;
- O comando `async` foi usado na declaração do método e, por conta disso, o método passou a ser uma **tarefa assíncrona**;
- Como o método é uma tarefa assíncrona, pudemos retornar o valor correspondente ao parâmetro de tipo indicado na `Task (IEnumerable<string>)` em vez do objeto `Task` em si;

- Usamos o comando `await` na instrução `Task.Delay(200)`. Somente podemos usar o comando `await` em métodos declarados como `async`, e somente em métodos que são, eles próprios, uma `Task`. Verifique que o método `Delay` é uma `Task` (retorna um objeto `Task`).

Dessa forma, o próprio método `ListarNomes` é, em si, uma tarefa assíncrona que retorna um `IEnumerable<string>`.

Por ser uma tarefa, o método `ListarNomes` passou a ser "awaitable" e, dessa forma, podemos executá-lo de duas maneiras:

a) `var lista = await ListarNomes();`

b) `var lista = ListarNomes().Result;`

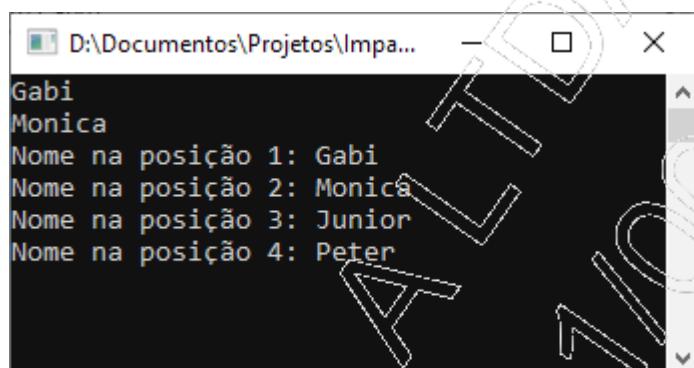
O item a) só pode ser usado em métodos declarados como `async`, como já mencionado antes.

Já o item b) pode ser executado em métodos não assíncronos, como é o caso do método `Main`.

Veja um exemplo de execução e seu resultado:

```
namespace Conceitos.CSharp.Cap11
{
    class Program
    {
        static async Task<IEnumerable<string>> ListarNomes()
        {
            var nomes = new List<string>()
            {
                "Gabi", "Monica", "Junior", "Peter"
            };
            foreach (var item in nomes)
            {
                if (item.Contains("a"))
                {
                    Console.WriteLine(item);
                    await Task.Delay(200);
                }
            }
            return nomes;
        }
    }
}
```

```
static void Main(string[] args)
{
    var lista = ListarNomes().Result;
    int x = 1;
    foreach (var item in lista)
    {
        Console.WriteLine($"Nome na posição {x++}: " + item);
    }
    Console.ReadKey();
}
```



Só usamos o comando **await** dentro de métodos declarados como **async**, e ele é aplicado a tarefas "awaitable", ou seja, definidas com retorno **Task** ou **Task<TResult>**.

### Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

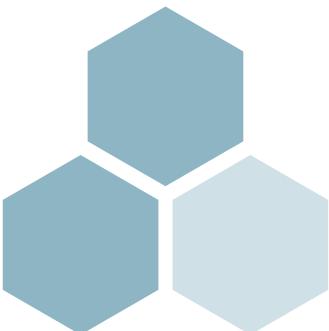
- A classe **Task** é usada para definir tarefas assíncronas. Podemos usá-la para instanciar objetos que representarão novas tarefas. Nesse caso, o parâmetro do construtor é um delegate do tipo **Action**;
- Para executar uma nova tarefa, podemos aplicar o método **Start** no objeto **Task**, ou usar o método **Run**, também de **Task**;
- Para aguardar o término de uma tarefa antes de prosseguir com o restante do código podemos usar o método **Wait** da classe **Task**;
- É comum uma tarefa (**Task**) retornar um valor. Nesse caso, usamos a versão genérica da classe **Task**, representada por **Task<TResult>**. Podemos extrair o valor retornado por meio da propriedade **Result**;
- Um método chamado de tarefa assíncrona pode ser definido com o uso do comando **async**. Nesse caso, podemos usar o comando **await** para aguardar uma tarefa na operação ser concluída;
- Somente podemos usar o comando **await** em métodos declarados como **async**, e em operações que são em si uma tarefa.



11

# Programação assíncrona

Teste seus conhecimentos



**1. Qual método da classe Thread é usado para realizar uma pausa no processo atual?**

- a) Sleep
- b) Delay
- c) Wait
- d) Pause
- e) Stop

**2. Para que um método seja considerado “awaitable”, ele deve ser declarado como:**

- a) Thread
- b) Task
- c) void
- d) object
- e) async

**3. Só é permitido usarmos o comando await em métodos declarados como:**

- a) await
- b) Task
- c) object
- d) void
- e) async

**4. Quando um método define um Task<TResult> como retorno, qual propriedade podemos usar para obter o valor de TResult?**

- a) Wait
- b) Task
- c) Result
- d) Value
- e) Não podemos obter o valor de retorno TResult, apenas Task<TResult>.

**5. O construtor da classe Task recebe como parâmetro um delegate do tipo:**

- a) Predicate
- b) Expression
- c) Action
- d) Func
- e) EventHandler



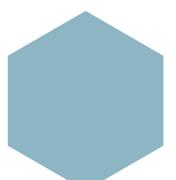


11

# Programação assíncrona



Mãos à obra!



## Laboratório 1

### A - Definindo tarefas assíncronas

#### Objetivos:

Neste laboratório, vamos definir algumas tarefas como assíncronas.

1. Crie um novo solution, vazio, chamado **Capítulo11.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models**, **Lab.Views**, **Lab.Interfaces**, **Lab.Arquivos** e **Lab.Dados** do Capítulo 10, na pasta do solution **Capítulo11.Labs**;
3. Adicione esses projetos ao novo solution;
4. Vamos tornar alguns métodos do nosso projeto assíncronos. Nos capítulos futuros, outros métodos também serão assíncronos. No projeto **Lab.Models**, abra a classe **Conta**. Nessa classe, transforme o método **RegistrarMovimento** em uma tarefa (Task):

```
namespace Lab.Models
{
    public abstract class Conta
    {
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }

        public List<Movimento> Movimentos { get; set; }

        public Cliente ClienteInfo { get; set; }

        public Conta(int Banco, string Agencia, string Conta)
        {
            if(this.Movimentos == null)
            {
                this.Movimentos = new List<Movimento>();
            }

            this.NumeroBanco = Banco;
            this.NumeroAgencia = Agencia;
            this.NumeroConta = Conta;
        }

        protected async Task RegistrarMovimento(double valor, Operacoes operacao)
        {
            if (valor <= 0)
            {
                throw new
                    ArgumentException("O valor deve ser positivo");
            }
        }
    }
}
```

```
this.Movimentos.Add(new Movimento()
{
    Data = DateTime.Now,
    Historico = operacao == Operacoes.Saque ? "SAQUE" :
    "DEPÓSITO",
    Operacao = operacao,
    Valor = valor
});
}

public abstract string MostrarExtrato();

public abstract void EfetuarOperacao(double valor,
Operacoes operacao = Operacoes.Deposito);

public override string ToString()
{
    return this.NumeroAgencia + "/" + this.NumeroConta;
}
}
```

5. Abra as classes **ContaCorrente** e **ContaEspecial**. Nessas classes, chame o método **RegistrarMovimento** com o comando **await**. Para isso, torne o método **EfetuarOperacao** assíncrono, ou seja, declarado como **async**:

#### • Classe ContaCorrente

```
namespace Lab.Models
{
    public class ContaCorrente : Conta
    {
        public double Saldo { get; protected set; }
        //public Cliente ClienteInfo { get; set; }

        public ContaCorrente(int Banco, string Agencia, string Conta):
            base(Banco, Agencia, Conta)
        {
        }

        public ContaCorrente(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public async override void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            if(valor <= 0)
            {
                throw new
                    ArgumentException("O valor deve ser positivo");
            }
        }
    }
}
```

```
        if(valor > this.Saldo)
        {
            throw new
                InvalidOperationException("Saldo insuficiente");
        }

        switch (operacao)
        {
            case Operacoes.Deposito:
                this.Saldo += valor;
                break;
            case Operacoes.Saque:
                if(valor <= this.Saldo)
                {
                    this.Saldo -= valor;
                }
                break;
        }
        await base.RegistrarMovimento(valor, operacao);
    }

    public virtual string Exibir()
    {
        string cliente = this.ClienteInfo != null ?
            this.ClienteInfo.Exibir() + '\n' : "";

        return $"{cliente}" +
            $"Banco: {this.NumeroBanco}\n" +
            $"Agência: {this.NumeroAgencia}\n" +
            $"Conta: {this.NumeroConta}\n" +
            $"Saldo Atual: {this.Saldo}";
    }

    public override string MostrarExtrato()
    {
        StringBuilder builder = new StringBuilder();
        builder.Append($"Cliente: {ClienteInfo.Nome}\n")
            .Append($"Banco: {NumeroBanco}\n")
            .Append($"Agência: {NumeroAgencia}\n")
            .Append($"Conta: {NumeroConta}\n")
            .Append(new string('-', 35) + '\n');

        if(this.Movimentos.Count() == 0)
        {
            builder.Append("Nenhum movimento registrado para esta
conta");
        }
        else
        {
            foreach (var item in this.Movimentos)
            {
                builder.Append($"{item}\n");
            }
        }
        builder.Append(new string('-', 35) + '\n');
        builder.Append($"Saldo: {this.Saldo:c}");
    }

    return builder.ToString();
}
}
```

- Classe ContaEspecial

```
namespace Lab.Models
{
    public class ContaEspecial : ContaCorrente
    {
        private double _limite;
        public double Limite
        {
            get => _limite;
            set => _limite = (value <= 0 ? value :
                throw new
                    InvalidOperationException("O Limite deve ser
positivo"));
        }

        public ContaEspecial(int Banco, string Agencia, string Conta) :
            base(Banco, Agencia, Conta)
        { }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
        }

        public ContaEspecial(int Banco, string Agencia, string Conta,
            Cliente cliente, double Limite)
            : this(Banco, Agencia, Conta)
        {
            this.ClienteInfo = cliente;
            this.Limite = Limite;
        }

        //métodos sobrescritos
        public async override void EfetuarOperacao(double valor,
            Operacoes operacao = Operacoes.Deposito)
        {
            if (valor <= 0)
            {
                throw new
                    ArgumentException("O valor deve ser positivo");
            }
            if (valor > this.Saldo)
            {
                throw new
                    InvalidOperationException("Saldo insuficiente");
            }
            switch (operacao)
            {
```

```
        case Operacoes.Deposito:
            this.Saldo += valor;
            break;
        case Operacoes.Saque:
            if (valor <= (this.Saldo + this.Limite))
            {
                this.Saldo -= valor;
            }
            break;
    }
    await base.RegistrarMovimento(valor, operacao);
}

public override string Exibir()
{
    return $"{base.Exibir()}\n" +
        $"Limite: {this.Limite}\n" +
        $"Salto Disponível: {this.Saldo + this.Limite}";
}

public override string MostrarExtrato()
{
    return new StringBuilder(base.MostrarExtrato())
        .Append($"\nLimite: {this.Limite}")
        .Append($"\nSalto Disponível: {this.Limite + this.
Saldo}")
        .ToString();
}
```

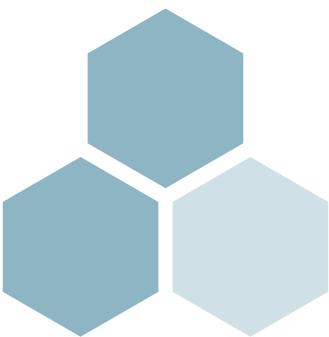
6. Teste a aplicação com essa nova configuração.



12

# Arquivos

- A classe File;
- Métodos adicionais;
- A classe Directory;
- Streams.



### 12.1. Introdução

A manipulação de arquivos é um processo natural em praticamente todo tipo de aplicação. Podemos mencionar os logs de erros gerados em um sistema, armazenamento de configurações de uma aplicação, dentre muitos outros exemplos. Entender o mecanismo de leitura e gravação de arquivos a partir de uma aplicação é tarefa indispensável.

O Framework .NET fornece o namespace **System.IO**, que contém um grande número de classes responsáveis por facilitar a tarefa de manipulação de arquivos. O propósito deste capítulo é apresentar as principais classes desse namespace.

### 12.2. A classe File

A classe **File** disponibiliza diversos métodos para realizarmos operações relacionadas à leitura e gravação de arquivos.

#### 12.2.1. Lendo dados de arquivos

Podemos obter dados de um arquivo através dos seguintes métodos da classe **File**:

- **ReadAllText**
- **ReadAllLines**
- **ReadAllBytes**

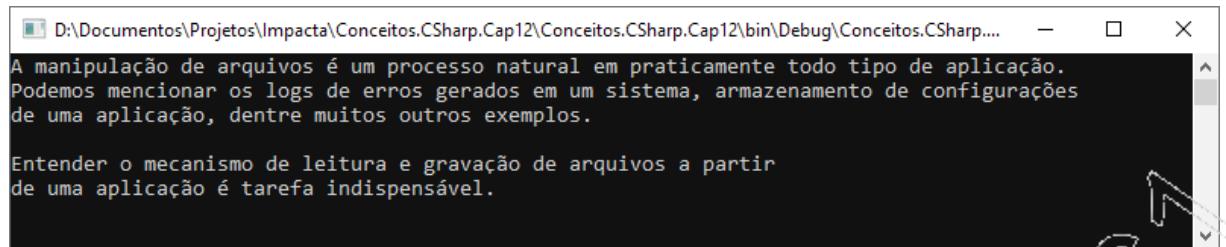
##### 12.2.1.1. Método ReadAllText

Este método permite obter o conteúdo de um arquivo e transferi-lo para uma string. Veja o exemplo:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\conteudo.txt";
            string texto = File.ReadAllText(path);

            Console.WriteLine(texto);
            Console.ReadKey();
        }
    }
}
```

O arquivo mencionado, **conteudo.txt**, apresenta parte do texto contido na introdução deste capítulo. Sua execução produz o resultado:



D:\Documentos\Projetos\Impacta\Conceitos.CSharp.Cap12\Conceitos.CSharp.Cap12\bin\Debug\Conceitos.CSharp....

```
A manipulação de arquivos é um processo natural em praticamente todo tipo de aplicação.  
Podemos mencionar os logs de erros gerados em um sistema, armazenamento de configurações  
de uma aplicação, dentre muitos outros exemplos.  
  
Entender o mecanismo de leitura e gravação de arquivos a partir  
de uma aplicação é tarefa indispensável.
```

### 12.2.1.2. Método ReadAllLines

Este método realiza a leitura de um arquivo e armazena cada linha em um novo elemento de um array de string. Vamos apresentar o mesmo exemplo anterior, só que considerando este método:

```
namespace Conceitos.CSharp.Cap12  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string path = @"D:\Documentos\Arquivos\conteudo.txt";  
            string[] linhas = File.ReadAllLines(path);  
  
            foreach (var item in linhas)  
            {  
                Console.WriteLine(item);  
            }  
  
            Console.ReadKey();  
        }  
    }  
}
```

Aparentemente, o resultado é o mesmo. A diferença é que foi exibida cada uma das linhas armazenadas.

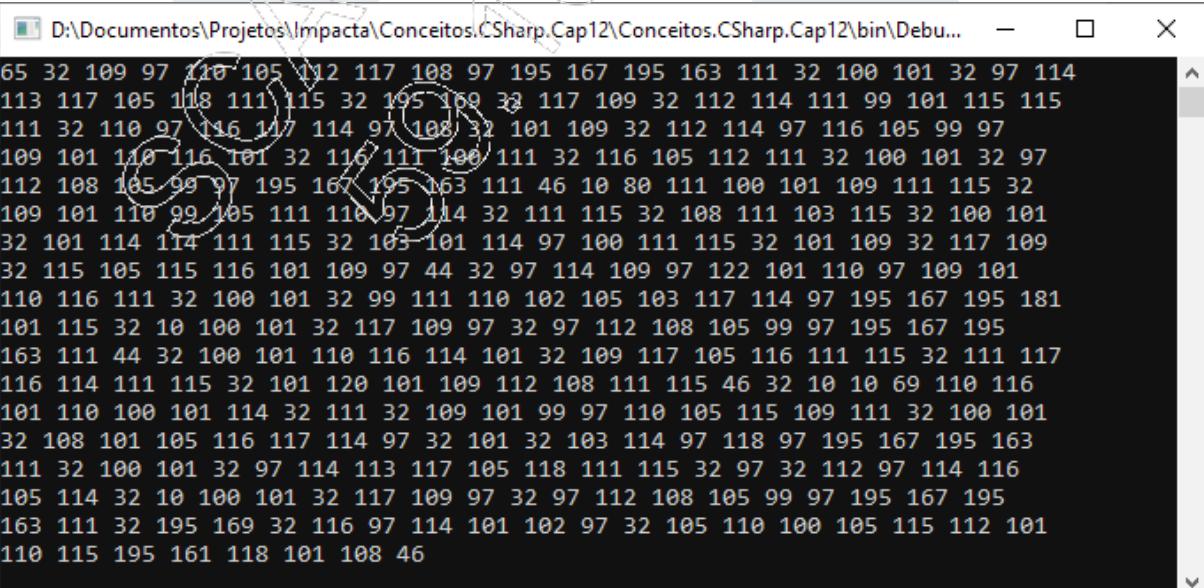
## 12.2.1.3. Método ReadAllBytes

Este método obtém o conteúdo do arquivo e o armazena no formato binário, em um array de bytes. Novamente, utilizaremos o mesmo arquivo:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\conteudo.txt";
            byte[] bytes = File.ReadAllBytes(path);

            for (int i = 0; i < bytes.Length; i++)
            {
                Console.Write(bytes[i] + " ");
                if(i > 0 && i % 20 == 0)
                {
                    Console.WriteLine();
                }
            }
            Console.ReadKey();
        }
    }
}
```

O método **ReadAllBytes** obtém o byte de cada caractere. No caso de um arquivo texto, é o valor numérico **Unicode** de cada caractere. Observe que o conteúdo do arquivo começa com a letra A, cujo valor Unicode é 65. Os demais caracteres seguem o mesmo procedimento. Veja o resultado:



```
D:\Documentos\Projetos\Impacta\Conceitos.CSharp.Cap12\Conceitos.CSharp.Cap12\bin\Debug... - □ ×
65 32 109 97 110 105 112 117 108 97 195 167 195 163 111 32 100 101 32 97 114
113 117 105 118 111 115 32 195 109 32 117 109 32 112 114 111 99 101 115 115
111 32 110 97 116 117 114 97 108 32 101 109 32 112 114 97 116 105 99 97
109 101 110 116 101 32 116 111 100 111 32 116 105 112 111 32 100 101 32 97
112 108 105 99 97 195 167 195 163 111 46 10 80 111 100 101 109 111 115 32
109 101 110 99 105 111 116 97 114 32 111 115 32 108 111 103 115 32 100 101
32 101 114 114 111 115 32 105 101 114 97 100 111 115 32 101 109 32 117 109
32 115 105 115 116 101 109 97 44 32 97 114 109 97 122 101 110 97 109 101
110 116 111 32 100 101 32 99 111 110 102 105 103 117 114 97 195 167 195 181
101 115 32 10 100 101 32 117 109 97 32 97 112 108 105 99 97 195 167 195
163 111 44 32 100 101 110 116 114 101 32 109 117 105 116 111 115 32 111 117
116 114 111 115 32 101 120 101 109 112 108 111 115 46 32 10 10 69 110 116
101 110 100 101 114 32 111 32 109 101 99 97 110 105 115 109 111 32 100 101
32 108 101 105 116 117 114 97 32 101 32 103 114 97 118 97 195 167 195 163
111 32 100 101 32 97 114 113 117 105 118 111 115 32 97 32 112 97 114 116
105 114 32 10 100 101 32 117 109 97 32 97 112 108 105 99 97 195 167 195
163 111 32 195 169 32 116 97 114 101 102 97 32 105 110 100 105 115 112 101
110 115 195 161 118 101 108 46
```

## 12.2.2. Escrevendo dados em arquivos

Analogamente ao processo de leitura de arquivos, a classe `File` disponibiliza os seguintes métodos para gravação de arquivos:

- `WriteAllText`
- `WriteAllLines`
- `WriteAllBytes`
- `AppendAllText`
- `AppendAllLines`

### 12.2.2.1. Método `WriteAllText`

Este método escreve o conteúdo de uma string em um arquivo. Exemplo de utilização:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\saída1.txt";
            string conteúdo = "Conteúdo: Gravação usando WriteAllText";
            File.WriteAllText(path, conteúdo);
            Console.WriteLine("Arquivo criado");

            Console.ReadKey();
        }
    }
}
```

Podemos ver que o arquivo foi criado, e que seu conteúdo está disponível.

### 12.2.2.2. Método WriteAllLines

Este método considera o conteúdo de cada elemento de um array de string e os escreve em um arquivo. Exemplo de utilização:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\saída2.txt";
            string[] linhas = { "Curso 1: Java", "Curso 2: Node.js", "Curso
3: Asp.Net" };
            File.WriteAllLines(path, linhas);
            Console.WriteLine("Arquivo criado");

            Console.ReadKey();
        }
    }
}
```

Cada elemento do array é incluído em uma linha do arquivo.

### 12.2.2.3. Método WriteAllBytes

Este método considera um array de bytes e escreve seu conteúdo em um arquivo. Veja o exemplo:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\saída3.txt";
            byte[] bytes = { 73, 109, 112, 97, 99, 116, 97, 32, 84,
101, 99, 110, 111, 108, 111, 103, 105, 97 };
            File.WriteAllBytes(path, bytes);

            Console.WriteLine("Arquivo criado");

            Console.ReadKey();
        }
    }
}
```

Ao abrir o arquivo gerado, temos um texto: **Impacta Tecnologia**.

### 12.2.2.4. Método AppendAllText

Trata-se de um método complementar. Ele escreve o conteúdo de uma variável no final de um arquivo existente.

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\conteudo.txt";
            string texto = "\n\nLista de Tópicos\n";
            File.AppendAllText(path, texto);

            Console.WriteLine("Texto adicionado");

            Console.ReadKey();
        }
    }
}
```

Consideramos o arquivo **conteudo.txt** que usamos para leitura, e adicionamos o texto "**Lista de tópicos**" no final.

### 12.2.2.5. AppendAllLines

Similarmente, este método escreve o conteúdo de um array de string no final de um arquivo existente.

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\conteudo.txt";
            string[] linhas = { "Leitura", "Gravação" };
            File.AppendAllLines(path, linhas);

            Console.WriteLine("Linhas adicionadas");

            Console.ReadKey();
        }
    }
}
```

As linhas **Leitura** e **Gravação** foram adicionadas ao final do arquivo. Os caracteres de nova linha (\n) foram necessários porque não há quebra automática de linhas ao adicionar conteúdo no final de um arquivo existente.

### 12.3. Métodos adicionais

Os métodos descritos a seguir fornecem funcionalidades úteis na manipulação de arquivos. Sua utilização é bastante simples:

Método	Sintaxe Básica	Descrição
<b>Copy</b>	File.Copy(source, dest, overwrite)	Copia o conteúdo do arquivo em <b>source</b> em um novo arquivo em <b>dest</b> . O parâmetro booleano <b>overwrite</b> , quando <b>true</b> , permite sobrescrever o conteúdo do destino.
<b>Delete</b>	File.Delete(path)	Remove o arquivo em <b>path</b> .
<b>Exists</b>	File.Exists(path)	Verifica se o arquivo em <b>path</b> existe. O método retorna um valor booleano.
<b>GetCreationTime</b>	File.GetCreationTime(path)	Retorna um dado do tipo <b>DateTime</b> indicando a data e a hora em que o arquivo indicado em <b>path</b> foi criado.

### 12.4. A classe Directory

Além dos arquivos, manipular o local onde eles se localizam é uma tarefa tão importante quanto os arquivos em si. Podemos obter, por exemplo, o número de arquivos em um diretório, criar novos diretórios, remover diretórios, entre outras informações.

Assim como a classe **File**, a classe **Directory** possui métodos estáticos úteis para interagirmos com as pastas do sistema.

A seguir, apresentamos os principais métodos e um exemplo de utilização de cada um deles.

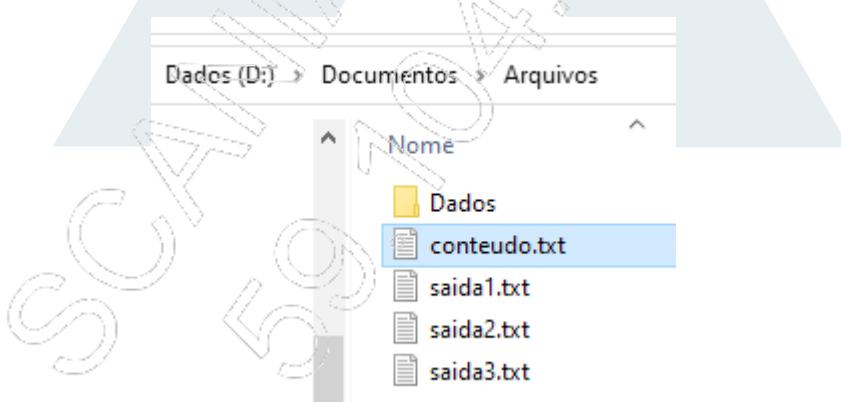
## 12.4.1. Método.CreateDirectory

Como o nome sugere, este método permite criar um novo diretório no sistema de arquivos:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\Dados";
            Directory.CreateDirectory(path);
            Console.WriteLine("Diretório criado");

            Console.ReadKey();
        }
    }
}
```

A pasta D:\Documentos\Arquivos já existia e a usamos em exemplos anteriores. A pasta Dados foi criada.



### 12.4.2. Método Delete

Este método é usado para remover o diretório indicado. Após ter criado o diretório Dados, vamos removê-lo:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\Dados";
            Directory.Delete(path);
            Console.WriteLine("Diretório removido");

            Console.ReadKey();
        }
    }
}
```

Quando o segundo parâmetro, **recursive**, é configurado com **false** e houver arquivos na pasta, uma exceção do tipo **IOException** é lançada.

```
Directory.Delete(path, false);
```

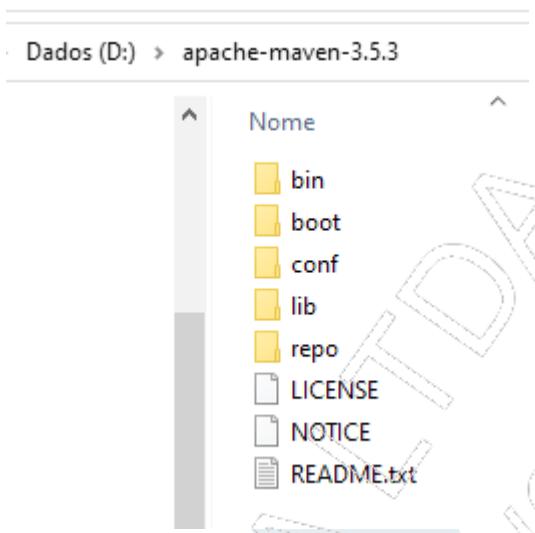
### 12.4.3. Método Exists

O método **Exists** verifica se um diretório existe no sistema de arquivos.

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\Dados";
            if (Directory.Exists(path))
            {
                Console.WriteLine("O diretório existe");
            }
            else
            {
                Console.WriteLine("O diretório não existe, ou foi
removido");
            }
            Console.ReadKey();
        }
    }
}
```

## 12.4.4. Método GetDirectories

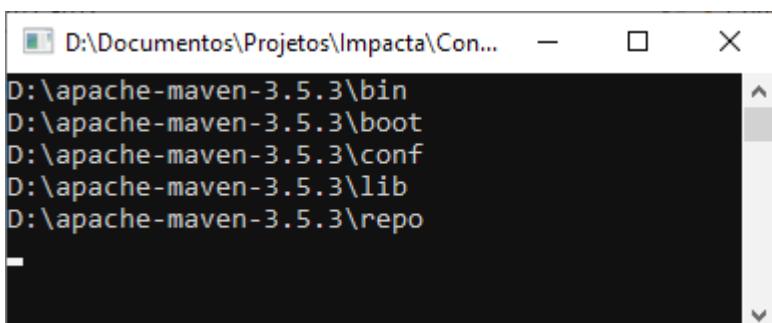
Este método lista os subdiretórios de um diretório indicado. Vamos ver a lista de diretórios na pasta D:\apache-maven-3.5.3:



```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\apache-maven-3.5.3";

            string[] pastas = Directory.GetDirectories(path);
            foreach (var item in pastas)
            {
                Console.WriteLine(item);
            }
            Console.ReadKey();
        }
    }
}
```

O resultado obtido é apresentado adiante:



```
D:\apache-maven-3.5.3\bin
D:\apache-maven-3.5.3\boot
D:\apache-maven-3.5.3\conf
D:\apache-maven-3.5.3\lib
D:\apache-maven-3.5.3\repo
```

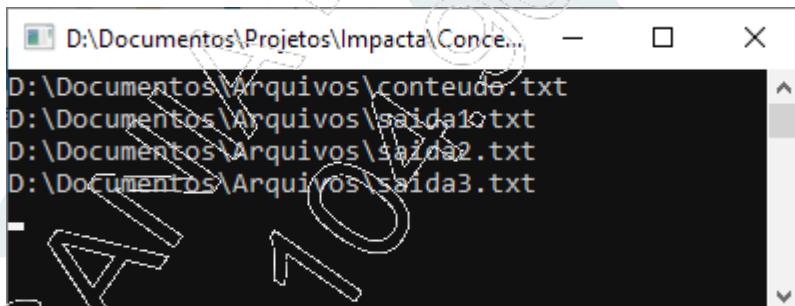
## 12.4.5. Método GetFiles

Analogamente, este método retorna uma lista dos arquivos presentes no diretório especificado.

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos";

            string[] pastas = Directory.GetFiles(path);
            foreach (var item in pastas)
            {
                Console.WriteLine(item);
            }
            Console.ReadKey();
        }
    }
}
```

O resultado é justamente o conjunto de arquivos usados neste capítulo.



## 12.5. Streams

Quando trabalhamos com arquivos de forma transacional, os dados são armazenados em memória e então transmitidos em uma simples operação. Como exemplo, podemos citar um vídeo contendo em torno de 100 gigabytes. Sua leitura certamente levará um tempo enorme, e o sistema poderá ficar bastante sobrecarregado por conta da grande quantidade de memória consumida.

Para evitar este inconveniente, o Framework .NET fornece um mecanismo de acesso a dados do sistema de arquivos por meio de **streams**. Streams são sequências de bytes cuja utilização ocorre em pequenas porções.

Tipicamente, os streams processam as seguintes operações:

- Leitura de pedaços de dados em variáveis, normalmente em array de bytes;
- Gravação de pedaços de dados de uma variável para um stream;
- Pesquisa e alteração de partes de dados a partir de uma determinada posição.

## 12.5.1. Manipulação de Streams

A plataforma .NET disponibiliza uma série de classes para a manipulação de streams, e a escolha por alguma destas classes leva em conta, especificamente:

- Qual o tipo de dados que se pretende ler ou gravar: texto ou binário;
- Onde as informações serão armazenadas: sistema de arquivos, na memória, ou na Web.

Podemos considerar as seguintes classes para a manipulação de Streams:

- **StreamReader**
- **StreamWriter**
- **BinaryReader**
- **BinaryWriter**

## 12.5.2. A classe StreamReader

Esta classe permite a leitura de dados textuais a partir de um stream. Exemplo:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\conteudo.txt";

            //Objeto usado para estabelecer o acesso ao stream
            FileStream file = new FileStream(path, FileMode.Open);

            //Objeto usado para ler os caracteres do FileStream
            StreamReader reader = new StreamReader(file);

            //obtém acesso ao próximo caractere no stream
            while(reader.Peek() != -1)
            {
                Console.Write((char)reader.Read()); //escreve o caractere lido
            }
            reader.Close();
            file.Close();

            Console.ReadKey();
        }
    }
}
```

Neste exemplo:

- Definimos um objeto **FileStream** para estabelecer o acesso ao stream;
- Criamos um objeto **StreamReader** para manipular sequencialmente os caracteres do stream;
- Em uma estrutura de repetição, obtemos uma referência a cada um dos caracteres do stream, lemos o caractere pelo seu código Unicode e o apresentamos na tela.

A utilização da classe **StreamReader** nos fornece um meio mais simplificado de leermos o conteúdo do arquivo: linha a linha e do início ao fim. O exemplo a seguir ilustra as duas maneiras:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\conteudo.txt";

            StreamReader reader = new StreamReader(path);

            //Forma 1 - leitura linha por linha
            string linha;
            while((linha = reader.ReadLine()) != null)
            {
                Console.WriteLine(linha);
            }

            Console.WriteLine(new string('-', 30));
            //Forma 2 - leitura de todo o arquivo
            Console.WriteLine(reader.ReadToEnd());

            reader.Close();
            Console.ReadKey();
        }
    }
}
```

O resultado foi omitido, mas é o conteúdo do arquivo.

## 12.5.3.A classe StreamWriter

Esta classe permite a gravação de dados textuais em um stream. É permitido usarmos o objeto **FileStream**, mas no exemplo vamos considerar o acesso ao arquivo pelo seu nome:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\novoarquivo.txt";

            //Obtém o objeto para acessar o fluxo
            StreamWriter writer = new StreamWriter(path);

            //define o conteúdo a ser armazenado no arquivo
            writer.WriteLine("Conteúdo do arquivo");
            writer.WriteLine("Mais conteúdo para o arquivo");

            //fecha o stream para liberar o conteúdo para o arquivo
            writer.Close();

            Console.WriteLine("Arquivo criado com sucesso");
            Console.ReadKey();
        }
    }
}
```

O exemplo anterior escreve o conteúdo no arquivo, sempre gerando um novo arquivo. Se quisermos que o novo conteúdo seja adicionado ao arquivo, caso este exista, devemos adicionar um segundo parâmetro ao construtor da classe **StreamWriter**, indicando que o arquivo deve ser aproveitado. Esse segundo parâmetro é booleano.

```
StreamWriter writer = new StreamWriter(path, true);
```

O valor default para esse parâmetro é **false**.

### 12.5.4. A classe BinaryReader

Esta classe permite a leitura de dados binários a partir de um stream. O exemplo adiante ilustra sua utilização:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\conteudo.txt";

            //Objeto usado para estabelecer o acesso ao stream
            FileStream file = new FileStream(path, FileMode.Open);

            //Objeto usado para manipular os streams obtidos por FileStream
            BinaryReader reader = new BinaryReader(file);

            //cria um array com a quantidade do inicio ao fim de bytes
            no stream
            byte[] bytes = new byte[reader.BaseStream.Length];

            //efetua a leitura dos bytes do arquivo, e os armazena no
            //array criado
            int valor;
            int posicao = 0;
            while((valor = reader.Read()) != -1)
            {
                bytes[posicao++] = (byte)valor;
            }

            Console.WriteLine("Bytes lidos com sucesso");
            //continua o programa...
            Console.ReadKey();
        }
    }
}
```

Neste exemplo:

- Criamos um objeto **FileStream** para permitir o acesso aos streams do arquivo;
- Criamos um objeto **BinaryReader** para ler os bytes do stream;
- Definimos um array de bytes com o tamanho obtido a partir do stream proveniente do arquivo;
- Em uma estrutura de repetição, realizamos a leitura de cada byte e o armazenamos em cada posição do array criado. O método **Read** realiza uma leitura sequencial, e retorna -1 quando não houver mais bytes a serem lidos.

## 12.5.5.A classe BinaryWriter

Esta classe permite a escrita de dados binários em um stream. Considere o exemplo a seguir:

```
namespace Conceitos.CSharp.Cap12
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Documentos\Arquivos\bytes.txt";

            //Objeto usado para estabelecer o acesso ao stream
            FileStream file = new FileStream(
                path,
                FileMode.Create,      //sempre cria um novo arquivo
                FileAccess.Write);    //executa apenas gravação

            //Objeto usado para manipular os streams obtidos por FileStream
            BinaryWriter writer = new BinaryWriter(file);

            //cria um array com os bytes a serem gravados
            byte[] dados = { 1, 4, 6, 7, 12, 33, 26, 98, 82, 101 };

            //escreve cada byte no stream
            foreach (var item in dados)
            {
                writer.Write(item);
            }

            //fecha cada um dos streams para liberar o conteúdo para o arquivo
            writer.Close();
            file.Close();

            Console.WriteLine("Bytes gravados com sucesso");
            //continua o programa...
            Console.ReadKey();
        }
    }
}
```

Neste exemplo:

- Criamos um objeto **FileStream** para gerar o acesso a stream, devidamente configurado para sempre gerar um novo stream, com saída para gravação;
- Criamos um objeto **BinaryWriter** para obter e escrever dados no stream;
- Definimos um array de bytes a serem armazenados no stream;
- Em uma estrutura de repetição, lemos cada byte do array e o armazenamos no stream;
- Fechamos os streams para liberar o conteúdo para o arquivo.

### Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

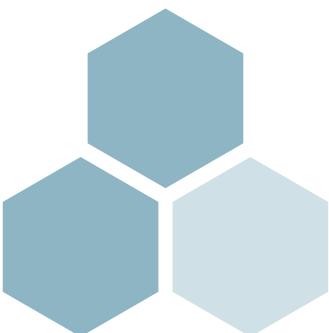
- O namespace **System.IO** nos fornece diversas classes para o gerenciamento de arquivos, no sistema de arquivos;
- A leitura e gravação dos arquivos com a classe **File** ocorre de forma transacional, ou seja, seu conteúdo é armazenado na memória antes se ser manipulado;
- Além da classe **File**, temos a classe **Directory**, para gerenciar o conteúdo de diretórios presentes no sistema de arquivos;
- Para otimizar o fluxo de dados de forma transacional, o Framework .NET disponibiliza um conjunto de classes que permite gerenciar dados no sistema de arquivos em pequenas partes. São os **Streams**.



12

# Arquivos

Teste seus conhecimentos



**1. Qual método da classe File permite obter todo o conteúdo de um arquivo texto e transferi-lo para uma string?**

- a) ReadAllFile
- b) ReadAllText
- c) Read
- d) ReadTextFile
- e) ReadFile

**2. O método ReadAllLines nos permite:**

- a) Obter cada um dos caracteres de um arquivo.
- b) Obter cada um dos espaços de um arquivo.
- c) Obter cada uma das linhas de um arquivo.
- d) Obter cada uma das palavras de um arquivo.
- e) Obter todo o conteúdo de um arquivo.

**3. O que faz o método Copy da classe File?**

- a) Copia os bytes de um arquivo em um novo arquivo texto.
- b) Copia o conteúdo de um arquivo em uma nova pasta.
- c) Copia o conteúdo de uma pasta em um novo arquivo.
- d) Copia o conteúdo do arquivo original em um novo arquivo.
- e) Copia o conteúdo do arquivo original em um fluxo de bytes.

**4. Qual método da classe Directory permite criar uma nova pasta?**

- a) CreateFolder
- b) Create
- c) CreateDirectory
- d) NewFolder
- e) NewDirectory

**5. O que são streams?**

- a) Streams são sequências de caracteres armazenados todos na memória.
- b) Streams são sequências de bytes armazenados integralmente na memória.
- c) Streams são sequências de bytes cuja utilização ocorre em pequenas porções.
- d) Streams são sequências de bytes gravados no arquivo para posterior leitura.
- e) Streams são objetos capazes de armazenar linhas de arquivos.





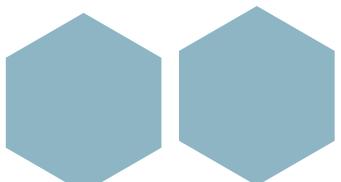
12

# Arquivos



Mãos à obra!

SCALPARE ALTA  
50° 104.997 10007-67



## Laboratório 1

### A – Escrevendo arquivos

#### Objetivos:

- Escrever um arquivo para armazenar o extrato de uma conta;
- Escrever um arquivo para armazenar as informações dos erros gerados nos eventos da aplicação;
- Incluir recursos para visualizar os arquivos gerados.

1. Crie um novo solution, vazio, chamado **Capítulo12.Labs**;
2. Copie os projetos **Lab.Utilitários**, **Lab.Models**, **Lab.Views**, **Lab.Interfaces**, **Lab.Arquivos** e **Lab.Dados** do Capítulo 11 na pasta do solution **Capítulo12.Labs**;
3. Adicionar esses projetos ao novo solution;
4. No projeto **Lab.Arquivos**, abra o arquivo **AcessoArquivo.cs**;
5. Nesta classe, escreva o método **GerarLog**, responsável por registrar o log de erros quando uma exceção for gerada na interface gráfica; o método **GerarExtrato**, para armazenar em um arquivo o extrato de uma determinada conta; o método **AbrirArquivo**, que permitirá abrir o extrato salvo pelo método **GerarExtrato**; e o método **AbrirLog**, para apresentar o conteúdo do arquivo de log gerado pelo método **GerarLog**:

```
namespace Lab.Arquivos
{
    public class AcessoArquivo
    {
        static string path = @"D:\Documentos\Projetos\Impacta\CSharp\Capítulo12.Labs";

        //método para gravar o extrato
        public static void GerarExtrato<T>(T conta) where T: Conta
        {
            try
            {
                if (!Directory.Exists(path + @"\Extrato"))
                {
                    Directory.CreateDirectory(path + @"\Extrato");
                }
                string arquivo = "extrato_" + $"{DateTime.Now:yyyyMMddHHmmssss}.txt";
                File.WriteAllText(path + @"\Extrato\" + arquivo, conta.ToString());
            }
            catch
            {
                throw;
            }
        }
    }
}
```

```
//método para gravar o arquivo de log
public static void GerarLog(string erro)
{
    try
    {
        if (!Directory.Exists(path + @"\Log"))
        {
            Directory.CreateDirectory(path + @"\Log");
        }
        StreamWriter writer = new StreamWriter(path + @"\Log\"
Erros.log",
writer.WriteLine(${[ {DateTime.Now:dd/MM/yyyy HH:mm:ss} ] - 
{erro}}");
        writer.Close();
    }
    catch
    {
        throw;
    }
}

//método para abrir um extrato selecionado
public static string AbrirExtrato(string caminho)
{
    try
    {
        StreamReader reader = new StreamReader(caminho);
        return reader.ReadToEnd();
    }
    catch
    {
        throw;
    }
}

//método para abrir e apresentar o log de erros
public static string AbrirLog()
{
    try
    {
        StreamReader reader = new StreamReader(path + @"\Log\"
Erros.log");
        return reader.ReadToEnd();
    }
    catch
    {
        throw;
    }
}
```

6. Agora, vamos realizar algumas alterações na interface gráfica, projeto **Lab.Views**. Primeiro, acesse a aba "**Operações Bancárias**" – elemento `<TabItem Header="Operações Bancárias">`. Adicione um botão para gravar o extrato (`salvarExtratoButton`):

```
<TabItem Header="Operações Bancárias">

    <Grid Background="LightSalmon">
        <Grid.RowDefinitions>
            <RowDefinition Height="100" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid Grid.Row="0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
                <RowDefinition Height="25"/>
            </Grid.RowDefinitions>

            <!--Componentes Label-->
            <Label Name="numeroContaLabel" Content="Conta:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="0" Grid.Column="0"/>
            <Label Name="operacaoLabel" Content="Operação:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="1" Grid.Column="0"/>
            <Label Name="valorLabel" Content="Valor:" VerticalAlignment="Center" HorizontalAlignment="Right" Grid.Row="2" Grid.Column="0"/>

            <!--Componentes TextBox e ComboBox-->
            <ComboBox Grid.Row="0" Grid.Column="1" Name="numeroContaComboBox" VerticalAlignment="Center" HorizontalAlignment="Left" Width="150">
            </ComboBox>

            <ComboBox Grid.Row="1" Grid.Column="1" Name="operacaoComboBox" VerticalAlignment="Center" HorizontalAlignment="Left" Width="150">
            </ComboBox>
        </Grid>
    </Grid>
</TabItem>
```

```
<TextBox Name="valorTextBox" Width="100"
VerticalAlignment="Center"
HorizontalAlignment="Left"
Grid.Row="2" Grid.Column="1"/>

<!--Botões para incluir operação e mostrar
extrato--&gt;
&lt;StackPanel Grid.Row="3" Grid.Column="1"
Orientation="Horizontal"
VerticalAlignment="Center"&gt;
    &lt;Button Name="executarButton"
Content="Executar"
VerticalAlignment="Center"
Width="100" Click="executarButton_
Click" /&gt;
    &lt;Button Name="extratoButton"
Content="Mostrar Extrato"
VerticalAlignment="Center"
Width="150" Click="extratoButton_
Click" /&gt;
    &lt;Button Name="salvarExtratoButton"
Content="Salvar Extrato"
VerticalAlignment="Center"
Width="150" Click="salvarExtratoButton_
Click" /&gt;
&lt;/StackPanel&gt;
&lt;/Grid&gt;
&lt;Grid Grid.Row="1"&gt;
    &lt;TextBox Name="extratoTextBox"
Height="250"
TextWrapping="Wrap"
AcceptsReturn="True"
VerticalScrollBarVisibility="Auto"
FontFamily="Courier New"
FontSize="16"/&gt;
&lt;/Grid&gt;
&lt;/Grid&gt;
&lt;/TabItem&gt;</pre>
```

7. Escreva o código para o evento click do botão **salvarExtratoButton**:

```
private void salvarExtratoButton_Click(object sender,
RoutedEventArgs e)
{
    try
    {
        var conta = (Conta)numeroContaComboBox.SelectedItem;
        AcessoArquivo.GerarExtrato<Conta>(conta);
        MessageBox.Show("Extrato armazenado com sucesso!");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message,
            "Erro",
            MessageBoxButton.OK,
            MessageBoxIcon.Error);
    }
}
```

8. No bloco **catch** de todos os eventos, realize uma chamada ao método **GerarLog** da classe **AcessoArquivo**:

```
namespace Lab.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            sexoComboBox.Items.Add(Sexos.Masculino);
            sexoComboBox.Items.Add(Sexos.Feminino);
            sexoComboBox.SelectedIndex = 0;

            operacaoComboBox.Items.Add(Operacoes.Deposito);
            operacaoComboBox.Items.Add(Operacoes.Saque);
            operacaoComboBox.SelectedIndex = 0;
        }

        private void incluirClienteButton_Click(object sender,
RoutedEventArgs e)
{
    try
    {
        //obtendo os dados do endereço
        Endereco endereco = new Endereco(
            ruaTextBox.Text,
            int.Parse(numeroTextBox.Text),
            cidadeTextBox.Text,
            cepTextBox.Text);
```

```
//obtendo os dados do cliente
int digitos = documentoTextBox.Text.Length;
Cliente cliente;

if (digitos == 11)
{
    cliente = new ClientePF(
        documentoTextBox.Text,
        nomeTextBox.Text,
        (Sexos)sexoComboBox.SelectedItem,
        int.Parse(idadeTextBox.Text),
        endereco);
}
else if (digitos == 14)
{
    cliente = new ClientePJ(
        documentoTextBox.Text,
        nomeTextBox.Text,
        (Sexos)sexoComboBox.SelectedItem,
        int.Parse(idadeTextBox.Text),
        endereco);
}
else
{
    throw new Exception("Documento inválido");
}

//adicionando o novo cliente na lista
Metodos.AdicionarCliente(cliente);

//vinculando a lista de clientes ao
//componente ComboBox
clienteComboBox.ItemsSource = Metodos.ListarClientes();
MessageBox.Show("Cliente incluído com sucesso!");
}
catch (Exception ex)
{
    AcessoArquivo.GerarLog(ex.Message);
    MessageBox.Show(ex.Message,
        "Erro",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}
```

```
//MessageBox.Show(cliente.Exibir());  
}  
  
private bool VerificarEspecial { get; set; }  
  
private void especialRadioButton_CheckedChanged(object sender,  
RoutedEventArgs e)  
{  
    try  
{  
        var radio = sender as RadioButton;  
        VerificarEspecial = (radio == especialRadioButton);  
  
        limiteLabel.IsEnabled = VerificarEspecial;  
        limiteTextBox.IsEnabled = VerificarEspecial;  
    }  
    catch (Exception ex)  
{  
        AcessoArquivo.GerarLog(ex.Message);  
        MessageBox.Show(ex.Message,  
            "Erro",  
            MessageBoxButton.OK,  
            MessageBoxIcon.Error);  
    }  
}  
  
private void incluirContaButton_Click(object sender, RoutedEventArgs  
e)  
{  
    try  
{  
        Conta conta;  
        var cliente = (Cliente)clienteComboBox.SelectedItem;  
  
        int banco = int.Parse(bancoTextBox.Text);  
        string agencia = agenciaTextBox.Text;  
        string numConta = contaTextBox.Text;  
  
        if (VerificarEspecial)  
        {  
            conta = new ContaEspecial(banco, agencia, numConta);  
            ((ContaEspecial)conta).Limite =  
                double.Parse(limiteTextBox.Text);  
        }  
        else  
        {  
            conta = new ContaCorrente(banco, agencia, numConta);  
        }  
        //vinculamos o cliente selecionado à nova conta  
        conta.ClienteInfo = cliente;  
  
        //adicionamos a nova conta à lista de contas do  
        //cliente selecionado  
        cliente.Contas.Add(conta);  
    }  
}
```

```
//incluímos a nova conta à lista global de contas
Metodos.AdicionarConta(conta);

//vinculamos a lista global de contas ao
//componente ComboBox
numeroContaComboBox.ItemsSource = Metodos.ListarContas();

    MessageBox.Show("Conta adicionada com sucesso!");
}
catch (Exception ex)
{
    AcessoArquivo.GerarLog(ex.Message);
    MessageBox.Show(ex.Message,
        "Erro",
        MessageBoxButton.OK,
        MessageBoxIcon.Error);
}

private void executarButton_Click(object sender,
    RoutedEventArgs e)
{
    try
    {
        //obtendo a conta selecionada
        var conta = (Conta)numeroContaComboBox.SelectedItem;

        //obtendo a operação
        var operacao = (Operacoes)operacaoComboBox.SelectedItem;

        //obtendo o valor da operação
        double valor = double.Parse(valorTextBox.Text);

        //executar a operação
        conta.EfetuarOperacao(valor, operacao);

        MessageBox.Show("Operação realizada com sucesso!");
        valorTextBox.Clear();
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message,
            "Erro",
            MessageBoxButton.OK,
            MessageBoxIcon.Error);
    }
}
```

```
private void extratoButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var conta = (Conta)numeroContaComboBox.SelectedItem;
        extratoTextBox.Text = conta.MostrarExtrato();
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message, "Erro",
            MessageBoxButton.OK,
            MessageBoxImage.Error);
    }
}

private void buscarButton_Click(object sender, RoutedEventArgs e)
{
    ListaElementos<Cliente> lista = new ListaElementos<Cliente>();
    lista.CriarElementos(Metodos.ListarClientes().ToArray());

    var item = lista.Buscar(p => p.Nome.Contains(buscaTextBox.Text));

    resultadoListBox.Items.Clear();
    resultadoListBox.Items.Add(item);
}

private void listarButton_Click(object sender, RoutedEventArgs e)
{
    ListaElementos<Cliente> lista = new ListaElementos<Cliente>();
    lista.CriarElementos(Metodos.ListarClientes().ToArray());

    var itens = lista.Listar(p => p.Nome.Contains(buscaTextBox.Text));

    resultadoListBox.Items.Clear();
    resultadoListBox.ItemsSource = itens;
}
```

```
        private void salvarExtratoButton_Click(object sender,
RoutedEventArgs e)
{
    try
    {
        var conta = (Conta)numeroContaComboBox.SelectedItem;
        AcessoArquivo.GerarExtrato<Conta>(conta);
        MessageBox.Show("Extrato armazenado com sucesso!");
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message,
                    "Erro",
                    MessageBoxButton.OK,
                    MessageBoxImage.Error);
    }
}
}
```

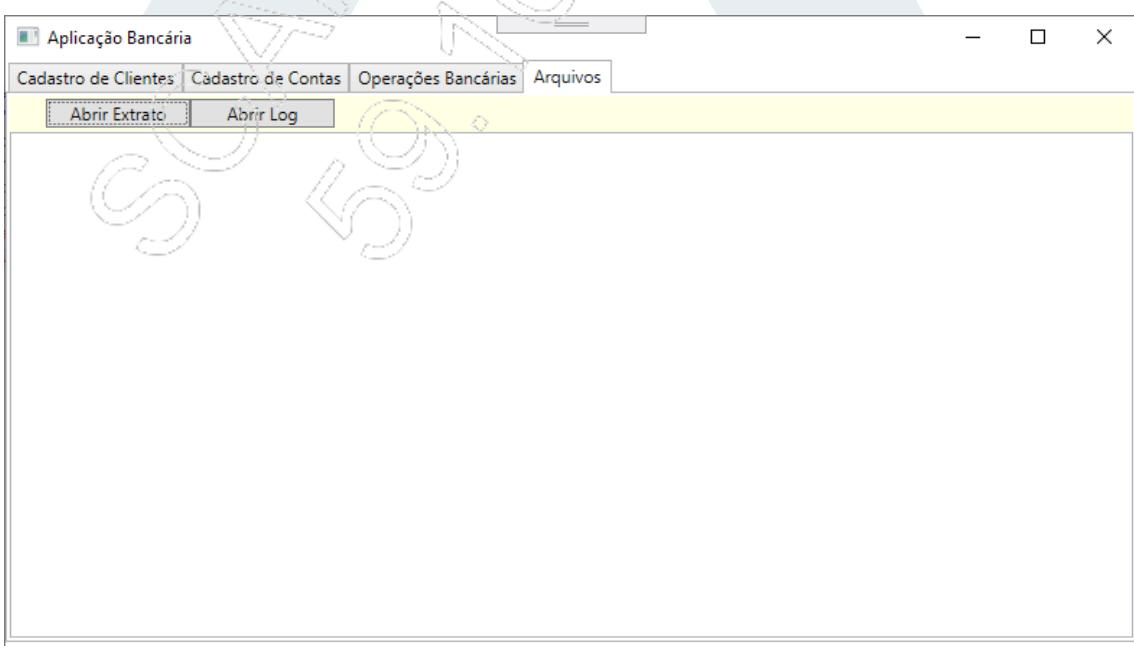
9. Na interface gráfica, adicione uma nova aba. Escreva a palavra "**Arquivos**" como valor do Header. Essa nova aba deverá ter apenas dois botões e uma caixa de textos com recurso para várias linhas. O primeiro botão se chamará **abrirExtratoButton**, e será usado para permitir o acesso aos arquivos de extratos. O segundo botão, **abrirLogButton** será usado para permitir a abertura do arquivo de log:

```
<TabItem Header="Arquivos">
    <Grid Background="LightYellow">
        <Grid.RowDefinitions>
            <RowDefinition Height="25" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid Grid.Row="0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="25" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
            </Grid.RowDefinitions>
        </Grid>
    </Grid>

```

```
<!--Botões para incluir operação e mostrar  
extrato-->  
<StackPanel Grid.Row="0" Grid.Column="1"  
Orientation="Horizontal"  
VerticalAlignment="Center">  
  
<Button Name="abrirExtratoButton"  
Content="Abrir Extrato"  
VerticalAlignment="Center"  
Width="100" />  
<Button Name="abrirLogButton"  
Content="Abrir Log"  
VerticalAlignment="Center"  
Width="100" />  
  
</StackPanel>  
</Grid>  
  
<Grid Grid.Row="1">  
<TextBox Name="arquivoTextBox"  
Height="350"  
TextWrapping="Wrap"  
AcceptsReturn="True"  
VerticalScrollBarVisibility="Auto"  
FontFamily="Courier New"  
FontSize="14"/>  
  
</Grid>  
</TabItem>
```

O resultado visual é mostrado a seguir:



10. Vamos escrever o código para o evento do botão **abrirExtratoButton**. Neste evento, usaremos a classe **OpenFileDialog** do namespace **Microsoft.Win32**. Essa classe permite o acesso a uma caixa de diálogo, para selecionar o arquivo a ser aberto. Deixaremos configurado o tipo de arquivo como sendo **\*.txt**, e a pasta onde os nossos extratos estão sendo armazenados;

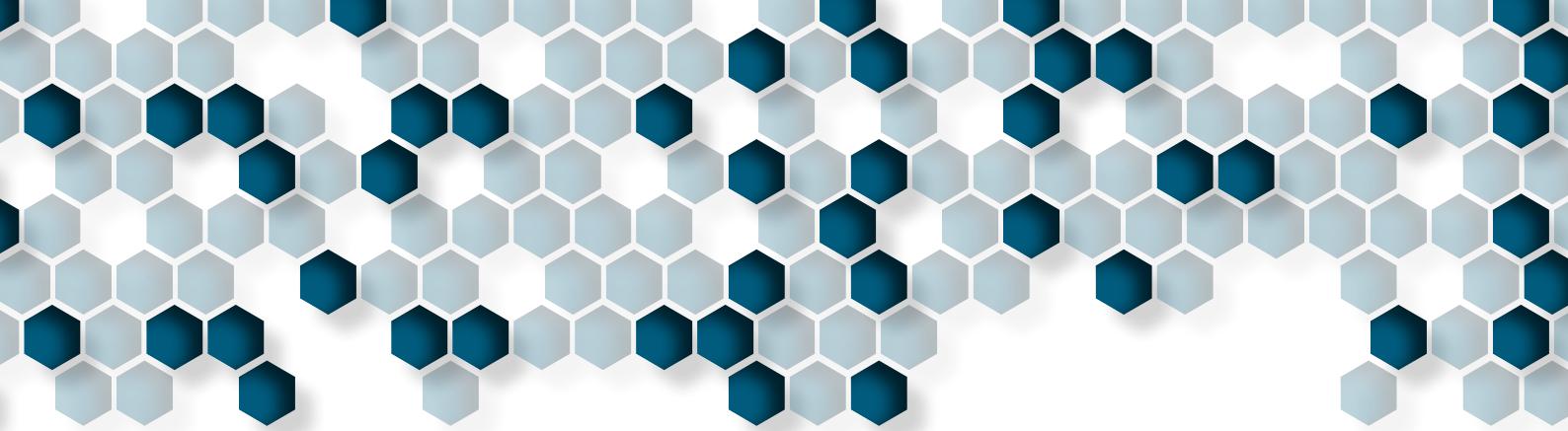
```
private void abrirExtratoButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        OpenFileDialog dialog = new OpenFileDialog();
        dialog.Filter = "Arquivos txt (*.txt)|*.txt";
        dialog.InitialDirectory =
            @"D:\Documentos\Projetos\Impacta\CSharp\Capitulo12.Labs\
            Extrato";

        if (dialog.ShowDialog() == true)
        {
            arquivoTextBox.Text = File.ReadAllText(dialog.
File Name);
        }
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message,
            "Erro",
            MessageBoxButton.OK,
            MessageBoxImage.Error);
    }
}
```

11. Para o botão **abrirLogButton**, vamos abrir o arquivo de log chamado **Erros.log**. Nesse evento, não disponibilizaremos a caixa de diálogos, pois o log é sempre armazenado no mesmo arquivo. Veja o código:

```
private void abrirLogButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        string arquivo =
            @"D:\Documentos\Projetos\Impacta\CSharp\Capitulo12.
Labs\Log\Erros.log";
        arquivoTextBox.Text = File.ReadAllText(arquivo);
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message,
            "Erro",
            MessageBoxButton.OK,
            MessageBoxImage.Error);
    }
}
```

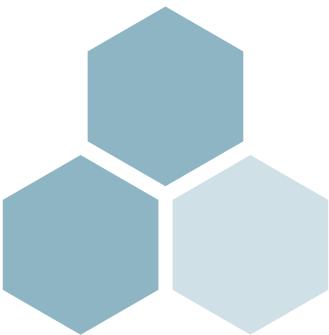
12. Teste a aplicação, de forma a contemplar todos os recursos adicionados neste laboratório.



# 13

## Acesso a dados com ADO.NET e Dapper

- ◆ Providers;
- ◆ Principais classes de acesso a dados;
- ◆ Definindo uma conexão com o SQL Server;
- ◆ Manipulando dados com Dapper;
- ◆ Executando instruções no banco de dados com Dapper;
- ◆ Trabalhando com ViewModel;
- ◆ Dapper Contrib.



## 13.1. Introdução

O .NET Framework oferece diversas maneiras de conectar, consultar e manipular informações de banco de dados. Todas elas usando, na sua estrutura básica, as classes do ADO.NET.

A arquitetura básica do ADO.NET é composta de **providers**, que são classes que fornecem acesso a fontes de dados, e de **consumers**, que são classes que utilizam as informações fornecidas pelos providers.

Uma fonte de dados pode ser um banco de dados SQL Server, Oracle, MySQL, arquivos XML, arquivos CSV, planilhas do Excel ou qualquer tipo de informação organizada.

## 13.2. Providers

O .NET Framework oferece quatro providers dentro da biblioteca de classes padrão:

Provider	Namespace	Destino
SQL Server	System.Data.SqlClient	Banco de dados SQL Server.
Oracle	System.Data.OracleClient	Banco de dados Oracle.
OLE DB	System.Data.OleDbClient	Qualquer banco com driver para o componente OLE DB.
ODBC	System.Data.ODBC	Qualquer banco com driver para o componente ODBC.

## 13.3. Principais classes de acesso a dados

Todos os providers implementam classes que herdam as classes ou interfaces do namespace **System.Data.Common**. Isso significa que a classe utilizada de qualquer provider terá as mesmas propriedades, métodos e eventos básicos de todos os outros. Não há diferença na programação de um provider ou de outro.

Classe base	Finalidade
DbConnection	Conexão com o banco de dados.
DbCommand	Execução de comandos.
DbDataReader	Leitura de dados.
DbDataAdapter	Transferência e sincronização de dados.

O .NET Provider para SQL Server, no namespace **System.Data.SqlClient**, fornece as seguintes classes derivadas das classes bases:

Classe	Classe base
<b>SqlConnection</b>	DbConnection
<b>SqlCommand</b>	DbCommand
<b>SqlDataReader</b>	DbDataReader
<b>SqlDataAdapter</b>	DbDataAdapter

O .NET Provider para OLE DB, no namespace **System.Data.SqlClient**, fornece as seguintes classes derivadas das classes base:

Classe	Classe base
<b>OleDbConnection</b>	DbConnection
<b>OleDbCommand</b>	DbCommand
<b>OleDbDataReader</b>	DbDataReader
<b>OleDbDataAdapter</b>	DbDataAdapter

O mesmo processo serve para qualquer outra classe de dados, mesmo que não faça parte do .NET Framework. A classe para conexão com MySQL, por exemplo, pode ser obtida pelo portal do MySQL. A implementação é exatamente a mesma: **MySqlConnection**, **MySqlCommand**, **MySqlDataReader** e **MySqlDataAdapter**.

## 13.4. Definindo uma conexão com SQL Server

A classe **SqlConnection** contém uma propriedade chamada **ConnectionString** para definir a conexão com o banco SQL Server. Uma vez criada a conexão, é possível abrir e fechar a conexão com os métodos **Open()** e **Close()**, respectivamente.

Vejamos, a seguir, um exemplo no qual é estabelecida uma conexão com um banco de dados do tipo SQL Server:

```
//String de conexão
string conexao = @"Data Source=localhost\sqlexpress;
    Initial Catalog=DBProdutos;
    Integrated Security=true";

//Conexão
var cn = new SqlConnection();
cn.ConnectionString = conexao;

//Conectar
cn.Open();

//Fechar a conexão
cn.Close();
```

## 13.5. Manipulação de dados com Dapper

Quanto mais frameworks de terceiros usamos, mais recursos consumimos do sistema. Isso porque, essencialmente, todos os frameworks utilizam como base o ADO.NET. Um exemplo típico é o **Entity Framework**, famoso pela sua facilidade de uso e pelo encapsulamento de quase todas as funcionalidades. Porém, todo encapsulamento tem seu preço: em sistemas de grande porte, se não aplicarmos suas funcionalidades de forma correta, corremos o risco de tornar nosso sistema pouco escalável e, consequentemente, pouco eficiente.

A solução para esse problema é trabalhar com o ADO.NET puro, uma vez que temos controle total sobre as operações que desejamos executar na aplicação sobre o banco de dados. Por outro lado, acabamos escrevendo muito código até mesmo para as operações mais simples.

O **Dapper** surgiu como uma solução para esse problema. Usamos a conexão fornecida pelo próprio ADO.NET, porém com os métodos escritos de forma otimizada pelo componente.

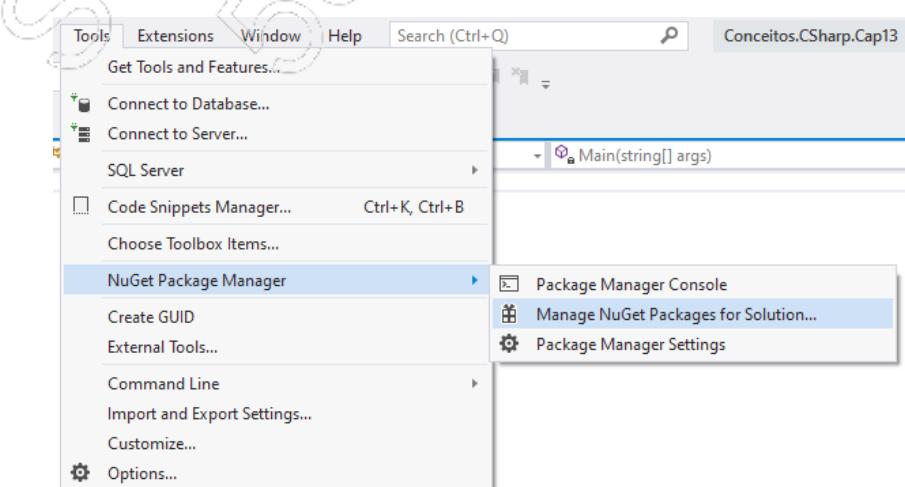
### 13.5.1. O que é o Dapper

O **Dapper** é um framework, também chamado de micro framework, que atua como um **Objeto de Mapeamento Relacional (ORM - Object Relational Mapping)**, cujo objetivo é mapear tabelas, colunas e campos do banco de dados em classes desenvolvidas no C#.

A utilização do **Dapper** é bastante simples. Primeiro, precisamos obter os recursos necessários, via download ou via **NuGet**. Usar o **NuGet** do Visual Studio é a forma mais eficiente para obtê-lo. O **NuGet** é um serviço do Visual Studio que permite obter recursos externos, baixá-los e configurá-los automaticamente nos nossos projetos.

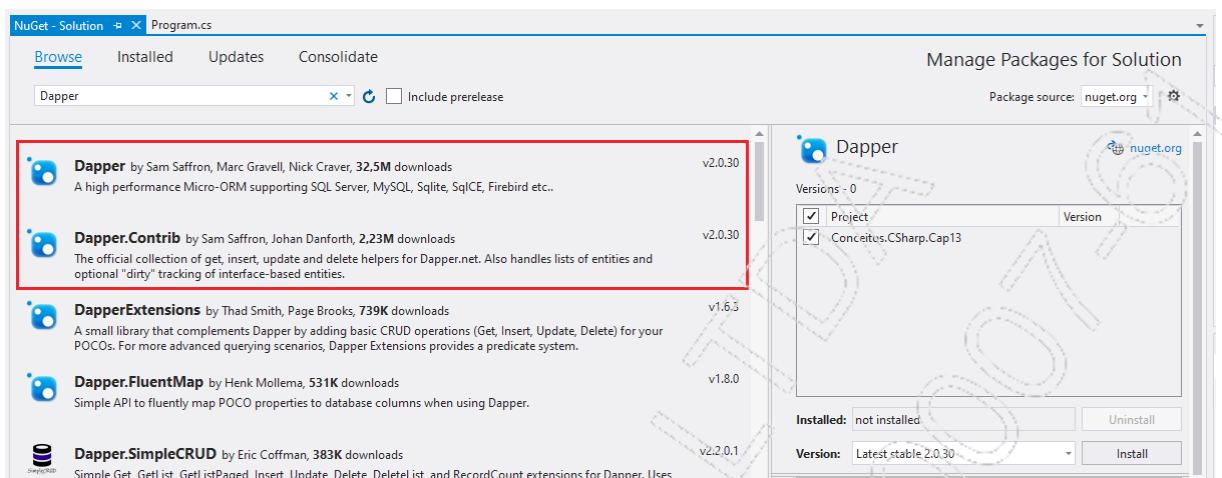
Considere um novo projeto chamado **Conceitos.CSharp.Cap13**. Vamos adicionar o **Dapper** a esse projeto via **NuGet**.

No menu **Tools**, acesse o item mostrado:



Na nova tela, selecione **Browse**. Digite **Dapper** e escolha as duas opções:

- **Dapper**
- **Dapper Contrib**



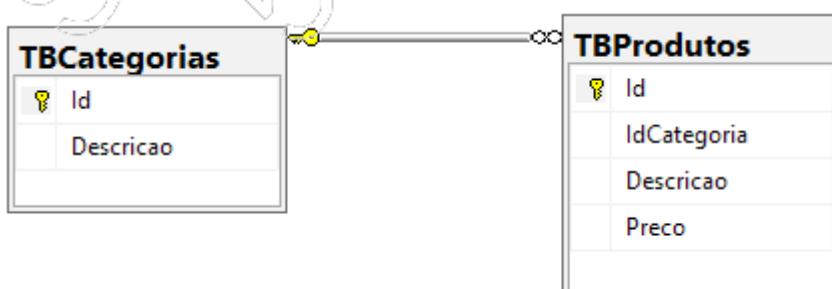
O **Dapper Contrib** é um recurso para facilitar tarefas relacionadas a **CRUD (Create, Recover, Update, Delete)** e não é obrigatório. O primeiro, **Dapper**, é essencial para nosso trabalho.

Uma vez adicionado, estamos prontos para trabalhar como **Dapper**.

## 13.5.2. Executando instruções no banco de dados com Dapper

Para apresentar os recursos de manipulação de dados com **Dapper**, consideraremos o banco de dados:

- Nome do banco de dados: **DbProdutos**
- Estrutura: Duas tabelas, **TBCategorias** e **TBProdutos**. O diagrama e o script para geração do banco são dados a seguir:



```
USE DbProdutos
GO

CREATE TABLE TBCategorias(
    Id INT IDENTITY(1,1),
    Descricao VARCHAR(20) NOT NULL,
    PRIMARY KEY(Id)
)
GO

CREATE TABLE TBProdutos (
    Id INT IDENTITY(1,1),
    IdCategoria INT NOT NULL,
    Descricao VARCHAR(50) NOT NULL,
    Preco FLOAT NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY(IdCategoria) REFERENCES TBCategorias(Id)
)
GO
```

### 13.5.3. Incluindo registros

Vamos apresentar um exemplo onde incluímos registros na tabela **TBCategorias**. A descrição da categoria é informada no programa:

```
namespace Conceitos.CSharp.Cap13
{
    class Program
    {
        static void Main(string[] args)
        {
            IncluirCategoria();
        }

        static string conexao = @"Integrated Security=SSPI;Persist
Security
Info=False;Initial Catalog=DbProdutos;Data Source=.\SQLEXPRESS";

        static void IncluirCategoria()
        {
            Console.WriteLine("Informe a categoria: ");
            string categoria = Console.ReadLine();

            using (var conn = new SqlConnection(conexao))
            {
                var registros = conn.Execute("INSERT INTO TBCategorias
(Descricao) VALUES (@Descricao)",
                    new { Descricao = categoria });

                Console.WriteLine("Registros inseridos: " + registros);
            }
        }
    }
}
```

O método **Execute** é um método de extensão disponibilizado pelo **Dapper**. Esse método estende a interface **IDbConnection**. Ele retorna o número de registros afetados (no caso do nosso exemplo, o número de registros incluídos).

### 13.5.4. Listando registros

Uma vez adicionadas algumas categorias, vamos listá-las. Para tanto, criaremos uma classe chamada **Categoria**, contendo propriedades com os mesmos nomes dos campos da tabela **TBCategorias**:

- Classe **Categoria**

```
namespace Conceitos.CSharp.Cap13
{
    public class Categoria
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
    }
}
```

- Método **ListarCategorias**

```
namespace Conceitos.CSharp.Cap13
{
    class Program
    {
        static void Main(string[] args)
        {
            ListarCategorias();
            Console.ReadKey();
        }

        static string conexao = @"Integrated Security=SSPI;Persist
Security
Info=False;Initial Catalog=DbProdutos;Data Source=.\\
SQLEXPRESS";
    }
}
```

```
static void ListarCategorias()
{
    using (var conn = new SqlConnection(conexao))
    {
        var categorias = conn.Query<Categoria>("SELECT * FROM
TBCategorias");
        foreach (var item in categorias)
        {
            Console.WriteLine("Id: " + item.Id);
            Console.WriteLine("Descrição: " + item.Descricao);
            Console.WriteLine("-----");
        }
    }
}
```



A variável `categorias` armazena o resultado da consulta, no formato `IEnumerable<Categoria>`. Após a execução, temos o resultado (este pode variar, dependendo das categorias que você adicionou):

```
D:\Documentos\Projetos\Impacta\Conceit... - □ X  
Id: 1  
Descrição: Informática  
-----  
Id: 2  
Descrição: Alimentação  
-----
```

### 13.5.5. Alterando Registros

Existem diversos recursos que podemos considerar ao trabalharmos com acesso a dados, além de incluir e listar. O método adiante permite realizar uma alteração de uma categoria, com base no seu código (este, por sua vez, pode ser informado de diversas formas):

```
static void AlterarCategoria()
{
    Console.Write("Informe o código da categoria: ");
    int codigo = int.Parse(Console.ReadLine());

    Console.Write("Informe a nova categoria: ");
    string categoria = Console.ReadLine();

    using (var conn = new SqlConnection(conexao))
    {
        var registros = conn.Execute("UPDATE TBCategorias SET " +
            "Descricao=@Descricao WHERE Id=@Id",
            new { Id= codigo, Descricao = categoria });

        Console.WriteLine("Registros alterados: " + registros);
    }
}
```

### 13.5.6. Removendo Registros

Analogamente, podemos remover registros com base em diversos critérios. O método de exemplo a seguir remove uma categoria com base no **Id**:

```
static void RemoverCategoria()
{
    Console.Write("Informe o código da categoria: ");
    int codigo = int.Parse(Console.ReadLine());

    using (var conn = new SqlConnection(conexao))
    {
        var registros = conn.Execute("DELETE FROM TBCategorias
            WHERE Id=@Id",
            new { Id = codigo });

        Console.WriteLine("Registros removidos: " + registros);
    }
}
```

### 13.6. Trabalhando com ViewModel

Considerando nosso modelo de banco de dados, temos duas tabelas. Acontece que é uma tarefa comum realizarmos consultas envolvendo queries que contenham junções, agrupamentos, somatórios etc. Em casos como esse, é conveniente criarmos classes com propriedades adequadas para armazenar o resultado de tais consultas. Essas classes são conhecidas como **ViewModels**. Este termo é usado nos cenários de consulta, e não de inclusão ou alteração.

Para ilustrar, vamos considerar os dados adiante no banco de dados:

The screenshot shows the SQL Server Management Studio interface with two tables displayed:

	Id	Descricao
1	1	Informática
2	2	Alimentação

	Id	IdCategoria	Descricao	Preco
1	1	1	Mouse	80
2	2	1	Pen Drive	120
3	3	2	Cup Cake	10
4	4	2	Chocolate Amargo	14,99
5	5	2	Hamburger Vegano	25

Consideremos, agora, as consultas:

```
use DbProdutos
go

select
    C.Descricao as Categoria,
    P.Descricao as Produto
from
    TBCategorias C, TBProdutos P
Where
    C.Id = P.IdCategoria
```

O resultado dessa consulta no banco de dados produz o resultado:

	Categoria	Produto
1	Informática	Mouse
2	Informática	Pen Drive
3	Alimentação	Cup Cake
4	Alimentação	Chocolate Amargo
5	Alimentação	Hamburger Vegano

Uma consulta através do Dapper será possível por meio da criação de uma classe, por exemplo, chamada **CategoriaProdutoVM**, com as propriedades:

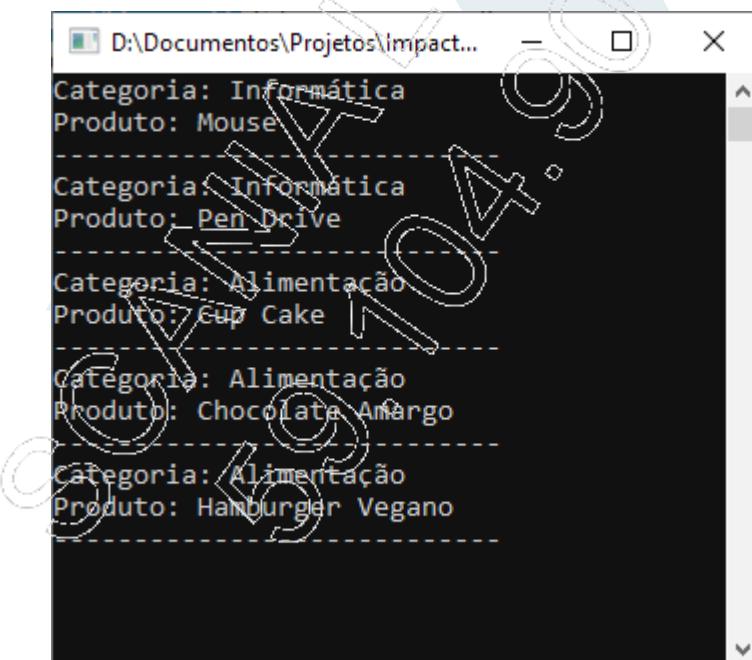
```
namespace Conceitos.CSharp.Cap13
{
    public class CategoriaProdutoVM
    {
        public string Categoria { get; set; }
        public string Produto { get; set; }
    }
}
```

O método **ListarProdutosVM** apresenta o resultado proveniente do banco de dados:

```
static void ListarProdutosVM()
{
    string sql = @"select
        C.Descricao as Categoria,
        P.Descricao as Produto
        from TBCategorias C, TBProdutos P
        Where C.Id = P.IdCategoria";

    using (var conn = new SqlConnection(conexao))
    {
        var produtos = conn.Query<CategoriaProdutoVM>(sql);
        foreach (var item in produtos)
        {
            Console.WriteLine("Categoria: " + item.Categoria);
            Console.WriteLine("Produto: " + item.Produto);
            Console.WriteLine("-----");
        }
    }
}
```

O resultado após a chamada a esse método é:



```
D:\Documentos\Projetos\Impact...
Categoria: Informática
Produto: Mouse
-----
Categoria: Informática
Produto: Pen Drive
-----
Categoria: Alimentação
Produto: Cup Cake
-----
Categoria: Alimentação
Produto: Chocolate Amargo
-----
Categoria: Alimentação
Produto: Hamburger Vegano
```

### 13.7. Dapper Contrib

O **Dapper Contrib** é uma parte do **Dapper** que fornece algumas simplificações nas tarefas mais comuns, como incluir, remover, alterar e buscar registros de uma tabela, devidamente mapeados para uma classe.

O **Dapper Contrib** estende a interface **IDbConnection** com os seguintes métodos:

- **Get**
- **GetAll**
- **Insert**
- **Update**
- **Delete**
- **DeleteAll**

Vamos apresentar exemplos desses métodos, manipulando um produto no banco de dados. Teremos uma classe chamada **Produto** e uma tabela chamada **TBProdutos**:

```
namespace Conceitos.CSharp.Cap13
{
    public class Produto
    {
        public int Id { get; set; }
        public int IdCategoria { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
    }
}
```

Agora, consideremos o método:

```
static void TestarDapperContrib()
{
    using (var conn = new SqlConnection(conexao))
    {
        //retorna o produto com Id = 1
        var produto = conn.Get<Produto>(1);
        Console.WriteLine($"Descrição: {produto.Descricao}\n" +
            $"Preço: {produto.Preco}");
    }
}
```

A execução desse método produz uma exceção:

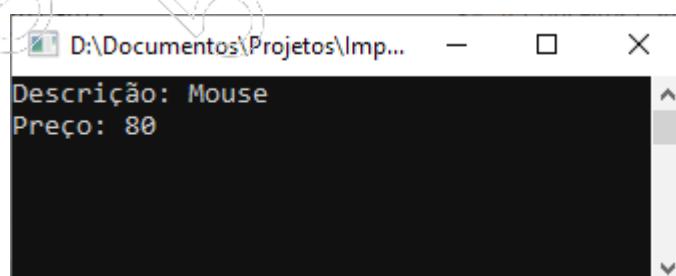
```
static void TestarDapperContrib()
{
    using (var conn = new SqlConnection(conexao))
    {
        //retorna o produto com Id = 1
        var produto = conn.Get<Produto>(1); ✖
        Console.WriteLine($"Descrição: {produto.Descricao}\n" +
            $"Preço: {produto.Preco}");
    }
}
```

Isso ocorre porque o **Dapper Contrib**, por padrão, considera o nome da tabela como sendo o nome da classe, no plural. Sendo assim, precisamos informar o nome correto da tabela que deve ser mapeada para a classe **Produto**.

Uma das formas de se fazer isso é adicionando um atributo **Table** na classe:

```
namespace Conceitos.CSharp.Cap13
{
    [Table("TBProdutos")]
    public class Produto
    {
        public int Id { get; set; }
        public int IdCategoria { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
    }
}
```

Essa alteração já é suficiente para produzir o resultado esperado no código apresentado anteriormente:



# Programando com C#

A outra forma de configurarmos a entidade com base no banco de dados é omitirmos o atributo **Table** e adicionarmos a instrução, no início do programa:

```
static void Main(string[] args)
{
    SqlMapperExtensions.TableNameMapper = entidade =>
    {
        if (entidade == typeof(Produto))
        {
            return "TBProdutos";
        }
        throw new Exception($"Entidade não suportada: {entidade}");
    };

    TestarDapperContrib();

    Console.ReadKey();
}
```

A vantagem dessa instrução é que a classe não fica cheia de atributos e, além disso, através do código, nós conseguimos alterá-lo dinamicamente.

A seguir, apresentaremos exemplos de instruções com os demais métodos:

- **GetAll**: Busca todos os registros com base na entidade informada;

```
//busca todos os produtos e os armazena na variável lista
var lista = conn.GetAll<Produto>();
```

- **Insert**: Insere um novo registro, com as informações da entidade fornecida na parametrização;

```
//inclui um novo produto. Observe que o Id não
//foi informado, por ser uma chave primária
var novoProduto = new Produto
{
    IdCategoria = 1,
    Descricao = "Laptop",
    Preco = 2000
};

var novo = conn.Insert<Produto>(novoProduto);
```

- **Update:** Atualiza um registro com as informações da entidade fornecida. Neste caso, a chave primária é necessária, pois é com base nela que os dados são alterados;

```
//altera os dados de um produto, como base no Id devidamente informado
var produtoAlterado = new Produto
{
    Id = 6,
    IdCategoria = 1,
    Descricao = "Laptop Novo",
    Preco = 3000
};

var alterado = conn.Update<Produto>(produtoAlterado);
```

- **Delete:** Remove um registro com a chave primária fornecida. Neste caso, apenas a chave primária é suficiente;

```
//remove o produto com base no Id (chave primária)
var removido = conn.Delete<Produto>(new Produto { Id = 6 });
```

- **DeleteAll:** Remove todos os registros mapeados para a entidade fornecida. É necessário cautela na utilização deste método.

```
//remove todos os registros - CUIDADO
var removeTodos = conn.DeleteAll<Produto>();
```

### Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- O ADO.NET é o nome dado ao conjunto de classes do .NET Framework usadas para manipular dados;
- As classes principais conectadas são **DbConnection**, **DbCommand**, **DbDataReader** e **DbDataAdapter**;
- As ações realizadas consistem em estabelecer conexão com o banco de dados, enviar comandos ao banco de dados e manipular os resultados obtidos;
- O **Dapper** é um micro serviço leve e rápido para acesso a dados;
- A melhor maneira de obter o **Dapper** para utilização no projeto é através do **NuGet**;
- Com o **Dapper**, podemos incluir, alterar, remover e buscar dados no banco de dados, sempre mapeando as informações para classes previamente definidas;
- Quando for necessário buscar dados que sejam formados pela composição de duas ou mais tabelas, criamos uma classe para acomodar estes novos dados. Essa classe é conhecida como **ViewModel**;
- Para tarefas envolvendo **CRUD**, podemos usar uma extensão do **Dapper** chamada **Dapper Contrib**.



13

# Acesso a dados com ADO.NET e Dapper

Teste seus conhecimentos



**1. Em que namespace são encontradas as classes de acesso a dados para o banco de dados SQL Server?**

- a) System.Data
- b) System.Common
- c) System.Data.SqlClient
- d) System.Data.SqlServer
- e) Todas as alternativas anteriores estão corretas.

**2. Em ADO.NET, o que significa Provider?**

- a) São classes que servem de base para outras classes de acesso a dados.
- b) São classes que acessam dados de um determinado banco ou usando determinado componente.
- c) São modelos de projetos em camadas.
- d) São classes que servem para conectar servidores de dados como SQL Server e Oracle usando código nativo .NET.
- e) É um conjunto de padrões de desenvolvimento para acesso a dados.

**3. O que é o Dapper?**

- a) Um micro framework que atua como um ORM.
- b) Um micro framework que atua como um Provider.
- c) Um micro framework que atua como uma string de conexão.
- d) Um micro framework usado no SQL Server.
- e) Um micro framework usado para integrar bancos de dados.

**4. Para incluir um novo registro no banco de dados, qual método do Dapper podemos usar?**

- a) Insert
- b) Include
- c) Execute
- d) Query
- e) Update

**5. Para obtermos um conjunto de registros originados de uma operação de join no banco de dados, a classe usada para mapear os resultados obtidos é conhecida como:**

- a) NewModel
- b) ViewModel
- c) View
- d) DataSet
- e) ModelText





13

# Acesso a dados com ADO.NET e Dapper



Mãos à obra!



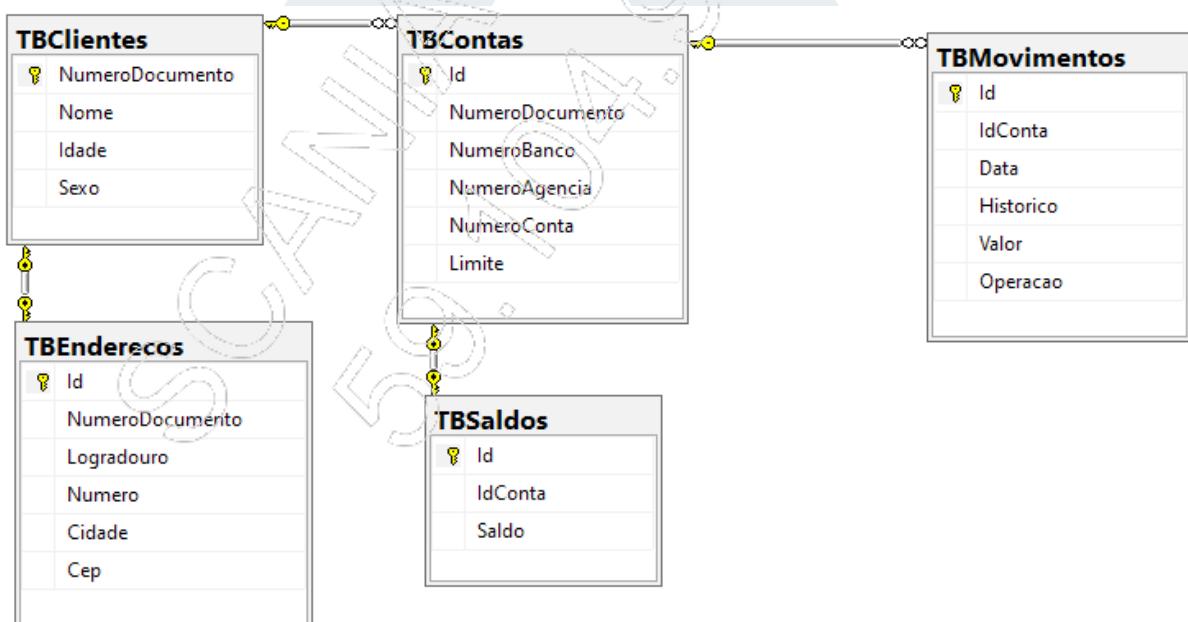
## Laboratório 1

### A – Manipulando dados e utilizando o Dapper

#### Objetivos:

- Criar um banco de dados para persistir os dados dos clientes, contas e movimentos;
- Implementar métodos capazes de manipular os dados, no sentido de inserir novos registros e realizar consultas. Será usado o Dapper como componente de mapeamento dos dados;
- Atualizar a interface gráfica de modo a transferir todo o controle das informações para o banco de dados.

1. Crie um novo solution, vazio, chamado **Capítulo13.Labs**;
2. Copie os projetos **Lab.Utilitarios**, **Lab.Models**, **Lab.Views**, **Lab.Interfaces**, **Lab.Arquivos** e **Lab.Dados** do Capítulo 12, na pasta do solution **Capítulo13.Labs**;
3. Adicione esses projetos ao novo solution;
4. Nossa primeira tarefa efetiva neste laboratório será criar o banco de dados a ser usado no projeto. No SQL Server, crie um banco de dados chamado **DBAplicacaoBancaria**;
5. Nesse banco de dados, adicione as tabelas indicadas a seguir:



O script para criação das tabelas é dado a seguir (fique atento às restrições criadas):

```
USE DBAplicacaoBancaria
GO

--tabela TBClientes
CREATE TABLE TBClientes
(
    NumeroDocumento VARCHAR(14) NOT NULL,
    Nome VARCHAR(50) NOT NULL,
    Idade INT NOT NULL,
    Sexo VARCHAR(1) NOT NULL,
    PRIMARY KEY(NumeroDocumento),
    CONSTRAINT CT_SEXO CHECK(Sexo IN ('M', 'F'))
)
GO

--tabela TBEndereco
CREATE TABLE TBEnderecos
(
    Id INT PRIMARY KEY IDENTITY(1,1),
    NumeroDocumento VARCHAR(14) NOT NULL,
    Logradouro VARCHAR(50) NOT NULL,
    Numero SMALLINT NOT NULL,
    Cidade VARCHAR(40) NOT NULL,
    Cep VARCHAR(8) NOT NULL,
    FOREIGN KEY (NumeroDocumento) REFERENCES TBClientes(NumeroDocumento),
    UNIQUE (NumeroDocumento)
)
GO

--tabela TBContas
CREATE TABLE TBContas
(
    Id INT PRIMARY KEY IDENTITY(1,1),
    NumeroDocumento VARCHAR(14) NOT NULL,
    NumeroBanco INT NOT NULL,
    NumeroAgencia VARCHAR(4) NOT NULL,
    NumeroConta VARCHAR(20) NOT NULL,
    Limite FLOAT NOT NULL,
    FOREIGN KEY (NumeroDocumento)
        REFERENCES TBClientes(NumeroDocumento)
)
GO

--tabela TBSaldos
CREATE TABLE TBSaldos
(
    Id INT PRIMARY KEY IDENTITY(1,1),
    IdConta INT NOT NULL,
    Saldo FLOAT NOT NULL,
    FOREIGN KEY (IdConta) REFERENCES TBContas(Id),
    UNIQUE (IdConta)
)
GO
```

```
--tabela TBMovimentos
CREATE TABLE TBMovimentos
(
    Id INT PRIMARY KEY IDENTITY(1,1),
    IdConta INT NOT NULL,
    Data DATETIME NOT NULL,
    Historico VARCHAR(20) NOT NULL,
    Valor FLOAT NOT NULL,
    Operacao INT NOT NULL,
    FOREIGN KEY (IdConta) REFERENCES TBContas(Id),
    CONSTRAINT CT_OPERACAO CHECK(Operacao IN (1,2))
)
GO
```

6. No projeto **Lab.Models**, vamos incluir uma propriedade chamada **Id** nas seguintes classes: **Conta** e **Movimento**. Faça o mesmo na estrutura **Endereco**. O objetivo é viabilizar o mapeamento das classes com o banco de dados através do **Dapper**. Para um cliente, a chave primária é o seu documento (CPF ou CNPJ):

- **Classe Conta**

```
namespace Lab.Models
{
    public abstract class Conta
    {
        public int Id { get; set; }
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }

        public List<Movimento> Movimentos { get; set; }

        public Cliente ClienteInfo { get; set; }

        public Conta(int Banco, string Agencia, string Conta)
        {
            if(this.Movimentos == null)
            {
                this.Movimentos = new List<Movimento>();
            }
            this.NumeroBanco = Banco;
            this.NumeroAgencia = Agencia;
            this.NumeroConta = Conta;
        }
    }
}
```

```
protected async Task RegistrarMovimento(double valor, Operacoes operacao)
{
    if (valor <= 0)
    {
        throw new ArgumentException("O valor deve ser positivo");
    }

    this.Movimentos.Add(new Movimento()
    {
        Data = DateTime.Now,
        Historico = operacao == Operacoes.Saque ? "SAQUE" :
        "DEPÓSITO",
        Operacao = operacao,
        Valor = valor
    });
}

public abstract string MostrarExtrato();

public abstract void EfetuarOperacao(double valor,
    Operacoes operacao = Operacoes.Deposito);

public override string ToString()
{
    return this.NumeroAgencia + "/" + this.NumeroConta;
}
}
```

- **Classe Movimento**

```
namespace Lab.Models
{
    public class Movimento
    {
        public int Id { get; set; }
        public DateTime Data { get; set; }
        public string Historico { get; set; }
        public Operacoes Operacao { get; set; }
        public double Valor { get; set; }

        public override string ToString()
        {
            return string.Format("{0:dd/MM/yyyy} {1,-10} {2,10:N2}",
                Data, Historico, Valor);
        }
    }
}
```

## ● Estrutura Endereco

```
namespace Lab.Models
{
    public struct Endereco
    {
        public int Id { get; set; }
        public string Logradouro { get; set; }
        public int Numero { get; set; }
        public string Cidade { get; set; }
        public string Cep { get; set; }

        public Endereco(int Id, string Logradouro, int Numero,
                        string Cidade, string Cep)
        {
            this.Id = Id;
            this.Logradouro = Logradouro;
            this.Numero = Numero;
            this.Cidade = Cidade;
            this.Cep = Cep;
        }

        public string Exibir()
        {
            return $"Logradouro: {this.Logradouro}\n" +
                $"Número: {this.Numero}\n" +
                $"Cidade: {this.Cidade}\n" +
                $"CEP: {this.Cep}";
        }
    }
}
```

Para utilizar o **Dapper**, necessitaremos da parametrização da classe tanto na inclusão como na consulta. O problema é que temos instâncias de **ClientePJ** e **ClientePF**. Por isso, ao realizar a consulta à lista de clientes, deveremos considerar que:

- O **Dapper** utilizar construtor padrão para gerar os objetos provenientes da consulta;
- Além do construtor padrão, não tem como gerar instâncias de classes abstratas. Por conta disso, deveremos:
  - Retirar o comando **abstract** da classe **Cliente**;
  - Adicionar o construtor padrão;
  - Adicionar a propriedade **NumeroDocumento** como **virtual**, e torná-la **override** nas subclasses.

Essas novas alterações são mostradas a seguir:

- Nova classe Cliente

```
namespace Lab.Models
{
    public class Cliente
    {
        public int Id { get; set; }

        public string Nome { get; set; }

        public virtual string NumeroDocumento { get; set; }

        private int _idade;
        public int Idade
        {
            get => _idade;
            set => _idade = (value >= 0 ? value :
                throw new InvalidOperationException("A idade não pode
ser negativa!"));
        }
        public Sexos Sexo { get; set; }
        public Endereco EnderecoResidencial { get; set; }

        public List<Conta> Contas { get; set; }

        public Cliente() { }

        public Cliente(string Nome, Sexos Sexo)
        {
            if(this.Contas == null)
            {
                this.Contas = new List<Conta>();
            }
            this.Nome = Nome;
            this.Sexo = Sexo;
        }

        public Cliente(string Nome, Sexos Sexo, int idade)
            : this(Nome, Sexo)
        {
            this.Idade = idade; //
        }

        public Cliente(string Nome, Sexos Sexo, Endereco endereco)
            : this(Nome, Sexo)
        {
            this.Idade = Idade; //
            this.EnderecoResidencial = endereco;
        }
    }
}
```

```
~Cliente()
{
    if(this.Contas != null)
    {
        this.Contas = null;
    }
}

public virtual string Exibir()
{
    return $"Nome: {this.Nome}\n" +
        $"Idade: {this.Idade}\n" +
        $"Sexo: {this.Sexo}\n" +
        $"ENDEREÇO DO CLIENTE:\n" +
        $"{this.EnderecoResidencial.Exibir()}";
}

public override string ToString()
{
    return this.Nome;
}
}
```

- **Nova classe ClientePF**

```
namespace Lab.Models
{
    public class ClientePF : Cliente, IDocumento
    {
        //propriedade implementada da interface
        private string _cpf;
        public override string NumeroDocumento
        {
            get => _cpf;
            set => _cpf = (value.ValidarCPF() ? value :
                throw new Exception("CPF Inválido"));
        }

        //método implementado da interface
        public string MostrarDocumento()
        {
            return $"CPF do cliente: {NumeroDocumento}";
        }
}
```

```
//construtores

public ClientePF(string Cpf, string Nome, Sexos Sexo)
    : base(Nome, Sexo)
{
    this.NumeroDocumento = Cpf;
}

public ClientePF(string Cpf, string Nome, Sexos Sexo,
    int Idade, Endereco endereco)
    : base(Nome, Sexo, Idade)
{
    this.NumeroDocumento = Cpf;
    base.EnderecoResidencial = endereco;
}

public override string Exibir()
{
    return $"{MostrarDocumento()}\n" +
        $"{base.Exibir()}";
}
}
```

- **Nova classe ClientePJ**

```
namespace Lab.Models
{
    public class ClientePJ : Cliente, IDocumento
    {
        //propriedade implementada da interface
        private string _cnpj;

        public override string NumeroDocumento
        {
            get => _cnpj;
            set => _cnpj = (value.Length == 14 ? value :
                throw new Exception("CNPJ Inválido"));
        }

        //método implementado da interface
        public string MostrarDocumento()
        {
            return $"CNPJ do cliente: {NumeroDocumento}";
        }
    }
}
```

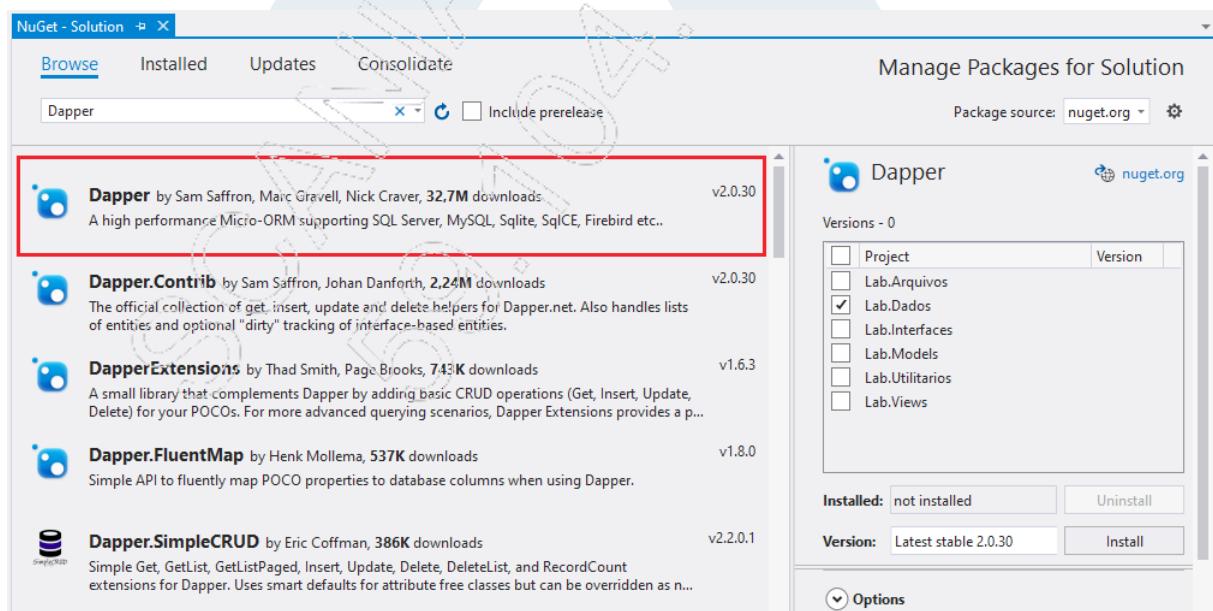
```
//construtores

public ClientePJ(string Cnpj, string Nome, Sexos Sexo)
    : base(Nome, Sexo)
{
    this.NumeroDocumento = Cnpj;
}

public ClientePJ(string Cnpj, string Nome, Sexos Sexo,
    int Idade, Endereco endereco)
    : base(Nome, Sexo, Idade)
{
    this.NumeroDocumento = Cnpj;
    base.EnderecoResidencial = endereco;
}

public override string Exibir()
{
    return $"{MostrarDocumento()}\n" +
        $"{base.Exibir()}";
}
}
```

7. No projeto **Lab.Dados**, adicione a referência ao **Dapper**. Use o **NuGet** para essa tarefa:



8. No projeto **Lab.Dados**, abra a classe **Dao** para adicionar a string de conexão e a classe **SqlConnection**, com seu construtor devidamente configurado:

```
namespace Lab.Dados
{
    public abstract class Dao<T, K> where T: class
    {
        protected string conexao = @"Integrated Security=SSPI;
                                         Persist Security Info=False;
                                         Initial Catalog=DBAplicacaoBancaria;
                                         Data Source=.\SQLEXPRESS";
        protected SqlConnection cn;

        public Dao()
        {
            cn = new SqlConnection(conexao);
        }

        public abstract int Incluir(T item);
        public abstract T Buscar(K chave);
        public abstract IEnumerable<T> Listar(K chave = default(K));
        public abstract int Remover(T item);
    }
}
```

9. No projeto **Lab.Dados**, abra a classe **ClientesDao**. Implemente os métodos dessa classe:

```
namespace Lab.Dados
{
    public class ClientesDao : Dao<Cliente, string>
    {
        public override Cliente Buscar(string chave)
        {
            Cliente cliente = null;
            try
            {
                cliente = cn.QueryFirst<Cliente>(
                    "SELECT NumeroDocumento, " +
                    "Nome, Idade, " +
                    "case Sexo" +
                    "    when 'F' then 1" +
                    "    when 'M' then 0 " +
                    "end as Sexo from TBClientes WHERE " +
                    "NumeroDocumento = @NumeroDocumento",
                    new { NumeroDocumento = chave });
            }
            catch
            {
                throw;
            }
            return cliente;
        }
}
```

```
public override int Incluir(Cliente item)
{
    var registros = 0;
    try
    {
        string documento;
        if(item is ClientePF)
        {
            documento = ((ClientePF)item).NumeroDocumento;
        }
        else
        {
            documento = ((ClientePJ)item).NumeroDocumento;
        }
        registros = cn.Execute("INSERT INTO TBClientes (" +
            "NumeroDocumento, Nome, Idade, Sexo) " +
            "VALUES (@NúmeroDocumento, @Nome, @Idade, @Sexo)",
            new {
                NumeroDocumento = documento,
                item.Nome,
                item.Idade,
                Sexo = (item.Sexo == Sexos.Masculino ? 'M' : 'F')
            });

        var endereco = cn.Execute("INSERT INTO TBEnderecos (" +
            "NúmeroDocumento, Logradouro, Numero, Cidade, Cep) " +
            "VALUES (@NúmeroDocumento, @Logradouro, @Número, @
Cidade, @Cep)",
            new {
                NúmeroDocumento = documento,
                item.EnderecoResidencial.Logradouro,
                item.EnderecoResidencial.Numero,
                item.EnderecoResidencial.Cidade,
                item.EnderecoResidencial.Cep
            });
    }
    catch
    {
        throw;
    }
    return registros;
}

public override IEnumerable<Cliente> Listar(string chave = null)
{
    List<Cliente> clientes = new List<Cliente>();

    try
    {
        var lista = cn.Query<Cliente>(
            "SELECT NumeroDocumento, " +
            "Nome, Idade," +
            "case Sexo" +
            "    when 'F' then 1" +
            "    when 'M' then 0 " +
            "end as Sexo from TBClientes");
    }
}
```

```
//separando os clientes por PF e PJ
foreach (var item in lista)
{
    var endereco = BuscarEndereco(item.NumeroDocumento);
    if(item.NumeroDocumento.Length == 11)
    {
        clientes.Add(new ClientePF(item.NumeroDocumento,
            item.Nome, item.Sexo, item.Idade, endereco));
    }
    else
    {
        clientes.Add(new ClientePJ(item.NumeroDocumento,
            item.Nome, item.Sexo, item.Idade, endereco));
    }
}
catch
{
    throw;
}
return clientes;
}

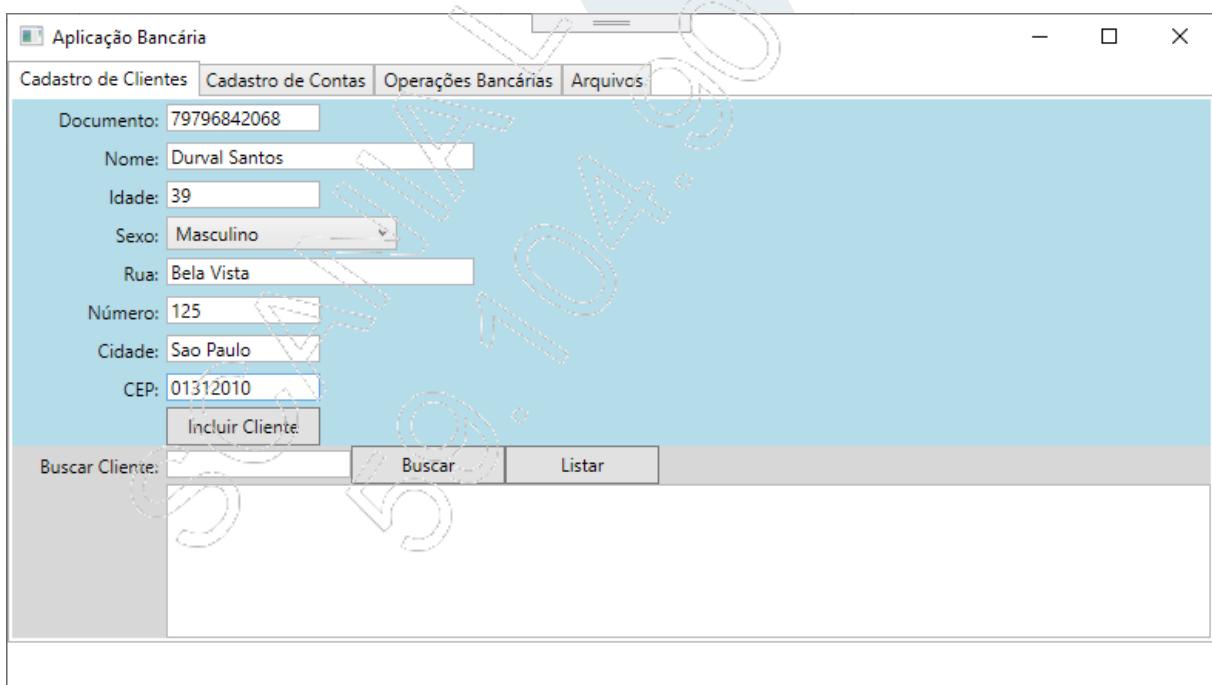
private Endereco BuscarEndereco(string documento)
{
    Endereco endereco;
    try
    {
        endereco = cn.QueryFirst<Endereco>(
            "SELECT * FROM TBEnderacos WHERE " +
            "NumeroDocumento = @NúmeroDocumento",
            new { NúmeroDocumento = documento });
    }
    catch
    {
        throw;
    }
    return endereco;
}

public override int Remover(Cliente item)
{
    string documento;
    var registros = 0;
    if (item is ClientePF)
    {
        documento = ((ClientePF)item).NúmeroDocumento;
    }
    else
    {
        documento = ((ClientePJ)item).NúmeroDocumento;
    }
    try
    {
```

```
registros = cn.ExecuteNonQuery("DELETE FROM TBClientes WHERE " +
    "NumeroDocumento = @NumeroDocumento", new
    {
        NumeroDocumento = documento
    });
}
catch
{
    throw;
}
return registros;
}
}
}
```

Observe que fizemos uma alteração na consulta aos clientes, no método **Listar**. Transformamos os valores do sexo, 'M' e 'F', em 0 e 1, respectivamente. Isso foi necessário por causa do tipo enumerado, que constitui o sexo:

10. Na aplicação (projeto **Lab.Views**), altere o código do evento **IncluirClienteButton** de forma a incluir o cliente no banco de dados, usando o método **Incluir** da classe **ClientesDao**. Lembre-se de referenciar o projeto **Lab.Dados**! Além disso, a listagem dos clientes será feita em dois lugares: no construtor da classe **MainWindow**, e no momento da inclusão:



```
private void incluirClienteButton_Click(object sender,
    RoutedEventArgs e)
{
    try
    {
        //obtendo os dados do endereço
        Endereco endereco = new Endereco(
            0,
            ruaTextBox.Text,
            int.Parse(numeroTextBox.Text),
            cidadeTextBox.Text,
            cepTextBox.Text);

        //obtendo os dados do cliente
        int digitos = documentoTextBox.Text.Length;
        Cliente cliente;

        if (digitos == 11)
        {
            cliente = new ClientePF(
                documentoTextBox.Text,
                nomeTextBox.Text,
                (Sexos)sexoComboBox.SelectedItem,
                int.Parse(idadeTextBox.Text),
                endereco);
        }
        else if (digitos == 14)
        {
            cliente = new ClientePJ(
                documentoTextBox.Text,
                nomeTextBox.Text,
                (Sexos)sexoComboBox.SelectedItem,
                int.Parse(idadeTextBox.Text),
                endereco);
        }
        else
        {
            throw new Exception("Documento inválido");
        }

        //adicionando o novo cliente na lista
        //Metodos.AdicionarCliente(cliente);
        clientesDao.Incluir(cliente);

        //vinculando a lista de clientes ao
        //componente ComboBox
        //clienteComboBox.ItemsSource = Metodos.ListarClientes();
        clienteComboBox.ItemsSource = clientesDao.Listar();

        MessageBox.Show("Cliente incluído com sucesso!");
    }
}
```

11. O construtor da classe também deve listar os clientes, além de instanciar a classe **ClientesDao**:

```
ClientesDao clientesDao;

public MainWindow()
{
    InitializeComponent();

    sexoComboBox.Items.Add(Sexos.Masculino);
    sexoComboBox.Items.Add(Sexos.Feminino);
    sexoComboBox.SelectedIndex = 0;

    operacaoComboBox.Items.Add(Operacoes.Deposito);
    operacaoComboBox.Items.Add(Operacoes.Saque);
    operacaoComboBox.SelectedIndex = 0;

    clientesDao = new ClientesDao();
    clienteComboBox.ItemsSource = clientesDao.Listar();
}
```

12. Analogamente ao que foi feito com a classe **Cliente**, faremos algumas alterações na classe **Conta**:

- Para este projeto, ela deixará de ser abstrata;
- Adicionaremos um construtor padrão;
- Adicionaremos a propriedade **NumeroDocumento**, para que haja o mapeamento através do Dapper.

Posteriormente, na classe **ContasDao**, faremos os ajustes necessários para referenciar o cliente de cada conta:

- **Nova classe Conta**

```
namespace Lab.Models
{
    public class Conta
    {
        public int Id { get; set; }
        public int NumeroBanco { get; set; }
        public string NumeroAgencia { get; set; }
        public string NumeroConta { get; set; }

        public string NumeroDocumento { get; set; }

        public List<Movimento> Movimentos { get; set; }

        public Cliente ClienteInfo { get; set; }
    }
}
```

```
public Conta() { }

public Conta(int Banco, string Agencia, string Conta)
{
    if(this.Movimentos == null)
    {
        this.Movimentos = new List<Movimento>();
    }

    this.NumeroBanco = Banco;
    this.NumeroAgencia = Agencia;
    this.NumeroConta = Conta;
}

protected async Task RegistrarMovimento(double valor, Operacoes operacao)
{
    if (valor <= 0)
    {
        throw new
            ArgumentException("O valor deve ser positivo");
    }

    this.Movimentos.Add(new Movimento()
    {
        Data = DateTime.Now,
        Historico = operacao == Operacoes.Saque ? "SAQUE" :
        "DEPÓSITO",
        Operacao = operacao,
        Valor = valor
    });
}

public virtual string MostrarExtrato()
{
    throw new Exception("Este método deve ser implementado");
}

public virtual void EfetuarOperacao(double valor,
    Operacoes operacao = Operacoes.Deposito) { }

public override string ToString()
{
    return this.NumeroAgencia + "/" + this.NumeroConta;
}
}
```

# Programando com C#

13. Implemente os métodos **Incluir** e **Listar** da classe **ContasDao**:

```
namespace Lab.Dados
{
    public class ContasDao : Dao<Conta, string>
    {
        public override Conta Buscar(string chave)
        {
            throw new NotImplementedException();
        }

        public override int Incluir(Conta item)
        {
            var registros = 0;
            double limite = 0;

            if(item is ContaEspecial)
            {
                limite = ((ContaEspecial)item).Limite;
            }
            try
            {
                registros = cn.Execute("INSERT INTO TBContas (" +
                    "NumeroDocumento, NumeroBanco, NumeroAgencia, " +
                    "NumeroConta, Limite) " +
                    "VALUES (@NumeroDocumento, @NumeroBanco, " +
                    "@NumeroAgencia, @NumeroConta, @Limite)",
                    new
                    {
                        item.NumeroDocumento,
                        item.NumeroBanco,
                        item.NumeroAgencia,
                        item.NumeroConta,
                        Limite = limite
                    });
            }
            catch
            {
                throw;
            }
            return registros;
        }

        public override IEnumerable<Conta> Listar(string chave = null)
        {
            List<Conta> contas = new List<Conta>();

            try
            {
                var lista = cn.Query<ContaEspecial>(
                    "SELECT * from TBContas");
            }
            catch
            {
                throw;
            }
            return contas;
        }
    }
}
```

```

//separando os clientes por PF e PJ
foreach (var item in lista)
{
    if (item.Limite == 0) //Conta Comum
    {
        var cc = new ContaCorrente(item.NumeroBanco,
            item.NumeroAgencia, item.NumeroConta);
        cc.Id = item.Id;
        cc.NumeroDocumento = item.NumeroDocumento;

        contas.Add(cc);
    }
    else
    {
        contas.Add(item);
    }
}
catch
{
    throw;
}
return contas;
}

public override int Remover(Conta item)
{
    throw new NotImplementedException();
}
}
}

```

14. Defina uma variável do tipo **ContasDao** e atualize o construtor de forma a instanciar a classe **ContasDao** e a listar as contas, preparando a aplicação para novas operações;

```

ClientesDao clientesDao;
ContasDao contasDao;

public MainWindow()
{
    InitializeComponent();

    sexoComboBox.Items.Add(Sexos.Masculino);
    sexoComboBox.Items.Add(Sexos.Feminino);
    sexoComboBox.SelectedIndex = 0;

    operacaoComboBox.Items.Add(Operacoes.Deposito);
    operacaoComboBox.Items.Add(Operacoes.Saque);
    operacaoComboBox.SelectedIndex = 0;

    clientesDao = new ClientesDao();
contasDao = new ContasDao();

    clienteComboBox.ItemsSource = clientesDao.Listar();
numeroContaComboBox.ItemsSource = contasDao.Listar();
}

```

15. Atualize o evento do botão `incluirContaButton`:

```
private void incluirContaButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        Conta conta;
        var cliente = (Cliente)clienteComboBox.SelectedItem;

        int banco = int.Parse(bancoTextBox.Text);
        string agencia = agenciaTextBox.Text;
        string numConta = contaTextBox.Text;

        if (VerificarEspecial)
        {
            conta = new ContaEspecial(banco, agencia, numConta);
            ((ContaEspecial)conta).Limite =
                double.Parse(limiteTextBox.Text);
        }
        else
        {
            conta = new ContaCorrente(banco, agencia, numConta);
        }
        //vinculamos o cliente selecionado à nova conta
        conta.ClienteInfo = cliente;

        //adicionamos o número do documento
        conta.NumeroDocumento = cliente.NumeroDocumento;

        //adicionamos a nova conta à lista de contas do
        //cliente selecionado
        cliente.Contas.Add(conta);

        //incluímos a nova conta à lista global de contas
        //Metodos.AdicionarConta(conta);
        contasDao.Incluir(conta);

        //vinculamos a lista global de contas ao
        //componente ComboBox
        //numeroContaComboBox.ItemsSource = Metodos.ListarContas();
        numeroContaComboBox.ItemsSource = contasDao.Listar();

        MessageBox.Show("Conta adicionada com sucesso!");
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message,
                      "Erro",
                      MessageBoxButton.OK,
                      MessageBoxImage.Error);
    }
}
```

16. Inclua a propriedade **IdConta** na classe **Movimento**, para viabilizar o mapeamento com o **Dapper**:

```
namespace Lab.Models
{
    public class Movimento
    {
        public int Id { get; set; }
        public int IdConta { get; set; }
        public DateTime Data { get; set; }
        public string Historico { get; set; }
        public Operacoes Operacao { get; set; }
        public double Valor { get; set; }

        public override string ToString()
        {
            return string.Format("{0:dd/MM/yyyy} {1,-10} {2,10:N2}",
                Data, Historico, Valor);
        }
    }
}
```

17. No projeto **Lab.Models**, crie uma nova classe chamada **ValorSaldo**. Essa classe será usada para mapear o saldo da conta corrente, na ocasião do registro do movimento:

```
namespace Lab.Models
{
    public class ValorSaldo
    {
        public int Id { get; set; }
        public int IdConta { get; set; }
        public double Saldo { get; set; }
    }
}
```

18. No projeto **Lab.Dados**, crie a classe **MovimentosDao**, capaz de registrar um movimento e listar os movimentos, com base na conta selecionada. Essa classe deve ser subclasse de **Dao**. Nessa classe, implemente os métodos **Incluir** e **Listar**:

```
namespace Lab.Dados
{
    public class MovimentosDao : Dao<Movimento, int>
    {
        public override Movimento Buscar(int chave)
        {
            throw new NotImplementedException();
        }

        public override int Incluir(Movimento item)
        {
            var registros = 0;

            try
            {

```

# Programando com C#

```
//registra um novo movimento
registros = cn.Execute("INSERT INTO TBMovimentos (" +
    "IdConta, Data, Historico, Valor, Operacao) " +
    "VALUES (" +
        "@IdConta, @Data, @Historico, @Valor, @Operacao)",
    new
    {
        item.IdConta, item.Data, item.Historico,
        item.Valor, item.Operacao = item.Operacao
    });

//verifica se o movimento já foi realizado. Neste caso, a
//tabela TBSaldos já possui um registro.

var saldo = cn.QueryFirstOrDefault<ValorSaldo>("SELECT
* FROM " +
    "TBSaldos WHERE IdConta=@IdConta",
    new { item.IdConta });

if(saldo == null)
{
    //SE não existir, ela é criada com o valor do saldo
    cn.Execute("INSERT INTO TBSaldos (IdConta,Saldo) " +
        "VALUES (@IdConta,@Saldo)", new {
            item.IdConta, Saldo = item.Valor
        });
}
else
{
    //caso contrário, o saldo é atualizado
    var saldoAtual = (item.Operacao == Operacoes.Deposito)
?
    saldo.Saldo + item.Valor : saldo.Saldo - item.
    Valor;

    cn.Execute("UPDATE TBSaldos SET Saldo=@Saldo WHERE
        IdConta = @IdConta", new
    {
        item.IdConta,
        Saldo = saldoAtual
    });
}
catch
{
    throw;
}
return registros;
}
```

```
public override IEnumerable<Movimento> Listar(int chave = 0)
{
    IEnumerable<Movimento> movimentos = new List<Movimento>();
    try
    {
        movimentos = cn.Query<Movimento>(
```

```
        "SELECT * FROM TBMovimentos WHERE IdConta=@IdConta",
        new { IdConta = chave });
    }
    catch
    {
        throw;
    }
    return movimentos;
}

public override int Remover(Movimento item)
{
    throw new NotImplementedException();
}
}
```

19. No projeto **Lab.Views**, atualize o construtor, incluindo uma referência à classe **MovimentosDao**:

```
ClientesDao clientesDao;
ContasDao contasDao;
MovimentosDao movimentosDao;

public MainWindow()
{
    InitializeComponent();

    sexoComboBox.Items.Add(Sexos.Masculino);
    sexoComboBox.Items.Add(Sexos.Feminino);
    sexoComboBox.SelectedIndex = 0;

    operacaoComboBox.Items.Add(Operacoes.Deposito);
    operacaoComboBox.Items.Add(Operacoes.Saque);
    operacaoComboBox.SelectedIndex = 0;

    clientesDao = new ClientesDao();
    contasDao = new ContasDao();
    movimentosDao = new MovimentosDao();

    clienteComboBox.ItemsSource = clientesDao.Listar();
    numeroContaComboBox.ItemsSource = contasDao.Listar();

}
```

20. Para registrar o movimento, adicionamos este movimento no banco de dados e, em seguida, atualizamos a propriedade **Movimentos** da classe **Conta**. O extrato não sofre alterações:

```
private void executarButton_Click(object sender,
    RoutedEventArgs e)
{
    try
    {
        //obtendo a conta selecionada
        var conta = (Conta)numeroContaComboBox.SelectedItem;

        //obtendo a operação
        var operacao = (Operacoes)operacaoComboBox.SelectedItem;

        //obtendo o valor da operação
        double valor = double.Parse(valorTextBox.Text);

        //executar a operação
        conta.EfetuarOperacao(valor, operacao);

        var movimento = new Movimento()
        {
            IdConta = conta.Id,
            Data = DateTime.Now,
            Operacao = operacao,
            Valor = valor,
            Historico = operacao == Operacoes.Saque ?
                "SAQUE" : "DEPÓSITO"
        };

        movimentosDao.Incluir(movimento);

        MessageBox.Show("Operação realizada com sucesso!");
        valorTextBox.Clear();
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message,
            "Erro",
            MessageBoxButton.OK,
            MessageBoxIcon.Error);
    }
}
```

21. No evento do botão **extratoButton**, atualizamos a lista de movimentos:

```
private void extratoButton_Click(object sender,
    RoutedEventArgs e)
{
    try
    {
        var conta = (Conta)numeroContaComboBox.SelectedItem;

        //Buscar o estrato da conta selecionada
        conta.Movimentos = movimentosDao.Listar(conta.Id).
        ToList();

        conta.ClienteInfo = clientesDao.Buscar(conta.
        NumeroDocumento);

        extratoTextBox.Text = conta.MostrarExtrato();
    }
    catch (Exception ex)
    {
        AcessoArquivo.GerarLog(ex.Message);
        MessageBox.Show(ex.Message,
            "Erro",
            MessageBoxButton.OK,
            MessageBoxImage.Error);
    }
}
```

Como sugestão de continuidade, procure variar as consultas, implementar novos métodos e novas operações. Isso o ajudará a fixar melhor os conceitos estudados ao longo do treinamento.



## Referências bibliográficas

MONTE EVEREST PARTICIPAÇÕES E EMPREENDIMENTOS (Org.); CELSO DE SOUZA, Emilio. *Desenvolvimento Web com ASP.NET e Visual Studio 2017*. São Paulo: Monte Everest Participações e Empreendimentos Ltda., 2017. 1142 p.

