# Programação em C#

Aprendizagem + Tecnologias de Programação de Sistemas de Informação

## UC00607 - Desenvolver programas complexos em linguagem de programação estruturada

FORMADOR/A
Bruno Silva

DATA
Outubro de 2026

PESSOAS 2030
PORTUGAL 2030
Cofinanciado pela União Europeia
Os Fundos Europeus mais próximos de si.

# Índice

Capítulo 1 - Introdução ....4
1.1 - Lógica de programação ....4
1.2 - Algoritmos ....5
Capítulo 2 – Linguagens de Programação ....7
2.1 - Níveis da classificação das linguagens (alto e baixo nível) ....7
2.2 - Software de tradução, interpretador e compilador ....8
Capítulo 3 – Instalação Visual Studio ....9
3.1 – Preparação e requisitos mínimos ....9
3.2 – Descarregar e Instalar do IDE Visual Studio ....10
Capítulo 4 – Resolução de problemas pós-instalação ....17
Capítulo 5 – Como criar um projeto com a linguagem C# ....21
Capítulo 5.1 – Como criar um projeto C# ....21
Capítulo 5.2 – Possíveis erros na criação dos projetos ....23
Capítulo 5.3 – Ambiente de trabalho em editor C# ....27
Capítulo 6 – Criação de múltiplos projetos em Visual Studio ....35
Capítulo 7 – Introdução à programação C# ....37
Capítulo 7.1 – Instruções de escrita de valores na consola ....37
Capítulo 7.1.1 – Sequências de Escape ....37
Capítulo 7.2 – Tipos de Dados e as Variáveis ....38
Capítulo 7.2.1 – Constantes e Variáveis ....39
Capítulo 7.2.2 – Escrita de mensagens (com a inclusão das variáveis) ....40
Capítulo 7.3 – Instruções leitura de valores (a partir do teclado) ....43
Capítulo 8 – Estruturas de decisão ....44
Capítulo 8.1 – Estrutura de decisão simples ....45
Capítulo 8.2 – Estrutura de decisão dupla ....46
Capítulo 8.3 – Estrutura de decisão dupla encadeada (IF … Else if … Else) ....48
Capítulo 8.4 – Estrutura de decisão seletiva (Switch … Case) ....49
Capítulo 9 – Estruturas de repetição ....50
Capítulo 9.1 – Estruturas de Repetição FOR (contador automático) ....50
Capítulo 9.2 – Estruturas de Repetição While e Do … While (contador automático) ....52
Capítulo 9.2.1 – Estruturas de Repetição WHILE (contador manual) ....52
Capítulo 9.2.2 – Estruturas de Repetição DO …WHILE (contador manual) ....53
Capítulo 9.2.3 – Instruções Break e Continue ....55
Capítulo 10 – Subprogramas ....56
Capítulo 10.1 – Exemplo de código sem subprogramas vs com subprogramas ....56
Capítulo 10.2 – Vantagens da utilização dos subprogramas ....57
Capítulo 10.3 – Subprogramas: Procedimentos e Funções ....57

Capítulo 10.3.1 – Procedimentos ..... 58
Capítulo 10.3.2 – Funções ..... 59
Capítulo 10.3.3 – Subprogramas - procedimentos e funções (por parâmetros ou referência) ..... 60

Capítulo 11 – POO: Programação Orientada a Objetos ..... 63

11.1 - Programação estruturada (sequencial) ..... 63
11.2 - Programação Orientada a Objetos (POO) ..... 64
**11.3 – Conceitos de Programação Orientada a Objetos** ..... 65

11.3.1 - Encapsulamento de informação ..... 65
**11.3.2 - Composição / Herança** ..... 66
**11.3.3 - Polimorfismo** ..... 66

11.4 - Paradigmas de Programação: Objetos, Classes e Instâncias ..... 66
11.5 - Paradigmas de Programação ..... 67

Capítulo 12 – Classes ..... 68

Capítulo 12.1 – Construção de uma classe ..... 69
Capítulo 12.1.1 – Variáveis de instância ..... 72
Capítulo 12.1.2 – Construtores (sem e com parâmetros) ..... 72
Capítulo 12.1.3 – Métodos de instância: Get e Set ..... 73
Capítulo 12.1.4 – Métodos adicionais ..... 75
Capítulo 12.2 – Instância de Classes: Criação de novos objetos a partir de uma Classe ..... 77

Capítulo 13 – Interfaces ..... 80

13.1 – O que são as Interfaces? ..... 80
13.2 – Como criar uma interface? ..... 80

Capítulo 14 – Coleções (estruturas de dados) ..... 87

Capítulo 14.1 – Introdução as coleções ..... 87
Capítulo 14.2 – Principais coleções da Plataforma .NET ..... 88
Capítulo 14.2.1 – Filas de Espera - Queue<T> ..... 89
Capítulo 14.2.2 – Pilhas - Stack<T> ..... 91
Capítulo 14.2.3 – Listas – List<T> ..... 93
Capítulo 14.2.4 – Listas Ordenadas – SortedList<key, Value> ..... 100

Bibliografia ..... 107

# Capítulo 1 - Introdução

## 1.1 - Lógica de programação

Antes de começar a programar, temos de ser capazes de **estruturar o nosso raciocínio.** Quando é proposto um problema, temos de conseguir utilizar a **lógica** que atribuímos ao problema, de forma a encontrar uma **sequência de passos** que conduzam à sua resolução sem erros.



**Então o que é a Lógica**?

- Lógica de programação é a técnica de encadear pensamentos para atingir determinado objetivo. As pessoas utilizam a lógica no quotidiano sem dar conta.

**Então e o que é são as instruções em algoritmos**?

- ***Na linguagem comum***, entendemos por instruções como, um conjunto de normas definidas, para a realização de algo.

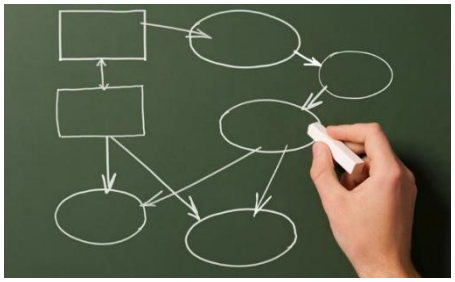
  **Exemplo**: Se quisermos fazer um bolo, precisamos de:

  - partir os ovos;
  - separar as gemas das claras;
  - Entre muito mais …

***Em informática***, uma ***instrução***, é a ação elementar a ser executada.

## 1.2 - Algoritmos

Um **algoritmo** é, portanto, uma sequência ***finita*** e ***logicamente ordenada de ações***, para chegar à ***solução de um problema***.






Além disso, a definição das ações tem de **ser clara, precisa e sem ambiguidades**, ou seja, um ***algoritmo não pode ter diferentes interpretações***.

A ***algoritmia*** (formulação de algoritmos), permite-nos separar as ***3 fases distintas*** na resolução do problema:




1. ***a análise do problema (***especificação do problema, análise de requisitos, pressupostos, etc…***);***
2. ***a conceção do algoritmo (***Sequência finita e ordenada de ações que conduzem à solução de um problema***);***
3. ***a tradução desse algoritmo na linguagem de programação*** (Conjunto de instruções destinadas a serem processadas num sistema informático para realizar alguma tarefa***);***

***Resumindo, um algoritmo*** é uma ***descrição (passo-a-passo)***, de uma metodologia que conduz à ***resolução*** de um ***problema*** ou à ***execução*** de uma ***tarefa***.

***A programação consiste na codificação desse algoritmo***, segundo uma ***linguagem de programação específica***.

Desta forma, é possível conceber algoritmos, e deixar para mais tarde, as questões mais específicas com a implementação do algoritmo numa determinada linguagem de programação.

A ***programação*** consiste na ***codificação precisa desse algoritmo***, segundo uma linguagem de programação específica. **Exemplos de linguagens de programação**:

- Visual Basic,
- C / C++;
- C#;
- Java;
- Entre outros…








***NOTA IMPORTANTE:*** Um mesmo algoritmo pode servir de base à codificação de um programa em ***diferentes linguagens*** e ***ambientes de programação*** ou ***com diferentes técnicas de implementação***.

Podemos considerar como ***algoritmos*** coisas tão diversificados como:

- A substituição de uma lâmpada fundida num automóvel;
- A substituição de um pneu num automóvel;
- Uma receita de culinária;
- A resolução de um problema de matemática;

# Capítulo 2 – Linguagens de Programação

Neste capítulo, vamos explorar um pouco sobre linguagens de baixo e alto nível, e as traduções realizadas pelos mesmos.

## 2.1 - Níveis da classificação das linguagens (alto e baixo nível)

**Níveis das Linguagens de Programação** - expressão que pretende traduzir uma maior ou menor proximidade relativamente à linguagem do computador.

| | | |
|---|---|---|
| **Linguagens de programação** | **Alto Nível** | O programador não tem que saber nem preocupar-se com a forma como a máquina resolve as operações que ele manda executar. |
| | **Baixo Nível** | Cada instrução (ordem para o CPU) está associada a uma sequência de "0" e "1", designada por palavra. |

**Níveis da classificação das linguagens (Baixo Nível)**

- **Linguagem Máquina (formato binário)** - codificação das instruções e dados para formato binário do computador, de acordo com a estrutura ou arquitetura do CPU. Corresponde a sequências codificadas de *bits*.




- **Linguagem Assembly (mnemónicas) -** nível de codificação ainda muito próximo da linguagem máquina, mas baseada em mnemónicas (abreviaturas ou caracteres sugestivos das palavras), representando as operações.

**Implementação da expressão: Z = A + B - (C + D)**

*linguagem máquina vs linguagem assembly*

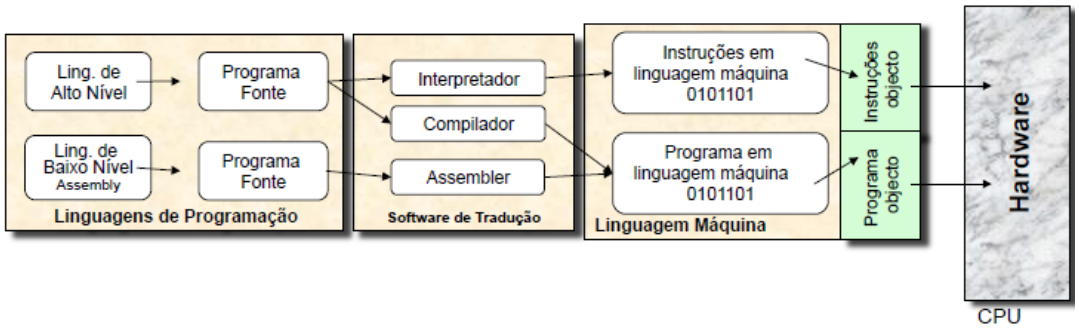
| Instrução | Linguagem máquina | Linguagem assembly | Significado |
|---|---|---|---|
| 1 | 011011010100 | S, ADD C,D | - Fazer: S ← C+D |
| 2 | 011001010101 | R, SUB B,S | - Fazer: R ← B-S |
| 3 | 001010101110 | E, ADD A,R | - Fazer: E ← A+R |
| 4 | 010000000000 | HLT | - Finalizar |

- Numa **Linguagem de Alto Nível,** o programador utiliza um conjunto mais abrangente de instruções que escreve nos seus programas, tendo apenas que saber escrevê-las corretamente e dar-lhes uma ordenação lógica, com vista a obter o resultado.

  **Algumas linguagens:** Fortran, Cobol, Basic, Pascal, C, Linguagens Orientadas por Objetos (OOP ou POO)

## 2.2 - Software de tradução, interpretador e compilador

**Software de Tradução** - Para converter um programa escrito em qualquer uma das linguagens de alto nível para a linguagem máquina, não basta pedir ao computador que leia o programa, é necessário um *software* específico para fazer a tradução para os "0" e "1" inteligíveis ao computador.




- **Interpretador** - traduz instrução a instrução à medida que o programa vai sendo lido e executado, sendo a tradução feita simultaneamente. O programa necessitará de recorrer ao interpretador, sempre que é executada uma instrução.

- **Compilador** - traduz a totalidade das instruções para um programa em linguagem máquina (ou intermédia), o qual poderá ser executado diretamente pelo computador (ou máquina virtual) e independentemente do *software* que efetua a tradução.

# Capítulo 3 – Instalação Visual Studio

## 3.1 – Preparação e requisitos mínimos

Antes de começar a instalação do Visual Studio, tome atenção aos seguintes pontos:

***a)*** Verifique os requisitos do sistema. Estes requisitos ajudam a saber se o computador tem suporte para o editor Visual Studio.

***b)*** Tenha as atualizações mais recentes do Windows. Essas atualizações garantem que o computador tenha as atualizações de segurança mais recentes e os componentes do sistema obrigatórios para o processo de instalação do Visual Studio.

***c)*** Se houver algum processo de reiniciar o computador, deve efetuar o mesmo. A reinicialização garante que todas as instalações ou atualizações pendentes não impeçam a instalação do Visual Studio.

***d)*** Liberte espaço. Remova arquivos e aplicações desnecessários do sistema, por exemplo, executando o aplicativo Limpeza de Disco.

Pode instalar o Visual Studio lado a lado com versões anteriores.

Para obter mais informações, consulte a seção do ***Direcionamento e compatibilidade da plataforma Visual Studio*** e instalar versões do Visual Studio no mesmo sistema.

Em termos de requisitos mínimos, temos as seguintes especificações:

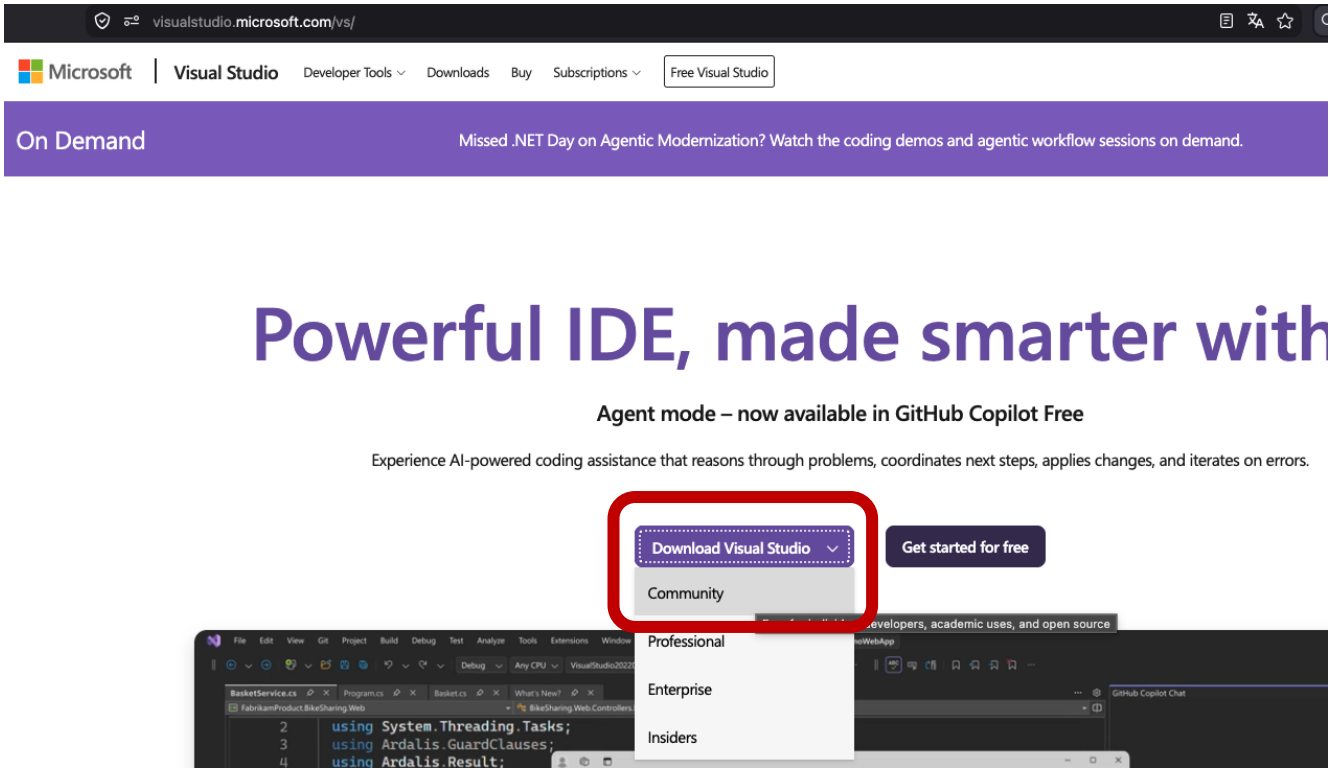
https://learn.microsoft.com/pt-pt/visualstudio/releases/2026/vs-system-requirements

## 3.2 – Descarregar e Instalar do IDE Visual Studio

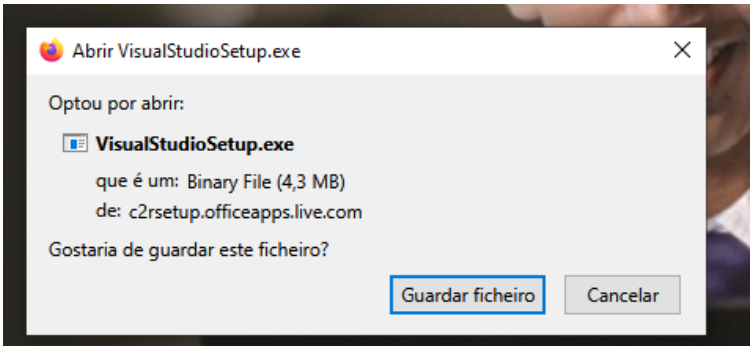
Deve começar por fazer o download do instalador do Visual Studio.

Hiperligação: https://visualstudio.microsoft.com/vs/

Para este módulo, vamos retirar a versão gratuita, neste caso, a versão Community:

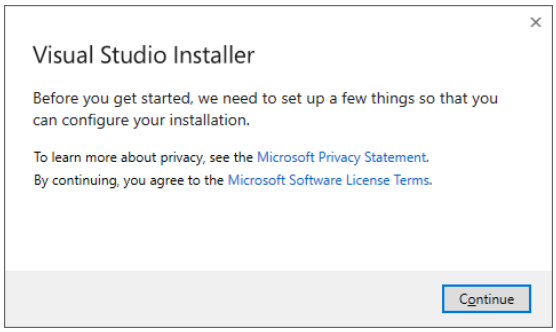


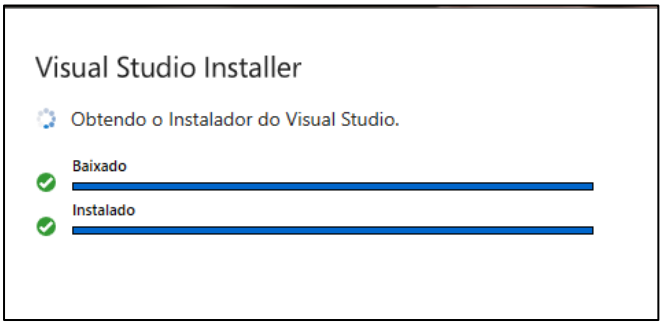

Este, por sua vez, vai descarregar o pacote da instalação do software:




Após o download, deve executar o ficheiro para instalar o editor do Visual Studio. Esse instalador inclui tudo o que precisa para instalar e personalizar o Visual Studio. Se receber um aviso de Controlo de Conta de Utilizador, escolha **Sim**, para dar autorização.

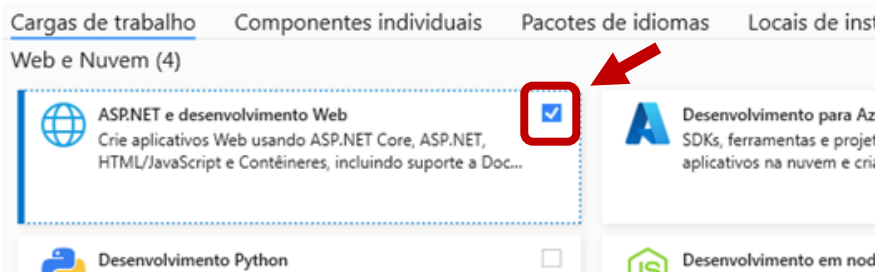
Será solicitado uma confirmação dos **Termos de Licença** e da **Política de Privacidade** da Microsoft. Escolha **Continuar**.



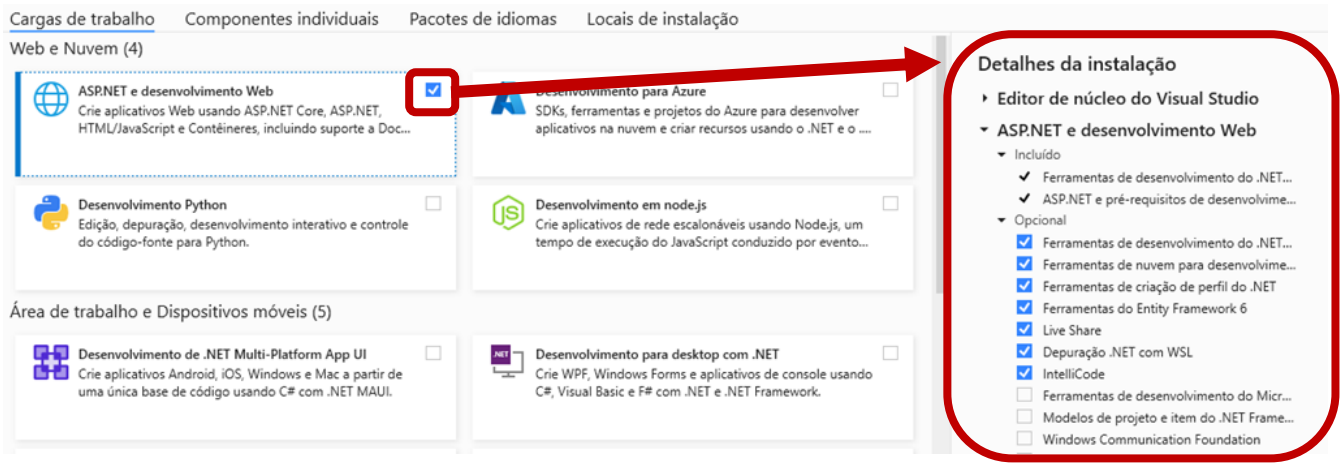

Quando o instalador do Visual Studio efetuar a pré-instalação, poderá personalizar a instalação selecionando os conjuntos de recursos desejados.

***Nota Importante***: Neste processo, temos de selecionar a cargas de trabalho que desejamos instalar. Para tal, ***deve escolher o grupo*** e ***selecionar a caixa de seleção*** que está no ***canto superior direito***.




Cada carga de trabalho tem mais detalhes de instalação, ou seja, poderá instalar pacotes adicionais de trabalho que podem estar desabilitados. Isto irá permitir decidir os recursos necessários para o desenvolvimento.

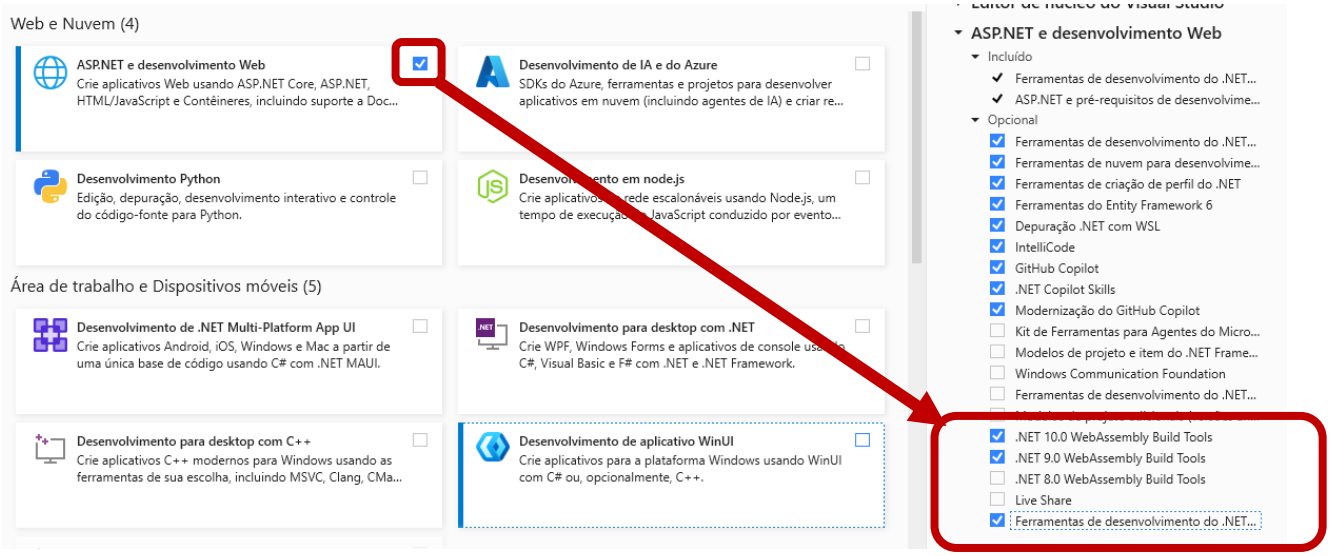
***Exemplo***: se escolher a carga de trabalho ***ASP.NET e desenvolvimento para a Web***, irá aparecer uma barra lateral no lado direito da janela para verificar as opções que já vem selecionadas por defeito, mas se sentir a necessidade de instalar algo mais específico, poderá analisar as opções adicionais oferecidas nesta versão.




***Cada carga de trabalho tem opções de personalização diferente.*** Normalmente, as opções que já estão selecionados por defeito, permitem efetuar grande parte do trabalho para os programadores que estão numa fase inicial.

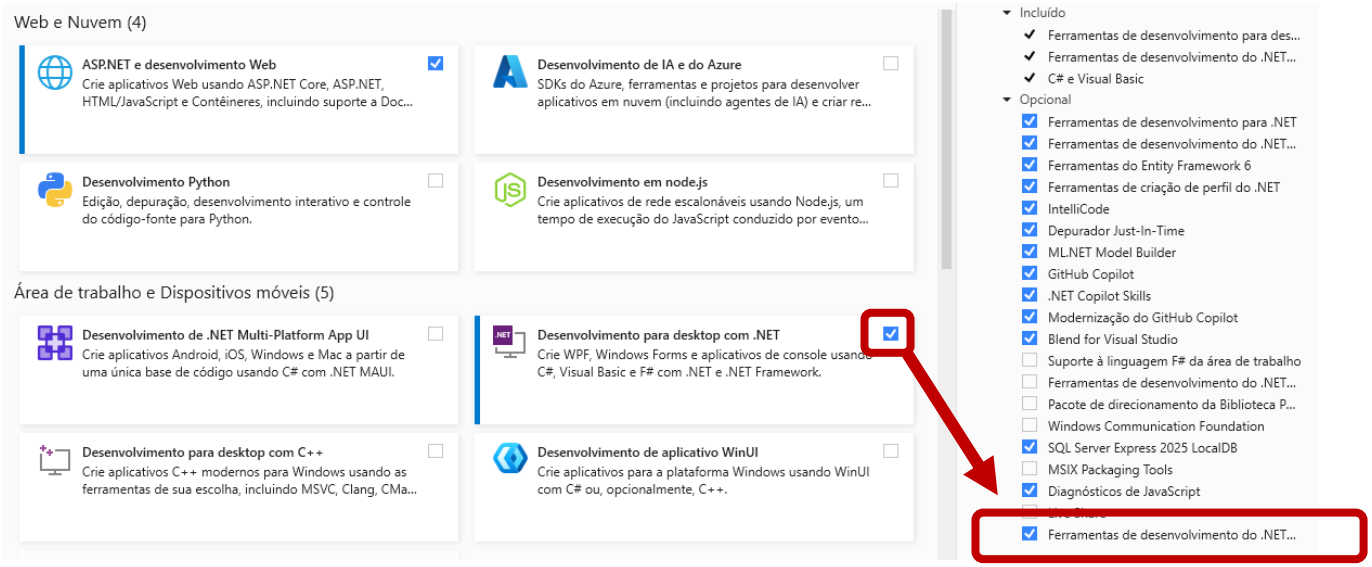
## *Opção 1: ASP.NET e desenvolvimento Web*

- Dentro da carga de trabalho, temos de selecionar algumas opções adicionais:
  - .NET 10.0 WebAssembly Build Tools
  - .NET 9.0 WebAssembly Build Tools
  - Ferramentas de desenvolvimento do .NET



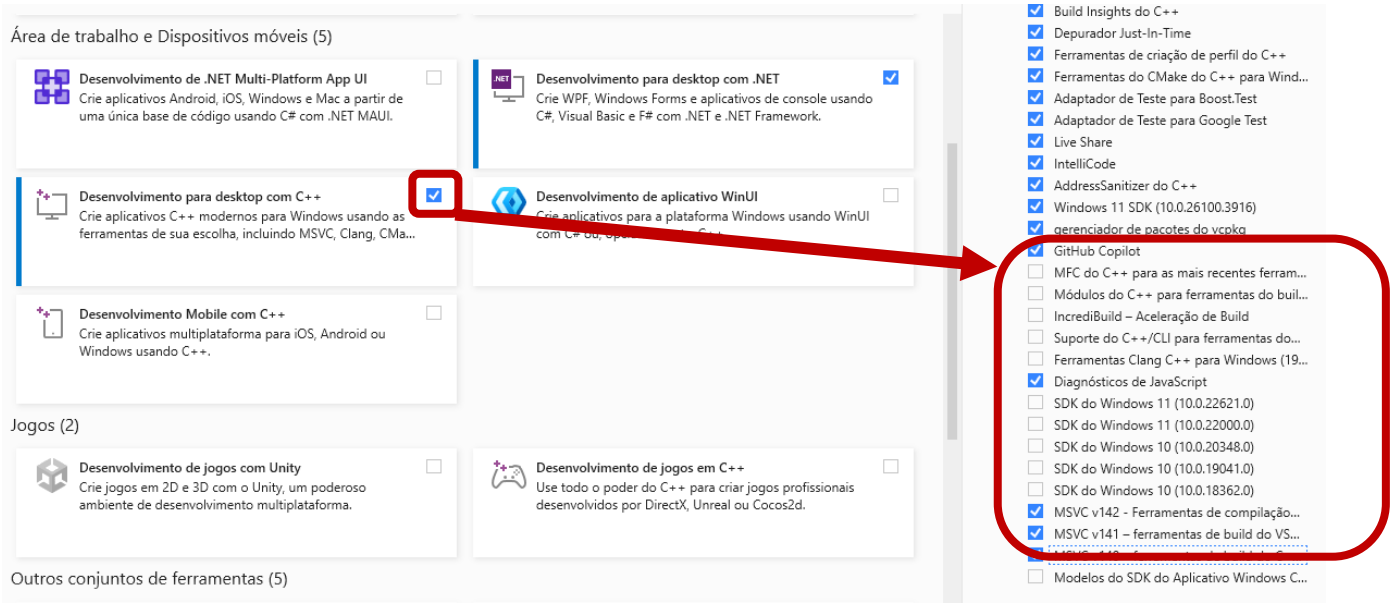
### *Opção 2: Desenvolvimento para desktop com .NET*

- Dentro da carga de trabalho, temos de selecionar algumas opções adicionais:
    - Ferramentas de desenvolvimento do .NET



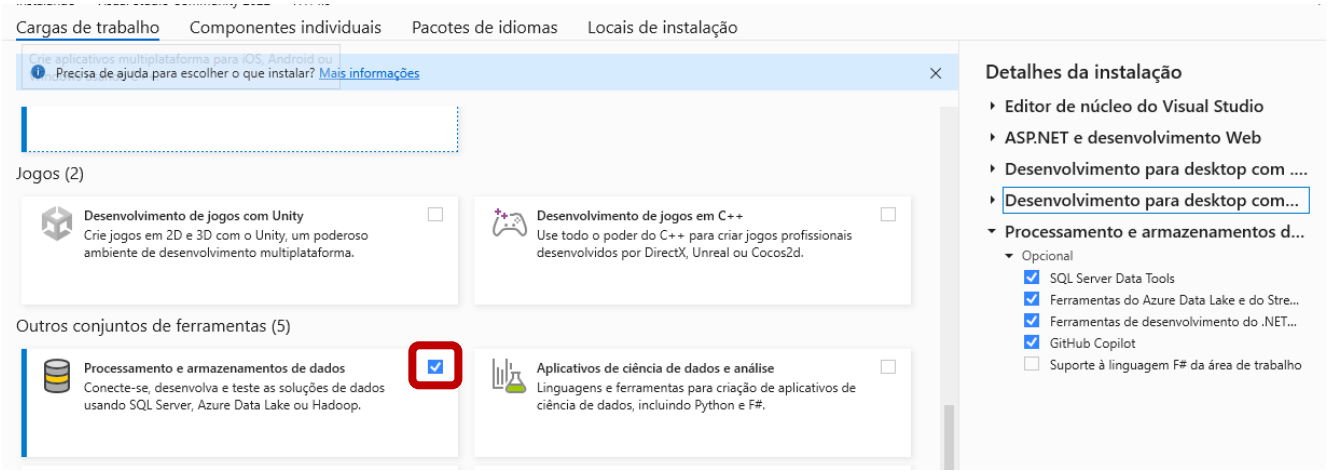

### *Opção 3: Desenvolvimento para desktop com C++*

- Dentro da carga de trabalho, temos de selecionar algumas opções adicionais:
    - Ferramentas de build MSVC v143
    - Diagnósticos de Javascript
    - MSVC v142 – Ferramentas de compilação VS 2019 C++
    - MSVC v141 – Ferramentas de build VS 2017 C++



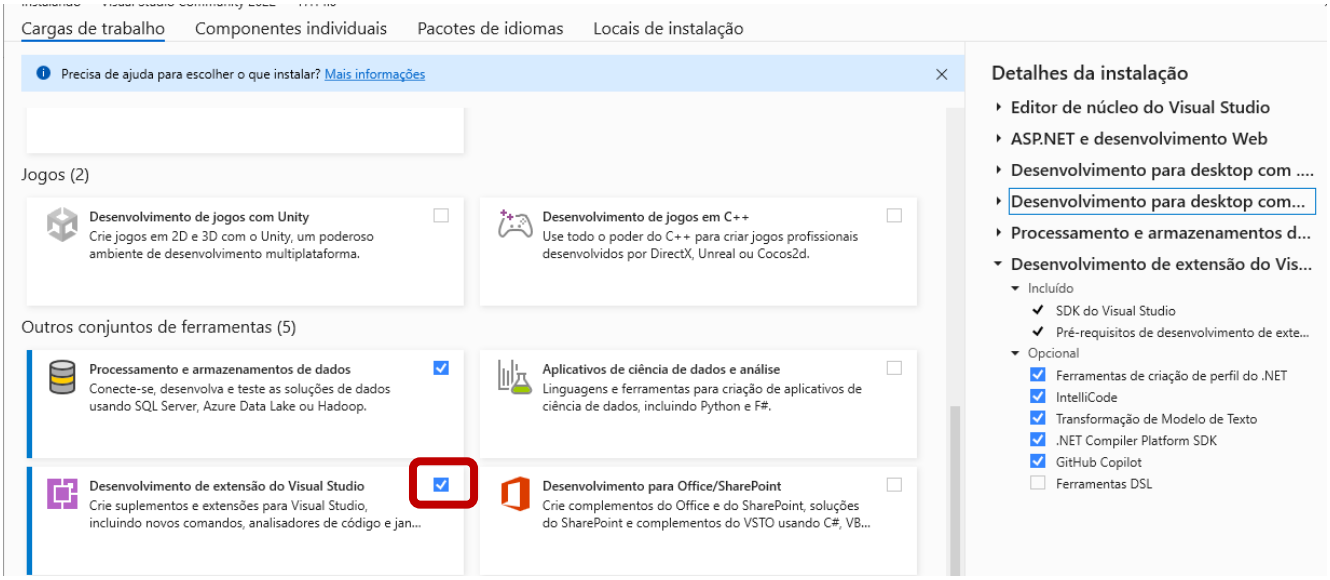
## *Opção 4: Processamento e armazenamento de dados*

- Dentro da carga de trabalho, ***não precisa de selecionar nenhuma opção adicional***.




## *Opção 5: Desenvolvimento de extensão do Visual Studio*

- Dentro da carga de trabalho, ***não precisa de selecionar nenhuma opção adicional***.



### *Opção 6: Desenvolvimento para Linux, Mac e embarcados com C++*

- Dentro da carga de trabalho, ***não precisa de selecionar nenhuma opção adicional***.

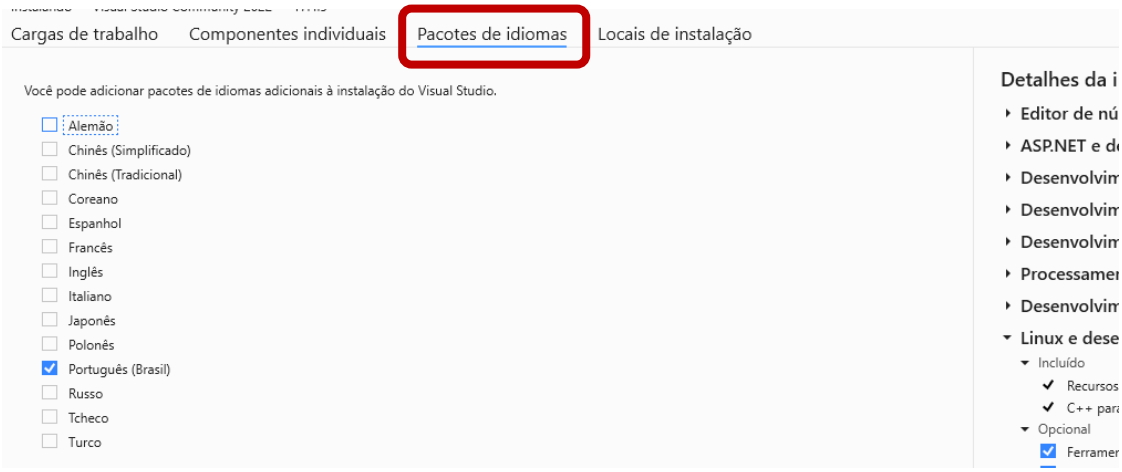



### *Opção 7: Seleção Idiomas e Locais de Instalação*

#### *7.1 - Seleção Idiomas*

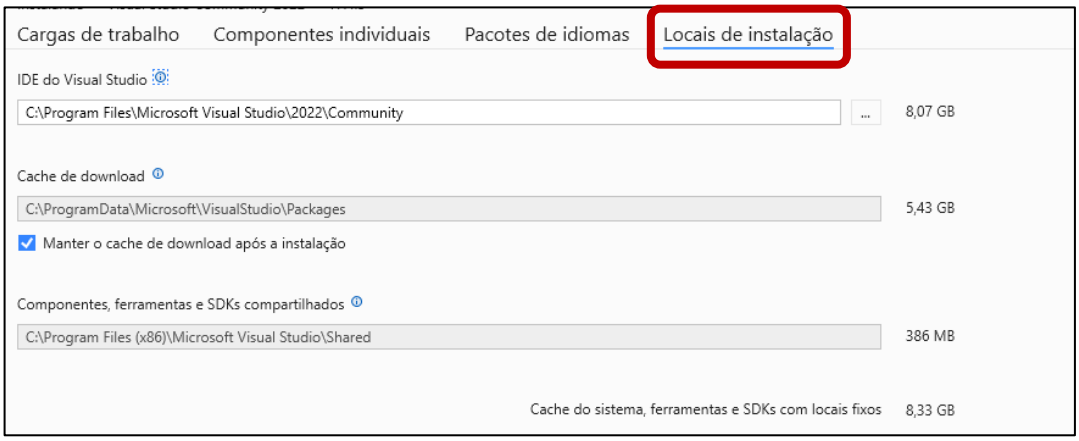
- Nesta seção "***Pacotes de idiomas***", basta ***selecionar os idiomas***. Isto vai permite com que o ***editor ajuste a linguagem mais apropriada*** para uma melhor compreensão dos conteúdos e navegação do mesmo.

- ***Por defeito***, se a instalação do ***editor reconhecer a sua zona geográfica***, automaticamente preenche a opção mais apropriada (***mas é apenas uma recomendação***).



### *7.2 - Locais de Instalação no dispositivo*

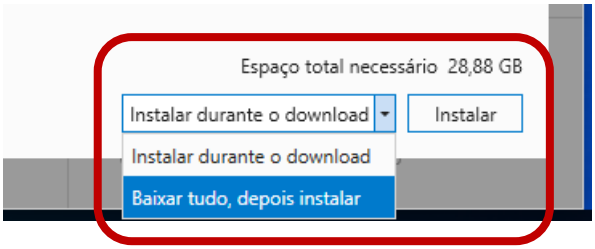
- Nesta seção "***Locais de instalação***", pode indicar onde deseja instalar o editor. Embora que, a primeira opção seja possível alterar, as outras duas são endereçadas para a localização onde o sistema operativo está instalado.




- ***Nota Importante***: Desde a versão Visual Studio 2017 e na seção "***Locais de instalação***", pode indicar como processo de instalação pode ser feito. No canto inferior direito existe uma caixa de seleção com 2 opções:

  - ***Forma 1***: ***Instalar durante o download*** – esta opção permite descarregar os ficheiros de instalação e efetuar a instalação dos mesmos sem estar a espera que o seja feito todo o download dos conteúdos;

  - ***Forma 2***: ***Baixar tudo, depois instalar*** – esta opção permite em primeiro descarregar todos os ficheiros da internet e só depois é que faz a instalação dos todos os conteúdos.



# Capítulo 4 – Resolução de problemas pós-instalação

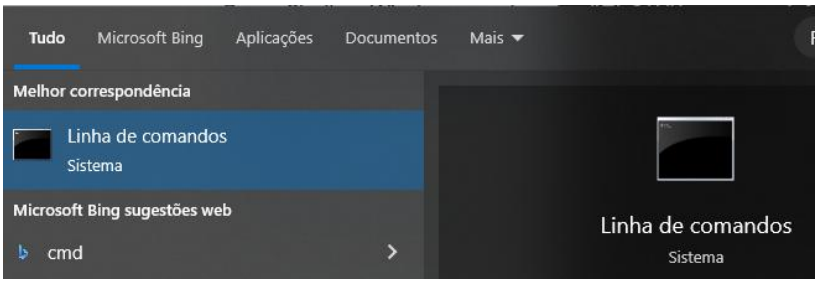
Por norma o Visual Studio já vem com as versões dos núcleos e bibliotecas mais atualizadas. Mas, no entanto, é necessário perceber se a versão atual está instalada e atualizada para garantir que temos todos os recursos à mão e sem ter restrições de utilização.

Como tal, temos de verificar se o sistema operativo tem o Kit do Software de Desenvolvimento, também conhecido com SDK (Software Development Kit), que é um conjunto de ferramentas que permite aos programadores criar, compilar e executar aplicações na plataforma .NET.

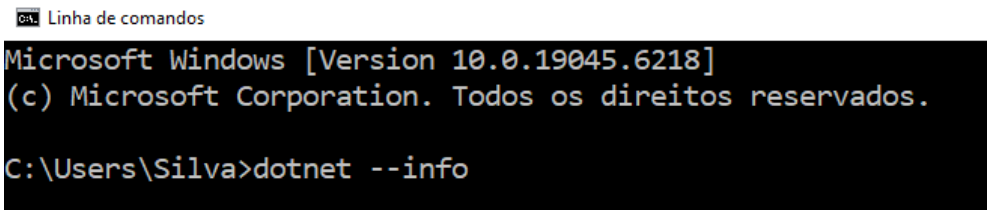
Este Kit inclui o runtime, as bibliotecas e ferramentas da linha de comando .NET, além de recursos para compilar, empacotar e publicar o código. Essencialmente, o SDK fornece tudo o que um programador precisa para construir e implementar software usando linguagens como C#, como aplicações web, móveis e serviços web ou na cloud (nuvem).

Como tal, vamos primeiro verificar qual a versão do SDK .NET que temos nos vossos sistemas. Faça o seguinte passo:

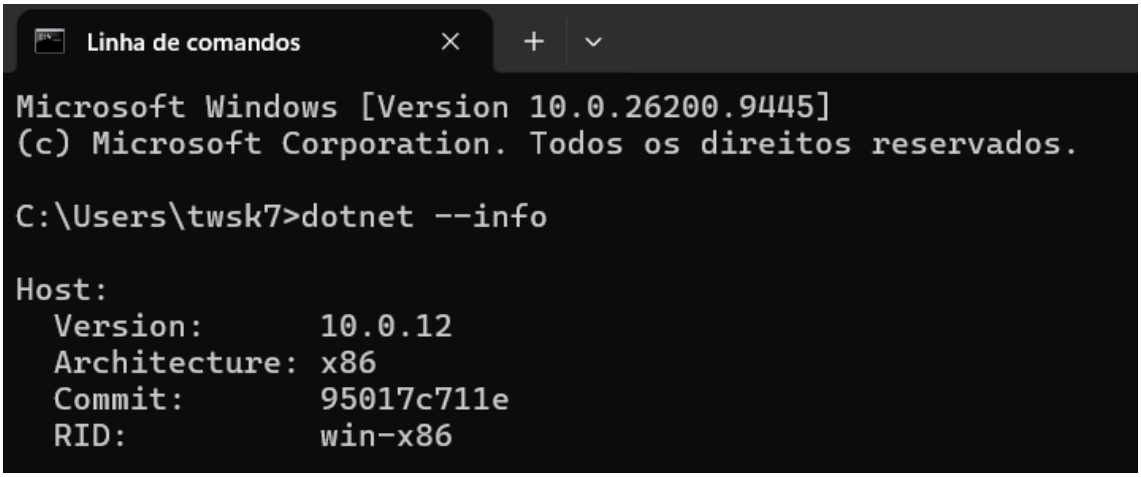
**Passo 1 –** Vamos ao menu do iniciar ou pesquisa do Windows e procurar pela aplicação da Linha de Comandos ou Terminal, bastando digitar as palavras “cmd”:




**Passo 2 –** Digite o comando dotnet –info (com espaço entre as duas expressões):




**Passo 3 –** Verifique qual é a versão instalada no computador:

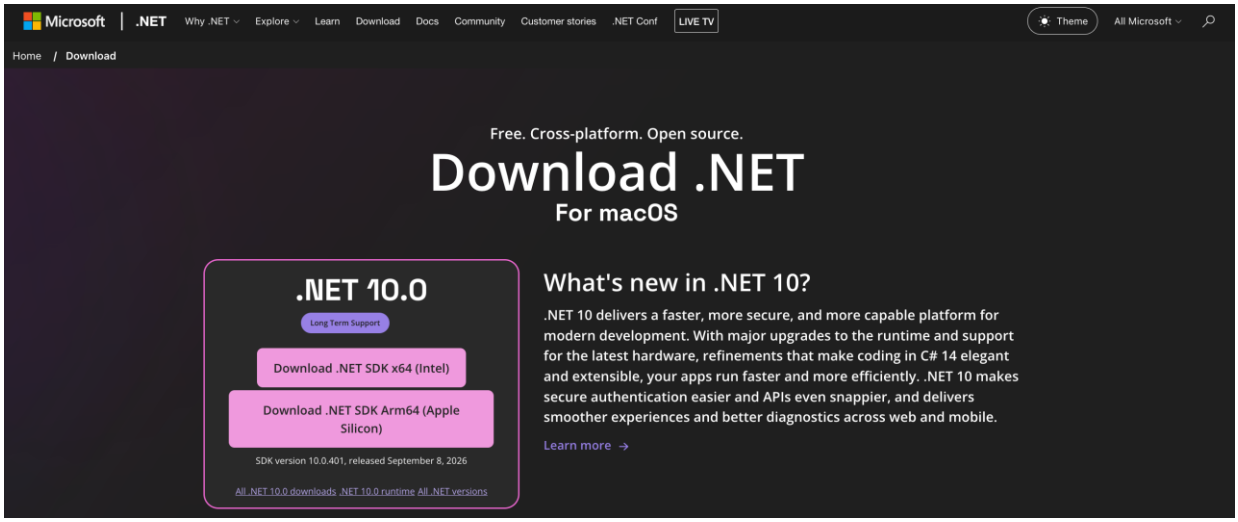



***Se não aparecer o bloco com as informações do SDK, muito provavelmente, não deverá ter a versão mais atualizada do SDK .Net ou faltam ficheiros.***

***Como tal, vamos seguir os seguintes passos (faz o seguinte vim instalar a versão mais atual da biblioteca):***

**Passo 1 -** Para retirar a biblioteca com o software .Net Core SDK, vamos ao seguinte link: https://dotnet.microsoft.com/en-us/download mais especificamente:

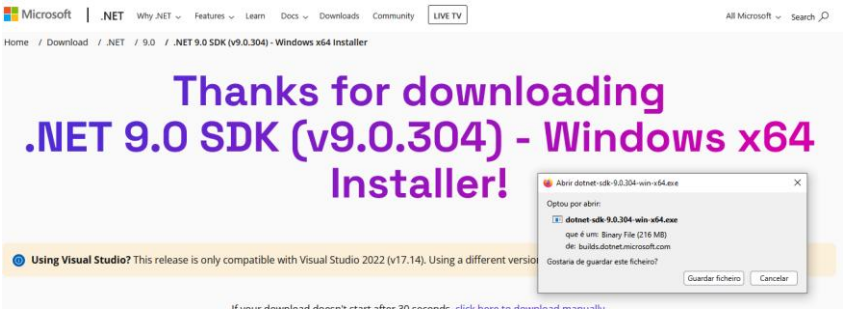
- **Computadores com processadores Intel/AMD** devem clicar na opção “Download .NET SDK x64 (Intel)”;
- **Computadores com processadores ARM64 (Mac Apple Silicon)** devem clicar na opção “Download .NET SDK Arm64 (Apple Silicon)”;



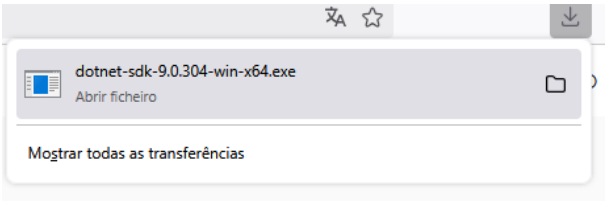

**Nota importante:** O SDK é compatível com Windows, Linux e MacOS. Caso necessite outras versões, basta clicar na seta que está a frente da caixa do download e será exibida as outras opções:



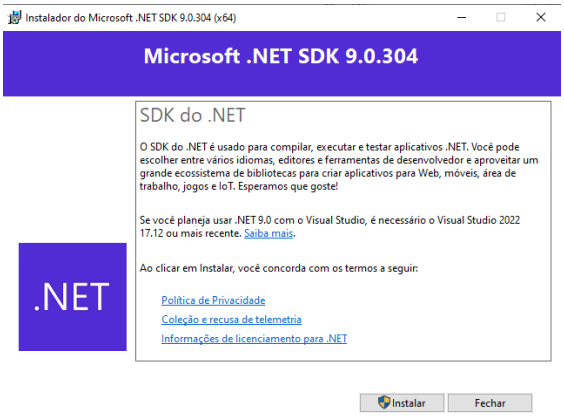

**Passo 3 –** Quando carregar a nova página o navegador pode perguntar se deseja descarregar a aplicação para instalar ou então nos navegadores mais recentes o download é inicializado automaticamente:



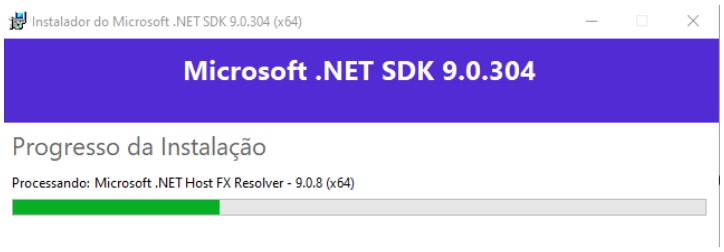

**Passo 4 –** Após o download, execute a aplicação:



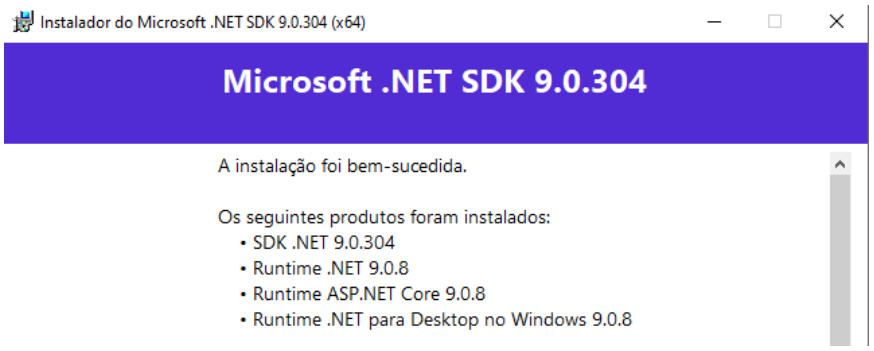

**Passo 5 –** A instalação do SDK é simples e fácil de realizar. Como tal, deve ler o conteúdo textual e para que serve esta instalação, e de seguida, clicar no botão Instalar:



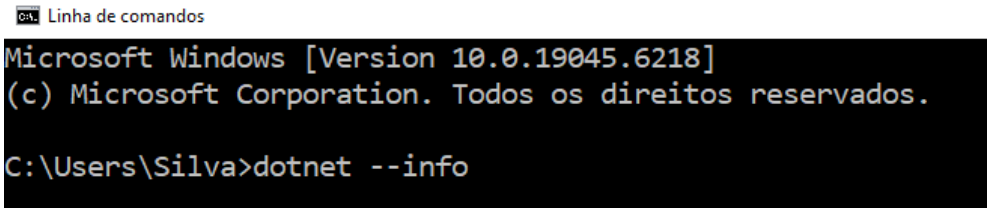
Este por sua vez começa logo a fazer a atualização ou instalação do SDK no sistema operativo:




No final será exibida a janela se tudo estiver instalado como deve ser:




**Volte a abrir uma linha de comandos e** digite o comando dotnet –info (com espaço entre as duas expressões):



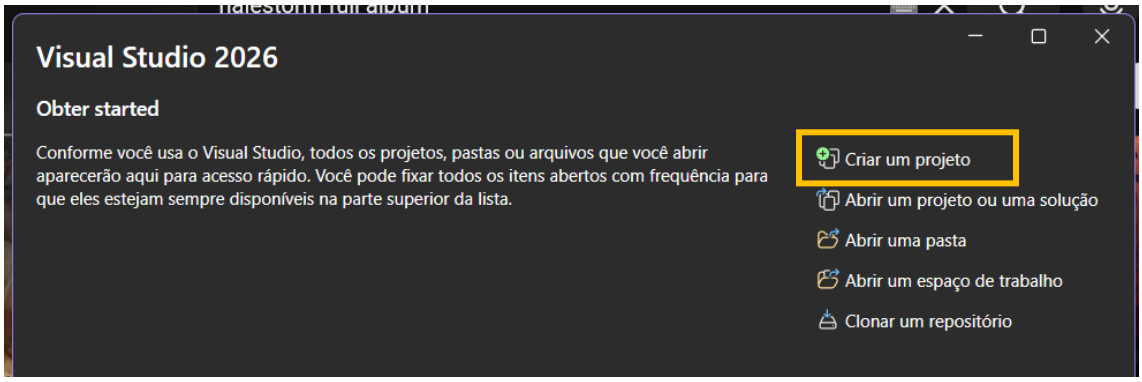

Se ainda assim ainda não aparecer o sdk temos de ir a forma manual.

# Capítulo 5 – Como criar um projeto com a linguagem C#

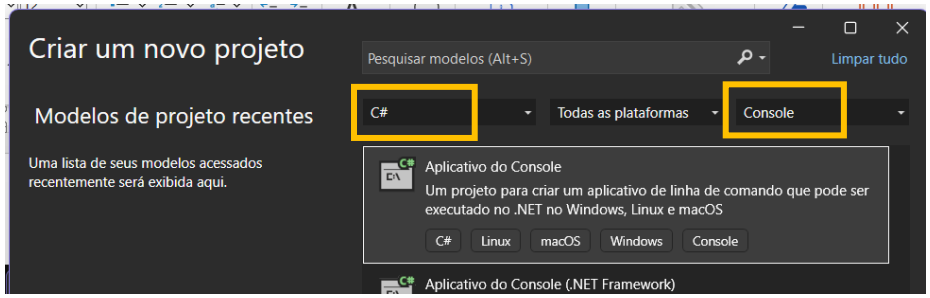
## Capítulo 5.1 – Como criar um projeto C#

Para fazer a criação de novos projetos não é efetuar os seguintes passos para a versão visual Studio 2022:

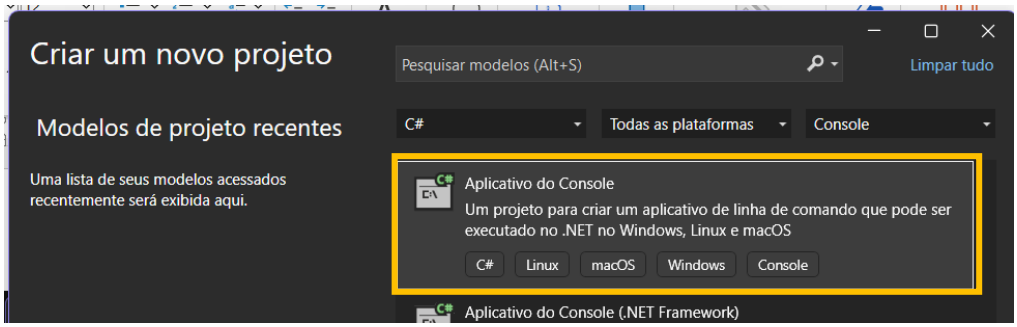
**Passo 1** – Abra o editor Visual Studio e selecionar a opção **Criar um novo projeto** no canto inferior direito da imagem:



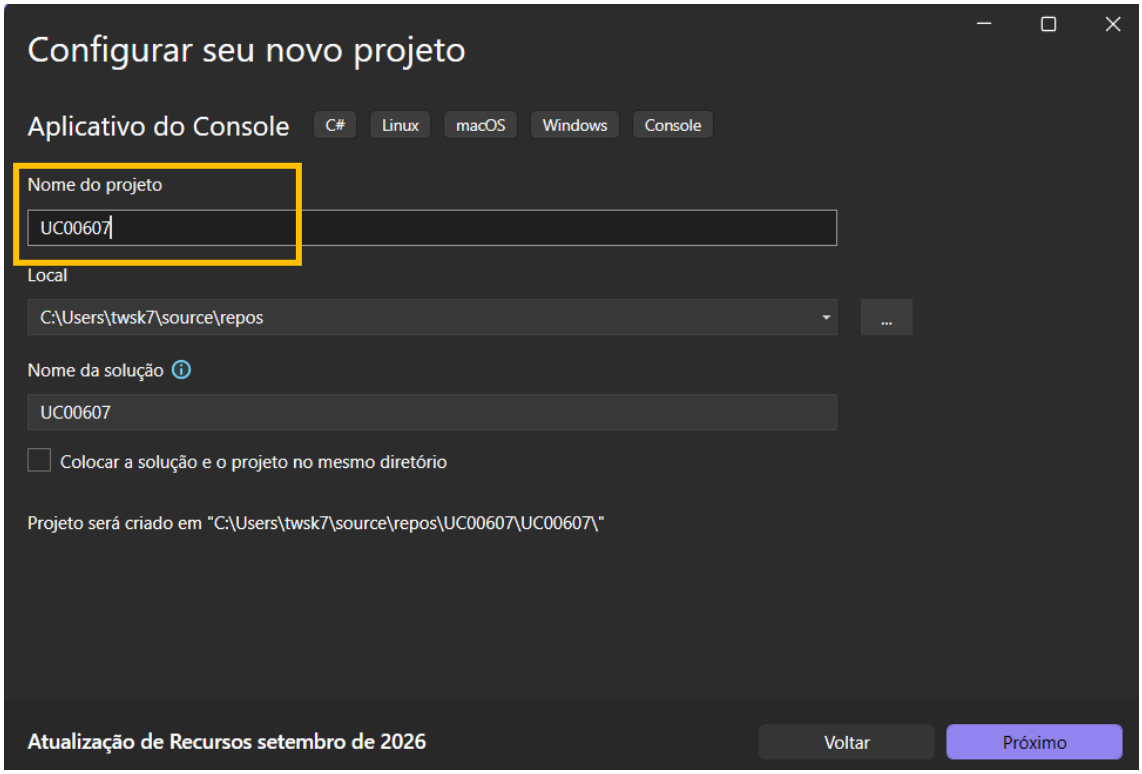

**Passo 2** - Na caixa de diálogo de criar um novo projeto, deve fazer a filtragem dos projetos do tipo c#. Como tal, deve filtrar por: linguagem **c#** e no último filtro deve indicar do **tipo Consola** (ou **Console**):



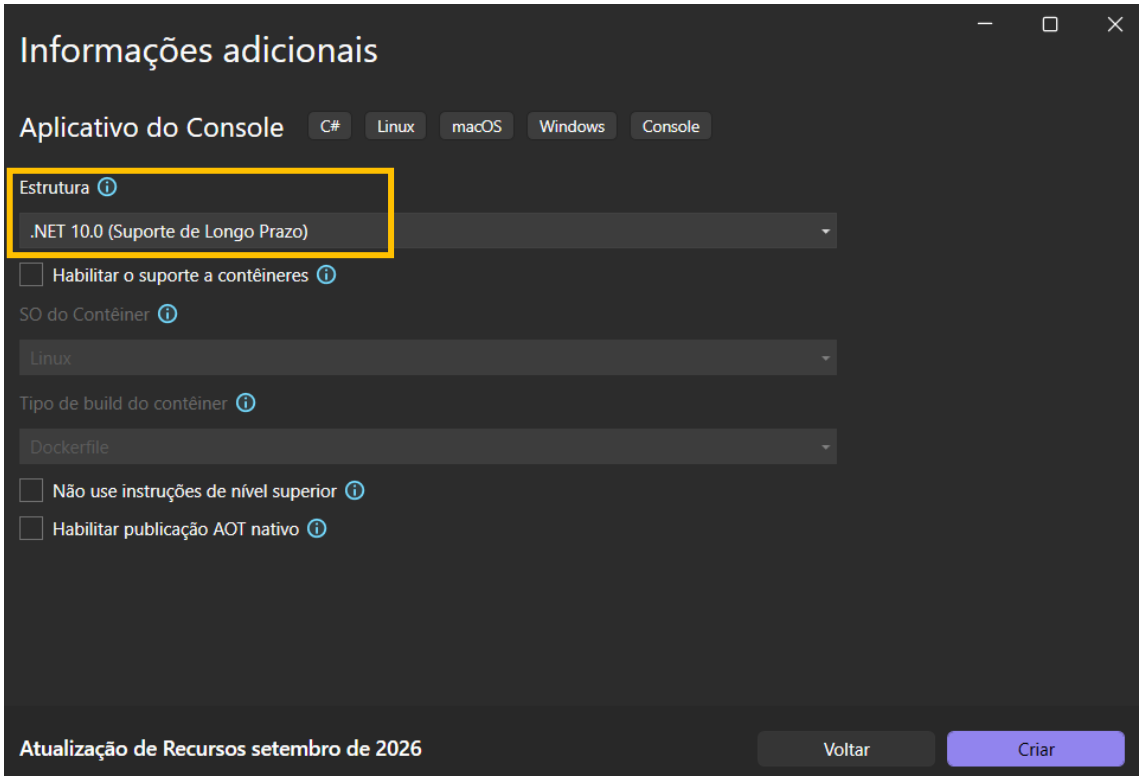

**Passo 3** – Depois, vamos escolher a opção "Aplicação de consola (.NET Core) ou Console App" e, em seguida, pressione **no botão Próximo:**



**Passo 4** – Dê um nome ao projeto, como por exemplo, "UC00607" e depois clique no botão **Próximo**:




**Passo 5** – Na última janela, vamos indicar qual é a versão, neste caso, a versão .NET 10 que queremos trabalhar e clicar no botão Criar:



## Capítulo 5.2 – Possíveis erros na criação dos projetos

**Passo 6** - Após a criação do projeto aparece o seguinte ambiente de trabalho (IDE)

Quando criamos projetos com a framework .NET Core, podemos ter problemas na geração dos ficheiros e no sistema de compilação do programa. Um dos erros mais comuns ocorre quando atualizamos o nosso Visual Studio para uma versão mais recente ou atualizamos a biblioteca do SDK .Net Core.

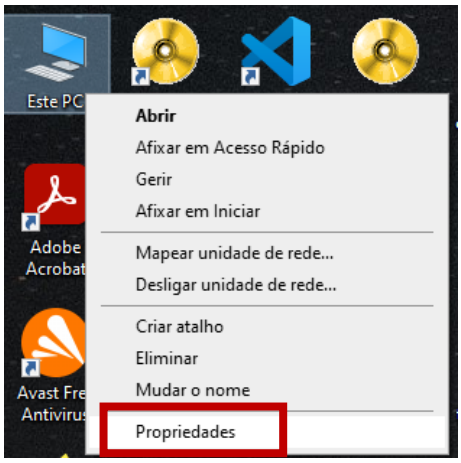
Por detrás do sistema, **os projetos precisam de saber onde está a localização da pasta com os ficheiros para executar os projetos .NET Core.** Essas informações estão localizadas nas propriedades avançadas do sistema, mais especificamente, **nas variáveis do sistema** e, por vezes, pode criar mais do que um caminho ao mesmo tempo, gerando assim conflito quando precisa de aceder a pasta com o caminho correto.

**Como tal, vamos ver uma situação muito comum:**

***Caso 1:*** Foi instalado o editor (IDE) Visual Studio 2022 com a versão do framework .NET Core 6 a bastante tempo. Depois de várias atualizações, houve uma duplicação nas variáveis do sistema e criou dois caminhos diferentes.

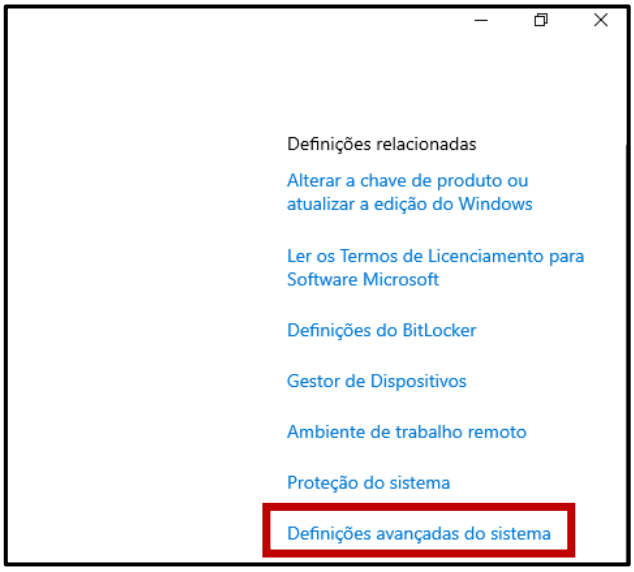
Desta forma, pode surgir problemas ao criar projetos com o Visual Studio e temos de indicar qual é o caminho correto. Como tal basta fazer os seguintes passos:

**Passo 1 –** Clicar com o lado direito do rato no ícone Este PC no ambiente de trabalho e selecionar a opção propriedades:

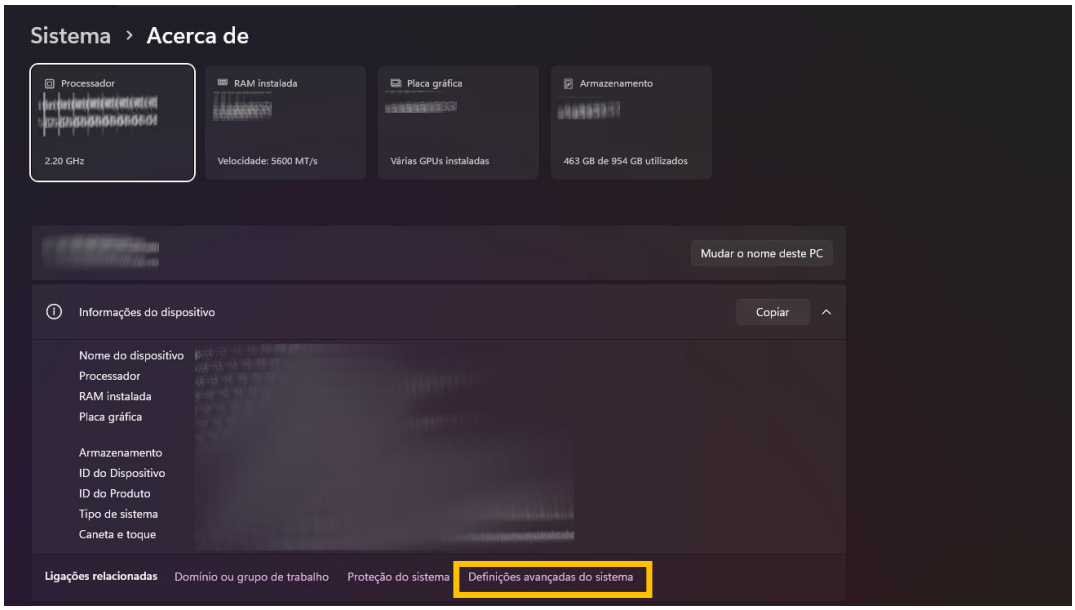


**Passo 2 –** Na nova janela, vamos ao lado direito e no meu lateral, deve selecionar a opção “Definições avançadas do sistema”:

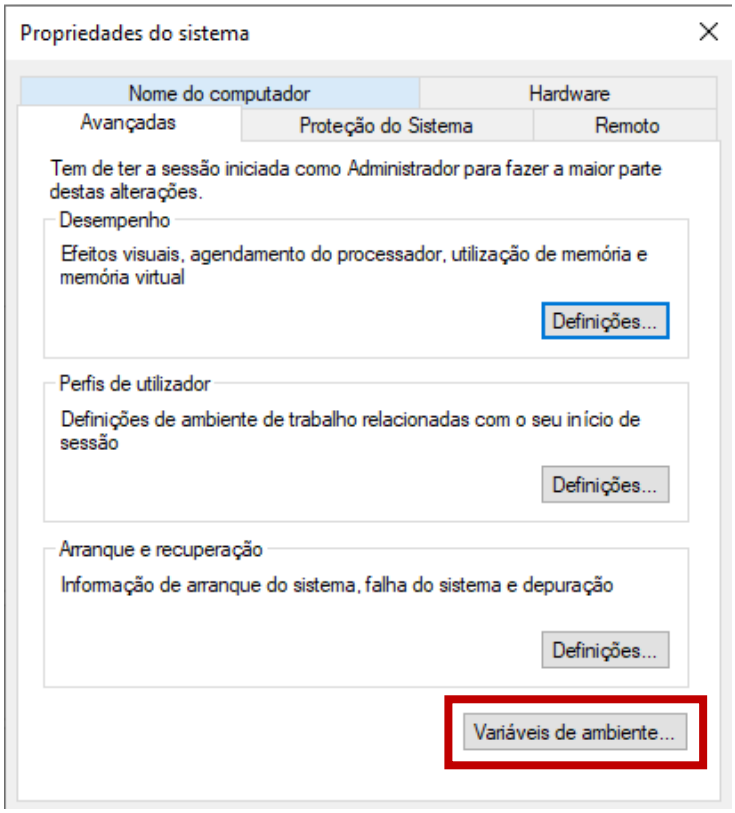
**Windows 10:**




**Windows 11:**

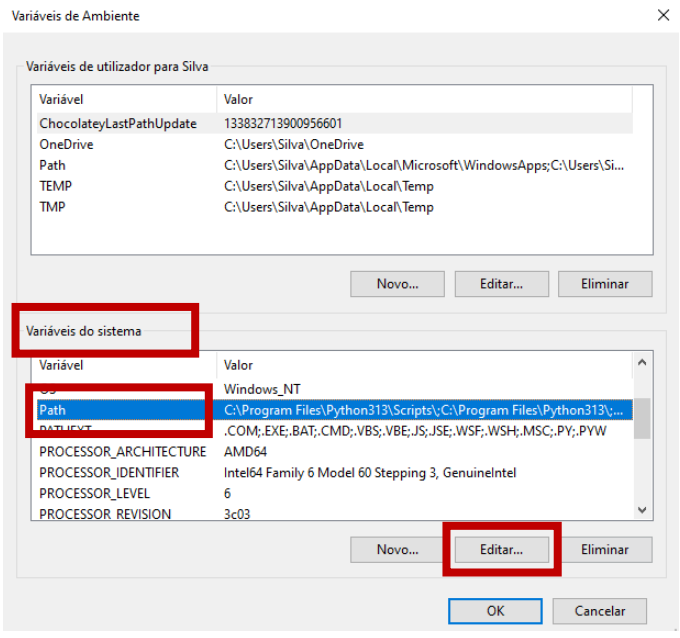



**Passo 3 –** Na nova janela, vamos clicar na opção para aceder às propriedades das “Variáveis de ambiente”:



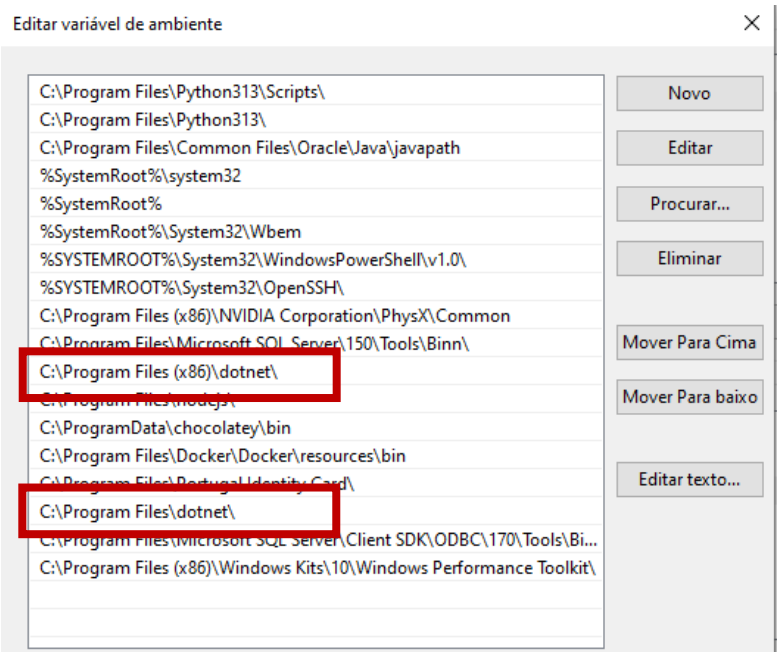

**Passo 4 –** Na nova janela, temos duas áreas: as variáveis do utilizador e as variáveis do sistema. Para o nosso contexto, queremos trabalhar na área das variáveis de sistema. Como tal vamos fazer os passos:

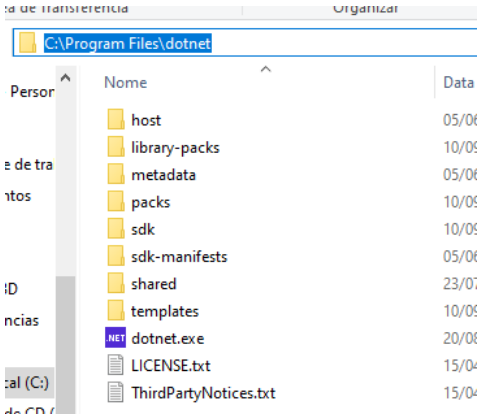
- na listagem de opções das **variáveis do sistema**, selecione a opção **Path**;
- clique no botão "**Editar**" para aceder a todos os caminhos do sistema;



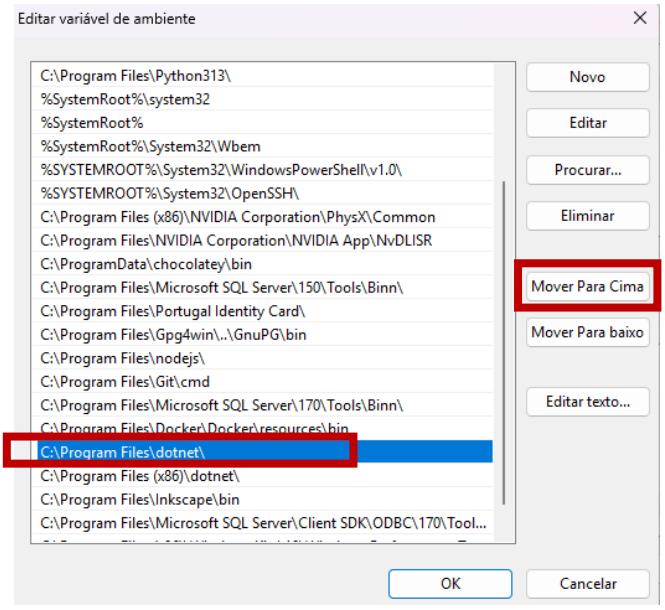
**Passo 5 – Se reparar com atenção**, temos **dois caminhos a apontar para o framework dotnet** (daí o **conflito na criação dos projetos**).

Para tal, **e como refere a documentação**, a instalação fica no diretório ***"C:\Program Files\dotnet"***. **Mas se reparar com atenção**, temos outra localização parecida **antes de chegar a localização correta** que é a versão dotnet x86 (que são para os computadores com processadores 32 bits, quando o nosso computador já tem a arquitetura de processadores de 64 bits):



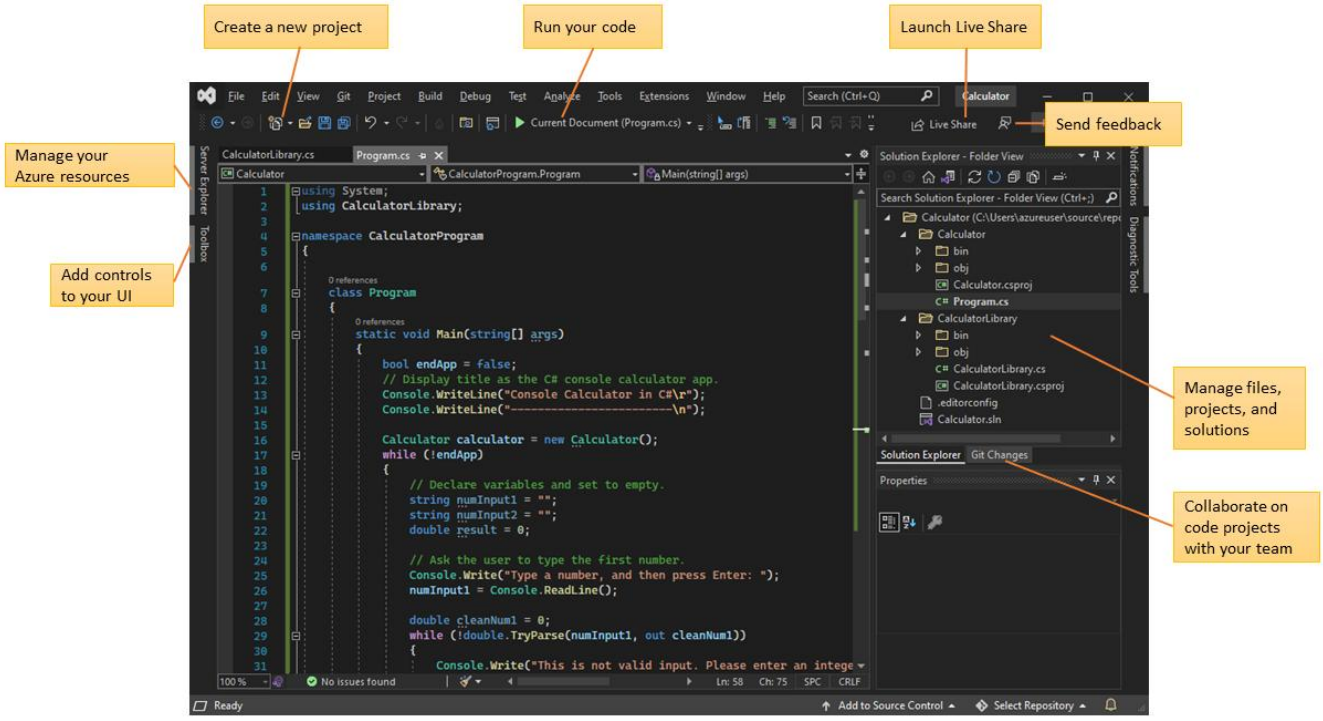

**Como tal e para corrigir este problema,** temos de mudar a posição de prioridade do carregamento correto da tecnologia dotnet (uma vez que os nossos sistemas vão carregar os dados pela ordem crescente). Basta selecionar o mover o registo "C:\Program Files\dotnet\" **para cima** do dotnet (x86):



## Capítulo 5.3 – Ambiente de trabalho em editor C#

**Passo 6** - Após a criação do projeto aparece o seguinte ambiente de trabalho (IDE)

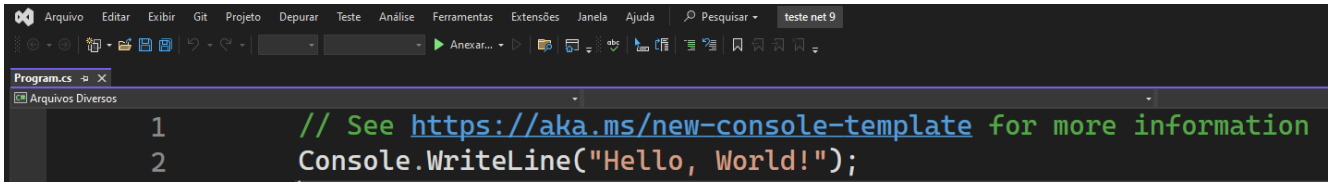
Um ambiente de desenvolvimento integrado (IDE) é um programa rico em recursos que suporta muitos aspetos do desenvolvimento de software. O IDE do Visual Studio é uma plataforma de inicialização para editar, depurar e criar código e, em seguida, publicar aplicações. Além do editor e depurador padrão que a maioria dos IDEs fornece, o Visual Studio inclui compiladores, ferramentas de autocompletar código, designers gráficos e muitos outros recursos para aprimorar o processo de desenvolvimento de software. O editor é constituído pela seguinte imagem:




A imagem anterior mostra o Visual Studio com um projeto aberto que mostra as principais janelas e funcionalidades:

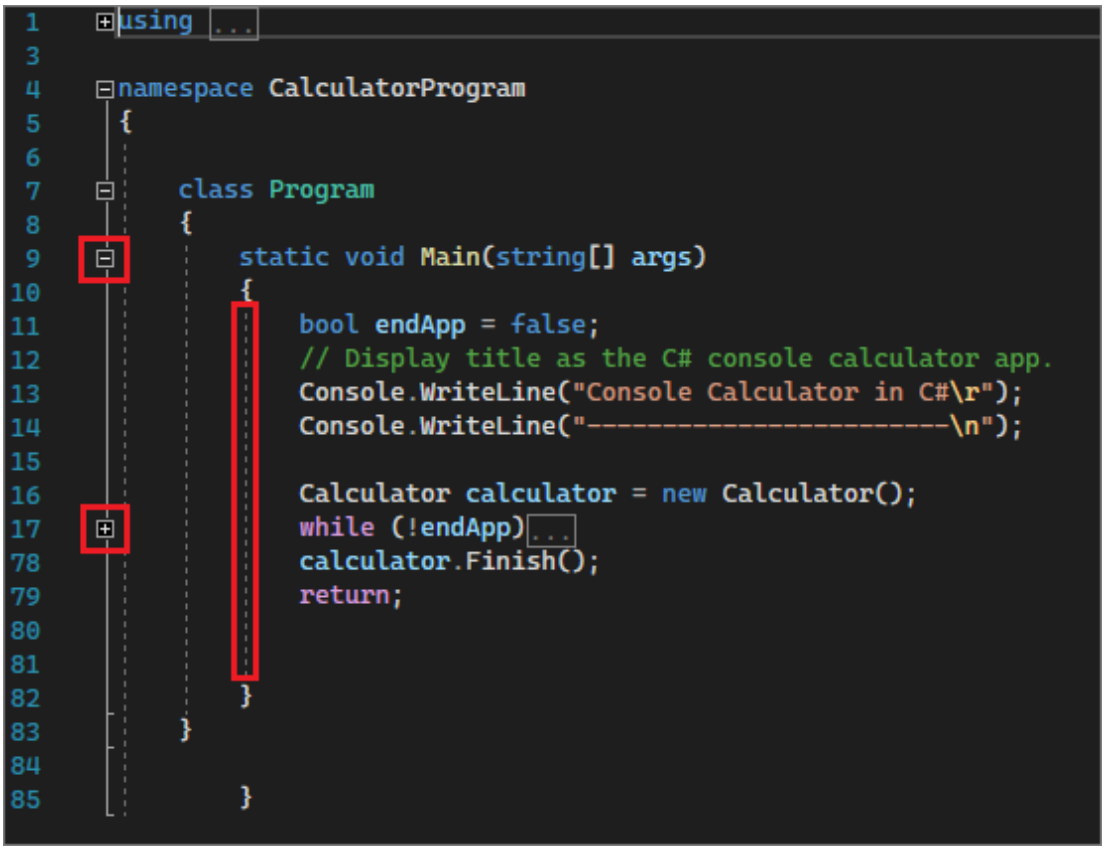
- No **Gerenciador de Soluções** (no canto superior direito), pode exibir, navegar e gerir os ficheiros de código. O **Gerenciador de Soluções** pode ajudar a organizar código agrupando os ficheiros em soluções e projetos.
- A **janela central do editor**, é o local onde vamos escrever e editar o código na maior parte do tempo. Na janela do editor, pode editar código ou criar uma interface;
- No **Git Changes** (no canto inferior direito), pode analisar itens de trabalho e compartilhar código com outras pessoas usando tecnologias de controlo de versões de código, como Git e GitHub.

**Passo 7** - O Visual Studio cria o projeto. Neste caso vai criar uma aplicação e coloca o seguinte código por padrão com a instrução ***Console.WriteLine()*** para mostrar a cadeia de caracteres "Olá, Mundo!" na janela da consola:



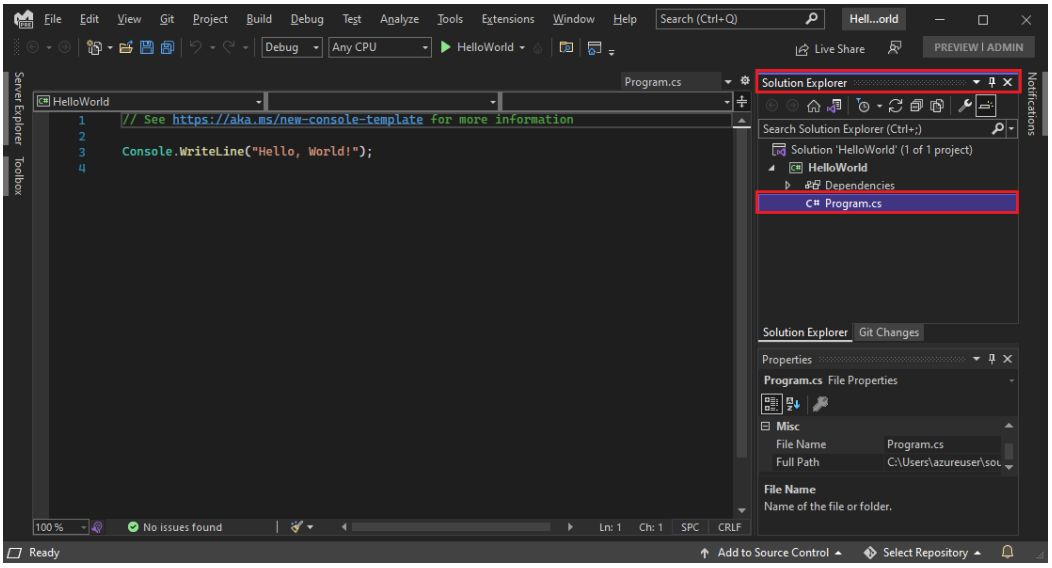

O código do C# é mostrado na janela do editor, que ocupa a maior parte do espaço. Observem que, as instruções no código (como o Console) tem cores automaticamente definidas, para indicar as diferentes partes do código, como palavras-chave e tipos.

Além disso, existem pequenas linhas verticais no código que indicam a correspondência de chaves e números de linha para assim ajudar a localizar o código posteriormente.



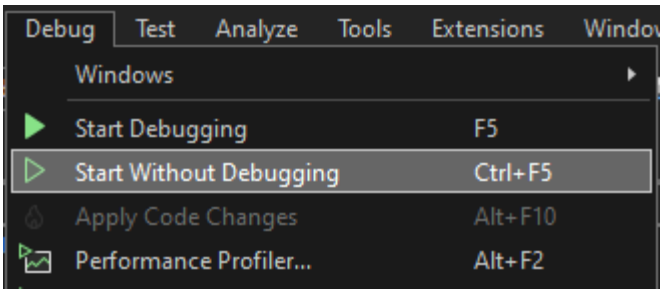

Ainda no editor de código, podemos navegar nos pequenos sinais de – ou +, para recolher ou expandir blocos de código. Este recurso de estruturação do código permite assim ocultar os blocos de código desnecessários, ajudando a minimizar o código que não é necessário.

Os arquivos de projeto estão listados no lado direito em uma janela chamada “Solution Explorer” (Diretório da aplicação). Há outros menus e outras janelas de ferramentas disponíveis, mas por enquanto, vamos seguir em frente.

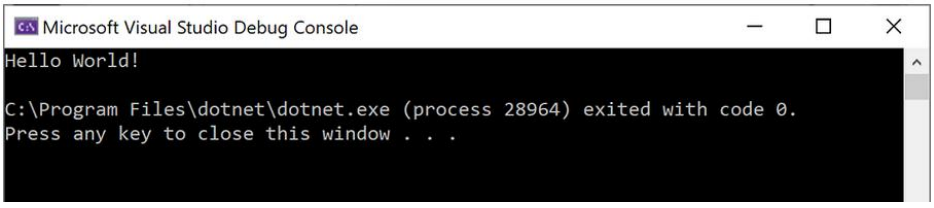



**Passo 8** – Depois de criado o novo programa, vamos agora executar o mesmo por 3 opções:

- utilizar a barra de menus, mais especificamente, a opção **Debug → Start Debugging**;
- o botão da barra de ferramentas com o sinal da seta verde;
- carregar na tecla “F5” do teclado;

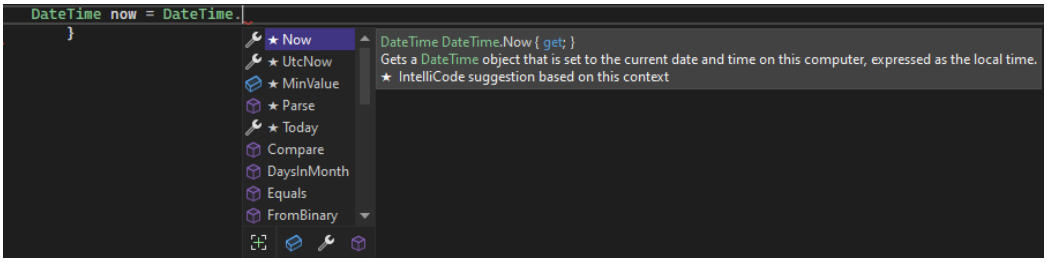



**Passo 9** - O Visual Studio compila o projeto e, caso não haja erros, executa-o. Uma vez que ainda não foi inserida nenhuma instrução e o projeto ser do tipo “Console Application”, o Visual Studio cria uma consola para executar o programa. Para fechar a janela da consola, pressione qualquer tecla do teclado.



**Nota Importante 1:** O Visual Studio utiliza a tecnologia IntelliSense que é um conjunto de recursos que exibem informações sobre o código diretamente no editor e, em alguns casos, escrevem pequenos pedaços de código ou permitem dar ajuda para procurar funcionalidades à medida que escreve código.

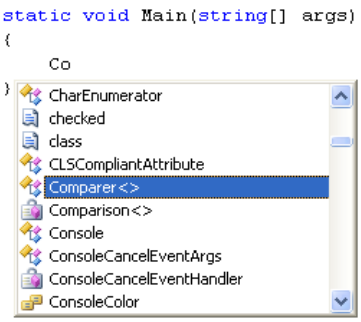
É como ter documentação básica embutida no editor, para que não precise procurar informações. A ilustração a seguir mostra como o IntelliSense exibe uma lista de membros para um tipo:




Para além disso, o editor é um ambiente de programação que ajuda imenso no processo de escrita do código permitindo, entre outras possibilidades:

- Completar a escrita de uma palavra (Word completion);
- Inserir blocos de instruções (snippet).

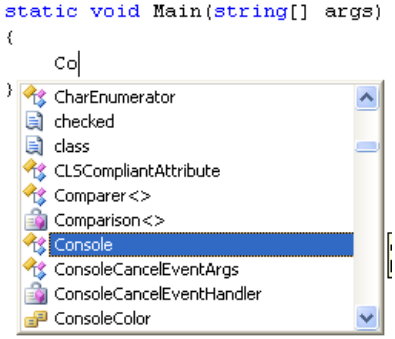
Ao iniciar a escrita com as letras “Co” o Visual Studio disponibiliza as várias palavras possíveis:



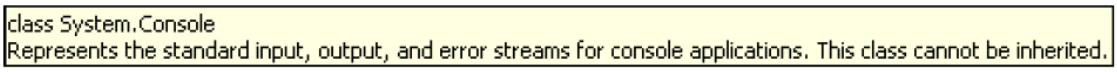

**Note que existe distinção entre maiúsculas e minúsculas (“C” e “c” são distintos).**

Se a janela de palavras não aparecer pode utilizar o botão da barra de ferramentas para a visualizar.

Com as teclas direcionais (↑ e ↓) escolher a palavra “Console”.



Ao selecionar uma palavra aparece uma pequena descrição da funcionalidade:




Com a tecla "***Tab*** ou ***Enter***" selecionar a palavra "Console".

```
static void Main(string[] args)
{
    Console
}
```

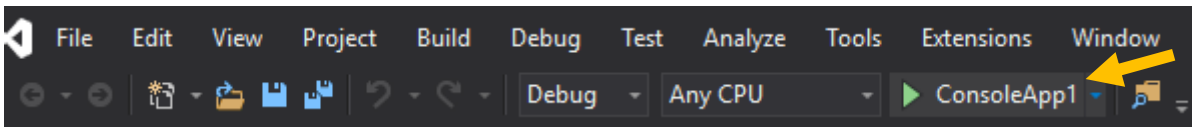
**Nota Importante 2:** Squiggles e Ações Rápidas

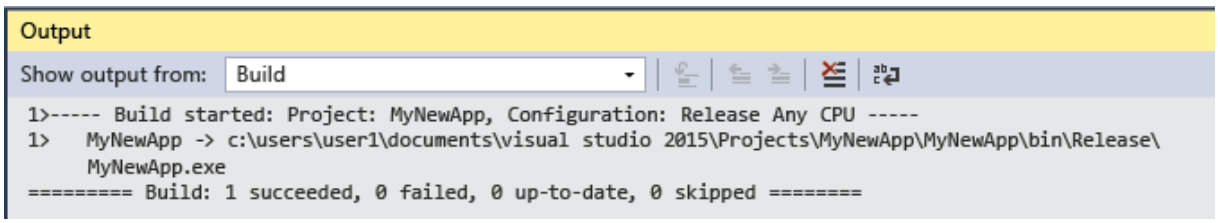
**Squiggles** são sublinhados ondulados que alertam sobre erros ou possíveis problemas no código enquanto digitamos algo. Essas pistas visuais ajudam a corrigir problemas imediatamente, sem esperar para descobrir erros durante a compilação ou o tempo de execução. Se passar o rato sobre um rabisco, verá mais informações sobre o erro. Também pode aparecer uma lâmpada na margem esquerda da linha onde está o rabisco mostrando *Ações Rápidas* que podem fazer para corrigir o erro.




**Passo 10 - Compilar código e Depurador**

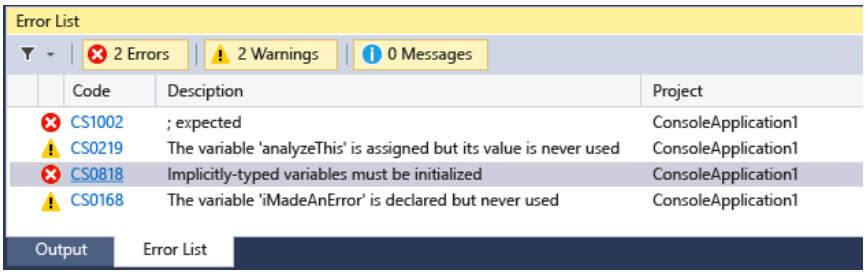
Para correr o código no Visual Studio, basta ir ao botão de Iniciar Depuração na barra de ferramentas ou carregar na tecla de atalho **F5** (**Depurar > Iniciar Depuração**)



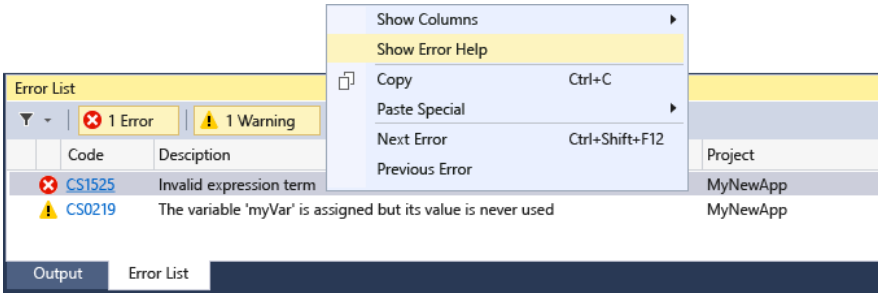
- ***Compilar código e Depurador***

Se ao compilar o código de um programa houver um erro a informar que a aplicação não pode arrancar, isto será sinal que existe algum tipo de erro. Para tal devemos inspecionar a lista de erros que aparece mais abaixo, mais especificamente na caixa ***Error List***.



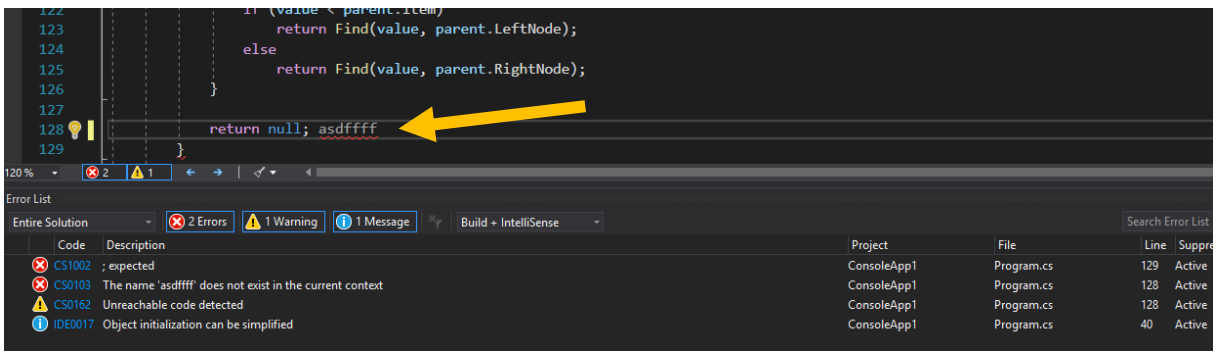

Por vezes os erros são muito fáceis de identificar, como por exemplo a falta de um ponto e virgula (;) ou até mesmo um nome de variável incorreto ou mal associado.

Por outro lado, também pode haver erros mais complexos (com código confuso ou mal escrito. Para mostrar o detalhe do erro temos de ***clicar no código do erro*** para ver o erro mais detalhadamente.



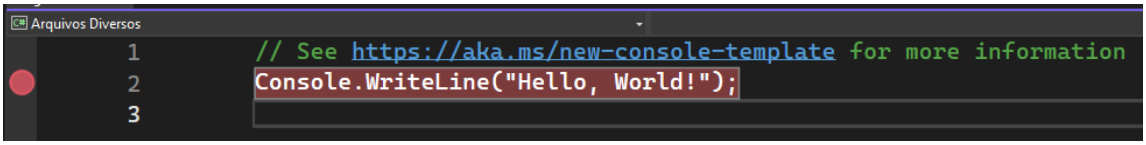

- ***Compilar código e Depurador - Usar a análise de código***

Se houver algum tipo de erro aquando digitamos o código na aplicação, estas serão facilmente detetadas na Lista de Erros e também no editor do código, mais especificamente com um rabisco a vermelho.

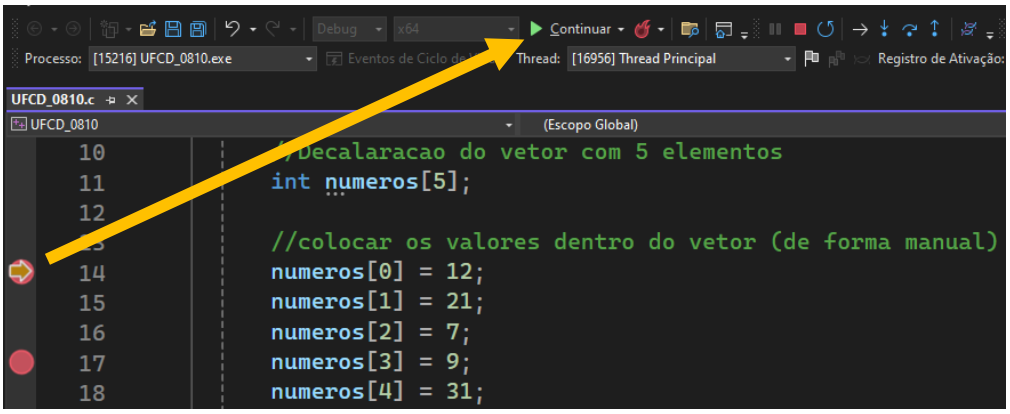


- ***Compilar código e Depurador – Pontos de paragem e inspeção de código***

  Se pretender parar o código num determinado ponto para verificar e inspecionar o que está a passar na linha de código (ou antes de lá chegar), podemos colocar pontos de paragem. Para fazer isso, basta clicar com o lado esquerdo do rato atrás do número da linha que deseja parar o programa quando este estiver a correr:

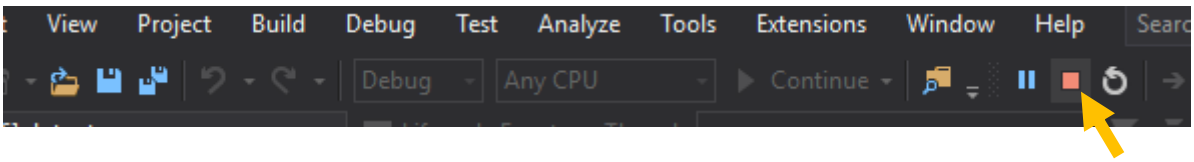



  Feito isso, quando o programa chegar a este ponto de paragem, o programador, pode analisar o que está a acontecer no programa e só depois disso, deve dar ordem para continuar a próxima execução do programa. Quando mais pontos tiver, mais paragens irá efetuar:




- ***Parar Depurador***

  Para parar a depuração **basta clicar no botão do Stop na farra de ferramentas** ou clicar na combinação de teclas ***SHIFT + F5***

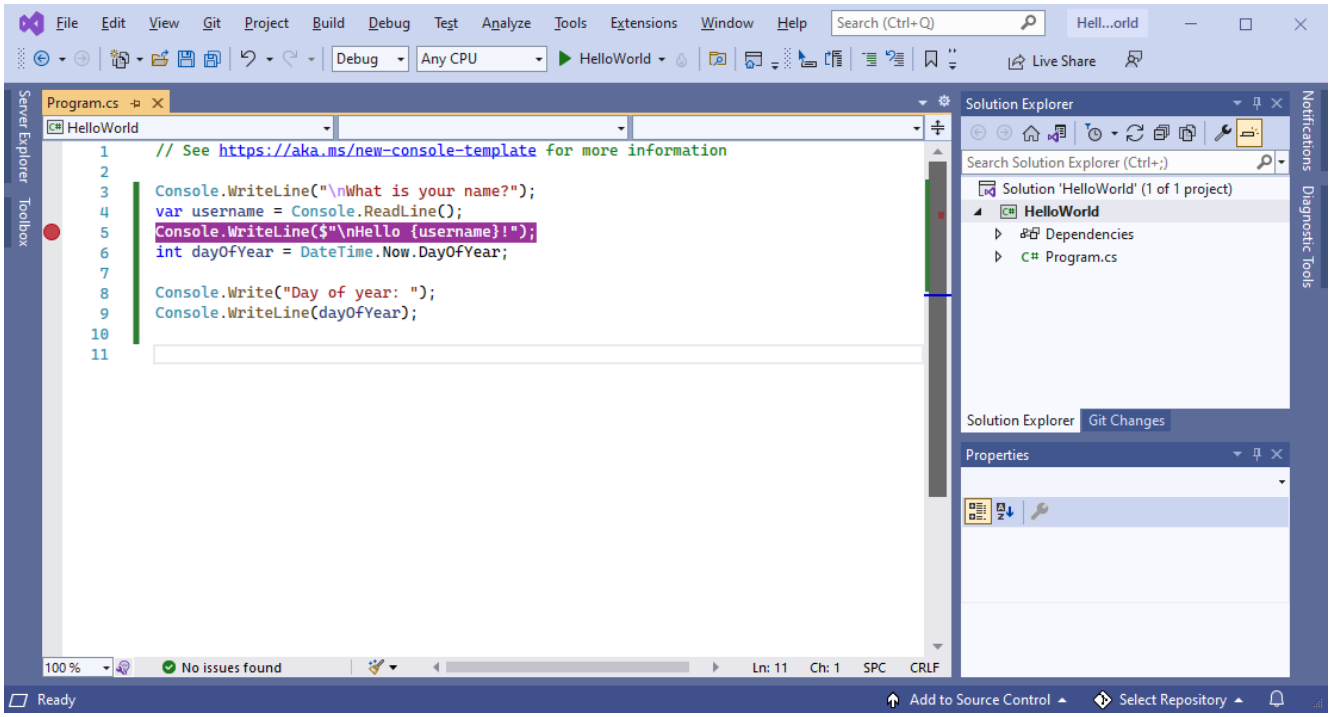


### Passo 11 - Personalizar o Visual Studio

Podem personalizar a interface do usuário do Visual Studio, incluindo a alteração do tema de cores padrão. Para alterar o tema de cores:

1. Na barra de menus, escolha Ferramentas → Opções para abrir a caixa de diálogo Opções.
2. Na página de opções Ambiente Geral>, altere a seleção Tema de Cores para Azul ou Claro e, em seguida, selecione OK.

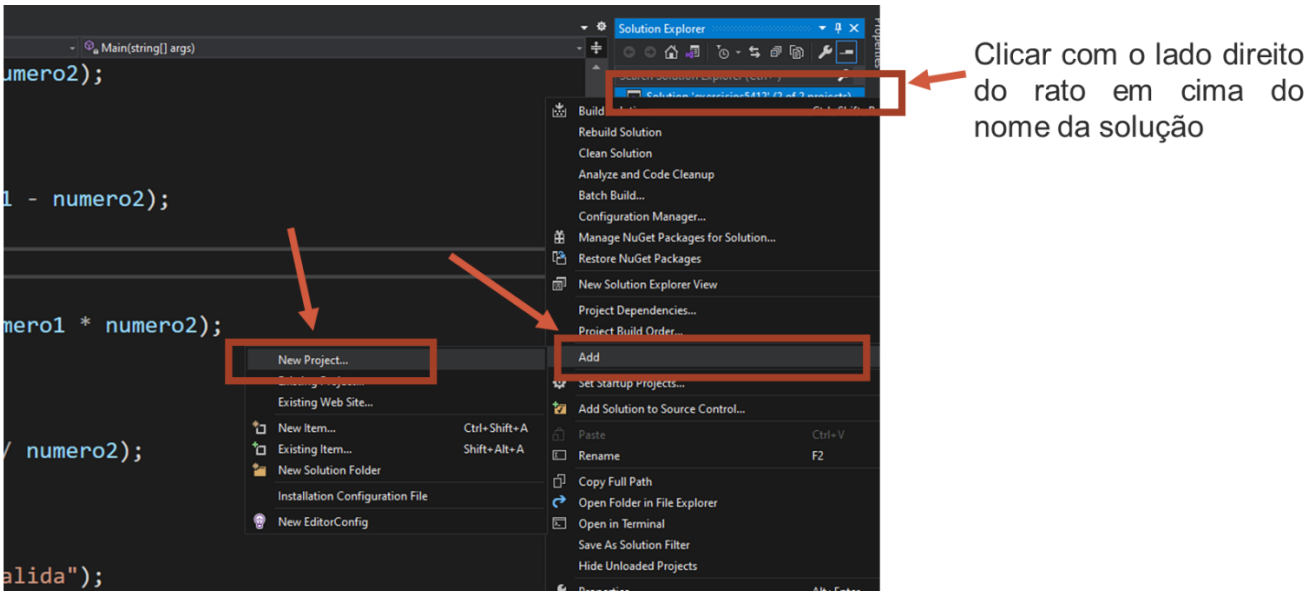
   O tema de cores para todo o IDE muda de acordo. A captura de tela a seguir mostra o tema Azul:



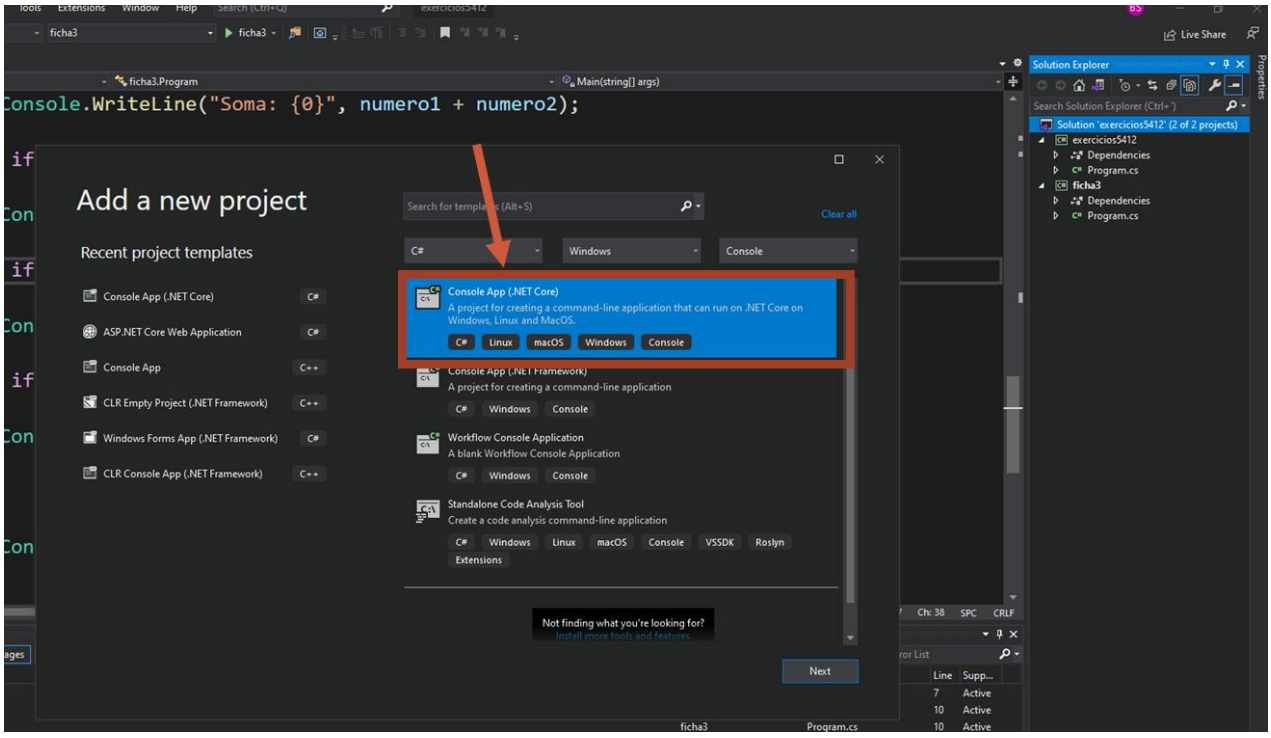
# Capítulo 6 – Criação de múltiplos projetos em Visual Studio

Para fazer a criação de múltiplos projetos na ferramenta IDE do Visual Studio fazemos os seguintes passos:

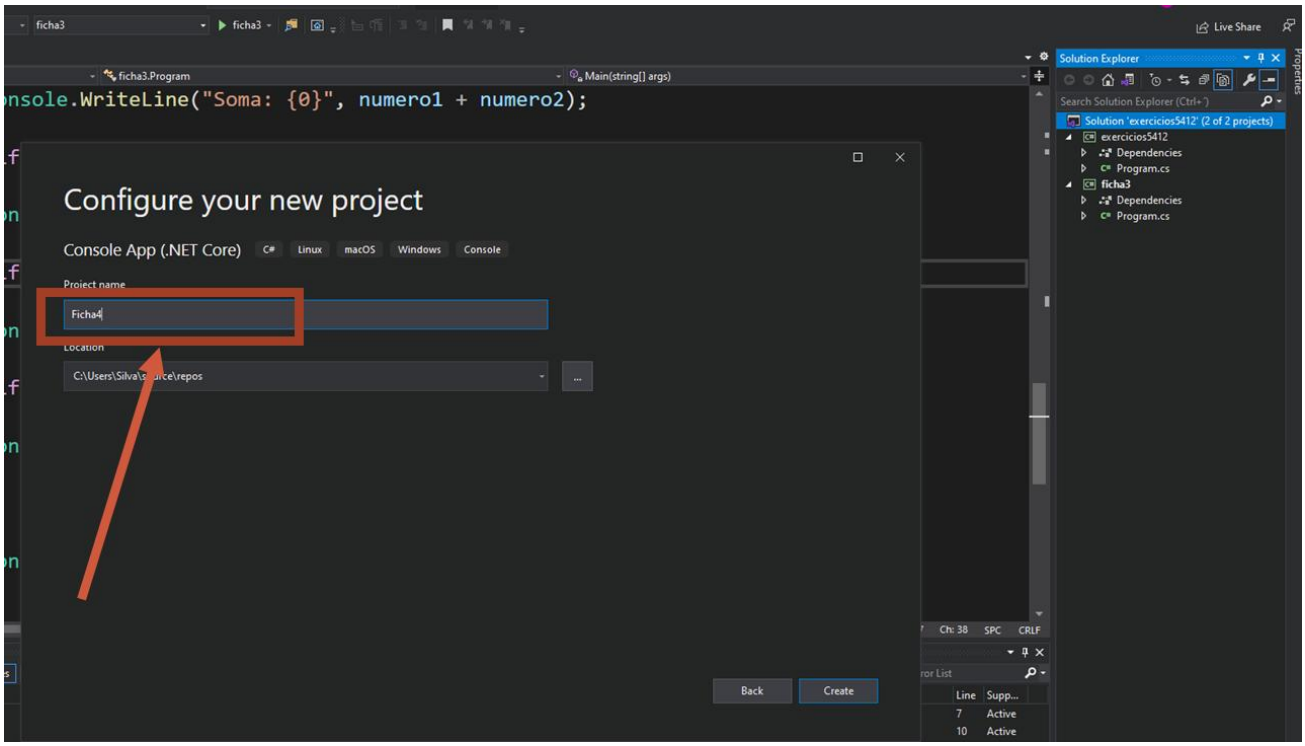
**Passo 1 –** Adicionar de um novo projeto dentro de outro:



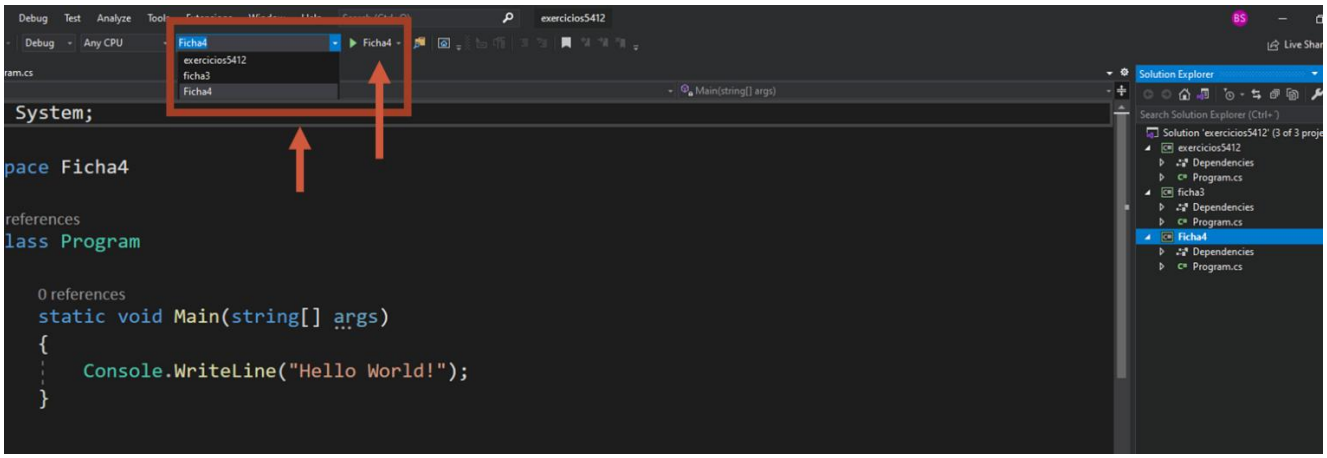

**Passo 2 –** Adicionar de um novo projeto C# do tipo consola:



**Passo 3 –** Atribuir um novo nome ao projeto:




**Passo 4 –** Ativar qual o projeto a executar (Depuração do código):



# Capítulo 7 – Introdução à programação C#

## Capítulo 7.1 – Instruções de escrita de valores na consola

As instruções de escrita correspondem, na linguagem C#, aos métodos da classe Console (que representa o ecrã e o teclado): Console.Write ou Console.WriteLine.

Como tal, vamos verificar **2 exemplos/formas de escrita de mensagens na consola**:

**Forma 1 - Escrita de mensagens sem quebra de linha (Console.Write)**

```
//Forma 1 - Escrita de mensagens (sem quebra de linha))
Console.Write("Isto é um exemplo de uma mensagem sem quebra de linha");
Console.Write("Mas, como podem ver, não houver quebra de linha");
```

```
Console de Depuração do Microsoft Visual Studio
Isto é um exemplo de uma mensagem sem quebra de linhaMas, como podem ver, não houver quebra de linha
```

**Forma 2 - Escrita de mensagens com quebra de linha (Console.WriteLine)**

```
//Forma 2 - Escrita de mensagens (com quebra de linha)
Console.WriteLine("Isto é um exemplo de uma mensagem com quebra de linha");
Console.WriteLine("Mas, como podem ver, houve quebra de linha");
```

```
Console de Depuração do Microsoft Visual Studio
Isto é um exemplo de uma mensagem com quebra de linha
Mas, como podem ver, houve quebra de linha
```

## Capítulo 7.1.1 – Sequências de Escape

As sequências de escape permitem fazer transformações no texto, como por exemplo, criar quebras de linha (/n), fazer tabulações no texto (/t), emitir som no computador (/a), entre outras opções:

**Sequências de Escape**

As sequências de escape podem ser incluídas na cadeia de caracteres.

| SEQUÊNCIA | DESCRIÇÃO |
|---|---|
| \' | Plica |
| \" | Aspa |
| \\ | Barra para trás |
| \0 | Valor nulo |
| \a | Som (*beep*) |
| \b | Um espaço para trás (*backspace*) |
| \f | Mudança de página (*form feed*) |
| \n | Nova linha |
| \r | Retornar ao início da linha (*carriage return*-CR) |
| \t | Tabulação horizontal |
| \v | Tabulação vertical |

```
//Outra forma de quebra de linha (instruções de escape)
Console.WriteLine("Exemplo de uma quebra de linha \n Veja o resultado");
```

```
Console de Depuração do Microsoft Visual Studio
Exemplo de uma quebra de linha
 Veja o resultado
```

## Capítulo 7.2 – Tipos de Dados e as Variáveis

Os tipos de dados são representações do valor do qual queremos trabalhar. Vamos ver os exemplos de tipos de dados mais usados:



Uma variável representa a área de memória que contém um valor dum determinado tipo de dados. A cada variável estão associados ***um tipo de dados*** e ***um nome***.

Para associar um determinado tipo de dados a uma dada variável, é necessário proceder-se à declaração da variável.

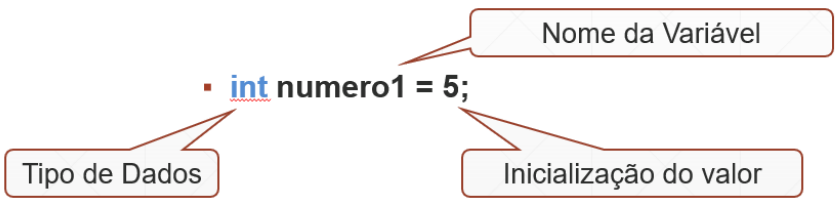



## Capítulo 7.2.1 – Constantes e Variáveis

**Uma variável** representa a área de memória que contém um valor dum determinado tipo de dados. A cada variável estão associados ***um tipo de dados*** e ***um nome***. Para associar um determinado tipo de dados a uma dada variável, é necessário proceder-se à declaração da variável.

Uma variável, deve ser declarada antes de ser utilizada, podendo ser inicializada imediatamente. Escreve-se o ***tipo de dados*** da variável/constante, seguido do nome para identificação e a inicialização da mesma.

***Exemplo***: Quero criar uma variável do tipo inteiro e para guardar números inteiros e iniciar com o número 5.



**Exemplo de criação de variáveis**

```
//declarar uma variável do tipo inteiro
int num1 = 5;

//declarar uma variável do tipo decimal
float num2 = 2.3; //permite até 8 casas decimais
double num3 = 5.11; //permite até 16 casas decimais

//declarar uma variável do tipo caracter (utilize aspa simples para colocar o caracter)
string texto = 'a'

//declarar uma variável do tipo textual (utilize aspas para colocar o texto)
string texto = "Olá Mundo"

//também pode declarar múltiplas variáveis ao mesmo tempo (agrupados pelo mesmo tipo de dados)
int numero_1 = 0, numero_2 = 5;
double nota1 = 0, nota2 =0, nota3 = 0;
string texto_1 = "Exemplo", texto_2 = "texto";
```

**Mas até agora apenas declaramos variáveis. Como podemos exibir as variáveis com o método de escrita de mensagens ( Console.Write() ou Consol.WriteLine() ) ?**

## Capítulo 7.2.2 – Escrita de mensagens (com a inclusão das variáveis)

**Forma 1 - Escrita de mensagens (com $ e chavetas {} )**

Nesta última forma, como queremos indicar a combinação de texto com variáveis, temos que, escrever o **símbolo $** atrás das aspas **para indicar que vamos combinar a escrita de texto** (que fica com a cor laranja) **com variáveis** que devem ficar **dentro de chavetas curvas { }**:

```
//Forma 3 - Escrita de mensagens (com variáveis incluídas)
int numero = 5;
Console.WriteLine($"Valor da variavel número: {numero}");
```

```
Console de Depuração do Microsoft Visual Studio
Valor da variavel número: 5
```

**Outro exemplo (mostrar/combinar mais do que uma variável ao mesmo tempo):**

```
string primeiro_nome = "Maria";
string ultimo_nome = "Sage";
Console.WriteLine($"Primeiro nome é: {primeiro_nome} e o último é: {ultimo_nome}");
```

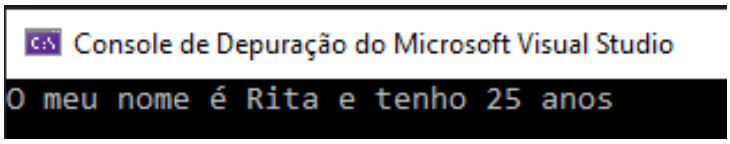
**Forma 2 - Escrita de mensagens (por concatenações – símbolo +)**

Também pode usar **outra forma de escrita** de mensagens (como feito na linguagem Java e entre outras), que passa por **colocar concatenações com o símbolo +** (mais):

```
int idade = 25;
string nome = "Rita";

//mensagem por concatenação
Console.WriteLine("O meu nome é " + nome + " e tenho " + idade + " anos");
```

**Resultado:**




**Exemplo da combinação de várias variáveis com o método Console.WriteLine()**

```
//Variáveis: são os contentores para guardar informação

//Temos de indicar qual o tipo do valor e dar um nome a variável
//Exemplos:

//variável para guradar números inteiros (int)
int numero = 5;
Console.WriteLine($"Valor da variavel número: {numero}");

//variável para guardar números decimais (double)
double nota = 14.5;
Console.WriteLine($"Valor da variavel nota: {nota}");

//variável para guardar apenas um caracter (char)
char opcao = 'a';
Console.WriteLine($"Valor da variavel opção: {opcao}");

//variável para guardar uma cadeia de caracteres (string)
string texto = "Hoje está um dia de sol!";
Console.WriteLine($"Valor da variavel texto: {texto}");

//variável do tipo booleano (apenas 1 de dois estados: verdadeiro ou falso)
bool estado = true;
Console.WriteLine($"Valor da variavel estado: {estado}");

//exemplo de declaração de múltiplas variaveis com o mesmo tipo de dados
int a = 5, b = 7, c = 1; //cada variavel o respetivo valor
Console.WriteLine($"Valor de a: {a}, valor de b: {b} e valor de c: {c}");
```

```
Console de Depuração do Microsoft Visual Studio
Valor da variavel número: 5
Valor da variavel nota: 14,5
Valor da variavel opção: a
Valor da variavel texto: Hoje está um dia de sol!
Valor da variavel estado: True
Valor de a: 5, valor de b: 7 e valor de c: 1
```



## Capítulo 7.3 – Instruções leitura de valores (a partir do teclado)

As instruções de escrita correspondem, na linguagem C#, aos métodos da classe **Console.ReadLine()** (ler valores através do teclado).

**Exemplo 1 – Ler um valor do teclado com a função Console.ReadLine()**

```
//Receber um valor Textual
Console.WriteLine("Insira um valor textual: ");
valor_texto = Console.ReadLine();
```

Mas, **quando lemos um valor do teclado, este vem sempre no formato textual**.

Quando é pretendido **ler um dado do tipo numérico** (inteiro ou decimal) em C#, devemos sempre **converter o que recebemos do teclado** de texto para numérico.

Como tal, vão verificar **2 formas de conversão** de texto para numérico:

- **Conversão para inteiros:** Convert.ToInt32

```
//Ler dados do teclado
int valor_inteiro = 0;

//Receber valor inteiro
Console.WriteLine("Insira um valor inteiro: ");
valor_inteiro = Convert.ToInt32(Console.ReadLine());
```

- **Conversão para decimais:** Convert.ToDouble

```
//Ler dados do teclado
double valor_decimal = 0;

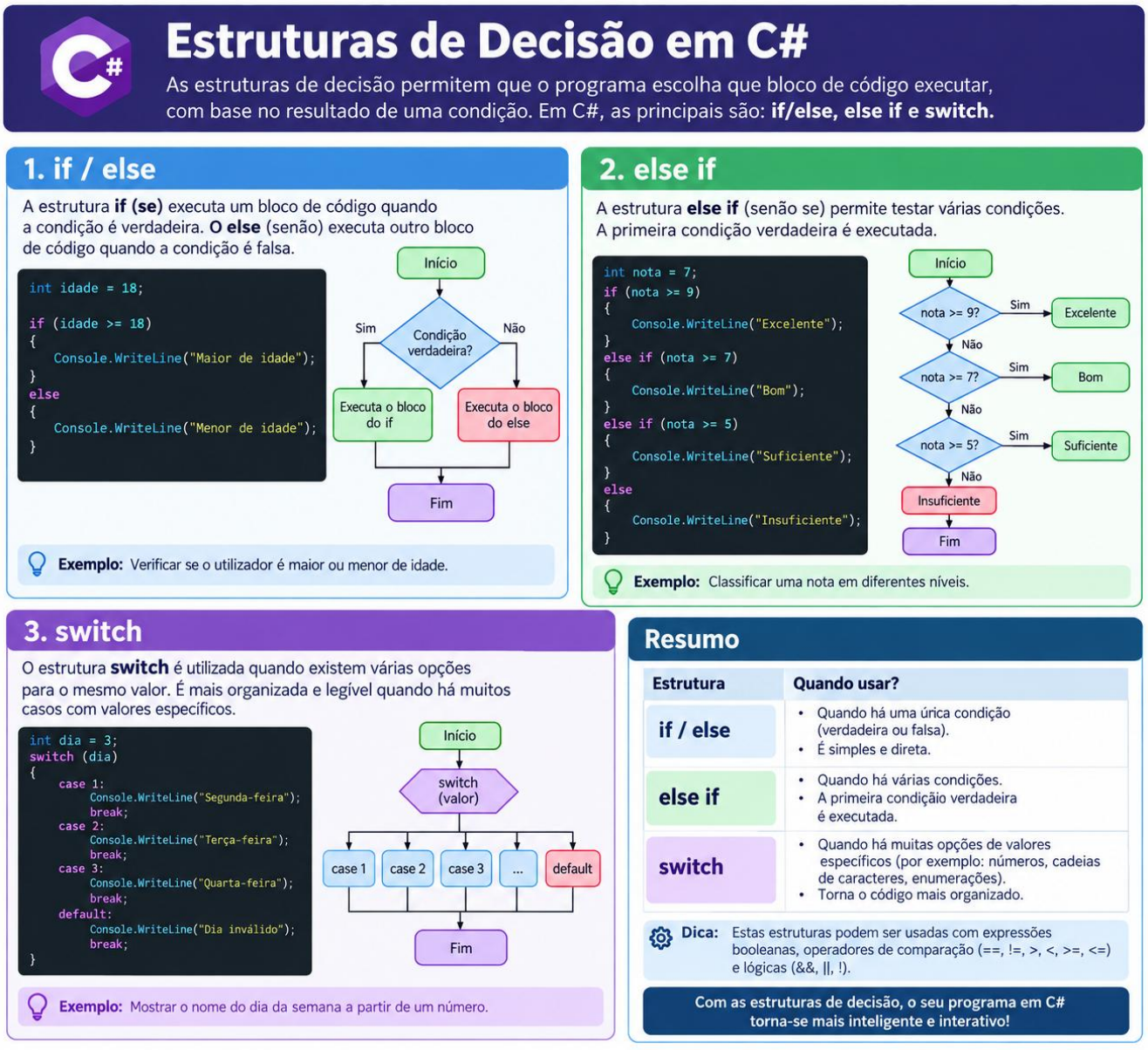
//Receber valor decimal
Console.WriteLine("Insira um valor decimal: ");
valor_decimal = Convert.ToDouble(Console.ReadLine());
```

# Capítulo 8 – Estruturas de decisão

Existem vários tipos de estruturas de controlo, cada uma das quais com comportamentos distintos em termos da sequência de execução das diversas instruções dentro dum programa. Estas condicionam a ordem pela qual as várias instruções serão executadas no programa, tal como vimos na descrição algorítmica.

Neste capítulo, vamos falar sobre as estruturas alternativas, mais especificamente:

- *Estruturas de decisão Simples / Dupla ( if … else )*
- *Estruturas de decisão Dupla Encadeada ( if … else if … else )*
- *Estruturas de decisão Seletiva ( switch e cases )*



## Capítulo 8.1 – Estrutura de decisão simples

Nesta estrutura de controlo, a instrução ou bloco de instruções serão executadas ***apenas se a condição testada for verdadeira***. Caso seja, a instrução ou bloco de instruções que serão executadas devem estar ***dentro de chavetas*** { código… } :

```
If (nota >= 9.5) { /*inicio do bloco de instruções*/
    Console.WriteLine("Nota positiva");
} /*fim do bloco de instruções*/
```

```
int nota = 10;

if(nota >= 9.5)
{
    Console.WriteLine("Nota Positiva!");
}
```

```
Microsoft Visual Studio Debug Console
Nota Positiva!
```

Também podemos conjugar condições, operadores aritméticos e operadores lógicos na estrutura de decisão condicional. Neste caso vamos utilizar o operador lógico ***&& (e)*** do qual obriga que as 2 condições têm de ser verdadeiras (basta 1 falhar para a condição ser falsa):

```
if (nota1 >= 9.5 && nota2 >= 9.5) {
    Console.WriteLine("Notas positivas");
}
```

Também podemos conjugar condições, operadores aritméticos e operadores lógicos na estrutura de decisão condicional. Neste caso vamos utilizar o operador lógico **|| *(ou)*** onde basta uma das 2 condições ser verdadeiras:

```
if (nota1 >= 9.5 || nota2 >= 9.5) {
        Console.WriteLine("Notas positivas");
}
```

```
static void Main(string[] args) {

    int nota = 10;

    if(nota >= 9.5)
    {
        Console.WriteLine("Nota Positiva!");
    }

}
```

```
Microsoft Visual Studio Debug Console
Nota Positiva!
```

## Capítulo 8.2 – Estrutura de decisão dupla

Nesta estrutura se a condição testada for verdadeira, será executado o bloco de instruções do caso verdadeiro, ***caso contrário***, será executado o bloco de instruções do caso falso.

As instruções ou blocos de instruções que serão executadas devem estar ***dentro de chavetas*** { código… }

```
If (nota >= 9.5) { /*inicio do bloco de instruções do caso verdadeiro*/
    Console.WriteLine("Nota positiva");
} /*fim do bloco de instruções do caso verdadeiro */
else { /*inicio do bloco de instruções do caso falso*/
    Console.WriteLine("Nota negativa");
} /*fim do bloco de instruções do caso falso */
```

```
int nota = 10;

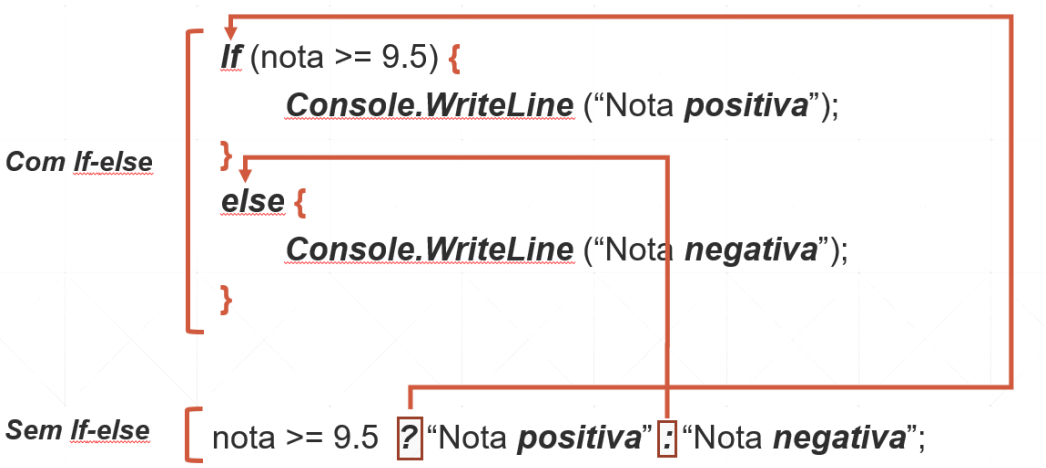
if(nota >= 9.5)
{
    Console.WriteLine("Nota Positiva!");
}
else
{
    Console.WriteLine("Nota Negativa!");
}
```

```
Microsoft Visual Studio Debug Console
Nota Positiva!
```

Também podemos por a estrutura de decisão sem ***if-else*** ou seja, utilizar a **estrutura trenária**. Diferente da instrução condicional ***if-else***, pode-se usar em qualquer sítio onde se espera uma expressão.

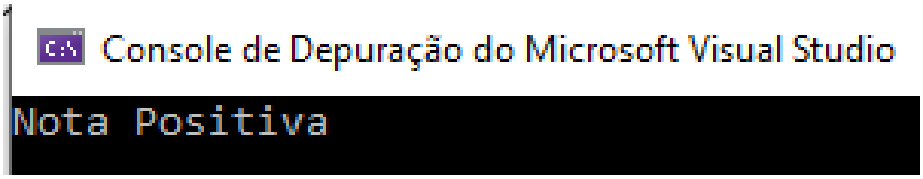
***expr1 ? expr2 : expr3***

- Assume ***expr2*** se ***expr1*** tiver um valor diferente de zero (verdade), e ***expr3*** no caso contrário (só avalia as expressões necessárias)
- ***"? :"*** é um operador ternário




```
int nota = 12;

Console.WriteLine(nota > 9.5 ? "Nota Positiva" : "Nota Negativa");
```



## Capítulo 8.3 – Estrutura de decisão dupla encadeada (IF … Else if … Else)

Estas decisões encontram-se limitadas ao resultado lógico da sua condição, pelo que ***podemos ter duas ou mais alternativas de decisão***, ou seja, optar entre ***duas ou mais instruções alternativas***.

As instruções ou blocos de instruções que serão executadas devem estar ***dentro de chavetas*** { código… }

```
Se (nota >= 0 && nota < 9.5) {
    Console.WriteLine("Nota negativa");
}
else if (nota >= 9.5 && nota < 20) {
    Console.WriteLine("Nota positiva");
}
else {
    Console.WriteLine("Nota inválida");
}
```

```
int nota = 14;

//Estrutura de decisão dupla encadeada

if (nota > 0 && nota < 9.5)
{
    Console.WriteLine("Nota negativa");
}
else if (nota >= 9.5 && nota <= 20)
{
    Console.WriteLine("Nota positiva");
}
else
{
    Console.WriteLine("Nota invalida!");
}
```

## Capítulo 8.4 – Estrutura de decisão seletiva (Switch … Case)

Utilizamos esta solução quando é necessário decidir entre **múltiplas** opções possíveis para a instrução ou bloco de instruções a executar. Torna-se assim ideal para criar menus de escolhas alternativas:

***switch* (*expressão*) {** /*inicio da instrução */
***case 1*: *instruções***
***case 2:* *instruções***
***(...)***
***default*: *instruções***
**}** /*fim da instrução */

***Regras do Switch:***

1. A estrutura ***Switch*** (Caso) inicia a execução pela avaliação da expressão do caso.
2. Depois de calculada a expressão do ***Switch***, o controlo é transferido para o ***caso mais apropriado***, por comparação do resultado da expressão do ***Switch***.
3. Se não houver nenhum caso que satisfaça a condição do ***Switch***, o case ***default*** será executado automaticamente.

***switch* (*opcaoMenu*) {** /*inicio da instrução */

***case 1*: *Console.WriteLine (“*Escolheu a opção 1!*”); break;***

**case *2*: *Console.WriteLine(“*Escolheu a opção 2!*”); break;***

***case x*: *Console.WriteLine(“*Escolheu a opção x!*”); break;***

**default: *Console.WriteLine(“*Essa opção não existe*”); break;***

**}** /*fim da instrução */

```
//Estrutura de decisão seletiva
int posicao = 0;

//pedir dados ao utilizador
Console.Write("Qual a posição que chegou: ");
posicao = Convert.ToInt32(Console.ReadLine());

//estrutura de decisão seletiva
switch (posicao)
{
    case 1: Console.WriteLine("Medalha de Ouro"); break;
    case 2: Console.WriteLine("Medalha de Prata"); break;
    case 3: Console.WriteLine("Medalha de Bronze"); break;
    default: Console.WriteLine("Folha de Participação"); break;
}
```

```
Console de Depuração do Microsoft Visual Studio
Qual a posição que chegou: 2
Medalha de Prata
```

# Capítulo 9 – Estruturas de repetição

Existem vários tipos de estruturas de repetição, cada uma das quais com comportamentos distintos em termos da sequência de execução das diversas instruções dentro dum programa. Estas condicionam a ordem pela qual as várias instruções serão executadas no programa, tal como vimos na descrição algorítmica.

Neste capítulo, vamos falar sobre as estruturas repetitivas, mais especificamente:

- ***Estrutura de repetição com contador automático:***
    - Implementação de estruturas de repetição ***For*** (Para);
- **Estrutura de repetição com contador manual:**
    - Implementação de estruturas de repetição ***While*** (Enquanto);
    - Implementação de estruturas de repetição ***Do .. While*** (Faz ... Enquanto);

## Capítulo 9.1 – Estruturas de Repetição FOR (contador automático)

Esta estrutura permite a ***repetição duma determinada instrução ou bloco de instruções***, durante ***um número de vezes predefinido no código*** e é a única repetição de forma ***automática***.

Por defeito, **cada repetição** corresponde **a um ciclo de instruções**, mas pode ser modificado (caso necessário).

### *Método de construção da estrutura de repetição FOR*

Esta estrutura deverá ter 3 parâmetros:

***FOR( <condição de inicio>; <condição de fim>; <ação no final do ciclo> )***

- ***condição de inicio -»*** iniciar a variável
- ***condição de fim -»*** qual a condição que irá satisfazer a condição
- ***ação no final do ciclo -»*** normalmente vamos dizer o que acontece a variável da condição de inicio: ***incrementa*** ou ***decrementa;***
- ***Exemplo Prático:***

```
for (int contador = 1; contador <= 10; contador++)
{
    // calcula a soma
    soma = soma + contador;
}
```

***Exemplo 1:*** Percorrer todos os números de 1 até 9:

```
//Estrutura de repetição FOR para percorrer os valores
for (int i = 1; i < 10; i++)
{
    Console.WriteLine($"Valor da variavel i = {i}");
}
```

```
Console de Depuração do Micro
Valor da variavel i = 1
Valor da variavel i = 2
Valor da variavel i = 3
Valor da variavel i = 4
Valor da variavel i = 5
Valor da variavel i = 6
Valor da variavel i = 7
Valor da variavel i = 8
Valor da variavel i = 9
```

***Exemplo 2:*** Somar todos os números de 1 a 5 (inclusive) de forma automática

Neste exemplo, quando começamos a estrutura de repetição, vamos ter ***2 valores*** em ter em conta e que serão determinantes para o ciclo do programa:

- ***a soma*** (pois será o nosso ***acumulador*** das somas);
- ***o contador*** (***variável i***, pois devemos dar ***+1*** ao contador de cada vez que percorre a estrutura de repetição, ou seja, de cada vez que corre o ciclo da estrutura de repetição);

```
for( int contador=1; contador <= 5; contador++){
    soma = soma + contador;
}
```

```
//Estrutura de repetição FOR para somar todos os valores
//de 1 a 5 (inclusive)
int soma = 0;

for (int i = 1; i <= 5; i++)
{
    soma = soma + i;
}

Console.WriteLine($"Soma Final = {soma}");
```

```
Console de Depuração do Microsoft Visual Studio
Soma Final = 15
```

# Capítulo 9.2 – Estruturas de Repetição While e Do … While (contador automático)

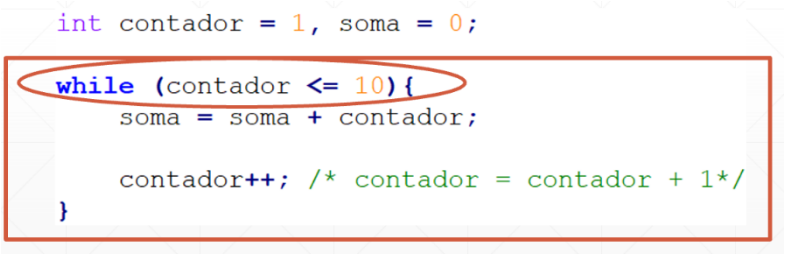
Estas diferem da estrutura de repetição FOR, porque utilizam uma condição manual. Dentro do bloco de instruções ***terá que ocorrer uma alteração que interfira com a condição***, caso contrário, ***entraria em ciclo infinito***.

Temos dois tipos:

- ***Enquanto*** *(While)*
- ***Faz … Enquanto*** *(Do … While)*

## Capítulo 9.2.1 – Estruturas de Repetição WHILE (contador manual)

O teste à condição é realizado ***antes da execução de qualquer instrução contida numa estrutura de repetição*** e enquanto a ***condição for verdadeira***, esta volta a executar a instrução ou bloco de instruções, caso contrário, este termina e sai do ciclo.



```
int contador = 1, soma = 0;

while (contador <= 10){
    soma = soma + contador;

    contador++; /* contador = contador + 1*/
}
```

***Exemplo da soma de todos os números de 1 a 10 com WHILE:***

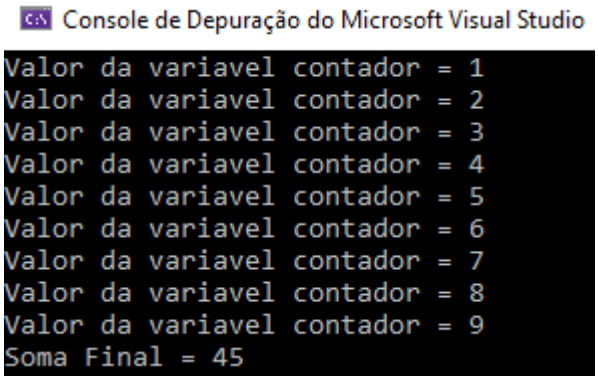
```
//estrutura de reptição while
int contador = 1;
int soma = 0;

while (contador < 10)
{
    Console.WriteLine($"Valor da variavel contador = {contador}");

    //calculo da soma
    soma = soma + contador;

    //gestão do contador de forma manual (caso contrário entra num ciclo infinito)
    contador++;
}

Console.WriteLine($"Soma Final = {soma}");
```




## Capítulo 9.2.2 – Estruturas de Repetição DO …WHILE (contador manual)

O teste à condição ***só é realizado depois de executar as instruções*** contidas numa estrutura de repetição e ***enquanto a condição for verdadeira***, volta a executar a instrução ou bloco de instruções, caso contrário, termina e sai do ciclo.

```
int contador = 1, soma = 0;

do{ /* vai funcionar sempre 1 vez, pois a condição está no fim */
    soma = soma + contador;

    contador++; /* contador = contador + 1*/
}while (contador <= 10) /*só testa a condição no final*/
```

***Exemplo da soma de todos os números de 1 a 10 com Do ... While:***

```
//estrutura de reptição do ... while
int contador = 1;
int soma = 0;

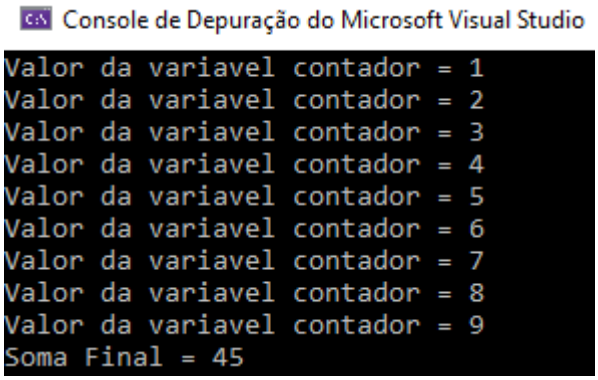
do
{
    Console.WriteLine($"Valor da variavel contador = {contador}");

    //calculo da soma
    soma = soma + contador;

    //gestão do contador de forma manual (caso contrário entra num ciclo infinito)
    contador++;

} while (contador < 10);

Console.WriteLine($"Soma Final = {soma}");
```




Este tipo de estrutura de repetição é ideal para mostrar menus, ou seja, **faz** mostrar os menus **enquanto** o utilizador não coloca uma opção válida. O objetivo é **fazer** mostrar os menus **enquanto** o utilizador não carregue na tecla 4:

```
//menu de escolha
int escolha = 0;

do
{
    Console.WriteLine("***** MENU *****");
    Console.WriteLine("1 - Inserir Dados");
    Console.WriteLine("2 - Alterar Dados");
    Console.WriteLine("3 - Eliminar Dados");
    Console.WriteLine("4 - Sair");

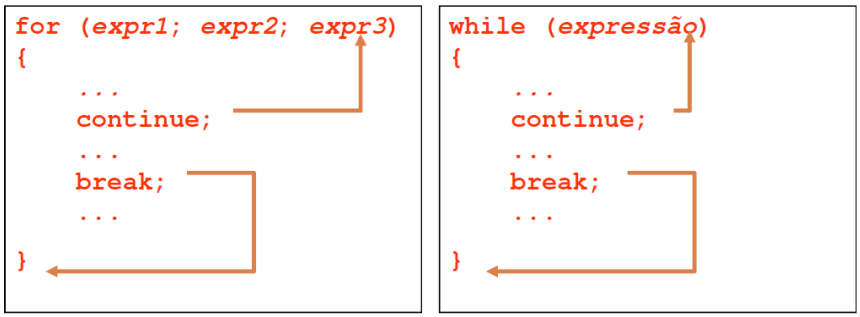
    Console.WriteLine("Escolha opção: ");
    escolha = Convert.ToInt32(Console.ReadLine());

    switch (escolha)
    {
        case 1: Console.WriteLine("Escolheu a opção inserir dados"); break;
        case 2: Console.WriteLine("Escolheu a opção alterar dados"); break;
        case 3: Console.WriteLine("Escolheu a opção eliminar dados"); break;
        case 4: Environment.Exit(0); break;

        default: Console.WriteLine("Opção Inválida"); break;
    }
} while (escolha != 4);
```

## Capítulo 9.2.3 – Instruções Break e Continue

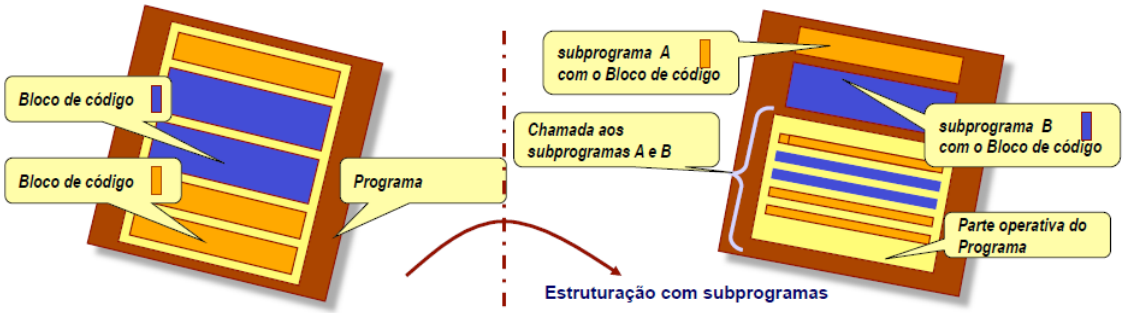
- ***break*** **-** sai do ciclo ou ***switch*** envolvente (mais interior)
- ***continue*** - salta para a iteração seguinte do ciclo envolvente (mais interior) no caso do ciclo for (expr1; expr2; expr3) ***salta para expr3***
    - no caso do ciclo ***while*** ou ***do (…) while*** salta para a expressão-argumento do ***while***.




# Resumo:

# Capítulo 10 – Subprogramas

Os subprogramas ajudam nesse ponto e têm uma solução mais simples permitindo assim reduzir trabalho e poupar linhas de código, em vez de repetir funções e procedimentos desnecessariamente.




Um programa corresponde à implementação de uma solução para um determinado problema. Mas, muitas vezes, os problemas são demasiado complexos.

Para tal, podemos dividir um problema grande em porções mais pequenas, por forma a facilitar a descoberta da solução.

## Capítulo 10.1 – Exemplo de código sem subprogramas vs com subprogramas

```
static void Main(string[] args){
    int resultado = 0;

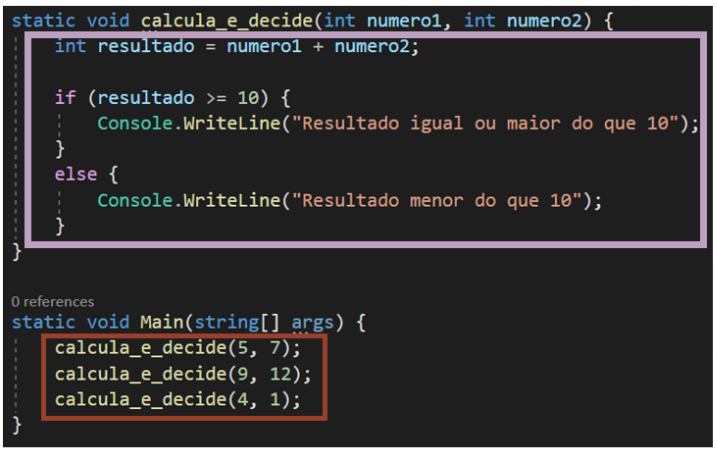
    resultado = 5 + 7;

    if (resultado >= 10) {
        Console.WriteLine("Resultado igual ou maior do que 10");
    }
    else{
        Console.WriteLine("Resultado menor do que 10");
    }

    resultado = 9 + 12;

    if (resultado >= 10) {
        Console.WriteLine("Resultado igual ou maior do que 10");
    }
    else{
        Console.WriteLine("Resultado menor do que 10");
    }
}
```

- Estamos a ***repetir*** a operação de ***soma*** e ***estrutura de decisão***.
- Se continuarmos com esta situação, ***isto vai ficar muito grande e confuso*** ☹




- O código repetido foi agrupado numa forma genérica e com a mesma tarefas;
- Contribuem para a resolução do problema geral e ***sem repetições desnecessárias***;

## Capítulo 10.2 – Vantagens da utilização dos subprogramas

- ***Clareza de código***: Criam-se secções do programa que executam tarefas específicas, com um conjunto de instruções a cumprir muito bem definido;
- ***Reutilização***: podemos usar o código devolvido em funções mais do que uma vez;
- ***Independência:*** uma função é relativamente independente do código que a chamou. Uma função pode modificar variáveis globais ou ponteiros, mas limitando-se aos dados fornecidos pela chamada de função
- O código de uma função só é executado quando esta é invocada em alguma parte do programa que está de algum modo ligada;
- Sempre que uma função é invocada, o programa que a invoca, fica "suspenso" temporariamente. Em seguida são executadas as instruções presentes na função.
- Uma vez terminada a função invocada, o programa retorna ao local onde esta foi invocada.

## Capítulo 10.3 – Subprogramas: Procedimentos e Funções

Um subprograma pode (ou não) ter que produzir algum resultado para o exterior (para quem lhe pediu que executasse a tarefa a que está destinado) , podendo assim ser classificado como:

- ***Procedimento*** (não retorna informação);
- ***Função*** (retorna informação);

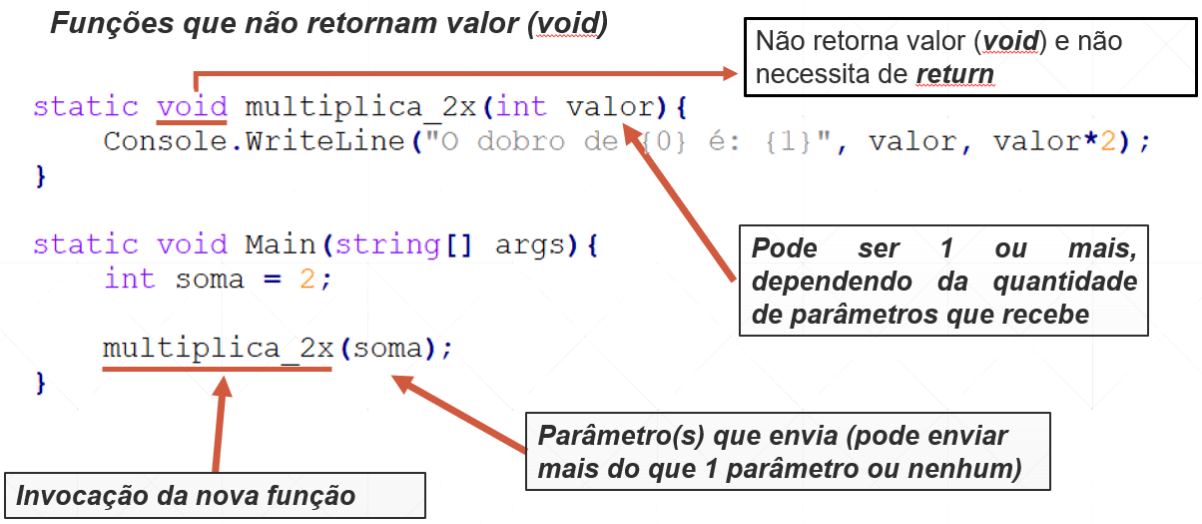
***Em C#*** os ***procedimentos*** e ***funções*** chamam-se ***métodos***.

## Capítulo 10.3.1 – Procedimentos

Um **Procedimento** é um subprograma que contém um conjunto de instruções a executar, de forma unária e atómica, para cumprir uma determinada tarefa.

- ***O procedimento pode ou não ter de receber dados de entrada*** e executa as instruções que o compõem ***até chegarem ao fim***;
- Não devem esquecer que, o número de dados de entrada de um procedimento deve ser igual a mesma quantidade de parâmetros que foi enviado do programa que invocou o procedimento.
- Um procedimento ***não retorna valor***, sendo isto identificado em C# pela palavra ***void,*** que é colocada ***antes do nome do método*** (sub-programa).

**Vamos ver um exemplo prático!**




```
//Procedimento que vai receber um parametro de entrada
1 reference
static void multiplica_2x(int num1)
{
    Console.WriteLine("O dobro de {0} é: {1}", num1, num1 * 2);
}

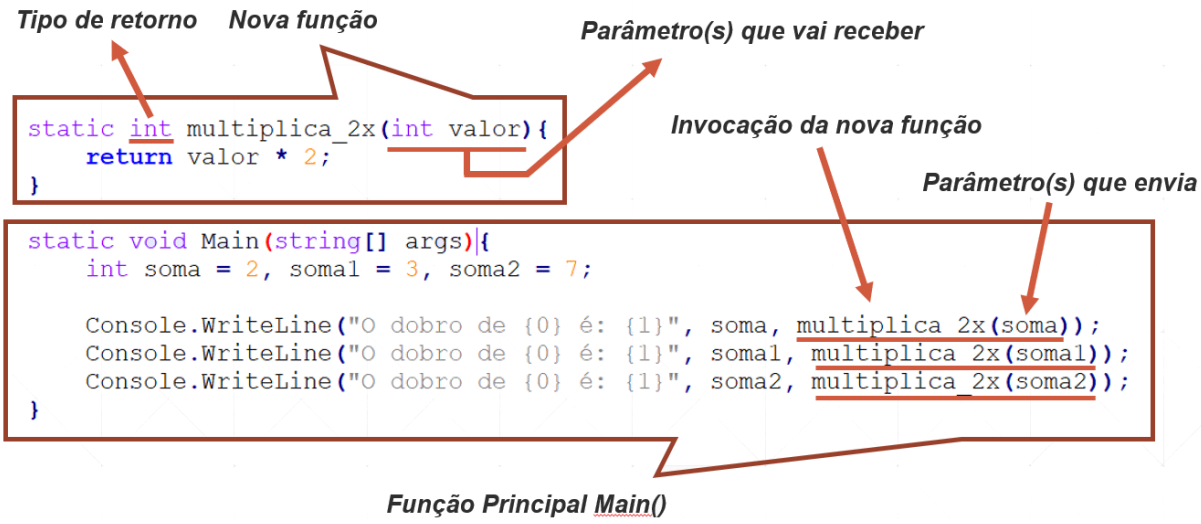
0 references
static void Main(string[] args)
{
    int numero1 = 2;
    //vamos invocar o procedimento multiplica_2x
    //que vai passar um parâmetro de entrada
    multiplica_2x(numero1);
}
```

## Capítulo 10.3.2 – Funções

Uma ***função*** é um subprograma idêntico a um procedimento, mas devolve um valor como resultado da sua execução.

- Uma função é um método que tem um valor de retorno (ao contrário do procedimento). A forma de uma função apresentar um valor de retorno é recorrer à instrução return;
- O return interrompe a execução do sub-programa chamado e o fluxo de execução é devolvido ao programa (ou subprograma) chamante, continuando a execução na instrução seguinte à chamada do subprograma;
- Também não devem esquecer que, o número de dados de entrada de uma função deve ser igual a mesma quantidade de parâmetros que foi enviado do programa que invocou a função.

**Vamos ver um exemplo prático!**




```
static int multiplica_2x(int valor)
{
    return valor * 2;
}

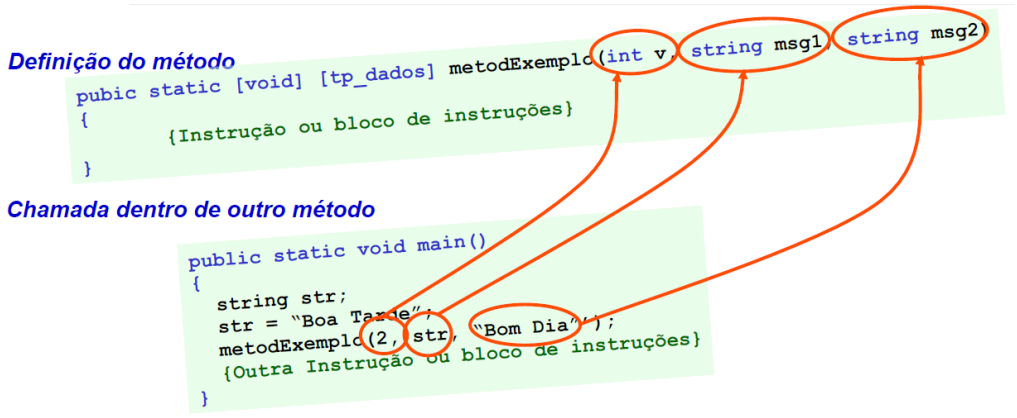
0 references
static void Main(string[] args)
{
    int soma = 2, soma1 = 3, soma2 = 7;

    Console.WriteLine("O dobro de {0} é: {1}", soma, multiplica_2x(soma));
    Console.WriteLine("O dobro de {0} é: {1}", soma1, multiplica_2x(soma1));
    Console.WriteLine("O dobro de {0} é: {1}", soma2, multiplica_2x(soma2));
}
```

## Capítulo 10.3.3 – Subprogramas - procedimentos e funções (por parâmetros ou referência)

Quando passamos parâmetros para um método, o C# vai adequar cada um desses aos parâmetros que se encontram descritos no método, respeitando a ordem pela qual os parâmetros são apresentados. Vamos ver um exemplo!

- O ***primeiro parâmetro declarado na definição do método <u>toma</u>*** o ***valor do primeiro argumento apresentado na chamada***, o ***segundo*** vai tomar o valor do ***segundo*** e ***assim sucessivamente***.




Quando utilizamos uma ***função*** ou ***procedimento*** podemos receber ou não valores de entrada. Mas dentro dos ***valores de entrada*** (***parâmetros),*** podemos ter ***2 tipos de parâmetros***:

- ***Por passagem por parâmetros –*** estamos a trabalhar com uma cópia dos dados que são passados para dentro de uma função ou procedimento. Qualquer alteração estará confinada à cópia efetuada, e nunca ao argumento original.

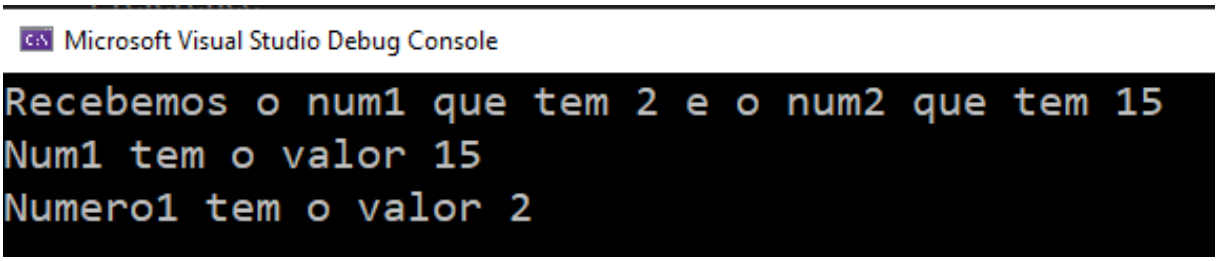
```
//Procedimento que vai receber 2 parametros de entrada
1 reference
static void multiplica_2x(int num1, int num2) {
    Console.WriteLine("Recebemos o num1 que tem {0} e o num2 que tem {1}", num1, num2);

    //Vamos alterar o valor da variavel num1
    num1 = 15;
    Console.WriteLine("Num1 tem o valor {0}", num1);
}

0 references
static void Main(string[] args) {
    int numero1 = 2, numero2 = 15;
    //vamos invocar o procedimento multiplica_2x
    //que vai passar dois parâmetros de entrada
    multiplica_2x(numero1, numero2);

    Console.WriteLine("Numero1 tem o valor {0}", numero1);
}
```

***Como podemos observar***, a alteração do valor de ***num1*** só teve efeito dentro sub-programa, pois estamos a ***trabalhar com uma cópia dos valores originais*** que foram ***passados do programa principal***;




- ***Por passagem por referência –*** neste caso, vamos trabalhar com os valores originais dos dados que são passados para dentro de uma função ou procedimento.

  As alterações realizadas nos parâmetros irão ter reflexo no programa que invocar a função. Para trabalhar sempre com os valores originais, utilizamos a palavra reservada ***ref*** antes do parâmetro de entrada.

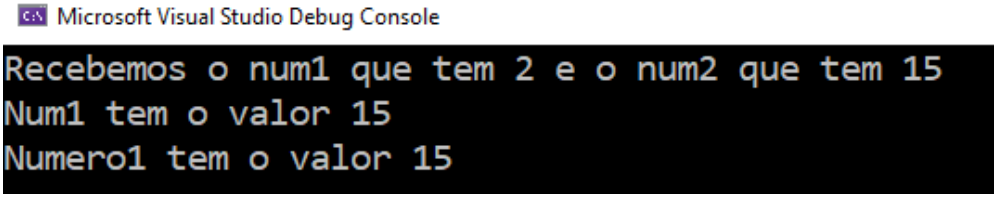
```
//Procedimento que vai receber 2 parametros de entrada
1 reference
static void multiplica_2x( ref int num1, ref int num2) {
    Console.WriteLine("Recebemos o num1 que tem {0} e o num2 que tem {1}", num1, num2);

    //Vamos alterar o valor da variavel num1
    num1 = 15;
    Console.WriteLine("Num1 tem o valor {0}", num1);
}

0 references
static void Main(string[] args) {
    int numero1 = 2, numero2 = 15;
    //vamos invocar o procedimento multiplica_2x
    //que vai passar dois parâmetros de entrada
    multiplica_2x( ref numero1, ref numero2);

    Console.WriteLine("Numero1 tem o valor {0}", numero1);
}
```

***Como podemos observar***, a alteração do valor de ***num1*** teve efeito ***dentro sub-programa*** e no ***programa principal***, pois estamos a ***trabalhar os valores originais*** que foram ***passados do programa principal.***



***Se alteramos o valor de uma variável ou parâmetro***, o mesmo será refletido em todo o programa, ***ao contrário da passagem por parâmetros***!

***Resumindo:***

- ***Por passagem por parâmetros –*** trabalha com uma cópia dos dados que são passados para dentro de um subprograma e qualquer alteração estará confinada à cópia efetuada, e nunca ao argumento original.
- ***Por passagem por referência –*** vamos trabalhar com os valores originais dos dados que são passados para de um subprograma e altera a informação em todo o programa;

# Capítulo 11 – POO: Programação Orientada a Objetos

***Antes das linguagens orientadas a objetos*** (anos 70 e 80) as linguagens estruturadas como o ***C*** e o ***Pascal*** tiveram um enorme sucesso.

Inicialmente o paradigma de programação orientado por objetos (***POO***) surgiu por via do software de simulação nos ***anos 60*** (***SIMULA-67***) e mais tarde por via da ***Engenharia de Software***.

É nos ***anos 90*** que a ***POO*** ganhou relevância dominando até hoje na indústria da informática.
Entre as linguagens que seguiram o paradigma da POO está o ***C++, Java***, ***Delphi***, ***C#, VisualBasic,*** entre outras;





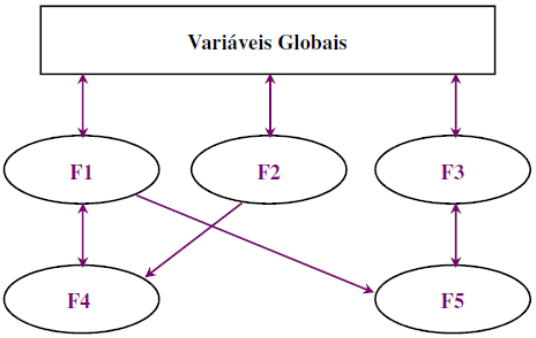



Então, em termos de paradigmas de programação, temos dois tipos de programação que são:

## 11.1 - Programação estruturada (sequencial)

Nas linguagens estruturadas, um programador pensa numa ou mais estruturas de dados para modelar o problema e em seguida desenvolve um conjunto de funções que atuam sobre estas estruturas de dados, ou seja, um ***pensamento mais voltado para o pensamento da máquina***.

**Programa = *Estrutura de Dados + Algoritmos***



***Características***:

- Muito usada para aprender a programar;
- Funciona bem para pequenos programas, mas quando as dimensões do programa aumentam, começam a aparecer problemas;
- À medida que os programas crescem torna-se mais difícil manter o código;
- O problema que acontece é que todas as operações têm acesso a todos os dados e basta uma pequena alteração num dos módulos pode ter implicações em todo o programa.
- Para programar em ***linguagem estruturada*** usamos variáveis e funções que executam certas regras que são desenvolvidas pelos programadores;

## 11.2 - Programação Orientada a Objetos (POO)

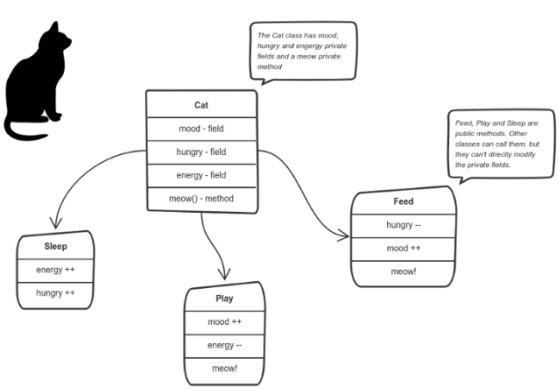
Nas ***linguagens estruturadas*** o problema era modelado em ***estruturas de dados*** e depois eram desenvolvidas ***operações / funções*** que atuavam sobre essas mesmas. O modelo funcionava (***e ainda funciona***) para programas com ***pouca complexidade***, ***mas***, qualquer ***alteração à estrutura*** implica que

- todas as ***funções*** / ***módulos*** ***tenham de ser revistos***;
- existe um ***elevado acoplamento (relacionamento) entre os módulos;***

A ***POO*** vem aliviar esse problema ***ao diminuir o acoplamento*** entre os módulos, onde as estruturas de dados são ***encapsuladas*** dentro dos ***objetos*** e dentro dos objetos continuam a existir funções que atuam sobre as estruturas de dados, ou seja,

***defende um pensamento de programação mais voltado ao pensamento natural (humano).***

**Programa = *Objetos + Relações***

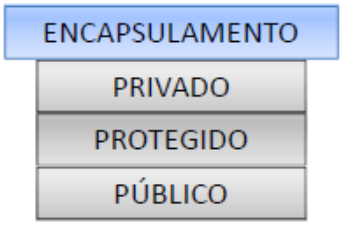


## 11.3 – Conceitos de Programação Orientada a Objetos

A programação orientada aos objetos assenta em três conceitos básicos fundamentais:

- **Encapsulamento de informação**
- **Composição / Herança**
- **Polimorfismo**

### 11.3.1 - Encapsulamento de informação

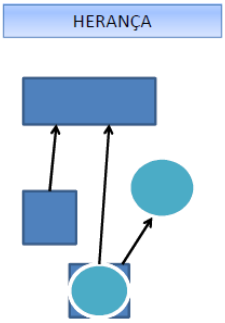



- O Objeto é uma embalagem de ***estados*** (propriedades), ***comportamentos*** (métodos) e tem uma ***identidade***;
- O estado dos objetos é acessível através de operações invocadas por intermédio de mensagens;
- Nem todos os estados e comportamentos podem ser livremente acedidos;

***Os modificadores de acesso controlam*** que parte do programa ***pode ou não utilizar a classe***. Estes podem ser de ***3 tipos***:

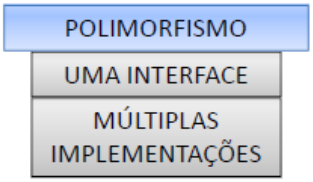
- ***Public*** (público) - é o nível de acesso que os atributos ou métodos podem ser acedidos em qualquer outro ponto do projeto e por outras classes;
- ***Private*** (privado) - é o nível de acesso mais restrito e permite definir que somente a própria classe pode aceder aos atributos ou métodos. Ou seja, nenhuma outra parte do projeto, nem mesmo as classes filhas (heranças), podem aceder a esse atributo ou método;
- ***Protected*** (protegido) – nível de acesso protegido que somente a própria classe e as classes que herdam da mesma é que podem aceder os atributos ou métodos. Dessa forma, ao instanciar a classe os elementos protegidos (protected) não podem ser acedidos diretamente, como ocorre com o public;

### 11.3.2 - Composição / Herança




- Classes derivadas têm um comportamento específico;
- Herança das propriedades e comportamento protegidos e públicos das classes derivadas;
- ***Todos herdamos características e comportamentos dos nossos pais!***

### 11.3.3 - Polimorfismo




- A fronteira/interface de acesso ao objeto é separada da própria implementação do objeto;
- Criação de classes com comportamento genérico;
- Um objeto pode pertencer a vários tipos com comportamentos específicos para cada tipo, ***mas tecnicamente diferentes, implementando a filosofia de "uma interface, múltiplos métodos".***

## 11.4 - Paradigmas de Programação: Objetos, Classes e Instâncias

- ***Objetos*** – é a conjugação de dados e operações numa entidade;
- ***Classe*** – Descreve os objetos que têm características comuns;
- ***Instância*** – Objeto que é descrito por uma classe. Todas as instâncias de uma mesma classe partilham as mesmas operações e possuem uma estrutura de dados interna comum;

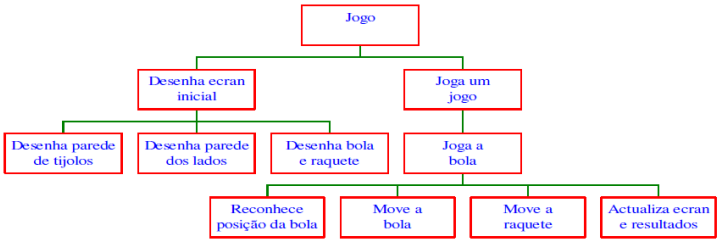
***Características***:

- **Produtividade** - Construir *software* é mais fácil e barato
- **Robustez** - O *software* é mais fiável (por reutilização e melhores regras)
- **Manutenção** - Muito mais fácil executar as melhorias e encontrar os "bugs"
- **Extensibilidade** - Muito mais fácil acomodar novos requisitos
- **Modularidade** - Abstração e encobrimento da informação
- **Naturalidade** - Programação orientada por abstrações do mundo real

## 11.5 - Paradigmas de Programação

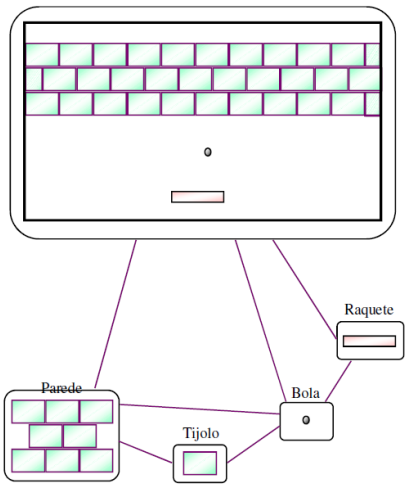
### Solução Tradicional

- Identificar as funções
- Identificar as relações entre as funções
- Identificar as características das funções




### Solução Orientada por Objectos

- Identificar os objectos
- Identificar as relações entre os objectos
- Identificar as características dos objectos



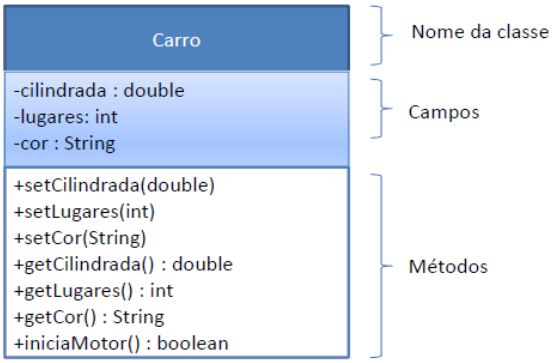
# Capítulo 12 – Classes

Uma ***classe*** é um conceito que define uma ***estrutura para a criação de objeto***. Uma classe ***define um objeto com base*** nas suas propriedades (os campos ou variáveis) e as suas ações ou métodos que definem o seu comportamento (funções).

A classe é especificada pelo programador, definindo a interface e a implementação dos objetos

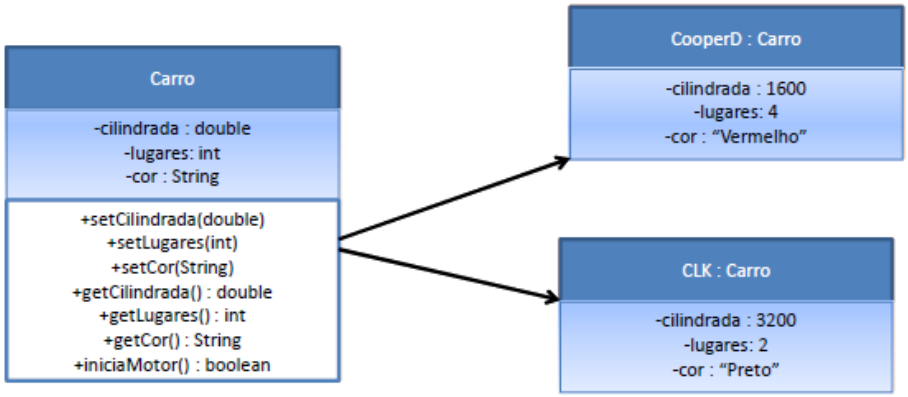
***Nesta classe temos:***

- a representação abstrata de um Carro (***classe***);
- que tem características (***campos***);
- e apresenta comportamentos (***métodos***);




***Como podemos analisar:***

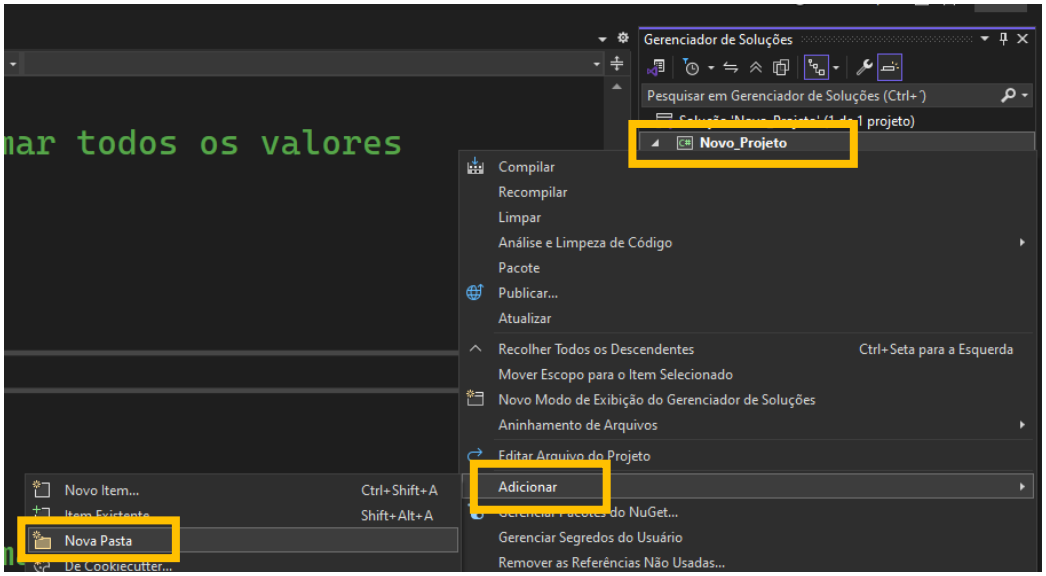
- A classe é um template a partir do qual "nasce" o objeto;
- O objeto é uma instância da classe, dá nome à classe e valores aos campos
- O objeto tem uma identidade e apesar do seu estado sofrer alterações, a identidade não é alterada!
- Se pintar o Mercedes CLK de Amarelo, continua a ser um CLK!

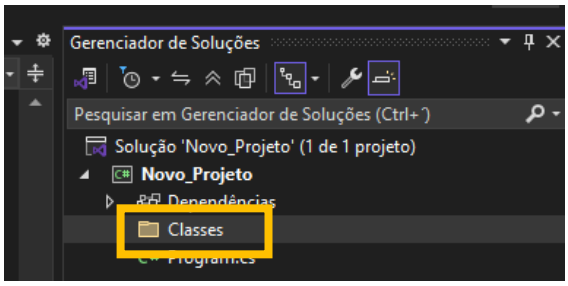


## Capítulo 12.1 – Construção de uma classe

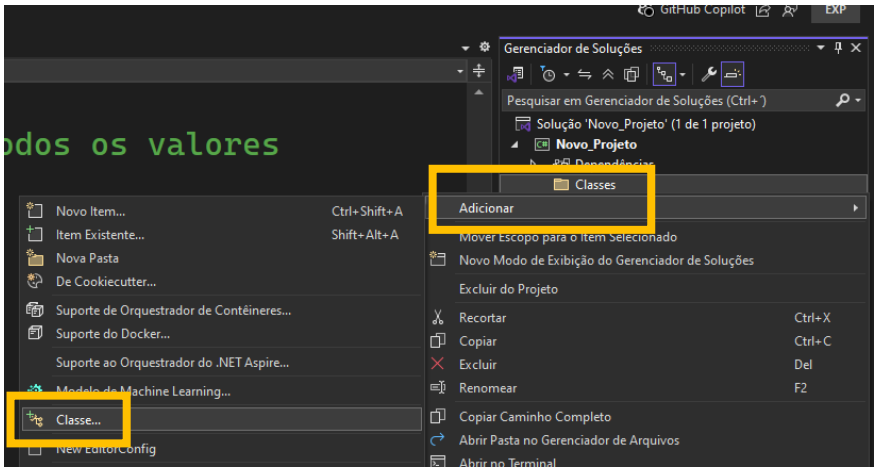
Para construir uma nova classe, vamos fazer os seguintes passos:

**Passo 1 –** Clique com o lado direito do rato em cima do nome do projeto e selecione a opção para criar uma nova pasta, como neste caso, a pasta com o nome “Classes”:

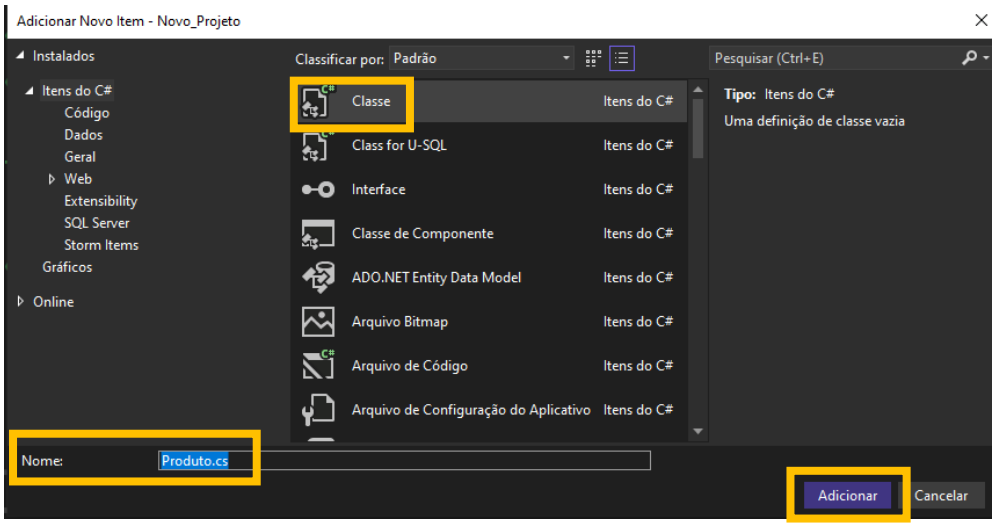


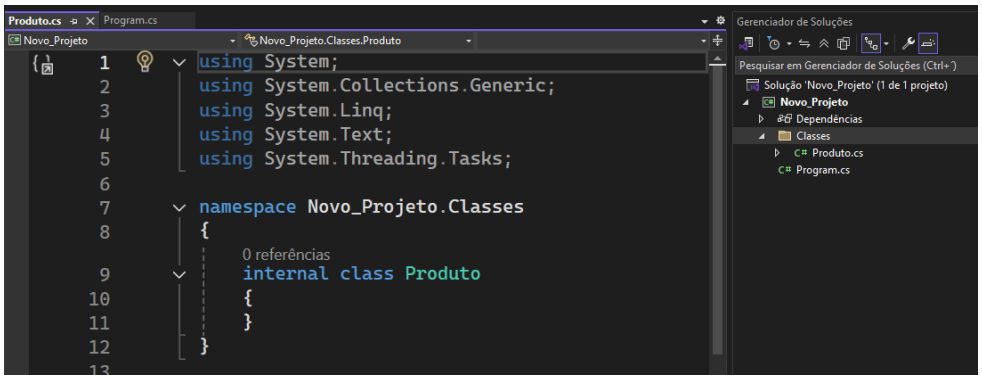

**Passo 2 –** Dentro da nova pasta, vamos pedir para criar um novo ficheiro do tipo Classe, selecionando as seguintes opções:



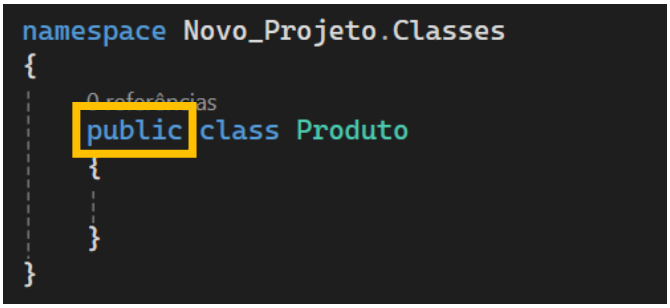
**Passo 3 –** De seguida, vai ser exibida a página de criação de ficheiros e recursos do C# e com a opção classe pré-selecionada. Mais abaixo, vamos dar o nome da classe, que neste exemplo, será o Produto. Se reparar com atenção, o programa já coloca por defeito a extensão .cs para que o sistema e o projeto saber que aquele ficheiro e do tipo de programação C#:




**Passo 4 –** No final, será construído o novo ficheiro dentro da pasta e com a seguinte estrutura:




**Passo 6 –** Para deixar a classe como pública e disponível em todo o projeto, alterar o modificador de acesso de **internal** para **public**:




- A ***declaração de classes*** em C# devem ***começar com letra Maiúscula***;

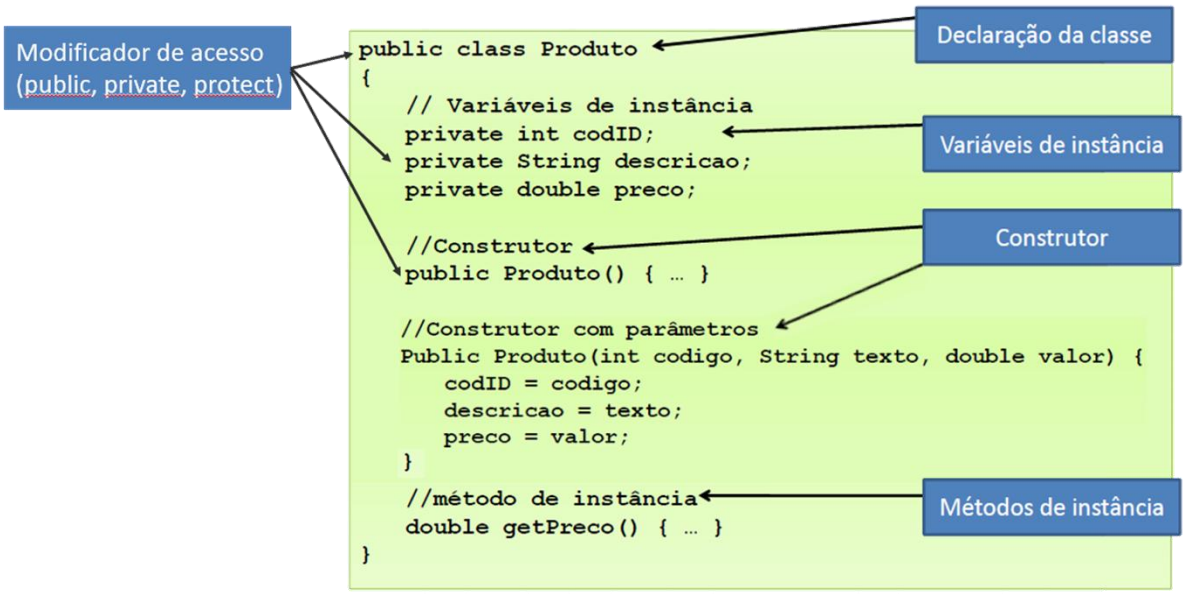
- O ficheiro onde está a definição da ***classe tem de ter o mesmo nome da classe*** e com ***extensão*** ".cs" (como já referido anteriormente);
- As palavras ***public*** / ***private*** são ***modificadores de acesso*** que ***controlam*** que parte do programa ***pode ou não utilizar a classe***.
  - ***Exemplo***: Uma classe ***pública*** pode ser utilizada em todo o lugar;

- ***Os modificadores de acesso controlam*** que parte do programa ***pode ou não utilizar a classe***.

  Estes podem ser de ***3 tipos***:

  - ***Public*** (público) - é o nível de acesso que os atributos ou métodos podem ser acedidos em qualquer outro ponto do projeto e por outras classes;
  - ***Private*** (privado) - é o nível de acesso mais restrito e permite definir que somente a própria classe pode aceder aos atributos ou métodos. Ou seja, nenhuma outra parte do projeto, nem mesmo as classes filhas (heranças), podem aceder a esse atributo ou método;
  - ***Protected*** (protegido) – nível de acesso protegido que somente a própria classe e as classes que herdam da mesma é que podem aceder os atributos ou métodos. Dessa forma, ao instanciar a classe os elementos protegidos (protected) não podem ser acedidos diretamente, como ocorre com o public;

**Exemplo de uma classe:**




**Vamos analisar o processo da construção de uma classe.**

## Capítulo 12.1.1 – Variáveis de instância

As variáveis de instância permitem representar as características de uma classe:

- A ***declaração de variáveis de instância*** em C# é feita ***dentro do bloco da definição de classe*** e ***nunca*** dentro de ***métodos*** ou ***construtores***;
- Para ***prevenir*** que outras classes ***possam aceder às variáveis de instância***, devemos utilizar a palavra reservada ***private*** antes da declaração.
- Neste exemplo, temos a classe **Produto** que vai ter 3 características: o ID, uma descrição e um preço;

```
0 referências
public class Produto
{
    //variaveis de instância
    private int codID;
    private string descricao;
    private double preco;
```

## Capítulo 12.1.2 – Construtores (sem e com parâmetros)

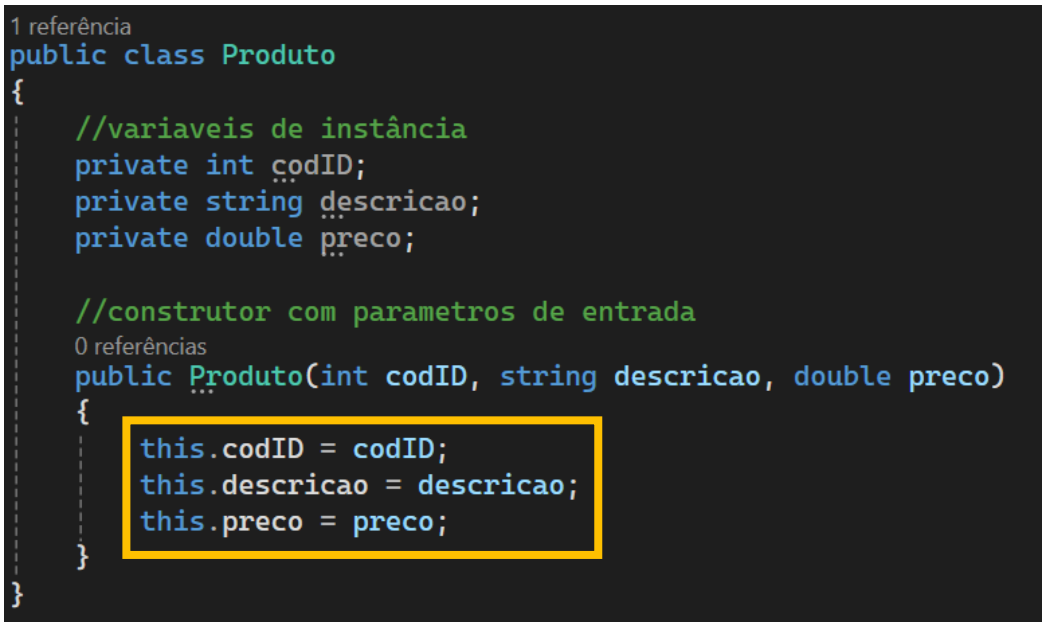
O ***construtor é um método especial***, por norma é o ***primeiro "método"*** que se ***constrói dentro da classe***. Não são métodos de instância porque apenas são utilizados para criar instâncias/objetos de uma dada classe;

Nos construtores ***vamos ter parâmetros de entrada no construtor*** e que vai ***receber/passar a informação dos parâmetros de entrada*** para as ***variáveis de instância*** da classe:

```
//Construtor com parâmetros
Public Produto(int codigo, String texto, double valor) {
    codID = codigo;
    descricao = texto;
    preco = valor;
}
```

***Se não especificarmos um construtor***, o C# ***cria um construtor por defeito*** que ***inicializa todas as variáveis de instância*** para ***valores por defeito*** (numéricos a 0, boolean a false/true e objetos a null)

- Utilizamos a palavra reservada ***this*** para ***identificar*** o que é a **variável de instância do parâmetro** da informação **do parâmetro de entrada do construtor**;
    - A palavra reservada ***this*** permite ter acesso a partir do código do objeto às ***variáveis***, ***métodos*** e ***construtores*** do próprio objeto.
    - A palavra reservada ***this*** é útil por exemplo, quando as variáveis dos parâmetros de entrada de um método são coincidentes com as variáveis de instância.




## Capítulo 12.1.3 – Métodos de instância: Get e Set

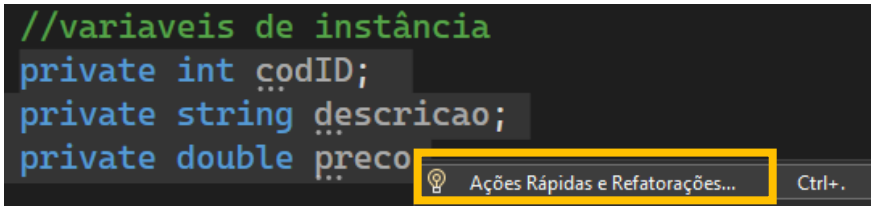
- ***Métodos de instância*** são responsáveis ***pelo comportamento das instâncias/objetos quando recebem/mostram os(as) valores/mensagens provenientes de alguma parte do programa***;
- ***Para que outras classes tenham acesso ao método*** é preciso designar o método com acesso público com a ***palavra reservada public***;
- ***Um método que não retorne qualquer tipo de dado*** deve ser colocada a palavra reservada ***void*** no ***tipo de dados de retorno***;
- Utilizamos a palavra reservada ***this*** para ***identificar*** o que é a **variável de instância do parâmetro** da informação **do parâmetro de entrada do construtor**;
    - A palavra reservada ***this*** permite ter acesso a partir do código do objeto às ***variáveis***, ***métodos*** e ***construtores*** do próprio objeto.
    - A palavra reservada ***this*** é útil por exemplo, quando as variáveis dos parâmetros de entrada de um método são coincidentes com as variáveis de instância.

```
public void setCodID(int codID) {
        this.codID = codID; //codID = codID não faz sentido
}
```

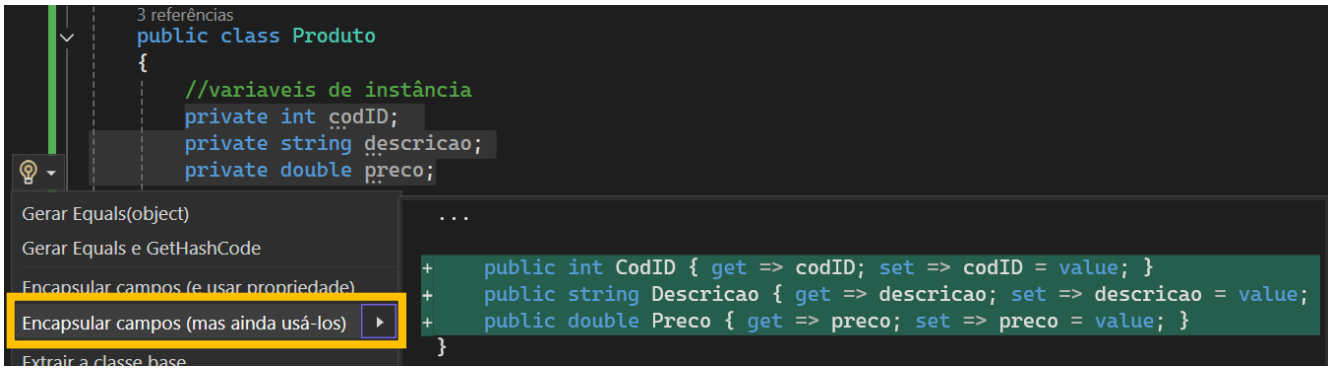
- É boa prática na codificação das classes C# construir métodos de instância ***set*** e ***get*** para ***cada variável de instância*** que existe definida na classe, em que:
  - O ***método get*** é utilizado para obter as informações das variáveis de instância do objeto criado a partir da classe;
  - O ***método set*** é utilizado para receber e alterar as informações das variáveis de instância do objeto criado a partir da classe;

Para **gerar** os **gets** e **sets** para cada variável de instância, vamos fazer os seguintes passos:

- **Passo 1** – **Selecione todas as variáveis de instância da classe** e **clique** em cima da seleção com **o lado direito do rato**. Será exibido o menu de opções e **deve selecionar a opção "Ações rápidas e Refatorações"**:




- **Passo 2 –** De seguida, selecione a opção " Encapsular campos (mas ainda usá-los)":




**Resultado Final:**

```
3 referências
public class Produto
{
    //variaveis de instância
    private int codID;
    private string descricao;
    private double preco;

    //construtor com parametros de entrada
    1 referência
    public Produto(int codID, string descricao, double preco)
    {
        this.codID = codID;
        this.descricao = descricao;
        this.preco = preco;
    }

    //métodos de instância (gerados a partir das variáveis de instância)
    0 referências
    public int CodID { get => codID; set => codID = value; }
    2 referências
    public string Descricao { get => descricao; set => descricao = value; }
    0 referências
    public double Preco { get => preco; set => preco = value; }
}
```

## Capítulo 12.1.4 – Métodos adicionais

Podemos criar **métodos adicionais** para designar comportamentos específicos, como por exemplo, **mostrar todos os dados das variáveis de instância de um objeto** criado com a classe Produto:

```
//Métodos adicionais

//Exemplo: mostrar todos os dados das variáveis de instância para um objeto
1 referência
public string Mostra_Produto()
{
    return "ID: " + this.codID + ", Descrição: " + this.descricao + ", Preço: " + this.preco;
}
```

**Exemplo da classe Produto (Completo):**

```
3 referências
public class Produto
{
    //variaveis de instância
    private int codID;
    private string descricao;
    private double preco;

    //construtor com parametros de entrada
    1 referência
    public Produto(int codID, string descricao, double preco)
    {
        this.codID = codID;
        this.descricao = descricao;
        this.preco = preco;
    }

    //métodos de instância (gerados a partir das variáveis de instância)
    0 referências
    public int CodID { get => codID; set => codID = value; }
    2 referências
    public string Descricao { get => descricao; set => descricao = value; }
    0 referências
    public double Preco { get => preco; set => preco = value; }

    //Métodos adicionais

    //Exemplo: mostrar todos os dados das variáveis de instância para um objeto
    1 referência
    public string Mostra_Produto()
    {
        return "ID: " + this.codID + ", Descrição: " + this.descricao + ", Preço: " + this.preco;
    }
}
```

## Capítulo 12.2 – Instância de Classes: Criação de novos objetos a partir de uma Classe

Quando pretendemos usar as classes para criar objetos e armazenar informação, temos de saber como invocar as mesmas. Como tal, as classes são tipos de referência para uma variável de um objeto de uma classe que contém uma referência ao endereço do objeto.

Desta forma, temos de fazer a criação da instância de uma classe para uma variavel, por forma a criar um novo objeto a partir dessa classe e colocar as informações pedidas no construtor dessa classe.

Para realizar as instâncias de classes utilizamos a palavra reservada **new**.

- A palavra reservada ***new*** trata-se de um ***operador que vai instanciar a classe***, ***alocando para isso espaço em memória para o objeto*** e retorna a referência/endereço desse objeto.
- ***Exemplo***: Vamos criar uma instância de uma classe, isto é, um novo objeto.

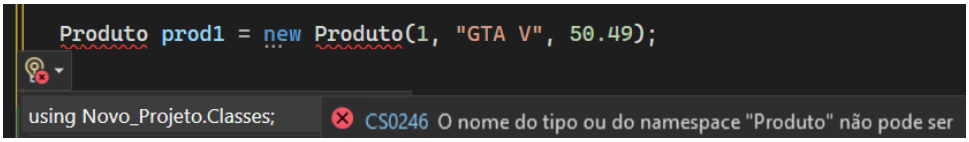
  ***Empregado*** eng1 = ***Empregado***("Magda Dionísio", 25);

- O operador ***new*** também tem a responsabilidade de:
  - ***invocar o construtor da classe;***
  - ***para inicializar um objeto;***

**Exemplo da criação de uma variável com base na classe Produto:**

```
Produto prod1 = new Produto(1, "GTA V", 50.49);
```

**Nota Importante:** Se a palavra que designa como classe estiver a dar erro, significa que a classe na está no mesmo sítio do ficheiro do ficheiro onde está a testar a criação do objeto. Como tal, temos de importar o local ou pasta de onde está o ficheiro da classe:

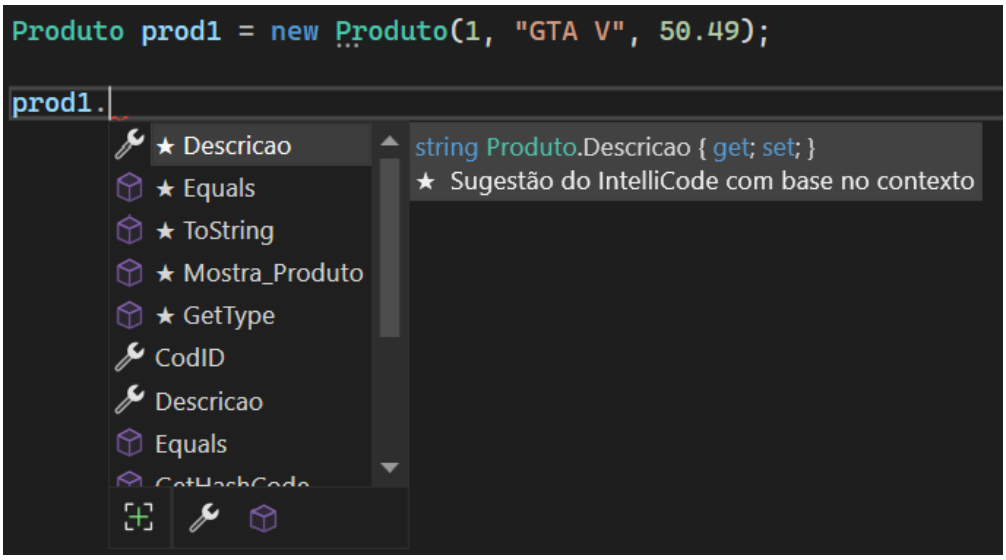



**Resultado da importação automática:**

```
using Novo_Projeto.Classes;

Produto prod1 = new Produto(1, "GTA V", 50.49);
```

Depois de criar o objeto em causa, basta invocar o nome da variável baseada na classe e poderá aceder as propriedades da mesma (como tal, basta colocar o símbolo **. (ponto final)** depois do nome da variável para obter a informação da classe):




Podemos assim fazer várias operações, como por exemplo:

**Mostrar** a informação de uma variável de instância do objeto (pelos métodos de instância **get**):

```
Produto prod1 = new Produto(1, "GTA V", 50.49);

Console.WriteLine("Valor do ID do prod1: " + prod1.CodID);
Console.WriteLine("Valor da descrição do prod1: " + prod1.Descricao);
```

```
Console de Depuração do Microsoft Visual Studio
Valor do ID do prod1: 1
Valor da descrição do prod1: GTA V
```

**Alterar** a informação de uma variável de instância do objeto (pelos métodos de instância **set**):

```
Produto prod1 = new Produto(1, "GTA V", 50.49);

//Alterar o valor da descricao do prod1
prod1.Descricao = "Super Mario";

Console.WriteLine("Valor da descrição do prod1: " + prod1.Descricao);
```

```
Console de Depuração do Microsoft Visual Studio
Valor da descrição do prod1: Super Mario
```

**Mostrar** a informação de todas as variáveis de instância do objeto com o método Mostra_Produto:

```
Produto prod1 = new Produto(1, "GTA V", 50.49);

Console.WriteLine("Dados prod1: " + prod1.Mostra_Produto());
```

```
Console de Depuração do Microsoft Visual Studio
Dados prod1: ID: 1, Descrição: GTA V, Preço: 50,49
```

# Capítulo 13 – Interfaces

## 13.1 – O que são as Interfaces?

Em C#, uma interface é um ficheiro que define um contrato do qual as classes podem implementar. Uma interface pode conter declarações de métodos, propriedades, eventos ou indexadores, mas sem implementação concreta.

Uma interface estabelece "o que" um tipo deve fazer, mas não "como" ele deve fazer. Isso é deixado para as classes que implementam a interface.

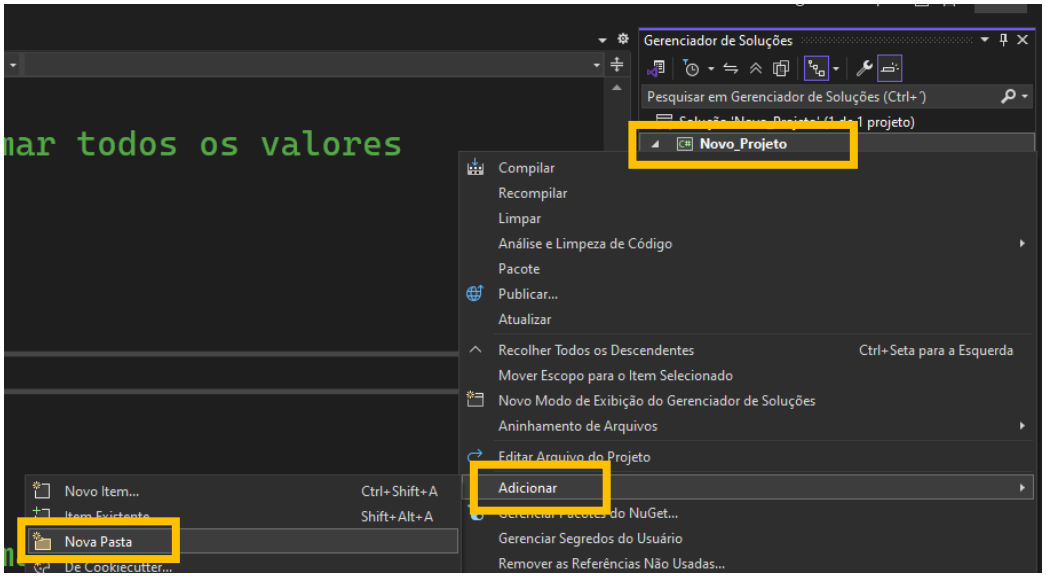
Resumindo, as Interfaces implementam quais os métodos e propriedades que uma classe deve implementar, para não haver mais funcionalidades do que é expectado (permitindo ao programador não colocar outras funcionalidades que não sejam necessárias).

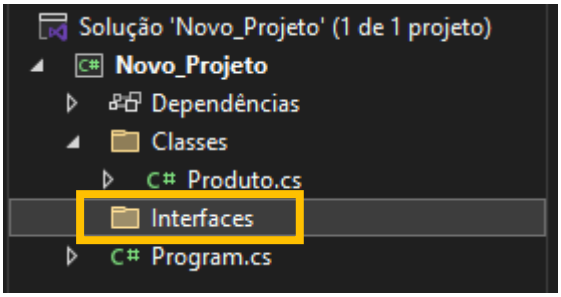
A regra será criar um ficheiro interface por cada classe que tiver no projeto, permitindo separar as funcionalidades dentro do projeto e de forma exclusiva.

## 13.2 – Como criar uma interface?

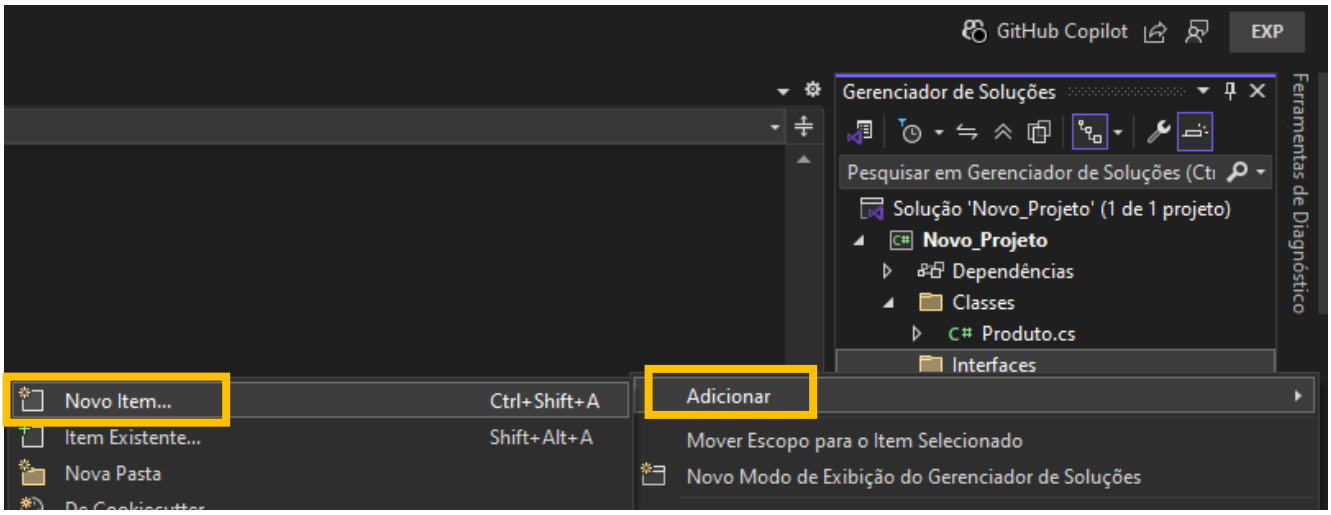
Como boa regra de criação de ficheiros de programação, para criar uma interface vamos dar o nome do ficheiro com a primeira letra em maiúscula (tal como nas classes), como segue o próximo exemplo:

**Passo 1 –** Clique com o lado direito do rato em cima do nome do projeto e selecione a opção para criar uma nova pasta, como neste caso, a pasta com o nome “Interfaces”:

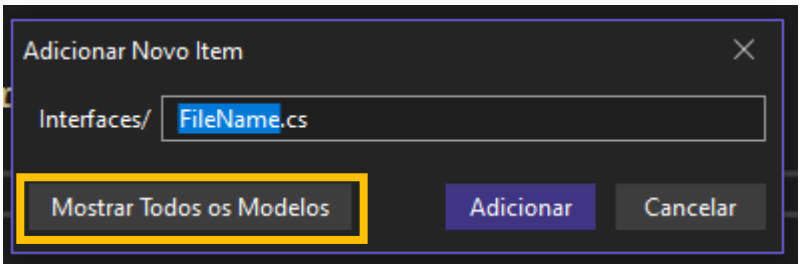





**Passo 2 –** Dentro da nova pasta, vamos pedir para criar um novo ficheiro do tipo Interface, selecionando as seguintes opções Adicionar → Novo item… :

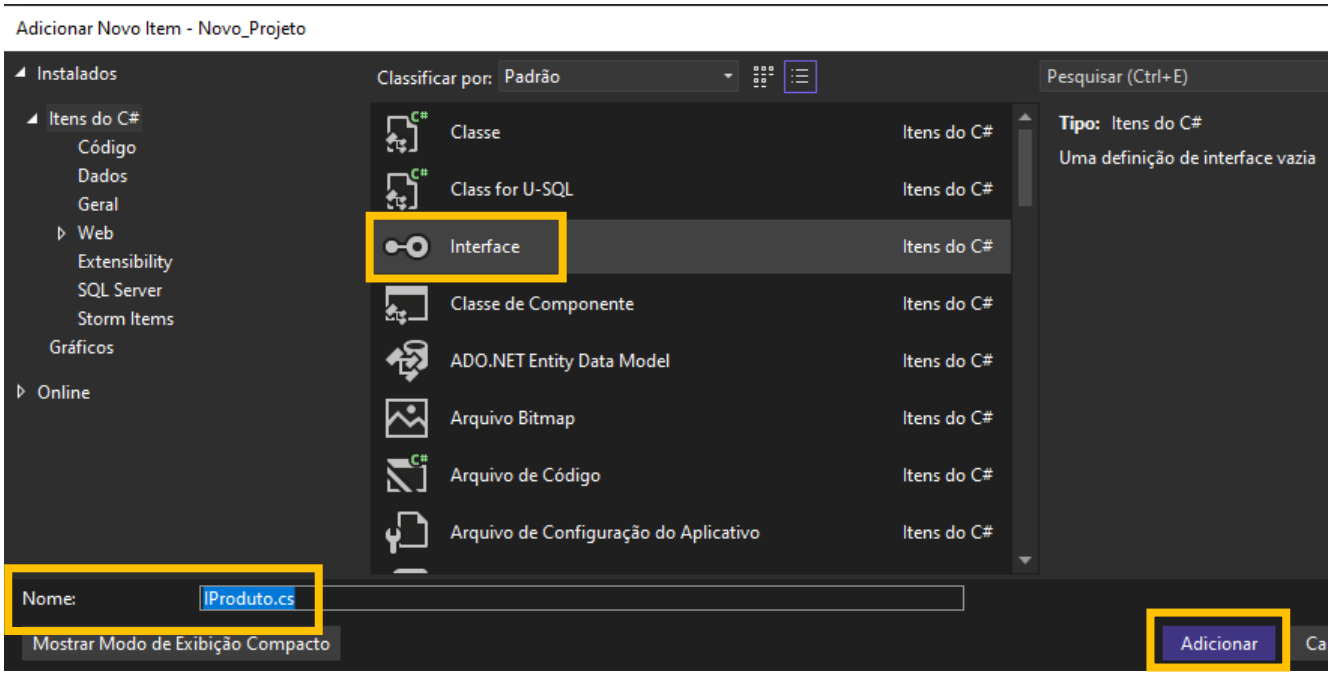



**Passo 3 –** Por vezes o editor, pode colocar a janela do novo item num formato minimizado. Como queremos selecionar o tipo de ficheiro Interface, vamos carregar no botão “Mostrar todos os modelos”:

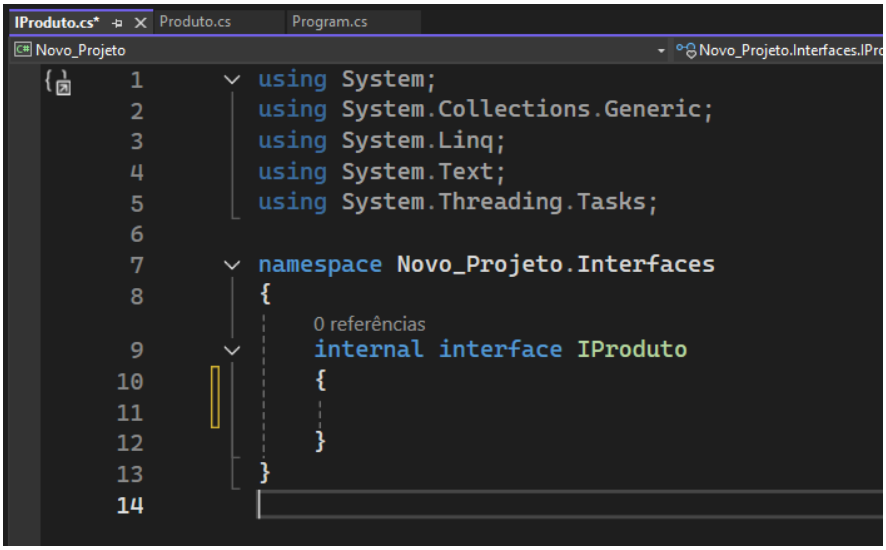



**Passo 4 –** Quando expandir a janela anterior, vamos selecionar o ficheiro do tipo “Interface” e dar o nome a começar por I (em maiúscula) e depois o nome do ficheiro. Para este exemplo, vamos criar a interface que depois vai interligar com a classe Produto.

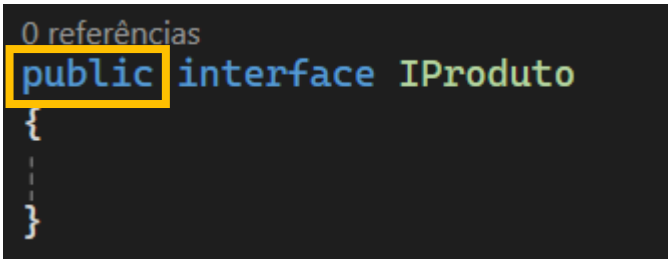
Assim sendo, como temos uma classe criada com o nome Produto, vamos criar a interface para depois interligar com a mesma e que vai ter o nome “IProduto.cs” (na maioria das vezes a extensão é dada automaticamente, bastando mudar somente o nome do ficheiro):



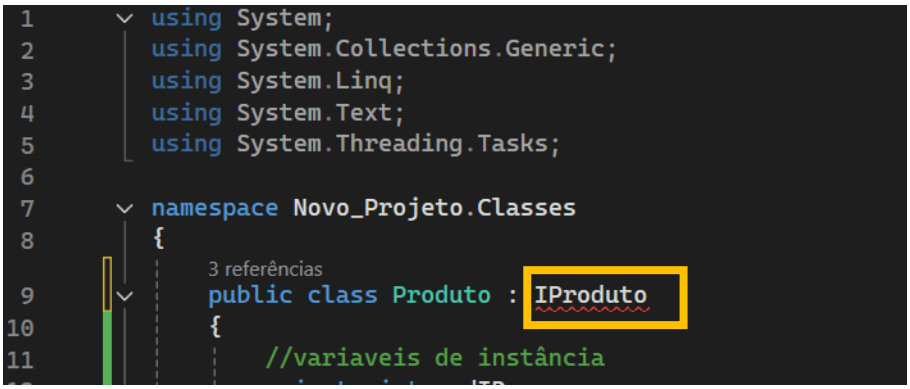

**Passo 5 –** Se tudo correr bem, será gerado o ficheiro dentro da pasta interfaces:




**Passo 6 –** Para deixar a interface como pública e disponível em todo o projeto, alterar o modificador de acesso de **internal** para **public**:

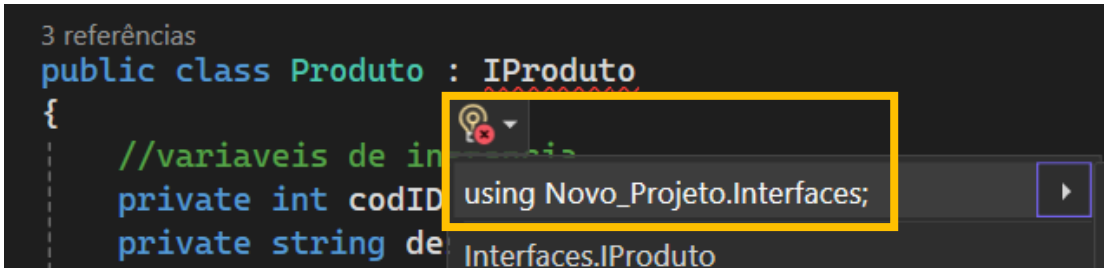


**Passo 7 –** Neste caso, para interligar a Interface (IProduto) a classe Produto, vamos abrir o ficheiro da classe Produto e no início do ficheiro, onde está declarada a criação da classe, vamos implementar a interface (que pode ser conjugado com a herança):

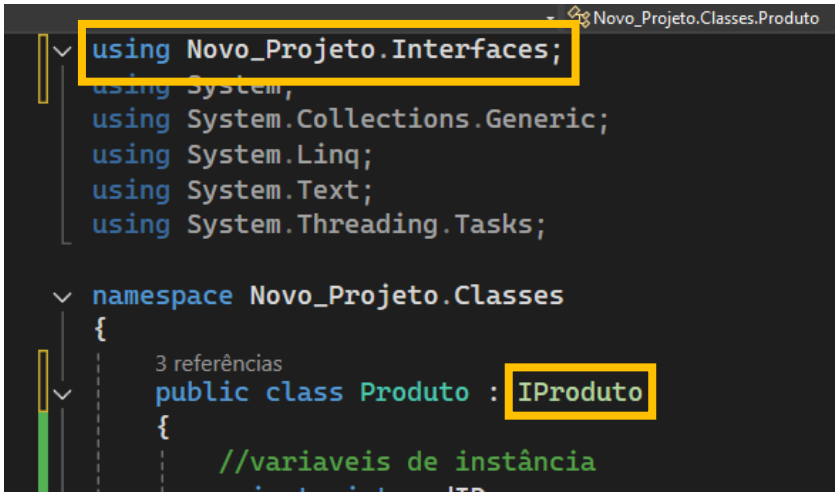



No entanto, repare que, o nome da interface está a vermelho. Como a interface está numa pasta diferente da localização da classe, temos de importar a ligação da localização do ficheiro da interface para o cabeçalho das bibliotecas (no topo do ficheiro).

Como tal, vamos clicar com o lado direito do rato em cima do nome da interface e selecionar a lâmpada de ações rápida e verificar se aparece a opção using e a localização da pasta interfaces no projeto:




Fazendo esses passos, o programa importa a localização mais acima, indicando que agora reconhece a pasta interfaces como parte da classe (se reparar o nome da interface mudou de cor):



**Passo 8 –** Dentro da Interface (IProduto) vamos colocar as regras que a classe Produto tem de seguir para não haver redundâncias de informação. Neste caso, quando fizemos a construção da classe, fizemos os seguintes passos:

- Variáveis de Instância
- Construtor
- Métodos get e set
- Métodos adicionais

Os 3 primeiros pontos são algo que pertencem na construção da classe. O que importa para a interface e saber o que implementou adicionalmente depois dos gets e sets, para que o utilizador não escreva métodos a mais do que é necessário. Como tal, no ficheiro da interface, apenas vamos indicar que a classe vai ter apenas o método adicional Mostra_Produto():

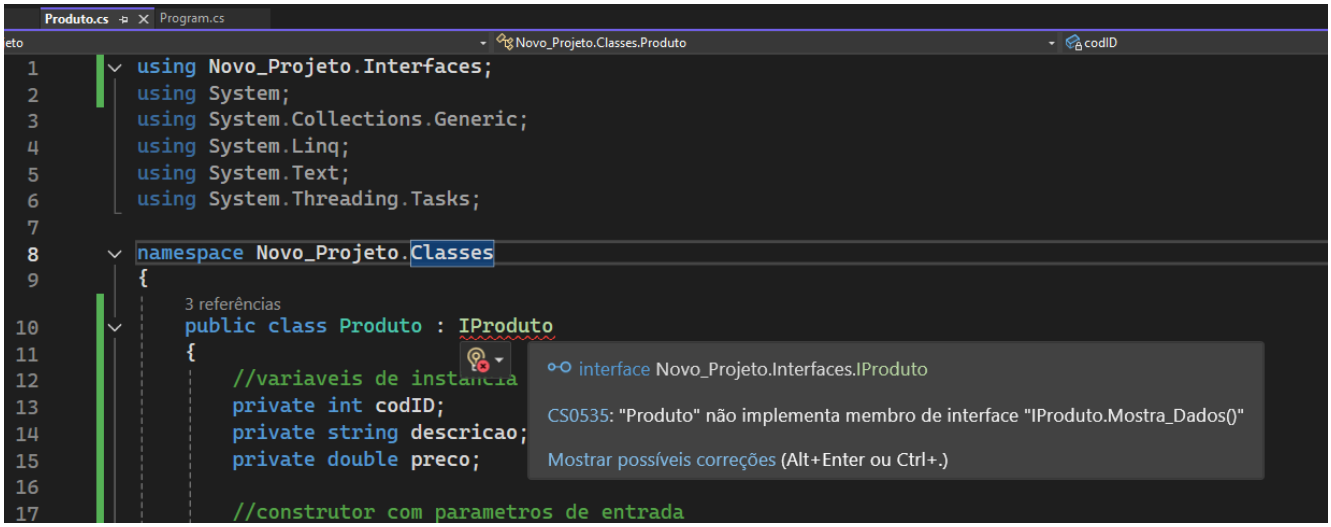
```
1 referência
public interface IProduto
{
    2 referências
    public string Mostra_Produto();
}
```

Como na classe já temos o método implementado, o contrato está bem implementado.

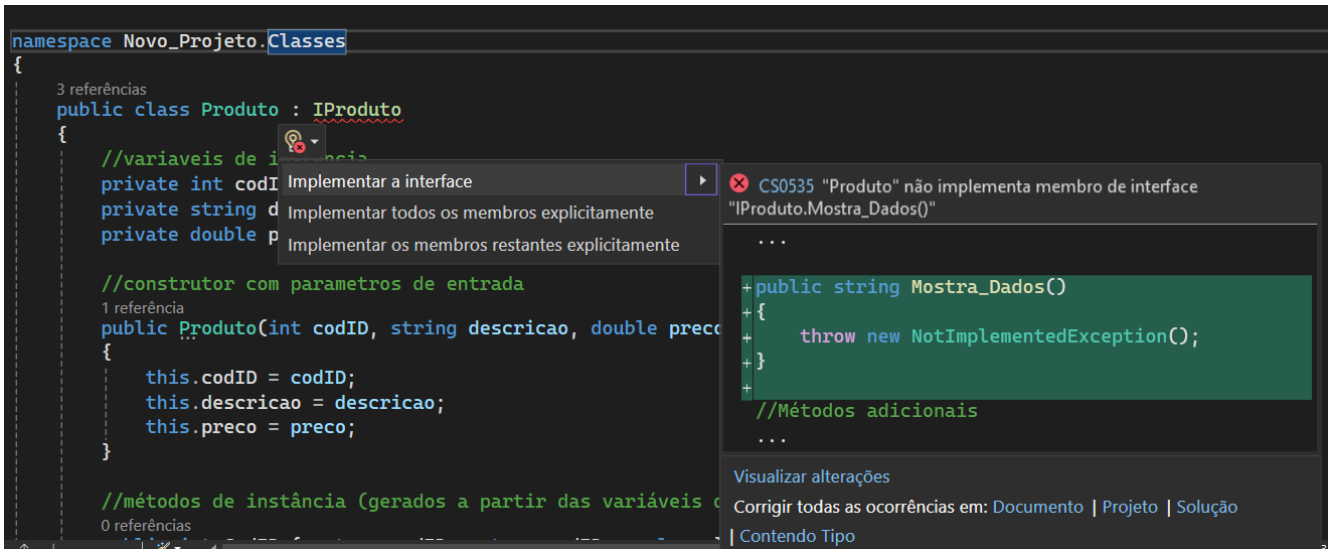
Mas testar a Interface e verificar se esta implementa as regras definidas na mesma, vamos fazer a seguinte experiência:

```
1 referência
public interface IProduto
{
    2 referências
    public string Mostra_Produto();
    0 referências
    public string Mostra_Dados();
}
```

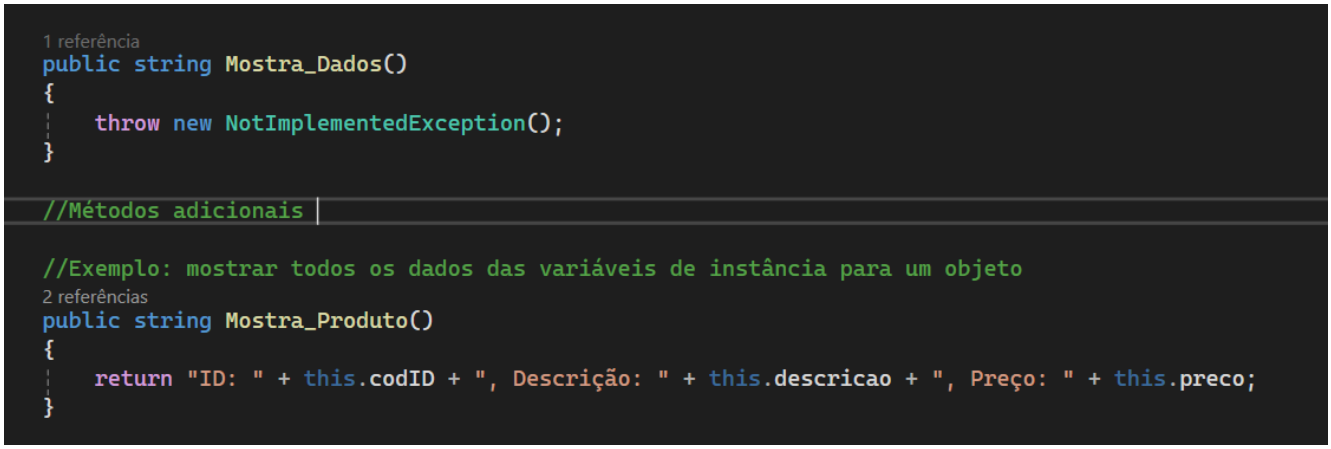
Como resultado e alterado o contrato entre a classe e a interface, quando voltarmos a classe, esta vai reclamar que ainda não tem o método criado:




**Caso seja para implementar o novo método adicional**, pode usar as ações rápidas e pedir ao visual studio para criar o método mais abaixo:




**Resultado:**



# Capítulo 14 – Coleções (estruturas de dados)

## Capítulo 14.1 – Introdução as coleções

**Arrays** (como já falado anteriormente nos casos dos vetores e matrizes) e **Coleções** são objetos que atuam como contentores de informação.

A utilização de arrays como contentores de dados é já nossa conhecida e utilizada extensivamente (lembrem-se a utilização de arrays para guardar notas das disciplinas, listas de números, etc.).

Os dados guardados nas coleções podem ser depois processados (cálculos sobre esses dados, decisões tomadas com base nos conteúdos dos dados, etc.)

As coleções são ***mais flexíveis que os vetores e as matrizes***, e são ideais para trabalhar com um ***número indeterminado de elementos***.

O ***nome coleções*** deriva do facto de se tratar de uma ***interface de programação*** para tratamento de ***coleções de objetos***.

Tal como uma matriz, uma ***coleção permite guardar múltiplos elementos***, ao ***contrário*** das matrizes, ***as coleções não têm um tamanho fixo***.

O ***tamanho de uma coleção aumenta automaticamente*** quando são adicionados elementos.

***Similares em:***

- Ambos permitem guardar múltiplos elementos.

***Diferentes em:***

- As matrizes estão embutidas na linguagem C#. As coleções são classes fornecidas pelo Visual Studio.
- As coleções oferecem métodos para efetuar operações (adicionar, remover, etc.).
- As matrizes têm tamanho fixo. As coleções têm tamanho variável.

## Capítulo 14.2 – Principais coleções da Plataforma .NET

| ***Classe Tracional*** ***(Collections)*** | ***Classe Genérica*** ***(Collections.Generic)*** | ***Descrição*** |
|---|---|---|
| ***ArrayList*** | ***List<T>*** | Funciona como uma tabela cujo tamanho cresce automaticamente. |
| ***---*** | ***LinkedList<T>*** | Armazena os objetos linearmente, utilizando uma lista ligada (ligando o valor antigo com o próximo valor) |
| ***HashTable*** | ***Dictionary<Key, Value>*** | Representa uma coleção de pares (chave, valor). A sua organização é baseada na chave, com acesso muito eficiente. Sem relação de ordem na chave. |
| ***SortedList*** | ***SortedList<Key,Value>*** | Semelhante a Dictionary, mas os pares (chave, valor) são mantidos ordenados, baseados no valor da chave. |
| ***Queue (Fila)*** | ***Queue<T>*** | Representa uma coleção de objetos em que o primeiro a ser colocado na coleção é o primeiro a sair ***(FIFO) – First In First Out***. |
| ***Stack (Pilha)*** | ***Stack<T>*** | Representa uma coleção de objetos em que o último a ser colocado na coleção é o primeiro a sair ***(LIFO) – Last In First Out***. |

**Nos capítulos a seguir vamos estudar as seguintes coleções:**

- Filas de espera – Queue<T>
- Pilhas – Stack<T>
- Listas – List<T>
- Listas Ordenadas – SortedList<key, Value>

## Capítulo 14.2.1 – Filas de Espera - Queue<T>

Uma fila de espera pode ser vista como uma fila de itens que estão à espera para serem processados.

- Quando chega ***um novo item*** ele é colocado ***no fim da fila***.
- Quando se deseja ***processar um item*** é retirado ***o primeiro da fila***.

Uma vez que os ***itens são retirados da fila pela mesma ordem que foram introduzidos***, também podemos chamar as filas por FIFO ***(first-in, first-out)***.

### *Métodos Principais:*

- ***Enqueue() –*** permite ***adicionar um objeto à fila;***
- ***Dequeue() –*** permite ***retirar um objeto da fila.***

### *Criar, adicionar e retirar elemento*

```
Declarar e criar (T = tipo dos elementos):
    Sintaxe: Queue<T> nome = new Queue<T>();
    Exemplo: Queue<int> fila = new Queue<int>();
```

```
Adicionar elemento (Método):
    Sintaxe: nome.Enqueue(elemento);
    Exemplo: fila.Enqueue(10);
```

```
Retirar elemento (Método):
    Sintaxe: T item = nome.Dequeue();
    Exemplo: int elemento = fila.Dequeue();
```

### *Testar, apagar e número de elementos*

```
Testar elemento (Método):
    Sintaxe: T item = nome.Peek();
        (Devolve o próximo elemento sem o retirar da fila)
    Exemplo: int elemento = fila.Peek();
```

```
Apagar todos os elementos (Método):
    Sintaxe: nome.Clear();
    Exemplo: fila.Clear();
```

```
Números de elementos (Propriedade):
    Sintaxe: T num = nome.Count;
    Exemplo: int numElementos = fila.Count;
```

**Exemplo:**

```
//Criar uma fila de espera
Queue<string> filaDeEspera = new Queue<string>();

//adicionar elementos com a função Enqueue()
filaDeEspera.Enqueue("Ana Rita");
filaDeEspera.Enqueue("Luis Martins");
filaDeEspera.Enqueue("Vitor Loureiro");

//Mostra quem é o primeiro elemento da lista (sem remover)
Console.WriteLine("Primeiro registo: " + filaDeEspera.Peek());

//verificar se contém o elemento a pesquisar
Console.WriteLine("Pesquisa por nome: " + filaDeEspera.Contains("Luis Martins"));

//Percorrer a lista
while (filaDeEspera.Count > 0)
{
    string proximo = filaDeEspera.Dequeue();
    Console.WriteLine("Proxima pessoa a ser atendido: " + proximo);
    Console.WriteLine("Dados na fila: " + filaDeEspera.Count());
}
```

```
Console de Depuração do Microsoft Visual Studio
Primeiro registo: Ana Rita
Pesquisa por nome: True
Proxima pessoa a ser atendido: Ana Rita
Dados na fila: 2
Proxima pessoa a ser atendido: Luis Martins
Dados na fila: 1
Proxima pessoa a ser atendido: Vitor Loureiro
Dados na fila: 0
```

## Capítulo 14.2.2 – Pilhas - Stack<T>

**Stacks (Pilhas):** é semelhante a ***Queue***, mas a ordem de saída é inversa à de entrada. O último elemento a entrar na pilha é o primeiro a sair.

- Quando chega ***um novo item*** ele é colocado ***no início da pilha***.
- Quando se deseja ***processar um item*** é retirado o ***primeiro da pilha***.

Uma vez que os ***itens são retirados da pilha pela ordem inversa com que foram introduzidos***, também é designada por fila ***LIFO (last-in, first out)***.

***Métodos Principais:***

- ***Push() –*** permite ***adicionar um objeto à pilha;***
- ***Pop() –*** permite ***retirar um objeto da pilha.***

***Criar, adicionar e retirar elemento***

```
Declarar e criar (T = tipo dos elementos):
    Sintaxe: Stack<T> nome = new Stack<T>();
    Exemplo: Stack<int> pilha = new Stack<int>();
```

```
Adicionar elemento (Método):
    Sintaxe: nome.Push(elemento);
    Exemplo: fila.Push(10);
```

```
Retirar elemento (Método):
    Sintaxe: T item = nome.Pop();
    Exemplo: int elemento = fila.Pop();
```

***Testar, apagar e número de elementos***

```
Testar elemento (Método):
    Sintaxe: T item = nome.Peek();
        (Devolve o próximo elemento sem o retirar da fila)
    Exemplo: int elemento = pilha.Peek();
```

```
Apagar todos os elementos (Método):
    Sintaxe: nome.Clear();
    Exemplo: pilha.Clear();
```

```
Números de elementos (Propriedade):
    Sintaxe: int num = nome.Count;
    Exemplo: int numElementos = pilha.Count;
```

**Exemplo:**

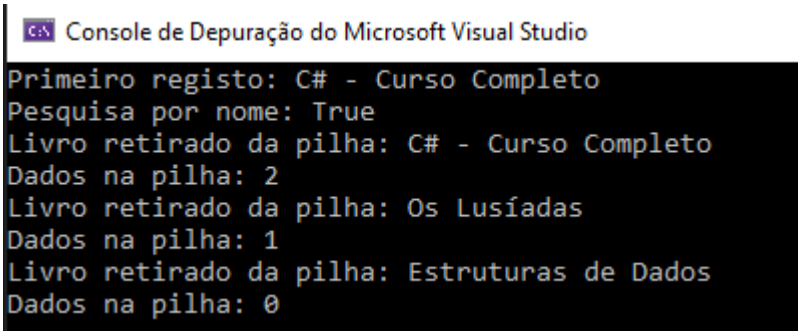
```
//Criar uma pilha de dados
Stack<string> pilhaDeLivros = new Stack<string>();

//adicionar elementos com a função Push()
pilhaDeLivros.Push("Estruturas de Dados");
pilhaDeLivros.Push("Os Lusíadas");
pilhaDeLivros.Push("C# - Curso Completo");

//Mostra quem é o primeiro elemento da lista (sem remover)
Console.WriteLine("Primeiro registo: " + pilhaDeLivros.Peek());

//verificar se contém o elemento a pesquisar
Console.WriteLine("Pesquisa por nome: " + pilhaDeLivros.Contains("Os Lusíadas"));

//Percorrer a pilha
while (pilhaDeLivros.Count > 0)
{
    string livro = pilhaDeLivros.Pop();
    Console.WriteLine("Livro retirado da pilha: " + livro);
    Console.WriteLine("Dados na pilha: " + pilhaDeLivros.Count());
}
```



## Capítulo 14.2.3 – Listas – List<T>

Uma ***lista*** permite ***guardar múltiplos elementos*** pela ***ordem desejada,*** ou seja***,*** implementa uma ***tabela dinâmica***, ultrapassando a ***limitação*** das tabelas (sempre que não existe mais espaço na tabela, ela cresce), como vimos nos arrays.

Para além das ***operações*** de ***adicionar e remover*** elementos, permite:

- ***aceder diretamente*** aos elementos;
- ***ordenar*** os elementos;
- ***pesquisar*** elementos.

Permite as ***funcionalidades de um vetor*** acrescentando uma ***maior flexibilidade***.

**Declarar e criar**

```
Declarar e criar (T = tipo dos elementos):
    Sintaxe: List<T> nome = new List<T>(); // declarar e criar
```

```
Exemplo1: List<int> lista = new List<int>(); // lista criada com 0 elementos
Exemplo2: List<int> lista = new List<int>(20); // lista criada com 0 elementos,
                                               // reservando espaço para 20 elementos
        if (lista = = null)  // se a lista ainda não foi criada
          lista = new List<int>();  // cria a lista
```

**Nota importante:** Pode criar listas com outro tipo de dados (double, string, etc…);

**Trabalhar com dados simples** (do tipo inteiro)

```
//criar lista de valores (tipo de dados simples)
List<int> lista_inteiros = new List<int>();
```

**Trabalhar com dados estruturados** com **classe** (**classe Produto**)

```
//criar lista de valores (usando classes)
List<Produto> lista_produtos = new List<Produto>();
```

***Adicionar elementos à lista***

Por omissão os elementos são adicionados no fim. No entanto, também se pode inserir elementos na posição desejada.

***A classe List<T> oferece dois métodos para efectuar estas operações:***

- ***Add:***
  ***Sintaxe:*** *nome.Add(elemento);*
- ***Insert:***
  ***Sintaxe:*** *nome. Insert(indice, elemento);*

**Trabalhar com dados simples** (do tipo inteiro)

```
//inserir dados com a função ADD()
lista_inteiros.Add(9);
lista_inteiros.Add(5);
lista_inteiros.Add(7);
lista_inteiros.Add(3);
lista_inteiros.Add(7);

//inserir dados numa indice especifico INSERT()
lista_inteiros.Insert(1, 31);
```

**Trabalhar com dados estruturados** com **classe** (**classe Produto**)

```
//inserir dados com a função ADD + construtor da classe Produto
lista_produtos.Add(new Produto(1, "GTA V", 59.99));
lista_produtos.Add(new Produto(2, "IPhone 17", 1000.99));
lista_produtos.Add(new Produto(3, "Huawei P70", 759.99));
lista_produtos.Add(new Produto(4, "Router AX3000", 49.99));

//inserir dados numa indice especifico INSERT()
lista_produtos.Insert(3, new Produto(5, "Impressora HP", 45.00));
```

### Propriedade Contagem do número de elementos

*Número de Elementos (só de leitura)*
*Sintaxe: nome.Count;*

```
Exemplo1:            // lista inicial   {10,5,15}
    int x = lista.Count;      // 3
    // lista.Count = 4;       // erro, count é Read-Only; só se pode ler o valor
```

**Trabalhar com dados simples** (do tipo inteiro)

```
//contar quantos elementos existem na lista com a função COUNT()
Console.WriteLine("Contagem: " + lista_inteiros.Count());
```

**Trabalhar com dados estruturados** com **classe** (**classe Produto**)

```
//contar quantos elementos existem na lista com a função COUNT()
Console.WriteLine("Contagem: " + lista_produtos.Count());
```

### Aceder a um elemento específico

*Do mesmo modo que se acede aos elementos de um vector (operador [ ])*
*Sintaxe: nome[índice]; // índice em [0..tamanho-1]*

```
Exemplo1:            // lista inicial   {10,5,15}
    lista[1] = 3;                //       {10,3,15}
    // lista[3] = 7;     // erro, esse elemento ainda não existe
```

**Trabalhar com dados simples** (do tipo inteiro)

```
//Aceder ao indice especifico
Console.WriteLine($"Valor do indice 3: {lista_inteiros[3]}");
```

**Trabalhar com dados estruturados** com **classe** (**classe Produto**)

```
//Aceder ao indice especifico
Console.WriteLine($"Valor do indice 3: {lista_produtos[3].Mostra_Produto()}");
```

**Percorrer a lista**

Podemos percorrer a lista com todos os elementos. Mas neste caso e para a próxima coleção, vamos usar uma 4 estrutura de repetição chamado **foreach()**. Esta estrutura de repetição permite percorrer cada valor individualmente (dado elementar simples ou classe) de uma coleção e não precisa de indicar índices (pois já percorre isso automaticamente).

Para poder utilizar a estrutura do **foreach**, vamos fazer a seguinte estrutura (onde podemos ler da seguinte forma: Para cada valor dentro da coleção):

- **Trabalhar com dados simples** (exemplo com números do tipo inteiro)
  Neste caso, temos de dizer que para cada valor que vai percorrer na coleção tem de armazenar o elemento numa variável temporária (neste caso **int valor** (onde **valor** é um nome sugestivo para a variável), e, mais à frente vamos indicar que essa informação está dentro (**in**) da coleção que está a percorrer (neste exemplo é a coleção **lista_inteiros**):

```
//Percorrer a lista com a estrutura de repetição FOREACH
foreach (int valor in lista_inteiros)
{
    Console.WriteLine(valor);
}
```

- **Trabalhar com dados estruturados** com **classe** (**classe Produto**)
  Neste caso, temos de dizer que para cada valor que vai percorrer na coleção tem de armazenar o elemento numa variável temporária (neste caso como estamos a trabalhar com uma classe Produto (dados estruturados), ou seja **Produto item** (onde **item** é um nome sugestivo para a variável), e, mais à frente vamos indicar que essa informação está dentro (**in**) da coleção que está a percorrer (neste exemplo é a coleção **lista_produtos**):

```
//Percorrer a lista com a estrutura de repetição FOREACH
foreach (Produto item in lista_produtos)
{
    Console.WriteLine(item.Mostra_Produto());
}
```

***Remover elementos da lista***

Não existe nenhuma predefinição do local, início ou fim, onde os elementos são removidos.

Pode-se remover um elemento através da sua identificação ou através do índice

*A classe List<T> oferece métodos para efectuar estas operações:*

- *Remove: // remover por identificação do elemento, removendo a 1.ª ocorrência*
  *Sintaxe: bool res = nome.Remove(elemento); // em res fica true , se op. c/ sucesso*
- *RemoveAt: // remover por índice*
  *Sintaxe: nome. RemoveAt(indice);*
- *Clear: // remove todos os elementos*
  *Sintaxe: nome. Clear();*

```
Exemplo1:          // lista inicial {10,5,15,5}
    bool res1 = lista.Remove(5)     // true   {10,15,5}
    bool res2 = lista.Remove(7)     // false, o elemento 7 não existe, {10,15,5}
    lista.Remove(15);               // true   {10,5}
Exemplo2:          // lista inicial {10,5,15,5}
    lista.RemoveAt(3)       // {10,5,15}
    lista.RemoveAt(0)       // {5,15}
```

**Trabalhar com dados simples** (do tipo inteiro)

```
//Remover dados da lista
//Remover por valor (devolve true se conseguiu ou caso contrário FALSE)
Console.WriteLine("Remover pelo valor: " + lista_inteiros.Remove(7));

//Remover por indice
lista_inteiros.RemoveAt(1);

//Limpar a lista com a função CLEAR()
lista_inteiros.Clear();
```

**Trabalhar com dados estruturados** com **classe** (**classe Produto**)

```
//Remover dados da lista
//Remover por valor (devolve 1 se conseguiu ou caso contrário 0)
Console.WriteLine("Remover pelo valor: " + lista_produtos.RemoveAll(x => x.CodID == 2));

//Remover por indice
lista_produtos.RemoveAt(1);

//Limpar a lista com a função CLEAR()
lista_produtos.Clear();
```

### *Ordenar elementos da lista*

Permite ***ordenar*** a lista por ordem ***crescente*** dos elementos

Para ordenar por ordem ***decrescente***:

***Percorrer*** a lista por ***ordem inversa ou ordenar*** e depois ***inverter*** a lista

***A classe List<T> oferece dois métodos para efectuar estas operações:***

- ***Sort:*** *// ordena a lista por ordem crescente*
  ***Sintaxe:*** ***nome.Sort();***
  ***Exemplo:*** *// em lista está {10,5,15}* → lista.Sort() // {5,10,15}
- ***Reverse:*** *// inverter a lista*
  ***Sintaxe:*** ***nome. Reverse();***
  ***Exemplo:*** *// em lista está {5,10,15}* → lista.Reverse() // {15,10,5}

### *Pesquisar elementos da lista*

- Permitem a ***pesquisa*** de elementos em lista ***ordenada*** ou ***não***
- Para pesquisar um elemento na lista pode-se:
- ***Percorrer*** a lista ***"manualmente"*** e procurar***;***
- ***Invocar*** um dos ***métodos*** oferecidos***.***
- Exemplos:

```
Exemplo1: Pesquisar em lista não ordenada     // lista inicial {10,5,15}
    bool res1 = lista.Contains(5);        // true
    bool res2 = lista.Contains(7);        // false
Exemplo2: Pesquisar e saber 1ª ocorrência     // lista inicial {10,5,15,5}
    int res1 = lista.IndexOf(5)   // res1=1
    int res2 = lista.IndexOf(30)  // res2=-1 (não encontra = -1)
Exemplo3: Pesquisar e saber última ocorrência  // lista inicial {10,5,15,5}
    int res1 = lista.LastIndexOf(5)       // res1=3
    inr res2 = lista.LastIndexOf(30)      // res2=-1 (não encontra = -1)
```

**Trabalhar com dados simples** (do tipo inteiro)

```
//Pesquisa por chave (devolve true se encontrar, senão devolve false)
bool res = lista_inteiros.Contains(3);
Console.WriteLine($"Encontrou a chave 3?: {res}");

//Verificar se o produto existe antes de inserir
//o simbolo ! permite indicar se não existe o que está à procura
if (!lista_inteiros.Contains(3))
{
    lista_inteiros.Add(3);
}
else
{
    Console.WriteLine("O inteiro já existe");
}
```

**Trabalhar com dados estruturados** com **classe** (**classe Produto**)

```
//Pesquisa por chave (devolve true se encontrar, senão devolve false)
bool resultado = lista_produtos.Any(x => x.CodID == 3);
Console.WriteLine($"Encontrou a chave 3?: {resultado}");

//Verificar se o produto existe antes de inserir
//o simbolo ! permite indicar se não existe o que está à procura
if (!lista_produtos.Any(x => x.CodID == 3))
{
    lista_produtos.Add(new Produto(3, "Huawei P70", 759.99));
}
else
{
    Console.WriteLine("O ID já existe");
}
```

## Capítulo 14.2.4 – Listas Ordenadas – SortedList<key, Value>

As listas ordenadas representam uma coleção de pares (chave, valor). A sua organização é baseada na chave, com acesso muito eficiente. Os pares (chave, valor) são mantidos ordenados, baseados no valor da chave.

- Uma ***lista ordenada*** permite ***guardar múltiplos elementos ordenadamente***.
- ***Um elemento*** é ***constituído*** por ***uma chave (key)*** e por ***um valor (value)***.
  - Par Chave-Valor ***(KeyValuePair)***
- Uma ***chave identifica*** um ***único elemento*** e funciona como ***um índice de acesso aos valores***.
- ***Não podem*** existir ***duas chaves iguais***.
- A ***ordem*** pela qual os ***valores são ordenados*** é ditada ***pela chave***.

A chave funciona como um índice de acesso aos valores. Vamos ver exemplos:

- **Conjunto de faturas**
  - Cada **fatura** tem um **número único (chave)** e tem associado um **custo total (valor)**.
- **Conjunto de imagens**
  - Cada **imagem** é identificada por um **código (chave)** e tem associado o **nome do ficheiro (valor)**.
- **Conjunto de músicas**
  - Cada **música** é identificada por um **código (chave)** e tem associado um **conjunto de detalhes (valor)**.

Mas, **também podemos usar classes** em vez de dados do tipo simples (int, string, double, etc…). Vamos ver alguns exemplos:

*Declarar e Criar:(Tk/Tv = tipo de chave/valor):*
*Sintaxe:* `SortedList<Tk,Tv> nome = new SortedList<Tk,Tv>();`

*Exemplo 1:* **(chave do tipo inteiro e valor do tipo texto)**
```
SortedList<int, string> lista = new SortedList<int, string>();
```
*Exemplo 2:* **(chave do tipo texto e valor do tipo real)**
```
SortedList<string, int> lista2 = new SortedList<string, int>();
```
*Exemplo 3:* **(chave do tipo inteiro e valor do tipo Classe)**
```
SortedList<int, Produto> lista3 = new SortedList<int, Produto>();
```

**Trabalhar com dados simples:** chave como **inteiro** (**int**) e valor com **texto** (**string**)

```
//Criar lista de valores (dados simples)
SortedList<int, string> dados = new SortedList<int, string>();
```

**Trabalhar com dados estruturados:** chave como **inteiro** (**int**) e valor com **classe** (**classe Produto**)

```
//Criar lista de valores (usando classes)
SortedList<int, Produto> produtos = new SortedList<int, Produto>();
```

***Adicionar elementos à lista***

- Os elementos são ***adicionados*** na ***ordem*** correta (***definida pela chave***).
- Para ***adicionar um elemento*** é necessário indicar o ***par chave-valor***.
- A classe ***SortedList<Tk,Tv>*** oferece um ***método*** para efetuar esta operação: ***Add***(chave, valor)
- Pode também utilizar-se o ***operador [ ]***, onde o índice é a chave.

***Adicionar elementos:***
***Sintaxe:*** ***nome.Add(chave, valor)***
***Exemplo1:*** ***(Adicionar o valor "texto" com a chave 10):***
*// a lista contém inicialmente {(5,"meu"),(15,"para"),(20,"ver")}*
***slista.Add(10, "texto");***
*// a lista fica {(5,"meu"),(10,"texto"),(15,"para"),(20,"ver")}*
*// slista.Add(5, "teu");  // erro, chave já existe*

**Trabalhar com dados simples:** chave como **inteiro** (**int**) e valor com **texto** (**string**)

```
//Criar lista de valores (dados simples)
SortedList<int, string> dados = new SortedList<int, string>();

//adicionar elementos
dados.Add(1, "GTA V");
dados.Add(2, "IPhone 17");
dados.Add(3, "Huawei P70");
dados.Add(4, "Router AX3000");
```

**Trabalhar com dados estruturados:** chave como **inteiro** (**int**) e valor com **classe** (**classe Produto**)

```
//Criar lista de valores (usando classes)
SortedList<int, Produto> produtos = new SortedList<int, Produto>();

//adicionar elementos
produtos.Add(1, new Produto(1, "GTA V", 59.99));
produtos.Add(2, new Produto(2, "IPhone 17", 1000.99));
produtos.Add(3, new Produto(3, "Huawei P70", 759.99));
produtos.Add(4, new Produto(4, "Router AX3000", 49.99));
```

***Propriedade Count***

```
Exemplo1: {(5,ab),(10,cd),(15,ef)}
    int x = slista.Count;       // 3
    // slista.Count = 4;        // erro, só se pode ler o valor
```

Quer sejam dados elementares ou com dados estruturados (por exemplo uma classe), a propriedade funciona da mesma forma:

- **Trabalhar com dados simples:** chave como **inteiro** (**int**) e valor com **texto** (**string**)

```
//Contar quantos elementos existem
Console.WriteLine($"Nº de elementos na lista ordenada: {dados.Count}");
```

- **Trabalhar com dados estruturados:** chave como **inteiro** (**int**) e valor com **classe** (**classe Produto**)

```
//Contar quantos elementos existem
Console.WriteLine($"Nº de elementos na lista ordenada: {produtos.Count}");
```

***Aceder aos valores***

Acede-se a um elemento da lista através da chave e não do índice.

- ***nome[chave]*** // obter/atribuir valor

```
Exemplo1: {(5,ab),(10,cd),(15,ef)}
    slista[15] = "gh";          // {(5,ab),(10,cd),(15,gh)}
    //slista[0];                // erro, essa chave não existe
Exemplo2: {(a,15),(d,5),(e,10)}
    slista['a'] = 20;           // {(a,20),(d,5),(e,10)}
    //slista['g'];              // erro, essa chave não existe
```

**Trabalhar com dados simples:** chave como **inteiro** (**int**) e valor com **texto** (**string**)

```
//Aceder ao valor de uma chave
Console.WriteLine($"Valor da chave 3: {dados[3]}");
```

**Trabalhar com dados estruturados:** chave como **inteiro** (**int**) e valor com **classe** (**classe Produto**)

```
//Aceder ao valor de uma chave
Console.WriteLine($"Valor da chave 3: {produtos[3].Mostra_Produto()}");
```

(Mas, neste caso, também pode obter os dados do CodID, Descrição e Preço em vez do MostraMensagem() que foi desenvolvido nos métodos adicionais da classe).

***Remover elementos à lista***

Normalmente um elemento é ***removido*** através da ***chave***.

Pode-se ***também remover*** um elemento através do ***índice relativo*** (menos utilizado).

A classe ***SortedList<Tk,Tv>*** oferece ***métodos*** para efetuar estas operações:

- ***Remove***(chave)
- ***Clear***() # remove todos os elementos

```
Remover através da chave:
    Sintaxe: bool res = nome.Remove(chave)
Exemplo1: {(5,ab),(10,cd),(15,ef)}
    bool res1 = slista.Remove(10)  // true   {(5,ab),(15,ef)}
    bool res2 = slista.Remove(7)   // false
    slista.Remove(15);             // true   {(5,ab)}

Remover por índice:
    Sintaxe: bool res = nome.RemoveAt(indice)
Exemplo2: {(5,ab),(10,cd),(15,ef)}
    bool res1 = slista.RemoveAt(2);        // true {(5,ab),(10,cd)}
    bool res2 = slista.RemoveAt(0);        // {(10,cd)}
```

Quer sejam dados elementares ou com dados estruturados (por exemplo uma classe), a propriedade funciona da mesma forma:

- **Trabalhar com dados simples:** chave como **inteiro** (**int**) e valor com **texto** (**string**)

```
//Remover dados atraves da chave
bool res = dados.Remove(3);
Console.WriteLine($"Produto 3 foi removido?: {res}");

//Apagar toda a lista
dados.Clear();
```

- **Trabalhar com dados estruturados:** chave como **inteiro** (**int**) e valor com **classe** (**classe Produto**)

```
//Remover dados atraves da chave
bool resultado = produtos.Remove(3);
Console.WriteLine($"Produto 3 foi removido?: {resultado}");

//Apagar toda a lista
produtos.Clear();
```

***Pesquisar elementos na Lista***

Permitem a ***pesquisa*** de elementos na ***lista ordenada***.

A ***pesquisa*** pode ser feita através:

- ***da chave***;
- ***do valor***;

A classe ***SortedList<Tk,Tv>*** oferece ***métodos*** para estas operações:

- ***ContainsKey***(k), ***ContainsValue***(v);

```
Verificar se um elemento existe através da chave:
    Sintaxe: bool res = nome.ContainsKey(chave)
Exemplo1: {(5,ab),(10,cd),(15,ef)}
    bool r1 = slista.ContainsKey(10)        // true
    boot r2 = slista.ContainsKey(7)         // false

Verificar se um elemento existe através do valor:
    Sintaxe: bool res = nome.ContainsValue(valor)
Exemplo2: {(5,ab),(10,cd),(15,ef)}
    bool r1 = slista.ContainsValue("cd")    // true
    boot r2 = slista.ContainsValue("gh")    // false
```

**Trabalhar com dados simples:** chave como **inteiro** (**int**) e valor com **texto** (**string**)

```
//Pesquisa por chave (devolve true se encontrar, senão devolve false)
bool res1 = dados.ContainsKey(2);
Console.WriteLine($"Encontrou a chave 2?: {res1}");
```

```
Encontrou a chave 3?: False
```

Também pode expandir a funcionalidade com uma estrutura de decisão simples e verificar se o produto não existe com a função ***ContainsKey()*** (o símbolo ! permite indicar se não existe o que está à procura), poderá adicionar algo, caso contrário, mostra a mensagem que o produto já existe:

```
//Verificar se o produto existe antes de inserir
//o simbolo ! permite indicar se não existe o que está à procura
if (!dados.ContainsKey(3)) {
    dados.Add(3, "Huawei P70");
}
else {
    Console.WriteLine("O produto já existe");
}
```

**Trabalhar com dados estruturados:** chave como **inteiro** (**int**) e valor com **classe** (**classe Produto**)

```
//Pesquisa por chave (devolve true se encontrar, senão devolve false)
bool result1 = produtos.ContainsKey(3);
Console.WriteLine($"Encontrou a chave 3?: {result1}");

//verificar se o produto existe antes de inserir
//o simbolo ! permite indicar se não existe o que está a procura
if (!produtos.ContainsKey(3)) {
    produtos.Add(3, new Produto(3, "Huawei P70", 759.99));
}
else {
    Console.WriteLine("O produto já existe");
}
```

***Percorrer elementos na Lista***

Podemos percorrer a lista sorteada por com todos os elementos (chave e valor), apenas chaves ou apenas valores:

- ***com chaves e valores (dados elementares)***

```
//Com chaves e valores
Console.WriteLine("\n***** Lista de Dados *****");
foreach (KeyValuePair<int, string> item in dados)
{
    //Obter a chave que está a percorrer atualmente
    int chave = item.Key;
    //Obter o valor que está a percorrer atualmente
    string valor = item.Value;

    //mostrar qual a chave e o valor do produto (com base na classe)
    Console.WriteLine($"Chave: {chave} | Valor Produto: {valor}");
}
```

- ***com chaves e valores (dados com classes)***

```
//Com chaves e valores
Console.WriteLine("\n***** Lista de Produtos (com base na classe Produto) *****");
foreach (KeyValuePair<int, Produto> item in produtos)
{
    //Obter a chave que está a percorrer atualmente
    int chave = item.Key;

    //Obter o valor que está a percorrer atualmente
    //Como estamos a trabalhar como uma estrutura de informação, temos de indicar
    //a classe que vai ler a estrutura da informação (neste caso Produto)
    Produto valor = item.Value;

    //mostrar qual a chave e o valor do produto (com base na classe)
    Console.WriteLine($"Chave: {chave} | Valor Produto: {valor.Mostra_Produto()}");

    //também pode aceder aos valores de cada variavel do produto
    Console.WriteLine($"ID do Produto: {valor.CodID}");
    Console.WriteLine($"Descrição do Produto: {valor.Descricao}");
    Console.WriteLine($"Preço do Produto: {valor.Preco}");
    Console.WriteLine("\n");
}
```

- ***apenas chaves (dados elementares)***

```
//Percorrer apenas chaves
//Utilize a propriedade da lista Keys para obter apenas as chaves
Console.WriteLine("\n***** Lista das chaves dos Dados *****");
foreach (int chave in dados.Keys)
{
    Console.WriteLine($"Chave: {chave}");
}
```

- ***apenas chaves (dados com classes)***

```
//Percorrer apenas chaves
//Utilize a propriedade da lista Keys para obter apenas as chaves
Console.WriteLine("\n***** Lista das chaves dos Produtos *****");
foreach (int chave in produtos.Keys)
{
    Console.WriteLine($"Chave: {chave}");
}
```

- ***apenas valores (dados elementares)***

```
//Percorrer apenas valores
//Utilize a propriedade da lista Values para obter apenas os valores
//Como estamos a trabalhar como uma estrutura de informação, temos de indicar
//a classe que vai ler a estrutura da informação (neste caso Produto)
Console.WriteLine("\n***** Lista dos valores dos Dados *****");
foreach (string item in dados.Values)
{
    Console.WriteLine($"Valor: {item}");
}
```

- ***apenas valores (dados com classes)***

```
//Percorrer apenas valores
//Utilize a propriedade da lista Values para obter apenas os valores
//Como estamos a trabalhar como uma estrutura de informação, temos de indicar
//a classe que vai ler a estrutura da informação (neste caso Produto)
Console.WriteLine("\n***** Lista dos valores dos Produtos *****");
foreach (Produto item in produtos.Values)
{
    Console.WriteLine($"Valor: {item.Mostra_Produto()}");
    Console.WriteLine($"ID do Produto: {item.CodID}");
    Console.WriteLine($"Descrição do Produto: {item.Descricao}");
    Console.WriteLine($"Preço do Produto: {item.Preco}");
    Console.WriteLine("\n");
}
```

# Bibliografia

OLIVEIRA, Ana; MENDES, Paulo; RAMOS, Carla. *Estruturas de dados em C#: Princípios e práticas*. 1. ed. Porto: Porto Editora, 2019.

PRICE, Mark. C# 13 and .NET 9 – Modern Cross-Platform Development Fundamentals. Packt, 2024.

MARQUES, Paulo. Curso Prático de C#. FCA 2016

LOUREIRO, Henrique. C# 7.0 com Visual Studio. FCA, 2017

RODRIGUES, Diego. APRENDA C# - Edição 2025: Domine o Desenvolvimento Escalável com Programação Moderna. StudioD21 Smart Tech Content, 2025

TRIGO, António; Henrique, Jorge. Aprenda a Programar com C#. Edições Sílabo, 2023

JAPIKSE Phil; TROELSEN Andrew. Pro C# 10 with .NET 6. 11ª Edição. Apress 2022