

E-Book

.NET

Developer

Fundamentos

dio.



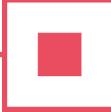
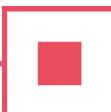
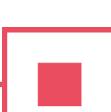
mensagem de recepção

Boas-vindas à nossa série de ebooks!
Preparamos estes materiais pensando em como transmitir mais conhecimento diversificado para você explorar diferentes meios de desenvolver suas habilidades.

Aproveite este material e bons estudos!



SUMÁRIO

- | | | |
|-----------|--|---|
| 04 | Introdução ao .net |  |
| 12 | Introdução às IDEs e configurações de ambiente |  |
| 24 | Sintaxe e seus tipos de dados |  |
| 39 | Tipos de operadores |  |
| 48 | Operadores aritméticos e a classe Math |  |
| 54 | Conhecendo as Estruturas de Repetição |  |
| 66 | Conhecendo a organização de um programa C# |  |
| 78 | Array e Listas |  |
| 88 | Comentários e boas práticas |  |
| 93 | Construindo um Sistema para um Estacionamento |  |

III Introdução ao .NET

História, Versões e usos

O .NET é uma plataforma Open Source de desenvolvimento unificado criada pela Microsoft, que permite a construção de sistemas e aplicações utilizando principalmente a linguagem de programação C#. O C# não é a única linguagem utilizada na plataforma .NET.

Possibilidades do .NET :

- Desktop
 - WPF
 - Windows Forms
 - UWP
- WEB
 - ASP.NET
- Cloud
 - Azure
- Mobile
 - Xamarin
- Gaming
 - Unity
- IoT
 - ARM32
 - ARM64
- AI
 - ML.NET
 - .NET for Apache Spark

Ferramentas para trabalhar com .NET e C# :

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- Command Line Interface



O começo do .NET

A Microsoft começou a trabalhar no C# no final dos anos 90, tendo sua primeira versão do framework lançada em 2002, com o objetivo de competir com o Java. No início a intenção era facilitar o desenvolvimento de aplicativos apenas no Windows, mas após a Microsoft perder um processo para a Sun Microsystems (responsável pelo Java), decidiram transformar o .NET em multiplataforma.

Linha do tempo do .NET

<https://time.graphics/pt/line/291016>

Diferenças .NET Framework e .NET (ou .NET Core)

O .NET Core surgiu depois do .NET framework e foi praticamente feita do zero com o objetivo de ser multiplataforma, visto que o .NET framework funciona apenas com Windows. Desde então, a Microsoft continua trabalhando em atualizações apenas no .NET Core.

WPF	Windows Forms	ASP.NET (4 & 5)	ASP.NET 5	Universal Windows Apps
			Core CLR	.Net Native
.NET Framework 4.6 <i>Full .NET Framework for any scenario and library support on Windows</i>			.NET Core 5 <i>Modular libraries & runtime optimized for server and cloud workloads</i>	
Common			NuGet packages .NET Core 5 Libraries .NET Framework 4.6 Libraries	
			Runtime Components Next gen JIT (RyuJIT) SIMD	
			Compilers .NET Compiler Platform (Roslyn) Languages innovation	

Versões .NET

A Microsoft decidiu encerrar novas implementações do .NET Framework na versão 4.8, focando agora no .NET Core que, para não confundir os usuários com as nomenclaturas entre Framework e Core, pulou da versão 3.1 para a versão 5.0, também removendo a palavra Core do nome.

Compilador do .NET e seu funcionamento

O compilador, de uma forma simplificada, é o programa que realiza a conversão de linguagem de alto nível para baixo nível. Ele é muito importante no .NET, pois sempre que o código for alterado, o programa deve ser compilado.



Linguagem de alto e baixo nível

- Linguagem de alto nível
 - A linguagem que entendemos e escrevemos nosso código fonte.
- Linguagem de baixo nível
 - A linguagem que a máquina entende. Possui pouca abstração, sendo difícil para um ser humano entender.

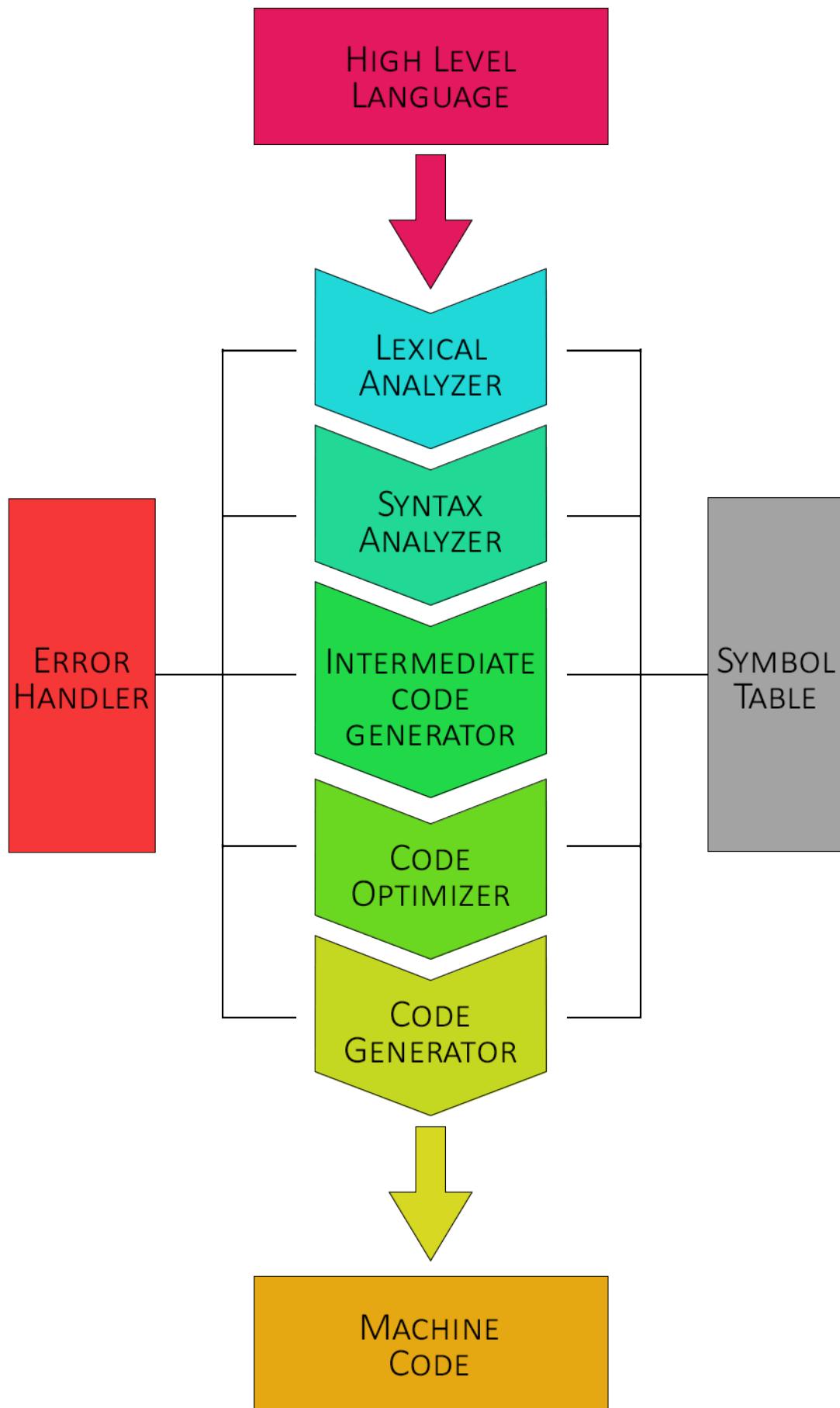
Exemplo de Hello, World em Assembly (baixo nível):

```
1 ; Exemplo de um Hello World em Assembly
2 ; ld -m elf_i386 -s -o hello hello.o
3 section .text align=0
4
5 global _start
6
7 mensagem db 'Hello world', 0x0a
8
9 len equ $ - mensagem
10
11 _start:
12     mov eax, 4 ;SYS_write
13     mov ebx, 1 ;Número do file descriptor (1=stdout)
14     mov ecx, mensagem ;Ponteiro para a string.
15     mov edx, len ; tamanho da mensagem
16     int 0x80
17
18     mov eax, 1
19     int 0x8
```

E um exemplo em C# (alto nível)

```
Console.WriteLine("Hello, World");
```

Fases de um compilador



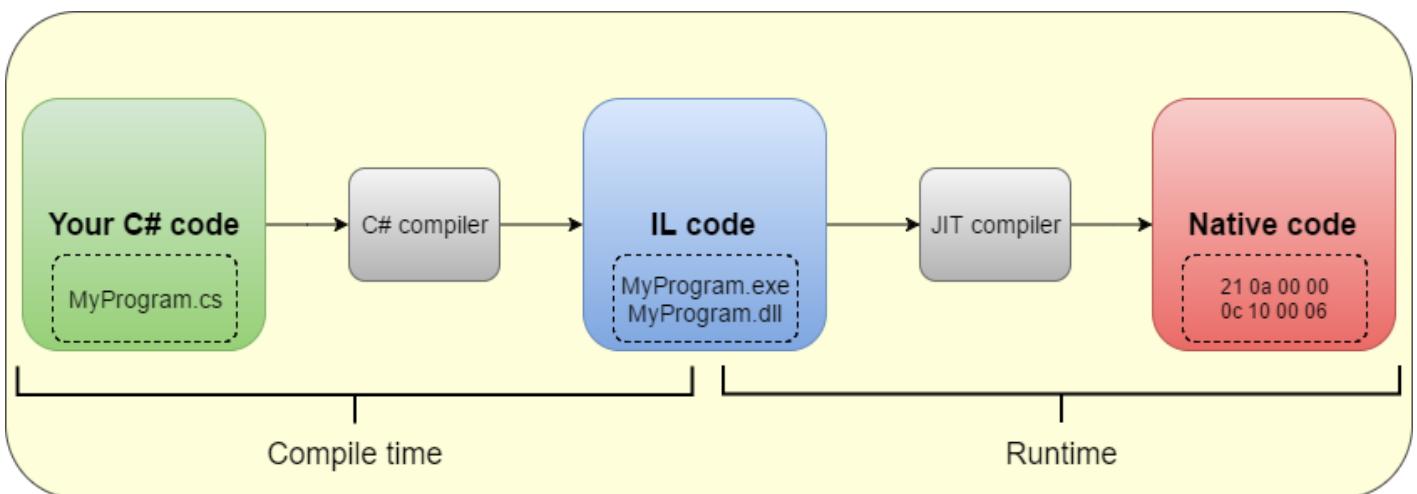
De forma simplificada, um compilador tem as seguintes etapas:

- High level Language
 - código escrito antes de passar pelo compilador
- Symbol Table
 - Consulta se o código digitado é válido da linguagem em questão
- Error Handler
 - Para cada etapa atual, será verificada a existência de erros e se ficou algum erro da etapa anterior
 - Caso encontre algum erro, o processo de compilação será interrompido
- Machine Code
 - Código de saída depois de passar pelo compilador

Etapas:

1. Lexical analyzer
 - a. Verifica se o que foi digitado faz sentido com a linguagem
2. Syntax analyzer
 - a. Verifica se a sintaxe digitada confere com a linguagem
3. Intermediate code generator
 - a. Gera um código um pouco mais próximo da linguagem de máquina, ficando no meio termo entre linguagem de alto e baixo nível e tornando mais fácil ser transformado em linguagem de máquina no final.
 - b. Serve para ser implementado numa máquina específica
4. Code optimizer
 - a. Verifica possíveis otimizações para o código , como remoções de comentários, variáveis sobrescritas, etc.
5. Code Generator
 - a. Gera o código final em linguagem de máquina

Compilador do .NET



Compile Time (Tempo de compilação)

- Your C# Code
 - Linguagem de alto nível
 - Diversas classes no projeto, que serão compiladas
- C# compiler
 - Pega todas as classes do projeto e faz o processo de compilação
 - Gera o código intermediário (IL code)
- IL code
 - Código independente de máquina, que não depende de uma arquitetura específica para executar.
 - Gera arquivos com extensões .exe e .dll
 - Programa pronto para ser executado

Runtime (Bibliotecas do .NET instaladas na máquina)

- JIT compiler
 - Na hora que o programa for executado, será compilado novamente pelo JIT (Just In Time) compiler
 - O código será convertido para a arquitetura específica da máquina em que foi executado
- Native code
 - Código de saída pronto para a arquitetura específica da máquina.

- Código Fonte:

```
BaseProcessor.cs X

public BaseProcessor(XmlNameTable nameTable, string[] schemaNames, EventHandler eventHandler, CompilationSettings compilationSettings)
{
    this.nameTable = nameTable;
    this.schemaNames = schemaNames;
    this.eventHandler = eventHandler;
    this.compilationSettings = compilationSettings;
    this.NsXml = nameTable.Add("http://www.w3.org/XML/1998/namespace");
}

protected void AddToTable(XmlSchemaObject item)
{
    if (qname.Name.Length == 0)
        return;
    XmlSchemaObject existingObject = GetObject(qname);
    if (existingObject != null)
    {
        if (existingObject == item)
            return;
        existingObject.Add(item);
    }
    else
        AddObject(qname, item);
}
```

- IL:

```
IL Viewer

IL_001e: stfld      class System.Xml.Schema.XmlSchemaCollection
                    nameTable
                    [71 7 - 71 73]
IL_0023: ldarg.0    // this
IL_0024: ldarg.1    // nameTable
IL_0025: ldstr      "http://www.w3.org/XML/1998/namespace"
IL_002a: callvirt   instance string System.Xml.XmlSchemaCollection::get_NameTable()
IL_002f: stfld      string System.Xml.Schema.XmlSchemaCollection::nameTable
IL_0034: ret

} // end of method BaseProcessor::ctor

.method family hidebysig specialname instance class System.Xml.Schema.XmlSchemaCollection get_NameTable() cil managed
{
    .maxstack 8
```

Compilador e Transpilador

Compilador é um programa que realiza a conversão de linguagem de alto nível para baixo nível.

Exemplo: C#, Java.

Transpilador é a conversão de uma linguagem ou implementação para outra. A sua saída permanece em linguagem de alto nível.

Exemplo: Typescript e Javascript.

NEM TODA LINGUAGEM É COMPIADA

Linguagem compilada são linguagens onde o código fonte é traduzido para o código de máquina.

Exemplo: C#, Java.

Linguagem interpretada são linguagens que fazem a leitura e interpretação diretamente do código fonte.

Exemplo: Javascript, PHP.

III Introdução às IDEs e configurações de ambiente

O que é uma IDE ?

Uma IDE (Integrated Development Environment), ou ambiente de desenvolvimento integrado, é um software que facilita e integra diversas facilidades para a escrita e depuração do código.

A IDE pode contar com diversas features essenciais para a programação. Uma delas é o Intellisense, que é um auto-completar do código. Além disso, pode-se utilizar esquema de cores para diferenciar classes, métodos, variáveis, etc.

Visual Studio

É a principal IDE para o .NET, com suporte para C#, C++, Python, Node.js, Unity e mobile.

Vantagens

- Versão gratuita (Community)
- Debug rico em detalhes
- Disponível para Windows e Mac

Desvantagens

- Não disponível para Linux
- Performance. Exige muitos recursos da máquina

Visual Studio Code

O VS Code é um editor de texto usado para facilitar o desenvolvimento de diversas linguagens. Ele não é considerado uma IDE, mas pode-se instalar diversas extensões para que ele se comporte como uma IDE.

Vantagens

- Totalmente gratuito
- Possibilidade de instalar extensões

- Disponível para Windows, Mac e Linux
- Muito leve em performance

Desvantagens

- É necessário uma configuração inicial
- Não muito intuitivo para algumas funcionalidades

Rider

Pode ser considerada uma IDE mais completa que o Visual Studio por conta das funcionalidades extras que oferece.

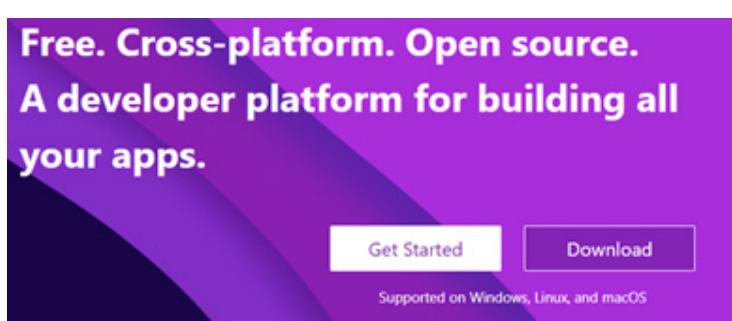
Vantagens

- Rico em funcionalidades
- Forte integração com o .NET
- Facilidade em trabalhar com o Unity
- Recomendações de refatoração de código
- Atalhos e comandos que aumentam a produtividade

Desvantagens

- Pago
- Performance. Exige muitos recursos da máquina

Instalando o .NET SDK



Site:

<https://dotnet.microsoft.com/>

Após Clicar em download, escolher em qual plataforma deseja instalar:
Window, Linux, macOS ou Docker

Para fins de desenvolvimento, selecione a opção SDK de acordo com a plataforma selecionada.

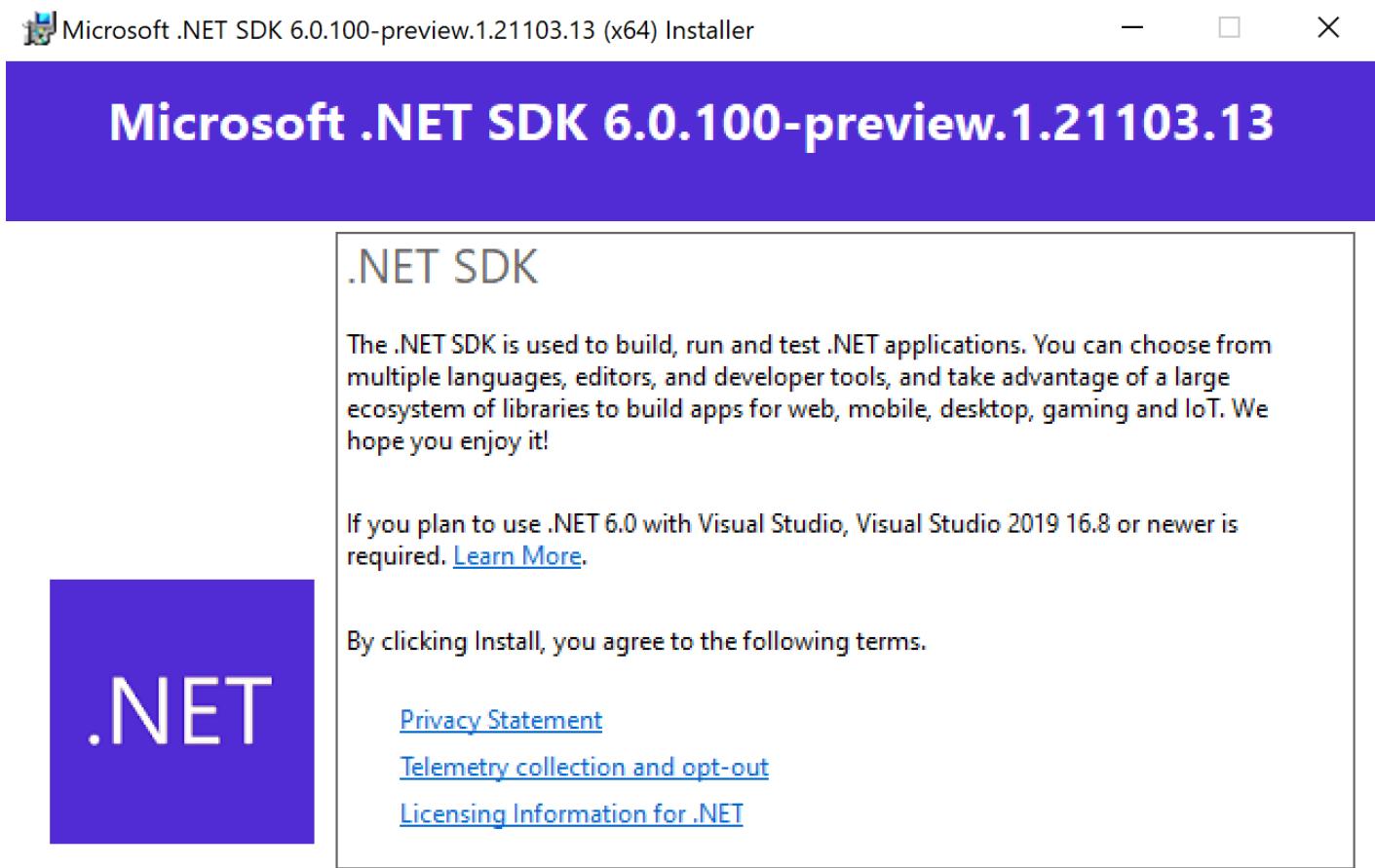
Para conferir outras versões, selecione:
All .NET Versions

.NET 6.0

LTS ⓘ

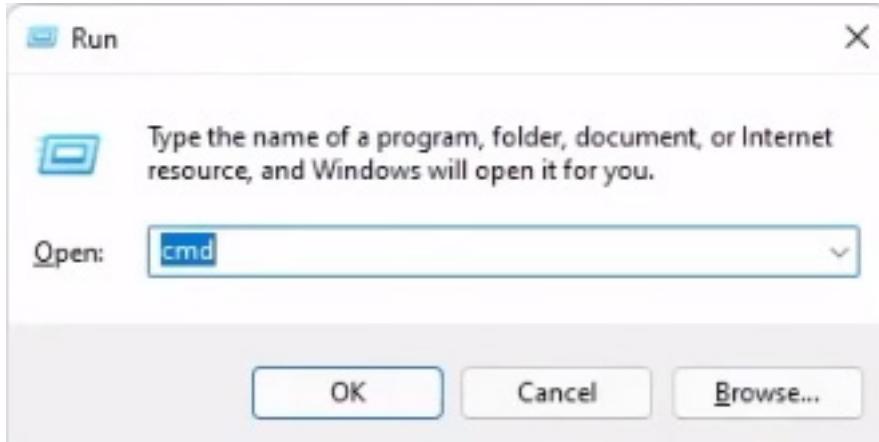
Após terminar de baixar, execute o instalador, a instalação é bem simples e intuitiva.

[Download .NET SDK x64](#) ⓘ



Podemos observar que o Runtime foi instalado junto do SDK pois além de desenvolver, a aplicação também precisará ser executada.

Para verificar no Windows se a instalação do SDK foi um sucesso, basta segurar a tecla com o símbolo do Windows no seu teclado e em seguida apertar a tecla R.



Na nova janela que surgir, digite **cmd** e clique em OK e abrirá o prompt de comando.

Prompt de comando:

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.22000.556]
(c) Microsoft Corporation. All rights reserved.

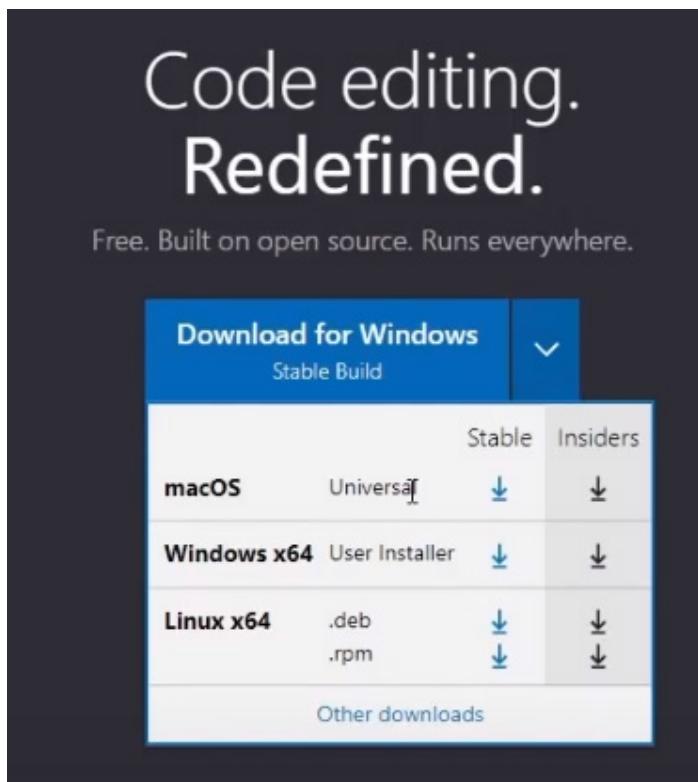
C:\Users\Buta>
```

```
.NET SDK (reflecting any global.json):
  Version: 6.0.201
  Commit: ef40e6aa06

Runtime Environment:
  OS Name: Windows
  OS Version: 10.0.22000
  OS Platform: Windows
  RID: win10-x64
  Base Path: C:\Program Files\dotnet\sdk\6.0.201\

Host (useful for support):
  Version: 6.0.3
  Commit: c24d9a9c91
```

Digite nele o seguinte: **dotnet --info**
Caso apareça algo parecido com a imagem, significa que ocorreu tudo bem na instalação e seu computador reconhece que uma versão do SDK



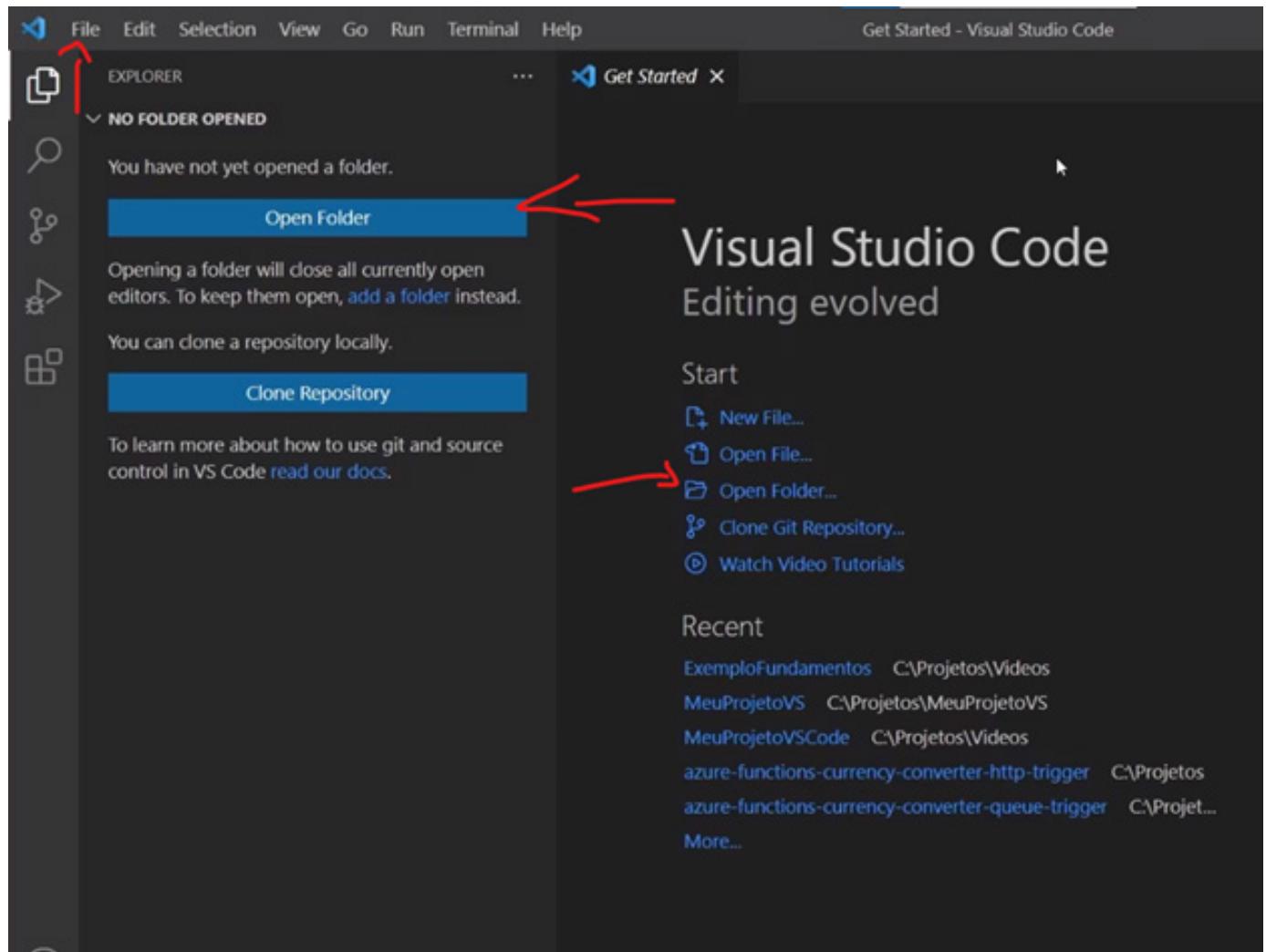
Instalando o VS Code

Site: <https://code.visualstudio.com>

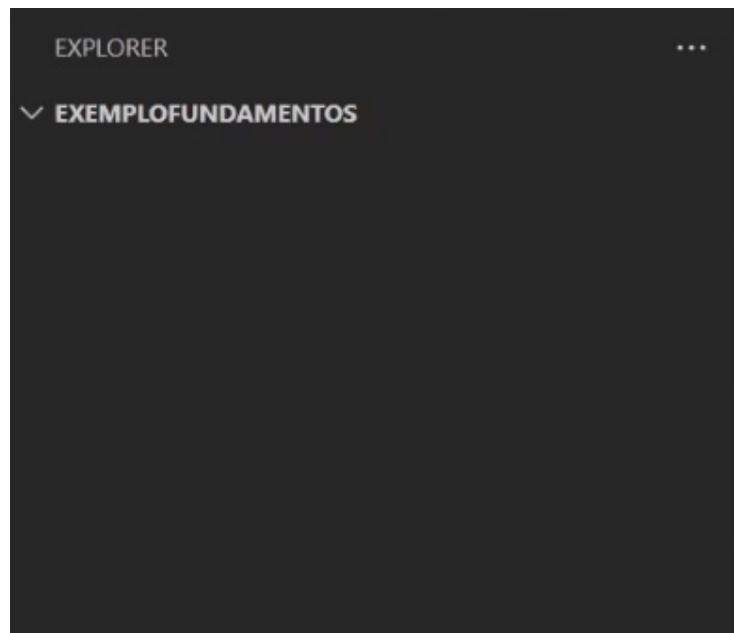
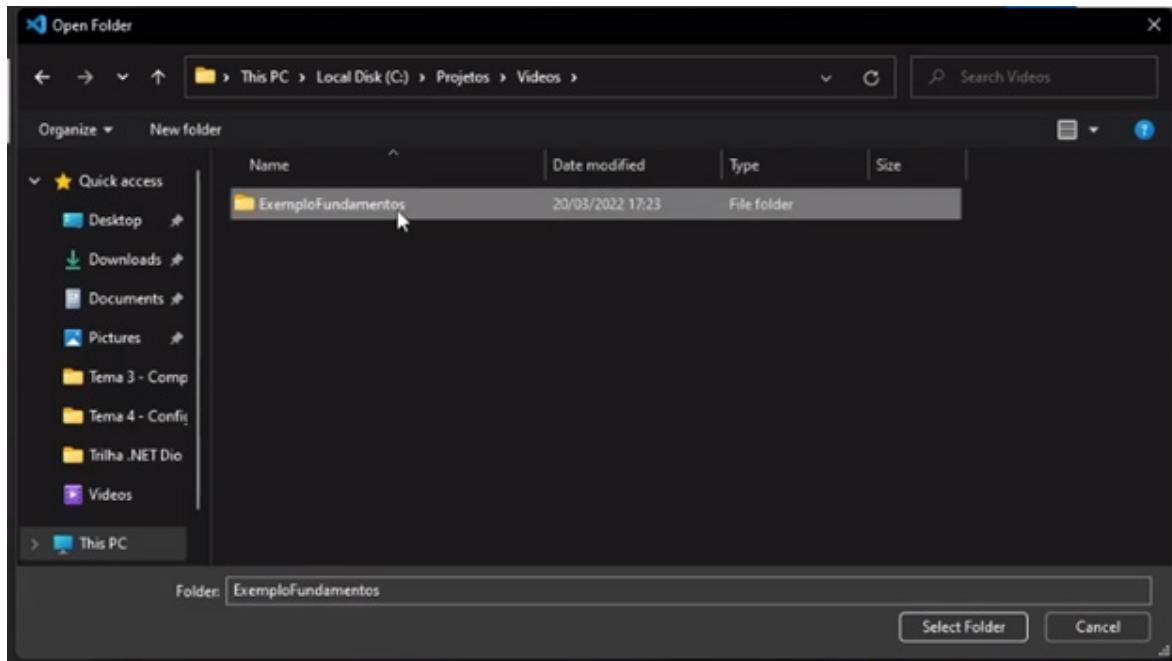
Após o download, execute o instalador. A instalação é simples e intuitiva.

Criando nosso projeto

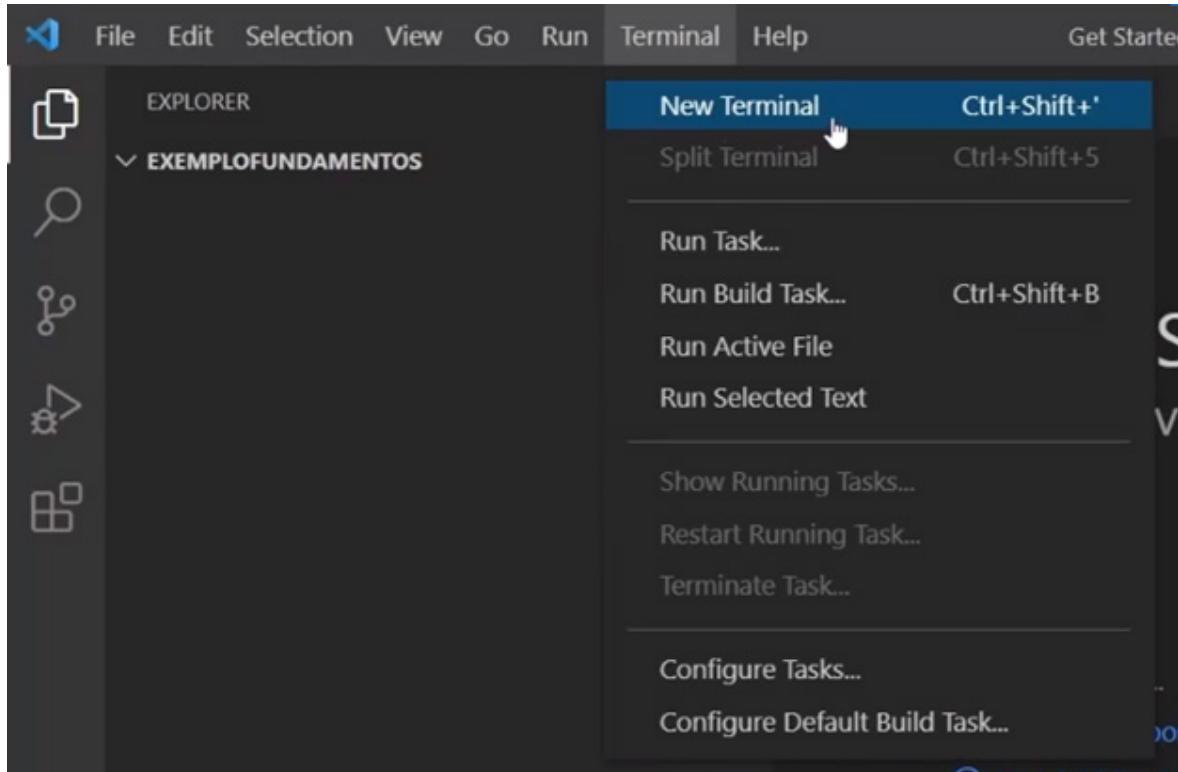
Primeiro, precisamos selecionar uma pasta vazia para dar início ao projeto. Ao abrir o VS Code, temos 3 opções para selecionar, conforme a imagem abaixo:



Criar uma pasta com o nome ExemploFundamentos e selecione-a:



Agora, abrir um novo terminal onde poderemos utilizar comandos do .NET para criar um projeto simples:



dotnet new console

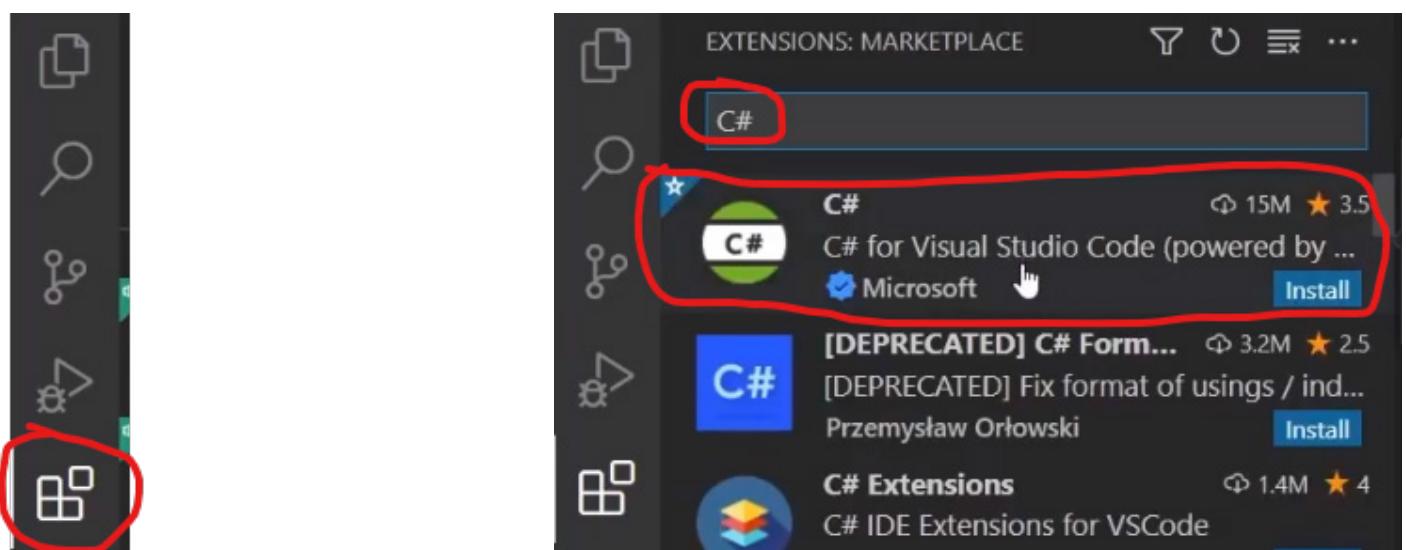
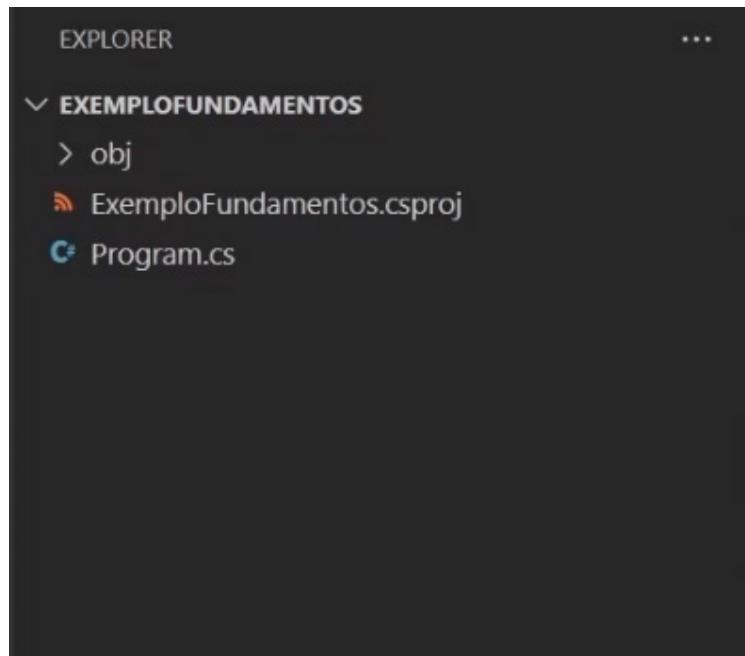
A screenshot of the Visual Studio Code terminal window. The tab bar at the top shows PROBLEMS, OUTPUT, TERMINAL (which is underlined in blue), and DEBUG CONSOLE. The terminal itself displays the following text:

```
PowerShell 7.2.2
Copyright (c) Microsoft Corporation.

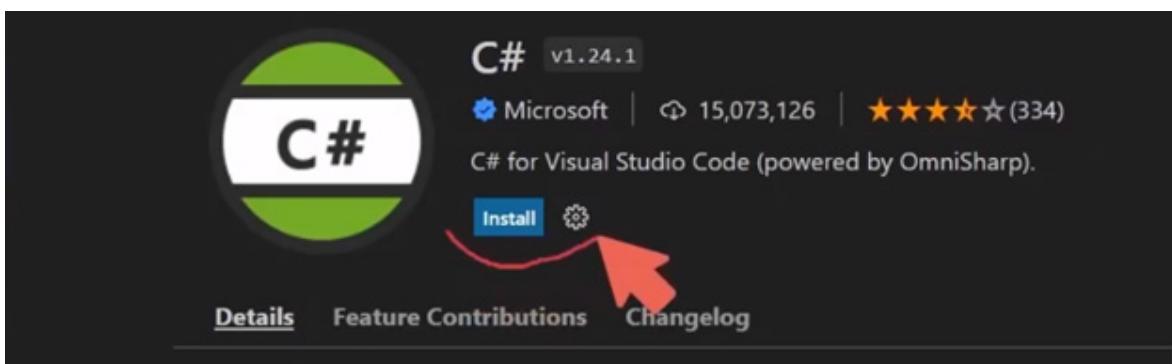
https://aka.ms/powershell
Type 'help' to get help.

PS C:\Projetos\Videos\ExemploFundamentos> dotnet new console[]
```

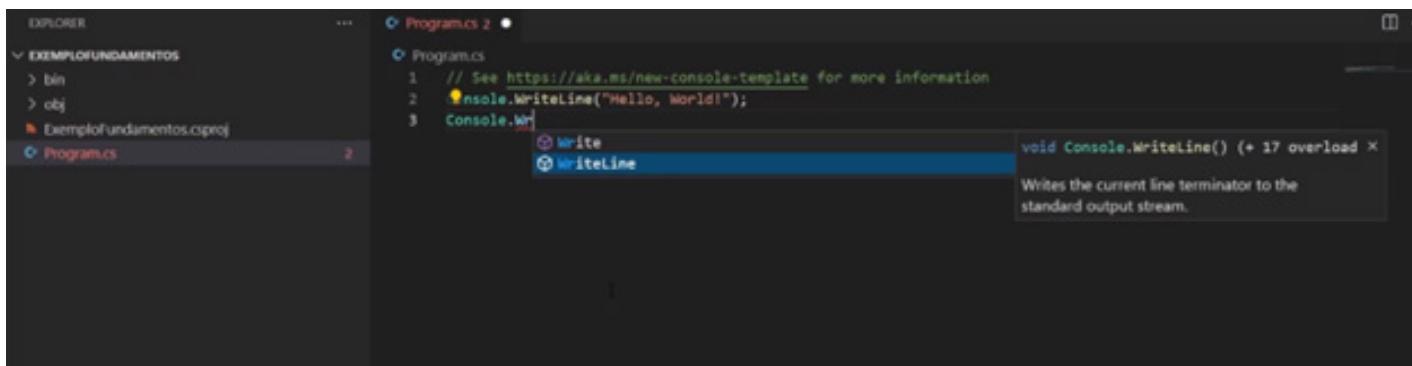
A large red arrow points from the bottom right towards the closing square bracket of the command line input.



Para ter o Intellisense da linguagem C#, precisamos instalar uma extensão para o VS Code



Após a instalação da extensão, podemos notar que ao começar a digitar no arquivo **.cs**, automaticamente o VS Code começa a dar sugestões para autocompletar o código.

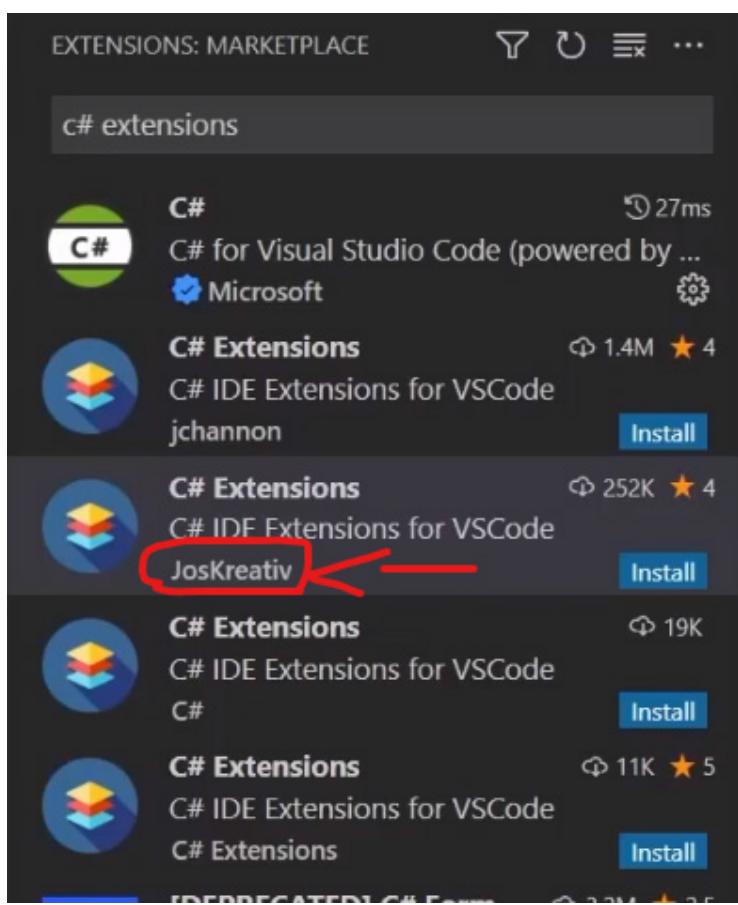
A screenshot of a terminal window titled 'Program.cs'. The window contains the following C# code:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
Console.WriteLine("Teste");
```

Depois, basta digitar no console o comando abaixo para executar seu primeiro programa em C# :

dotnet run

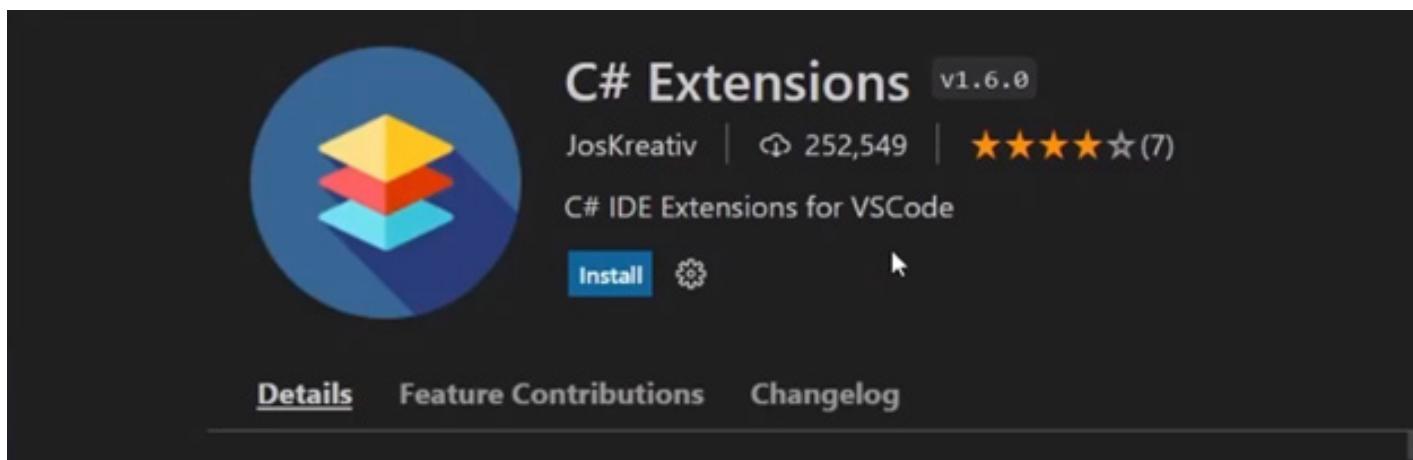
```
PS C:\Projetos\Videos\ExemploFundamentos> dotnet run
Hello, World!
Teste
```

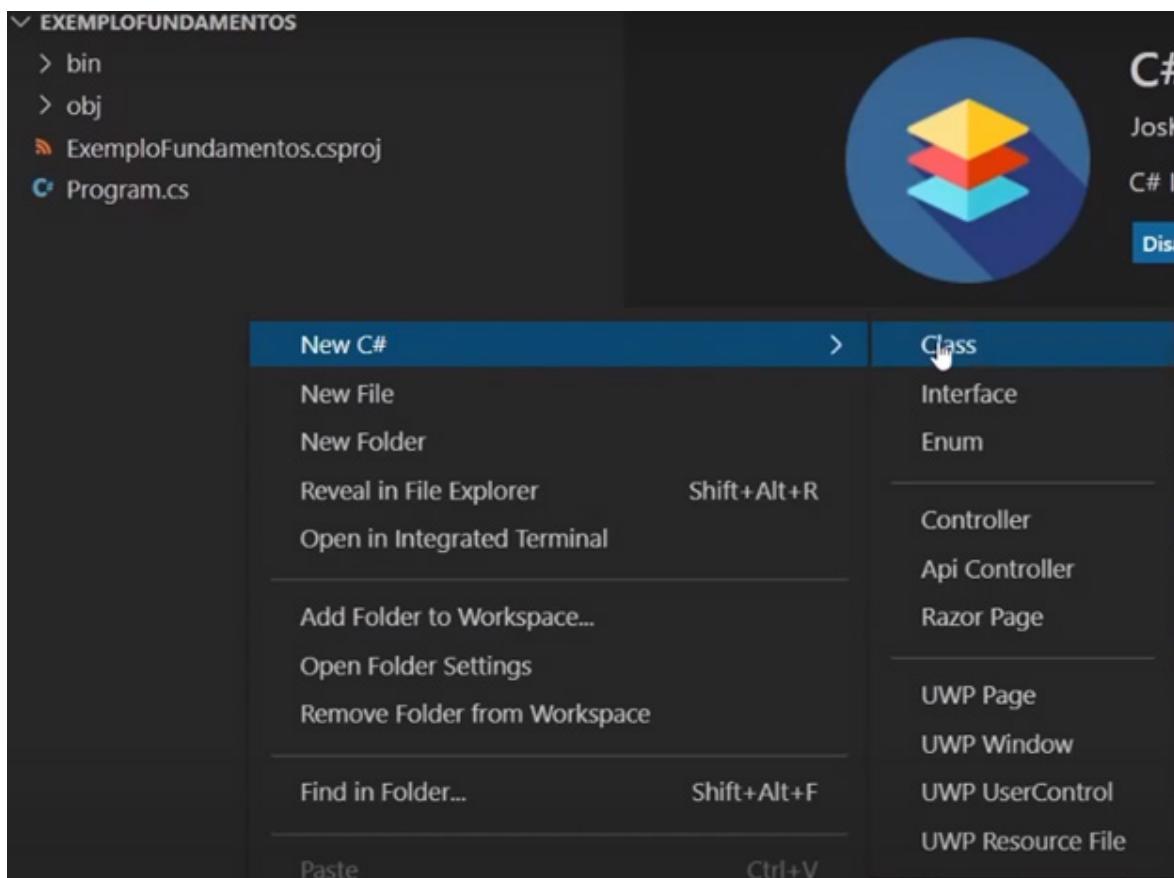


Extensões auxiliares

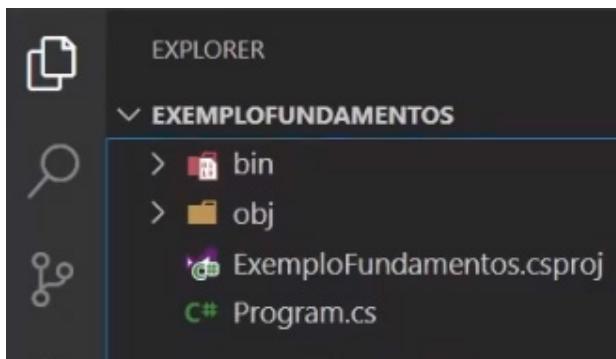
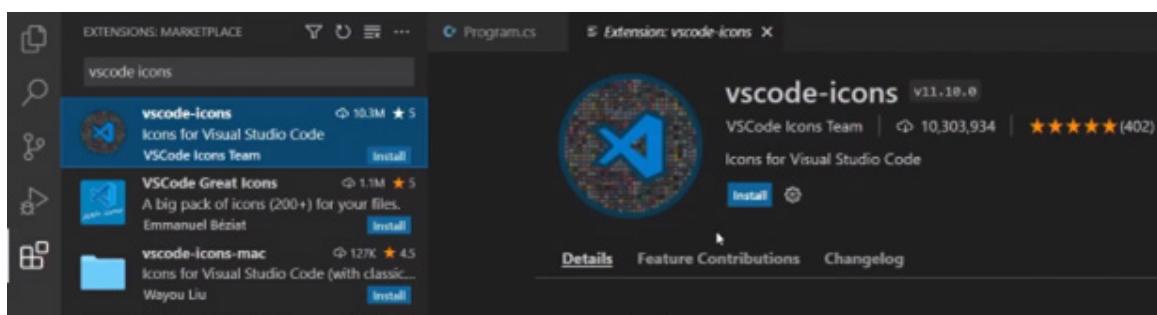
O VS Code pode oferecer diversas extensões complementares que fornecem funcionalidades extras. A que iremos instalar nos ajudará na criação de novas interfaces e classes.

Podemos observar que após a pesquisa pelo nome C# Extensions aparecem 4 opções diferentes. Vamos instalar aquela que se apresenta em segundo lugar no ranking de downloads, do criador JosKreativ, pois a que está em primeiro lugar parou de receber atualizações.





Agora, após instalar e ativar a extensão, ao clicar com o botão direito do mouse na área de arquivos do CS Code, nos é oferecido um menu denominado New C# com diversas opções.



Outra extensão interessante recomendada é a **vscode-icons**. Uma extensão mais visual, que muda os ícones dentro do VS Code.

Explorando o VS Code

Principal menu do VS Code:

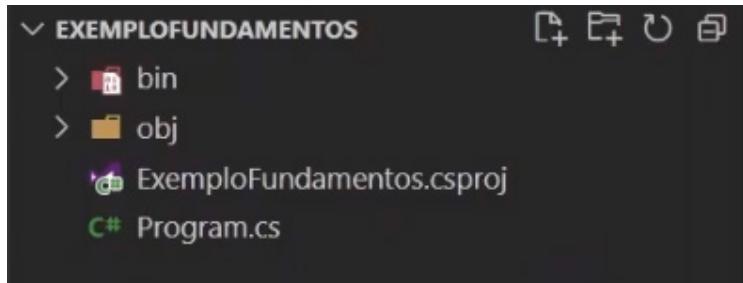


- Explorer
 - Representa a pasta do projeto
- Search
 - Pesquisar dentre os arquivos do projeto por nome, trechos de códigos, palavra-chave
 - Substituir palavras encontradas por outra da sua escolha, salvando a da sua escolha no lugar da que você pesquisou, onde quer que ela tenha sido encontrada.
- Source Control
 - Controle de versão do projeto com Git.
- Run and Debug
 - Depuração de código. Controlar o fluxo de execução do código.
- Extensions
- Account
 - Associar uma conta ao VS Code, para salvar configurações, extensões e etc.
- Manage
 - Configurações em geral do VS Code

III Sintaxe e seus tipos de dados

SINTAXE E INDENTAÇÃO

Entendendo a estrutura de um projeto



Podemos observar que ao criar um novo projeto, temos 2 pastas e 2 arquivos:

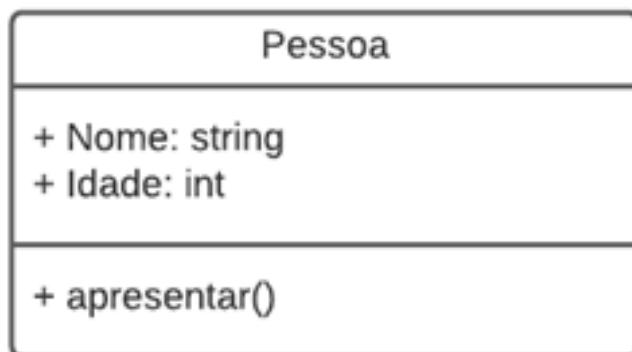
- bin
 - Pasta com os arquivos binários do projeto
 - Caso não exista na pasta do projeto, será gerada quando o projeto for compilado
- obj
 - Pasta com arquivos referentes à Debug
 - Caso não exista na pasta do projeto, será gerada quando o projeto for compilado
- ExemploFundamentos.csproj
 - Arquivo de metadados com configurações do projeto
 - Linguagem xml
- Program.cs
 - Arquivo de classe do C#

O conceito de classe

A classe está relacionada ao conceito de abstração da Orientação a Objetos. Mais a frente teremos mais sobre Orientação a Objetos. Mas qual o pilar da abstração?

Abstração nada mais é do que pegar um objeto do mundo real e transformá-lo em um objeto na programação.

Por exemplo: no mundo real numa determinada loja temos o comprador, o produto e o vendedor. Precisamos abstrair cada entidade dessa dinâmica do mundo real numa classe do nosso sistema. Então para o comprador e vendedor teremos a classe Pessoa.



Classe Pessoa

- Atributos
 - Nome
tipo : string
 - idade
tipo : inteiro
- Métodos
 - apresentar()

Dependendo do sistema, se não for uma loja de calçados, não faz sentido ter na Classe um atributo como Número do Calçado da pessoa.

Aqui temos um Objeto
instanciado da classe Pessoa.

Sendo o **Nome = Bob**,
Idade = 20
 e a representação
 do método **apresentar()**
 pelo balão de diálogo na imagem.

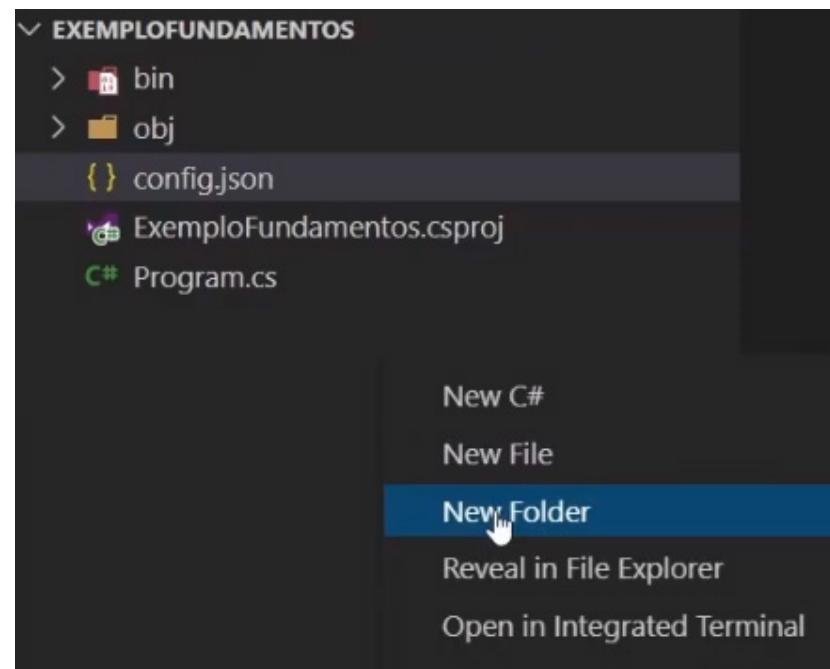


Objeto

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

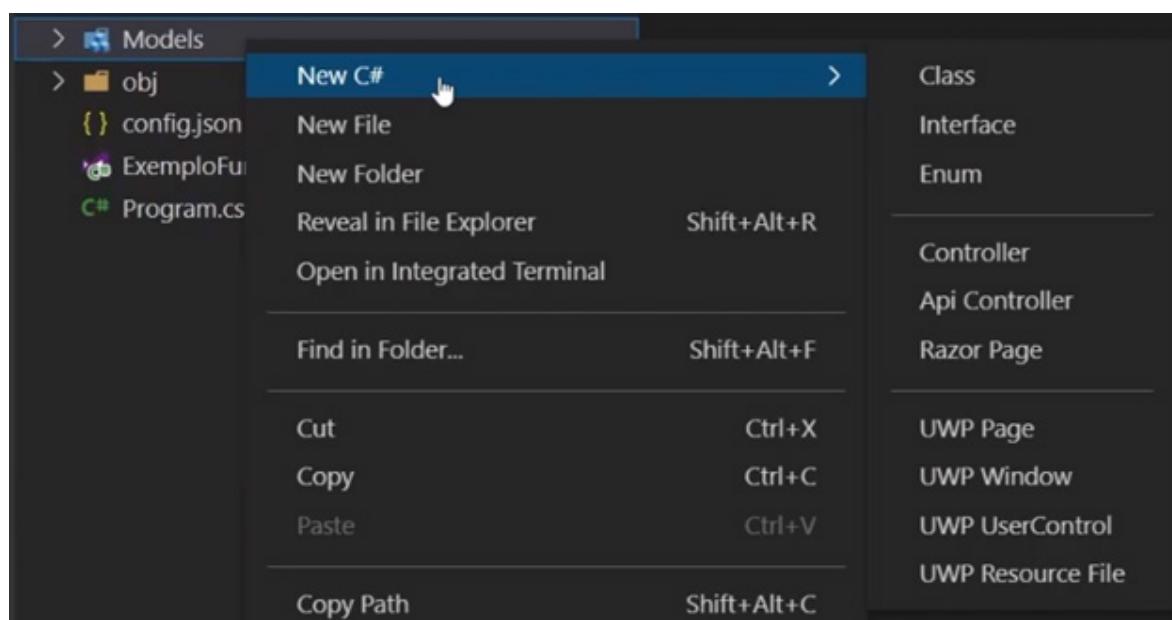
    public void Apresentar()
    {
        Console.WriteLine($"Olá! Meu nome é {Nome} e tenho {Idade} anos");
    }
}
```

O código da classe **Pessoa** escrita em C#.

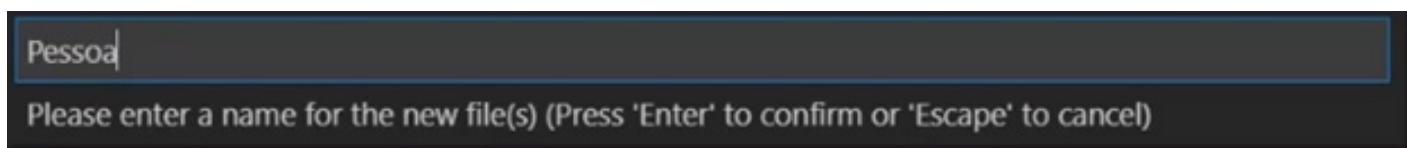


Clicando com o botão direito do mouse na área de arquivos do nosso projeto dentro do VS Code, selecione a opção **New Folder** e daremos à pasta o nome de Models:

Criada a pasta, clicando com o botão direito em cima dela e tendo a extensão C# extensions instalada e ativada, vamos à opção new C# > Class para criar um novo arquivo de classe no nosso projeto.



Por uma convenção, em várias linguagens de programação a primeira letra de cada palavra de um nome de classe sempre deve ser letra maiúscula, por exemplo: PessoaFisica. Dessa vez criaremos a classe **Pessoa**.



```

EXPLORER ... C# Pessoa.cs X
Models > C# Pessoa.cs
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5
6  namespace ExemploFundamentos.Models
7  {
8      public class Pessoa
9      {
10      }
11  }
12

```

Na classe **Pessoa**, dentro das chaves que vem depois de **public class Pessoa**, podemos utilizar o atalho **prop** para facilitar a escrita de uma nova propriedade:

```

C# Pessoa.cs 1 ●
Models > C# Pessoa.cs > {} ExemploFundamentos.Models > ExemploFundamentos.Models.Pessoa
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5
6  namespace ExemploFundamentos.Models
7  {
8      public class Pessoa
9      {
10          prop
11      }
12  }

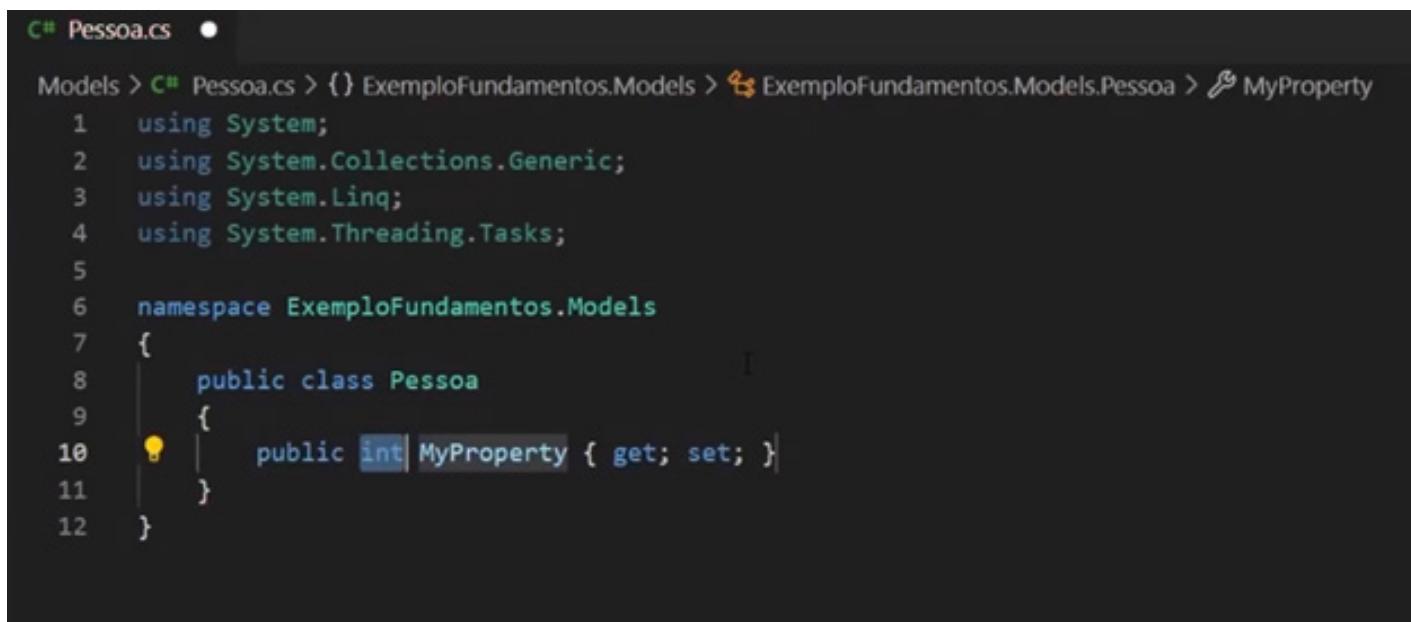
```

prop
propfull
propg
ParallelOptions
PreAllocatedOverlapped
ParallelMergeOptions
ParallelLoopState
ParallelLoopResult
GenericUriParserOptions
InvalidOperationException
PlatformNotSupportedException

An automatically implemented property. C# 3.0 or higher (C#)

public int MyProperty { get; set; }

Basta utilizar a tecla Tab do teclado para que o VS Code implemente o atalho.



A screenshot of the Visual Studio Code interface showing a C# file named Pessoa.cs. The code defines a class Pessoa with a public property MyProperty of type int. A yellow lightbulb icon is shown next to the property declaration, indicating a suggestion or code completion feature. The code is as follows:

```
C# Pessoa.cs •  
Models > C# Pessoa.cs > {} ExemploFundamentos.Models > ↗ ExemploFundamentos.Models.Pessoa > ↗ MyProperty  
1  using System;  
2  using System.Collections.Generic;  
3  using System.Linq;  
4  using System.Threading.Tasks;  
5  
6  namespace ExemploFundamentos.Models  
7  {  
8      public class Pessoa  
9      {  
10         public int MyProperty { get; set; }  
11     }  
12 }
```

Olhando mais a fundo para a propriedade criada, temos o tipo seguido do nome dessa propriedade:

```
public int MyProperty { get; set; }
```

De acordo com o modelo da classe Pessoa, precisamos alterar o tipo e o nome dessa propriedade:

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

Depois adicionamos mais uma prop para representar a idade:

```
public string Nome { get; set; }  
public int Idade { get; set; }
```

E por último o método:

```
public void Apresentar()  
{  
    Console.WriteLine($"Olá, meu nome é {Nome}, e tenho  
    {Idade} anos");  
}
```

Resultado final:

```
C# Pessoa.cs X

Models > C# Pessoa.cs > {} ExemploFundamentos.Models
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5
6  namespace ExemploFundamentos.Models
7  {
8      public class Pessoa
9      {
10          public string Nome { get; set; }
11          public int Idade { get; set; }
12
13          public void Apresentar()
14          {
15              Console.WriteLine($"Olá, meu nome é {Nome}, e tenho {Idade} anos");
16          }
17      }
18 }
```

Entendendo a estrutura do código

nameSpace ExemploFundamentos.Models

- Importante para organização
- Representa o caminho lógico para as classes que se encontram no mesmo domínio.
- A estrutura é com o nome do projeto, seguido de um ponto e a pasta onde a classe se encontra. Mas isso não significa o caminho real para a classe.

public class Pessoa

- Início da classe Pessoa, mostrando que o arquivo se trata efetivamente de uma classe

- Abertura e fechamento de chaves {} denota o início e fim do código

```
public string Nome { get; set; }  
public int Idade { get; set; }
```

- Propriedades da classe que serão úteis na implementação.

- get e set são métodos internos das propriedades.

- get é para pegar o valor da propriedade

- set para atribuir um novo valor para a propriedade

- Método ou função da classe

```
public void Apresentar()  
{  
    Console.WriteLine($"Olá, meu nome é {Nome}, e tenho {Idade} anos");  
}
```

- Nome e Idade têm coloração diferente pois estão relacionados às propriedades de mesmo nome dentro da classe.

- Os parênteses servem para indicar que se trata de uma função

- Um método/função pode receber parâmetros/argumentos na parte de dentro dos parênteses

- Um exemplo é o método **.WriteLine()** após a classe Console.

- A String que está escrita dentro dos parênteses é considerado um parâmetro/argumento.

Usando namespaces

Sabemos que a classe é uma abstração de um objeto da vida real. Porém a classe representa apenas o modelo, como por exemplo a planta-baixa de uma casa. É necessário instanciar a classe para que ela se torne um objeto.

Vamos abrir o arquivo principal **Program.cs**, apagar seu conteúdo prévio e escrever o seguinte:

The screenshot shows a dark-themed code editor with a sidebar containing project files: bin, Models (with Pessoa.cs), obj, config.json, ExemploFundamentos.csproj, and Program.cs. The main editor window displays the following C# code:

```
C# Program.cs
1 Pessoa p = new Pessoa();
```

The word "Pessoa" is underlined in red, indicating a syntax error. A yellow lightbulb icon is positioned next to the underlined word.

Podemos observar que um erro ocorre devido ao sublinhado em vermelho, porque o programa não sabe de onde vem essa classe Pessoa. Será necessário copiar o namespace da classe Pessoa e colar no arquivo principal:

The screenshot shows the same code editor and project structure as the previous screenshot. The main editor window now displays the following C# code:

```
C# Program.cs
1 using ExemploFundamentos.Models;
2
3 Pessoa p = new Pessoa();
```

The code is now valid, as the "using" directive specifies the namespace where the "Pessoa" class is located.

Usando a classe Pessoa

```
C# Program.cs
1  using ExemploFundamentos.Models;
2
3  Pessoa pessoa1 = new Pessoa();
4
5  pessoa1.Nome = "Buta";
6  pessoa1.Idade = 26;
7  pessoa1.Apresentar();|
```

- Instância da classe Pessoa na variável de nome pessoa1

`Pessoa pessoa1 = new Pessoa();`

- Atribuição para as propriedades Nome e Idade da variável pessoa1

`pessoa1.Nome = "Buta";`

`pessoa.Idade = 26;`

- Chamada do método

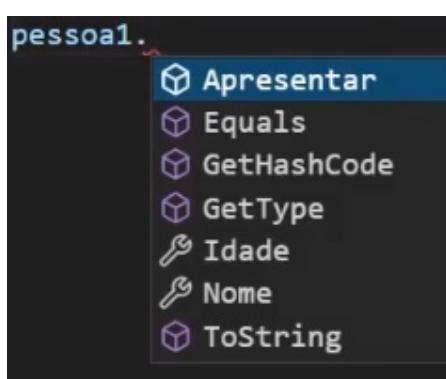
`pessoa1.Apresentar();`

Para executar, basta escrever o comando **dotnet run** no console:

The screenshot shows a terminal window with tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE. The TERMINAL tab is active, displaying the following text:

```
PS C:\Projetos\Videos\ExemploFundamentos> dotnet run
Olá, meu nome é Buta, e tenho 26 anos
PS C:\Projetos\Videos\ExemploFundamentos> []
```

Explorando a sintaxe



Ao escrever o nome da variável e colocar um ponto (.), teremos acesso às propriedades e métodos daquela variável.

Podemos quebrar uma String grande em partes menores adicionando o símbolo de **+** entre elas e pulando uma linha:

```
public void Apresentar()
{
    Console.WriteLine($"Olá, meu nome é " + $" {Nome}, e
tenho {Idade} anos");
}
```

Para quebrar uma linha no console, basta adicionar **\n** dentro da String:

```
Console.WriteLine($"Olá", meu nome é {Nome} \n e tenho
{Idade} anos");
```

A quebra de linha e os espaçamentos funcionam também nessa parte do código sem causar nenhum tipo de problema, desde que sejam apenas espaços em branco/vazios.

```
C# Program.cs
1  using ExemploFundamentos.Models;
2
3  Pessoa pessoa1 = new Pessoa();
4
5  pessoa1
6
7
8
9
10 .Nome = "Buta";
11
12 pessoa1
13 .Idade = 26;
14
15
16 pessoa1
17 .Apresentar();
18
```

Convenções case

Case são padrões de escrita de código. Tornam o código mais organizado e legível para humanos. Os mais usados no C# são o camelCase e o PascalCase.



camelCase

PascalCase

snake_case

spinal-case

Convenções cases no código

Nome da classe: PascalCase

```
public class Pessoa
```

Nome de propriedade: PascalCase

```
public int Idade { get; set; }
```

Nome de método: PascalCase

```
public void Apresentar();
```

Nome de variável : camelCase

```
Pessoa pessoa1 = new Pessoa();
```

Convenções de escrita de classe

Uma recomendação é nunca abreviar nome de classes:

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

Outra recomendação é que o nome do arquivo físico seja o mesmo nome da classe :

```
C# Pessoa.cs
> obj
< PessoaCopia
  < Pessoa.cs
  config.json
8     public class Pessoa
9     {
10         public string Nome { get; set; }
11         public int Idade { get; set; }
12         public string NomeRepresentanteLegalDaPessoaFisica { get; set; }
13     }
```

Convenção de nome e variável

Outra convenção é não utilizar caracteres especiais como pontuação, parênteses, chaves, colchetes, arroba, exclamação, etc no nome de classes, variáveis, propriedades e métodos/funções.

```
Pessoa pessoa:aFisicaRepresentacao = new Pessoa();
```

```
Pessoa pessoa{}aFisicaRepresentacao = new Pessoa();
```

O único caractere especial aceitável é o underline (_).

```
Pessoa pessoa_FisicaRepresentacao = new Pessoa();
```

TIPOS DE DADOS

Introdução e tipos inteiros

Type	Represents	Range	Default Value
string	A series of characters		
char	A Unicode character		
object	Object type		
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
decimal	decimal values with 28-29 significant digits	(+ or -)1.0 x 10e-28 to 7.9 x 10e28	0.0M
double	64 bit double-precision floating point type	(+/-)5.0 x 10 raise to -324 to (+/-)1.7 x 10 raise to 308	0.0D
int	32 bit signed integer type	-2,147,483,648 to 2,147,483,647	0
float	32 bit single-precision floating point type	-3.4 x 10 raise to 38 to + 3.4 x 10 raise to 38	0.0F
long	64 bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
uint	32 bit unsigned integer type	0 to 4,294,967,295	0
short	16 bit signed integer type	-32,768 to 32,767	0
ulong	64 bit unsigned integer type	0 to 18,446,744,073,709,551,615	0

O nome de uma pessoa é um dado do tipo texto (String) e a idade um dado do tipo inteiro (int ou integer).

- string
 - Conjunto de caracteres (char)
- char
 - Um caractere unicode
- object
- bool
 - Valores booleanos (boolean) que representam ou Verdadeiro ou Falso
- int
 - Representa os números inteiros, ou seja, sem casas decimais
- uint
 - Representa os números inteiros, ou seja, sem casas decimais
 - Representa apenas números positivos, ou seja, a partir do zero
 - Tem dobro do range de números positivos se comparado ao int
- long
 - Representa os números inteiros, ou seja, sem casas decimais
 - Tem o dobro de bits se comparado ao int

Números com casas decimais

- decimal
 - Mais preciso dos 3
 - Recomendado para representar valores monetários
- double
 - Mais preciso do que o float
- float

Declarando os tipos de variáveis

```
public string Nome {get; set}  
public int Idade {get; set;}  
public string NomeRepresentanteDaPessoaFisica {get; set;}
```

Uma variável do tipo String espera um valor entre aspas duplas (""):
`string apresentacao = "Olá, seja bem-vindo";`

Valores com casas decimais utilizam ponto(.) no lugar da vírgula (,):
`double altura = 1.80;`

Para que no console apareçam as 2 casas decimais, é necessário o método `.ToString("0.00")`:

```
Console.WriteLine("Valor da variável altura: " + altura.  
ToString("0.00"));
```

Valores do tipo decimal necessitam da letra “M” maiúscula ao final da variável apenas no caso de declaração direta:

```
decimal preco = 1.80M
```

Manipulando variáveis

Uma variável nada mais é do que um “pedaço” da memória reservado que armazena informação a ser trabalhada durante a execução do código.

Quando escrevemos o tipo antes do nome da variável, significa que estamos declarando-a. Quando é apenas o nome da variável, estamos alterando o valor de uma variável já existente. Por exemplo, a variável **quantidade** foi declarada com o valor 1 depois sobreescrita com o valor 10:

```
int quantidade = 1;  
Console.WriteLine("Valor da variável quantidade: " +  
quantidade);  
quantidade = 10;  
Console.WriteLine("Valor da variável quantidade: " +  
quantidade);
```

O tipo DateTime

É uma maneira de representar datas no C#. Com o método **.Now()** teremos a data atual do computador:

```
DateTime dataAtual = DateTime.Now  
Console.WriteLine(dataAtual);
```

Saída no console:

26/03/2022 10:31:13

Podemos também trabalhar de diversas formas, como adicionar horas, dias, meses, anos, etc:

```
DateTime dataAtual = DateTime.Now.AddDays(5);
```

Utilizando o método **toString()** pode-se formatar a saída do código para mostrar apenas dia/mes/ano:

```
Console.WriteLine(dataAtual.ToString("dd/MM/yyyy"));
```

Saída no console:

31/03/2022

III Tipos de operadores

OPERADORES DE ATRIBUIÇÃO

Introdução operador de atribuição

Serve para atribuir um valor (do lado direito do símbolo de =) para uma determinada variável (do lado esquerdo do símbolo de =):

```
int a = 10;  
int b = 20;  
int c = a + b;
```

Combinando operadores

Se quisermos, por exemplo, somar um valor a variável **c**, precisamos repeti-la após o sinal de = .

- Correta: **c = c + 5** ou **c += 5**
- Errada: **c = 5**

Ao executar **dotnet run**:

```
35
```

Assim como adição, podemos fazer as outras operações :

- Adição: **c += 5**
- Subtração: **c -= 5**
- Divisão: **c /= 5**
- Multiplicação: **c *= 5**

Convertendo tipos de variáveis

Supondo que temos uma variável do tipo inteiro:

```
int a = 5;
```

Se quisermos atribuir o mesmo valor, mas do tipo String,

```
int a = "5",
```

Isso porque já “avisamos” ao c# que essa variável será do tipo inteira. Para poder atribuir um valor a uma variável com tipo diferente, precisamos fazer uma conversão de tipo ou **casting**. Nesse caso, como sabemos que a variável é do tipo inteiro e o valor do tipo String, utilizaremos a classe **Convert** com o método **.ToInt32()**:

```
int a = Convert.ToInt("5");
```

Outra opção é utilizar tipo seguido do método **.Parse()**:

```
int a = int.Parse("5");
```

Tanto com o método **Parse** quanto com a classe **Convert** conseguimos converter os tipos de dados desde que os dados de entrada sejam compatíveis com o tipo da saída.

Diferença entre Convert e Parse

A diferença entre eles é o tratamento de valores do tipo **null**.

- Convert

```
int a = Convert.ToInt32(null);
```

Ao executar **dotnet run**:

```
0
```

O mais indicado pois é comum nos programas termos dados retornando **null**.

- Parse

```
int a = int.Parse(null);
```

Ao executar **dotnet run**:

```
>PS C:\Projetos\Videos\ExemploFundamentos> dotnet run
Unhandled exception. System.ArgumentNullException: Value cannot be null. (Parameter 's')
  at System.Int32.Parse(String s)
  at Program.<Main>$<String[] args> in C:\Projetos\Videos\ExemploFundamentos\Program.cs:line 6
PS C:\Projetos\Videos\ExemploFundamentos> []
```

Conversão para string

```
int inteiro = 5;
string a = inteiro;
Console.WriteLine(a);
```

Como o método **Parse** não existe para converter para String e utilizar a classe **Convert** seria redundante, basta utilizarmos o método **.ToString()**:

```
int inteiro = 5;
string a = inteiro.ToString();
```

Cast Implícito

Quando a conversão é automática. Por exemplo, ao atribuir uma variável do tipo **int** a uma outra do tipo **double**, temos o cast implícito:

```
int a = 5;  
double b = a;  
Console.WriteLine(b);
```

Ordem dos operadores

O C# respeita a ordem natural das operações matemáticas.

```
double a = 4 / 2 + 2;  
Console.WriteLine(b);
```

Quando queremos forçar uma ordem específica, utilizamos parênteses:

```
double a = 4 / (2 + 2);  
Console.WriteLine(a);
```

Convertendo de maneira segura

```
string a = "15-";  
int b = 0;  
int.TryParse(a, out b);  
Console.WriteLine(b);  
Console.WriteLine("Conversão realizada com  
sucesso!");
```

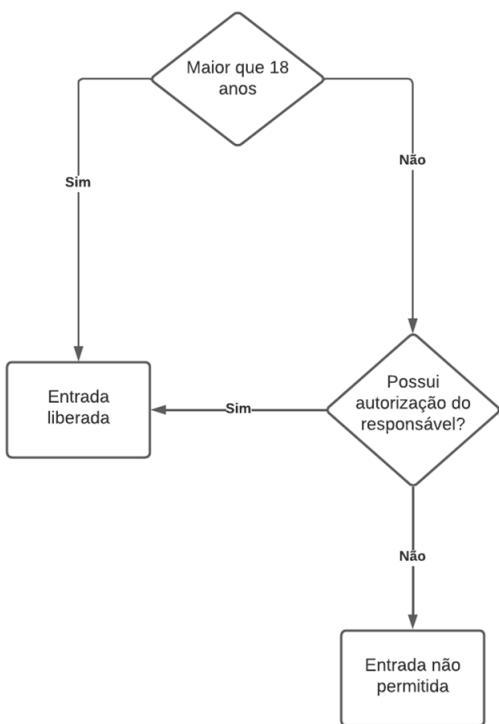
Com o **TryParse**, podemos converter um valor e, caso dê erro, o programa não será interrompido pois teremos sempre um resultado de saída. No nosso caso, como a variável **a** é um número inválido, o resultado de saída será 0 que é o valor de **b**.

Outra forma é declarar a variável já dentro do método:

```
int.TryParse(a, out int b);
```

OPERADORES LÓGICOS

Introdução operadores lógicos



Operadores lógicos servem para verificar uma determinada condição do código.

OR (Pipe ||)

No operador OU (OR), basta que uma das condições seja verdadeira para toda a sentença ser verdadeira.

Operador OR na prática

Exemplo 1:

```
bool ehMaiorDeIdade = false;
bool possuiAutorizacaoDoResponsavel = true;
```

```
if (ehMaiorDeIdade || possuiAutorizacaoDoResponsavel)
{
    Console.WriteLine("Entrada liberada!");
}
else
{
    Console.WriteLine("Entrada não liberada!");
}
```

Saída no console:

```
Entrada liberada!
```

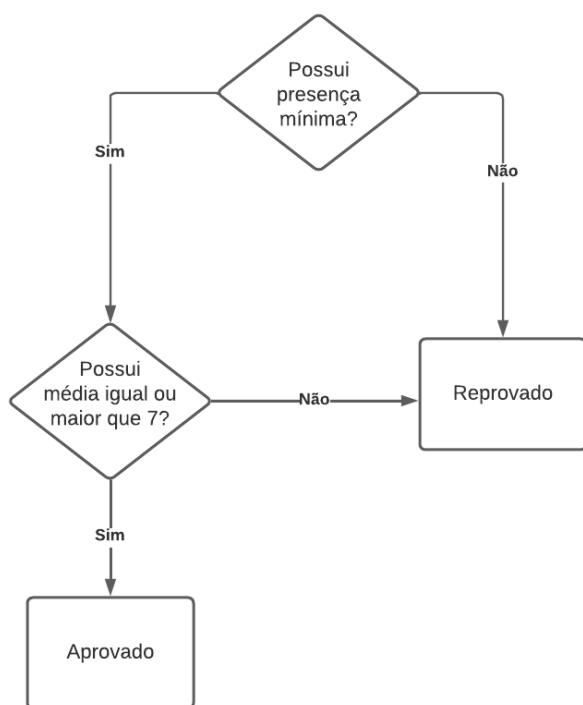
Exemplo 2:

```
bool ehMaiorDeIdade = false;  
bool possuiAutorizacaoDoResponsavel = false;  
  
if(ehMaiorDeIdade || possuiAutorizacaoDoResponsavel)  
{  
    Console.WriteLine("Entrada liberada!");  
}  
else  
{  
    Console.WriteLine("Entrada não liberada");  
}
```

Saída no console:

Entrada não liberada!

Introdução operador AND



No operador E (AND `&&`), para que a sentença seja verdadeira, todas as condições precisam ser verdadeiras.

Operador AND na prática

Exemplo 1: Todas as condições verdadeiras

```
bool possuiPresencaMinima = true;  
double media = 7.5;
```

```
if (possuiPresencaMinima && media >= 7)  
{  
    Console.WriteLine("Aprovado!");  
}  
else  
{  
    Console.WriteLine("Reprovado!");  
}
```

Saída no console:

```
Aprovado!
```

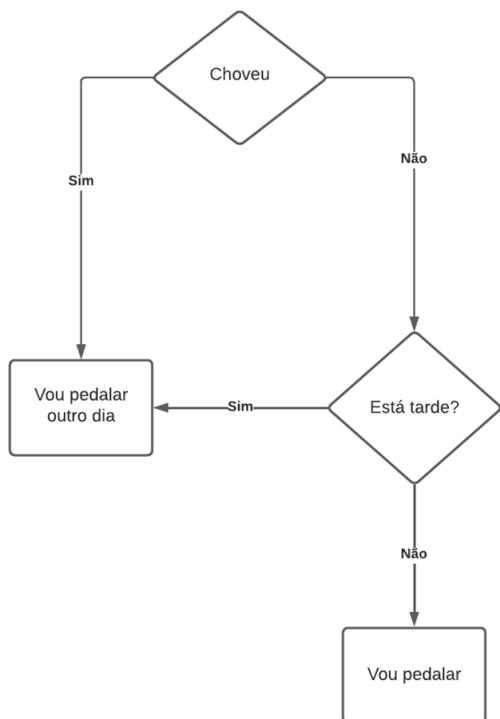
Exemplo 2: Apenas uma das condições é falsa

```
bool possuiPresencaMinima = false;  
double media = 7.5;  
  
if (possuiPresencaMinima && media >= 7)  
{  
    Console.WriteLine("Aprovado!");  
}  
else  
{  
    Console.WriteLine("Reprovado!");  
}
```

Saída no console:

Reprovado!

Introdução operador NOT



Operador de negação, cujo objetivo é inverter um valor booleano true ou false. No C# o operador NOT é representado por uma exclamação (!).

Operador NOT na prática

Exemplo 1:

```
bool choveu = false;
bool estaTarde = false;

if (!choveu && !estaTarde)
{
    Console.WriteLine("Vou pedalar");
}
else
{
    Console.WriteLine("Vou pedalar um outro dia");
}
```

Saída no console:

```
Vou pedalar
```

Exemplo 2:

```
bool choveu = true;
bool estaTarde = false;

if (!choveu && !estaTarde)
{
    Console.WriteLine("Vou pedalar");
}
else
{
    Console.WriteLine("Vou pedalar um outro dia");
}
```

Saída no console:

Vou pedalar um outro dia

III Operadores aritméticos e a classe Math

Introdução operadores aritméticos

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$x++$
--	Decrement	Decreases the value of a variable by 1	$x--$

- Adição +
- Subtração -
- Multiplicação *
- Divisão /
 - Retorna o resultado da divisão
- Módulo %
 - Retorna o resto da divisão
- Incremento ++
 - Incrementa +1 ao valor da variável
- Decremento –
 - Decrementa-1 ao valor da variável

Criando a nossa classe Calculadora

Criar uma classe **Calculadora.cs**:

```
namespace ExemploFundamentos.Models
{
    public class Calculadora
    {
        public void Somar(int x, int y)
        {
            Console.WriteLine($"{x} + {y} = {x + y}");
        }

        public void Subtrair(int x, int y)
        {
            Console.WriteLine($"{x} - {y} = {x - y}");
        }

        public void Multiplicar(int x, int y)
        {
            Console.WriteLine($"{x} X {y} = {x * y}");
        }

        public void Dividir(int x, int y)
        {
            Console.WriteLine($"{x} / {y} = {x / y}");
        }
    }
}
```

Já na classe **Program.cs**:

```
using ExemploFundamentos.Models;

Calculadora calc = New Calculadora();
calc.Somar(10, 30);
calc.Subtrair(10, 50);
calc.Multiplicar(15,45);
calc.Dividir(2,2);

using ExemploFundamentos.Models;

Calculadora calc = new Calculadora();

calc.Somar(10, 30);
calc.Subtrair(10, 50);
calc.Multiplicar(15, 45);
calc.Dividir(2, 2);
```

Saída no console:

```
10 + 30 = 40
10 - 50 = -40
15 x 45 = 675
2 / 2 = 1
```

Usando potência

De volta à classe **Calculadora.cs**, utilizaremos a classe **Math** junto do método **.Pow()** para representar a operação de potência:

```
public void Potencia(int x, int y)
{
    double pot = Math.Pow(x,y);
    Console.WriteLine($"'{x}^{y}' = {pot}");
}
```

Depois na classe **Program.cs**:

```
calc.Potencia(3,3);
```

Saída no console:

```
3^3 = 27
```

Funções Trigonométricas

Para representar seno, cosseno e tangente no C#, vamos utilizar novamente a classe **Math**. O valor do pi também pode ser representado pela mesma classe.

Na classe **Calculadora.cs**, faremos o código do Seno:

```
public void Seno(double angulo)
{
    double radiano = angulo * Math.PI / 180;
    double seno = Math.Sin(radiano);
    Console.WriteLine($"Seno de {angulo}º = {Math.
        Round(seno, 4)}");
}
```

Código do Cosseno:

```
public void Cosseno(double angulo)
{
    double radiano = angulo * Math.PI / 180;
    double cosseno = Math.Cos(radiano);
    Console.WriteLine($"Cosseno de {angulo}º = {Math.
        Round(cosseno, 4)}");
}
```

Código da Tangente :

```
public void Tangente(double angulo)
{
    double radiano = angulo * Math.PI / 180;
    double tangente = Math.Tan(radiano);
    Console.WriteLine($"Tangente de {angulo}º = {Math.
        Round(tangente, 4)}");
}
```

```
public void Tangente(double angulo)
{
    double radiano = angulo * Math.PI / 180;
    double tangente = Math.Tan(radiano);
    Console.WriteLine($"Tangente de {angulo}º = {Math.Round(tangente, 4)}");
```

E na classe **Program.cs**:

```
calc.Seno(30);  
calc.Cosseno(30);  
calc.Tangente(30);
```

Saída no console:

```
Seno de 30º = 0,5  
Cosseno de 30º = 0,866  
Tangente de 30º = 0,5774
```

Incremento e Decremento

```
int numeroIncremento = 10;  
  
Console.WriteLine(numeroIncremento);  
  
Console.WriteLine("Incrementando o 10");  
//numero = numero + 1;  
numeroIncremento++;  
  
💡 int numeroDecremento = 20;  
Console.WriteLine("Decrementando o 20");  
numeroDecremento--;  
  
Console.WriteLine(numeroDecremento);
```

Saída no console:

```
10  
Incrementando o 10  
Decrementando o 20  
19
```

Calculando Raiz Quadrada

Para calcular a raiz quadrada, novamente vamos utilizar a classe **Math** com o método **Sqrt()** que significa Square root ou raiz quadrada.

Classe **Calculadora.cs**:

```
public void RaizQuadrada(double x)
{
    double raiz = Math.Sqrt(x);
    Console.WriteLine($"Raiz quadrada de {x} = {raiz}");
}
```

E na classe **Program.cs**:

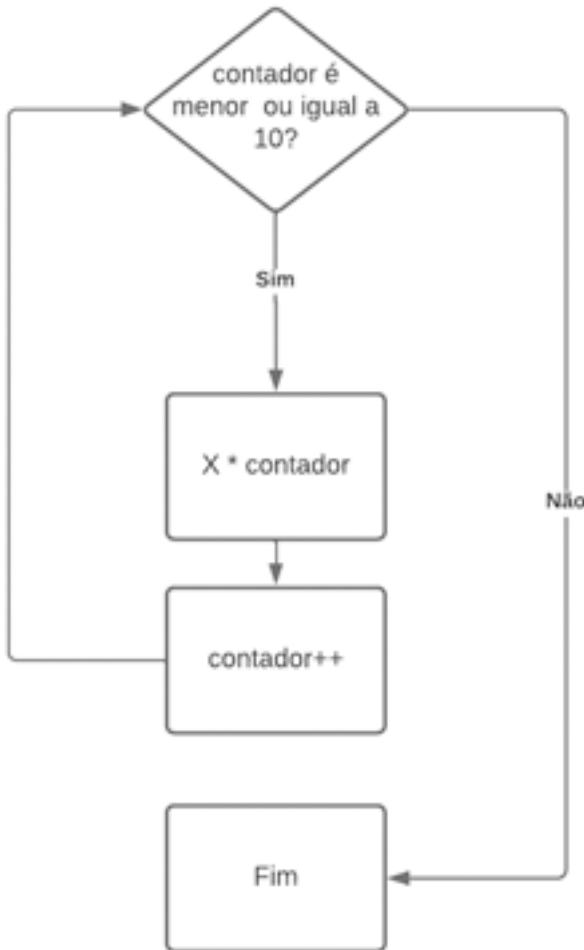
```
calc.RaizQuadrada(9);
```

Saída no console:

```
Raiz quadrada de 9 = 3
```

III Conhecendo as Estruturas de Repetição

Introdução estrutura de repetição



Laços ou estruturas de repetição são importantes em qualquer linguagem de programação. É importante entender essa estrutura para poder trabalhar de uma forma mais eficiente.

No exemplo da imagem, estamos utilizando uma variável de contador para que enquanto o valor deste contador seja menor ou igual a 10, o loop repita e esse contador tenha o incremento de +1.

Introdução ao FOR

O **FOR** é para quando precisamos repetir uma ação num determinado número de vezes. Por exemplo, no caso de uma tabuada podemos escrever diversas linhas de código como na imagem abaixo:

```
Console.WriteLine($"{numero} x 1 = {numero * 1}");
```

Porém, se utilizarmos o **FOR**, teremos o mesmo resultado mas de uma forma otimizada:

```
int numero = 5;

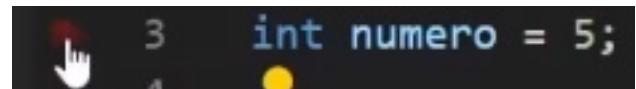
for(int contador = 0; contador <= 10; contador++)
{
    Console.WriteLine($"{numero} x {contador} = {numero * contador}");
}
```

Saída no console:

```
5 x 0 = 0
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Debugando o FOR

Para entender passo a passo como o loop **FOR** funciona, iremos utilizar a função de debugging de código. Para isso, vamos setar um breakpoint no nosso código, clicando na bolinha vermelha ao lado do número da linha:



Após setar o breakpoint, basta apertar a tecla F5 (Windows 10) para entrar no modo debugging:

```
C# Pessoa.cs    C# Program.cs X  C# Calculadora.cs
C# Program.cs
1  using ExemploFundamentos.Models;
2
3  int numero = 5;
4
5  for(D int contador = 0; contador <= 10; contador++)
6  {
7      Console.WriteLine($"{numero} x {contador} = {numero * contador}");
8  }
9
```

A seta e o fundo destacado em amarelo indicam onde está a execução do código em passo a passo. Para avançar com a execução, podemos pressionar a tecla F10 ou com o mouse clicar na opção do menu:

```
Program.cs
C# Program.cs X  C# Calculadora.cs
C# Program.cs
using ExemploFundamentos.Models;

int numero = 5;

for(D int contador = 0; contador <= 10; contador++)
{
```

▼ VARIABLES

▼ Locals

args [string[]]: {string[0]}
numero [int]: 5
contador [int]: 0

Ao avançarmos a execução, poderemos perceber na aba de debugging que a variável contador foi criada e que seu valor é igual a zero:

Após isso, ao avançar, a seta amarela vai para a parte da condição para entrar no loop **FOR** e, se for verdadeira, o código poderá prosseguir:

```
for(int contador = 0; D contador <= 10; contador++)
```

Avançando, será executada a linha com o Console e avançando novamente, a informação será impressa no console:

```
D Console.WriteLine($"{numero} x {contador} = {numero * contador}");
```

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
t supported by the current debugger code ty
Loaded 'C:\Projetos\Videos\ExemploFundament
Loaded 'C:\Program Files\dotnet\shared\Micr
is optimized and the debugger option 'Just
Loaded 'C:\Program Files\dotnet\shared\Micr
is optimized and the debugger option 'Just
Loaded 'C:\Program Files\dotnet\shared\Micr
le is optimized and the debugger option 'Ju
Loaded 'C:\Program Files\dotnet\shared\Micr
g symbols. Module is optimized and the debu
5 x 0 = 0
>
```

Depois disso, a execução retorna para a linha do laço **FOR** para executar a última instrução que é o incremento da variável contador:

```
for(int contador = 0; contador <= 10; D contador++)
{
    Console.WriteLine($"{numero} x {contador} = {nu
    ^
```

Clicando em avançar, podemos perceber que a variável foi incrementada e agora a seta amarela retorna para a condição do loop:

```
5  for(int contador = 0; D contador <= 10; contador++)
```

VARIABLES
Locals
args [string[]]: {string[0]}
numero [int]: 5
contador [int]: 11

Os passos de verificar a condição do loop, executar e incrementar a variável contador se repetirão até que a condição do loop não seja atendida, em outras palavras, quando a condição retornar FALSE.

Quando isso acontecer, a execução do programa será encerrada.

```
5  for(int contador = 0; D contador <= 10; contador++)
```

```
The program '[10632] ExemploFundamentos.dll' has exited with code 0 (0x0).
```

Introdução ao WHILE

While significa enquanto. Sua utilização deve ser gerenciada para que não ocorra um loop infinito.

Exemplo de um código que causaria um loop infinito:

```
int numero = 5;
int contador = 0;

while (contador <= 10)
{
    Console.WriteLine($"{numero} x {contador} = {numero * contador}");
}
```

PROBLEMS

OUTPUT

TERMINAL

```
5 x 0 = 0
5 x 0 = 0
5 x 0 = 0
5 x 0 = 0
5 x 0 = 0
5 x 0 = 0
5 x 0 = 0
5 x 0 = 0
```

Como a condição para o loop é sempre verdadeira, o código será executado de forma infinita até que seja forçado a parar.

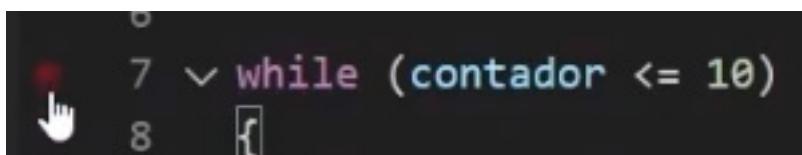
Para interromper a execução de um loop infinito no VS Code, basta clicar com o mouse na área do terminal, segurar a tecla Ctrl e apertar a tecla C (Windows 10).

Agora, para executar de forma correta o nosso código, precisamos adicionar um incremento à variável contador:

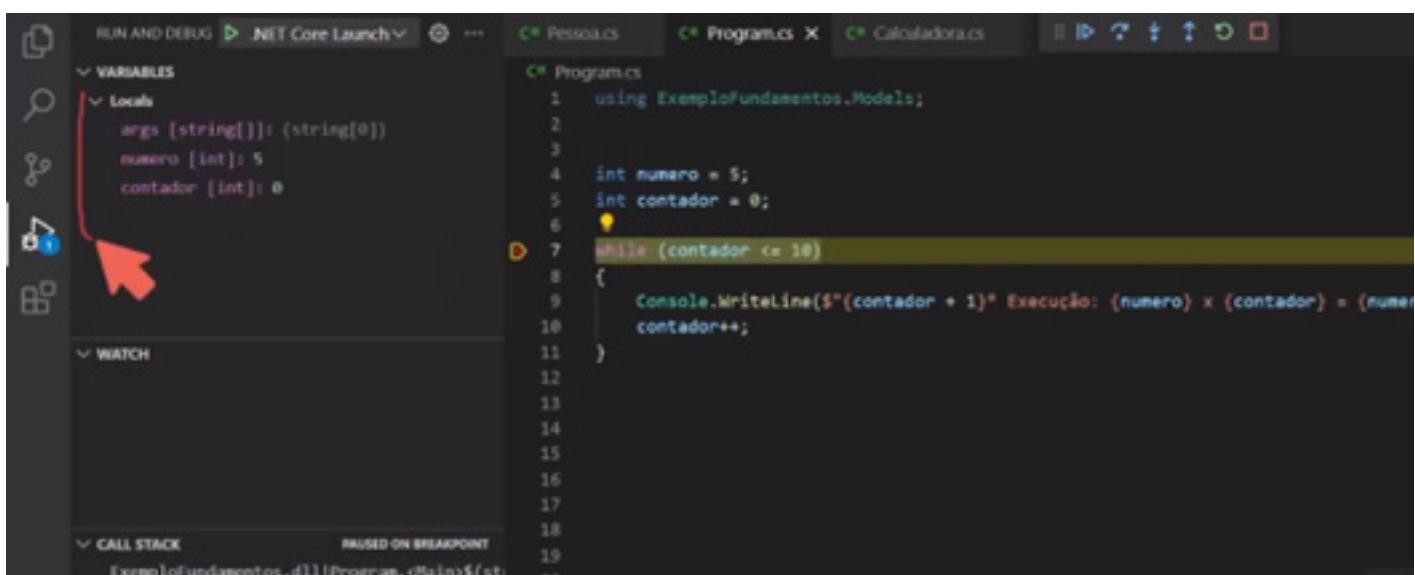
```
int numero = 5;
int contador = 0;

while (contador <= 10)
{
    Console.WriteLine($"{contador}° Execução: {numero} x {contador} = {numero * contador}");
    contador++;
}
```

Debugando o WHILE



Para debugar, adicionamos um breakpoint no início do **while** e, sem seguida, apontamos F5 no teclado (Windows 10):



Assim como fizemos no caso do **FOR**, vamos avançamos a execução do código até que a condição seja falsa:

```
while (contador <= 10)
{
    Console.WriteLine($"{contador + 1}º Execução: {numero} x {contador} = {numero * contador}");
    contador++;
}
```

```
while (contador <= 10)
{
    DConsole.WriteLine($"{contador + 1}º Execução: {numero} x {contador} = {numero * contador}");
    contador++;
}
```

```
while (contador <= 10)
{
    D
    Console.WriteLine($"{contador + 1}º Execução: {numero} x {contador} = {numero * contador}");
    Dcontador++;
}
```

VARIABLES

Locals

```
args [string[]]: {string[0]}\n  numero [int]: 5\n  contador [int]: 1
```

E podemos observar que assim que passamos pela execução do incremento, o valor da variável contador muda.

Interrompendo o fluxo de execução

Podemos interromper o fluxo do loop **while** através da palavra reservada **break**.

```
if(contador == 5)\n{\n    break;\n}
```

```
int numero = 5;\nint contador = 1;\n\nwhile (contador <= 10)\n{\n    Console.WriteLine($"{contador}º Execução: {numero} x {contador} = {numero * contador}");\n    contador++;\n\n    if (contador == 5)\n    {\n        break;\n    }\n}
```

Saída no console:

```
1º Execução: 5 x 1 = 5\n2º Execução: 5 x 2 = 10\n3º Execução: 5 x 3 = 15\n4º Execução: 5 x 4 = 20
```

Introdução ao DO WHILE

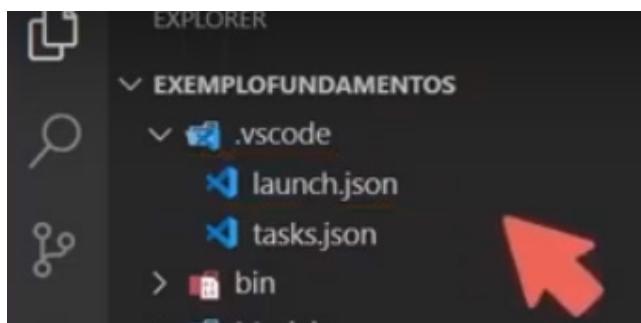
O DO WHILE é uma variação do WHILE, porém a verificação no caso do DO WHILE é feita após a execução do código.

```

4 int soma = 0, numero = 0;
5
6
7 do
8 {
9     Console.WriteLine("Digite um número (0 para parar)");
10    numero = Convert.ToInt32(Console.ReadLine());
11
12    soma += numero;
13
14 } while(numero != 0);
15
16 Console.WriteLine($"Total da soma dos números digitados é: {soma}");

```

Debugando o DO WHILE



O primeiro passo é alterar o tipo de saída do nosso terminal, porque o **Console.ReadLine()** não funciona no terminal de debug do VS Code. Para isso vamos abrir o arquivo **launch.json** dentro da pasta **.vscode**.

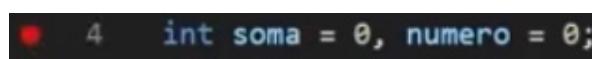
Dentro do arquivo, na linha 15 precisaremos alterar o valor do console de **internalConsole** para **integratedTerminal** e salvar o arquivo:

```

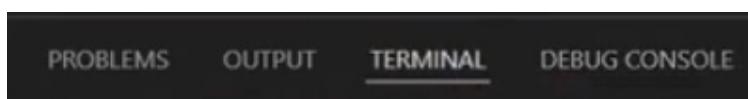
15 | | | | "console": "internalConsole",
15 | | | | "console": "integratedTerminal",

```

Feito isso, voltamos para nosso código e iniciamos o debug setando um breakpoint no início do nosso código e iniciar o debug:



```
C# Program.cs
1 using ExemploFundamentos.Models;
2
3
4 int soma = 0, numero = 0;
5
6
7 do
8 {
9     Console.WriteLine("Digite um número (0 para parar)");
10    numero = Convert.ToInt32(Console.ReadLine());
11
12    soma += numero;
13
14 } while(numero != 0);
15
16 Console.WriteLine($"Total da soma dos números digitados é: {soma}");
17
18
19
20
```



Por padrão, o console será aberto como DEBUG CONSOLE. Troque para TERMINAL.

```
4 int soma = 0, numero = 0;
5
6
7
8 {
9     Console.WriteLine("Digite um número (0 para parar)");
10    numero = Convert.ToInt32(Console.ReadLine());
11
12    soma += numero;
13
14 } while(numero != 0);
15
16 Console.WriteLine($"Total da soma dos números digitados é: {soma}");
17
```

Após avançar com a execução, podemos comprovar que o código avançou sem passar pelo while.

```
Digite um número (0 para parar)
5
```

Avançando mais alguma vezes, vamos notar que aparentemente o debugging parou. Isso vai ocorrer porque o programa estará esperando que entremos com um valor no console para continuar com a execução.

```
4 int soma = 0, numero = 0;
5
6
7 do
8 {
9     Console.WriteLine("Digite um número (0 para parar)");
10    numero = Convert.ToInt32(Console.ReadLine());
11
12    soma += numero;
13
14 } while(numero != 0);
15
16 Console.WriteLine($"Total da soma dos números digitados é: {soma}");
```

E após entrarmos com um valor para a variável **numero** o programa retoma a execução e podemos conferir a atribuição na aba lateral.

Continuando, o programa irá entrar na verificação do while e, como nesse caso a condição retornará TRUE, a execução voltará para o início do do e o programa continuará até a condição ser falsa. Quando isso acontecer, será escrita no console o valor da variável soma:

```
● 4 int soma = 0, numero = 0;
5
6
7 do
8 {
9     Console.WriteLine("Digite um número (0 para parar)");
10    numero = Convert.ToInt32(Console.ReadLine());
11
12    soma += numero;
13
D 14 } while(numero != 0);
15
16 Console.WriteLine($"Total da soma dos números digitados é: {soma}");
```

D 16 Console.WriteLine(\$"Total da soma dos números digitados é: {soma}");

Construindo um menu interativo

Vamos construir um menu interativo bem simples, que poderá ser aproveitado para exemplos futuros.

```
4 string opcao;
5
6
7 while(true)
8 {
9     Console.WriteLine("Digite a sua opção:");
10    Console.WriteLine("1 - Cadastrar cliente");
11    Console.WriteLine("2 - Buscar cliente");
12    Console.WriteLine("3 - Apagar cliente");
13    Console.WriteLine("4 - Encerrar");
14
15    opcao = Console.ReadLine();
```

```
16     switch(opcao)
17 {
18     case "1":
19         Console.WriteLine("Cadastro de cliente");
20         break;
21
22     case "2":
23         Console.WriteLine("Busca de cliente");
24         break;
25
26     case "3":
27         Console.WriteLine("Apagar cliente");
28         break;
29
30     case "4":
31         Console.WriteLine("Encerrar");
32         Environment.Exit(0);
33         break;
34
35     default:
36         Console.WriteLine("Opção inválida");
37         break;
38 }
39 }
```

Para podermos forçar a interrupção do programa quando a opção 4 for digitada, utilizamos o seguinte:

`Environment.Exit(0);`

Refatorando o menu

```
5     bool exibirMenu = true;
6
7     while(exibirMenu)
```

Para refatorar o nosso código, podemos criar uma variável do tipo **booleana** e iniciar seu valor como **true**. Depois, adicionar dentro do **case “4”** do switch uma nova atribuição a essa variável como **false**:

```
32     case "4":
33         Console.WriteLine("Encerrar");
34         exibirMenu = false;
35         break;
```

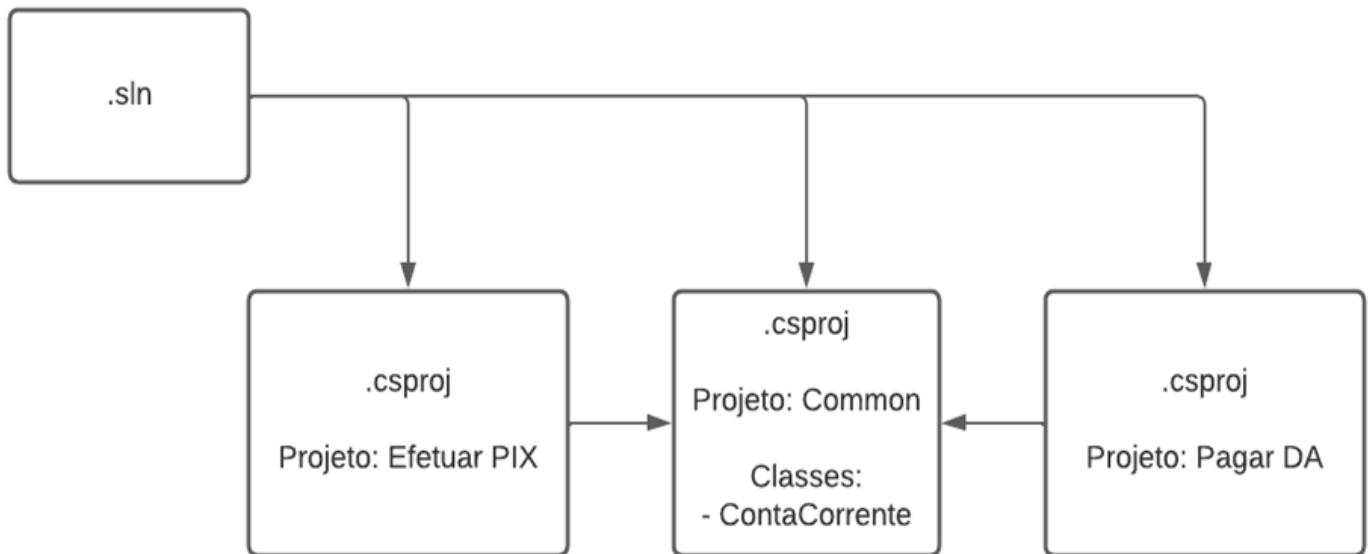
III Conhecendo a organização de um programa C#

Estrutura de um programa C#

Arquivos de projeto

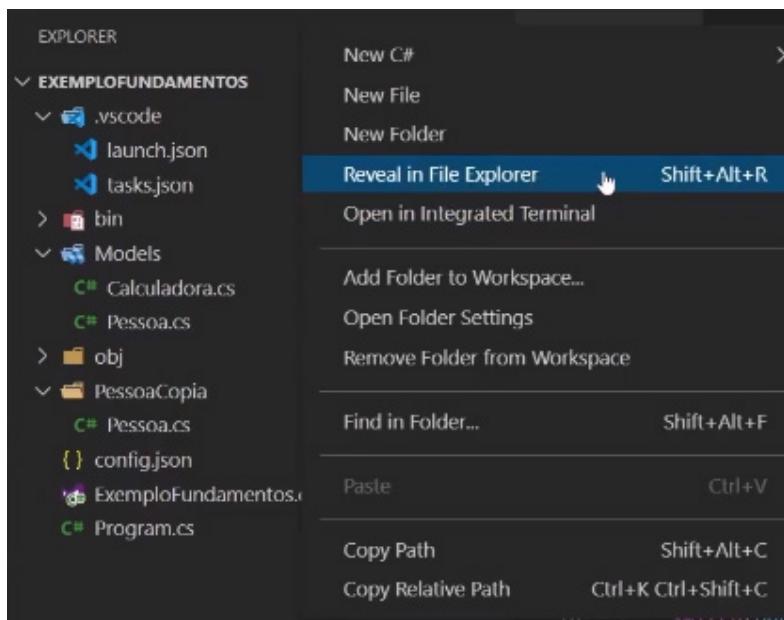
- .csproj
 - Contém informações referentes a um projeto (build, debug, versão)
- .sln
 - Contém informações que carregam um agrupamento de projetos.

No projeto de um banco, por exemplo, tanto o projeto Efetuar PIX quanto o Pagar DA dependeriam de uma conta-corrente para funcionarem. Para evitar que o código da conta corrente fique duplicado, o ideal é criar um terceiro projeto, como podemos ver na imagem a seguir:



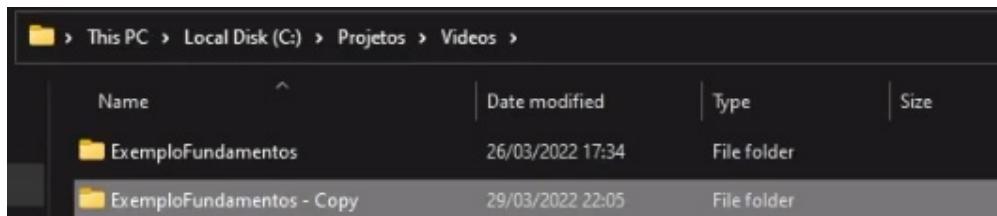
Nesse projeto denominado **Common** serão colocadas todas as classes de uso comum a outros projetos, como no nosso exemplo a ContaCorrente é de uso comum aos outros projetos.

Criando nosso novo projeto



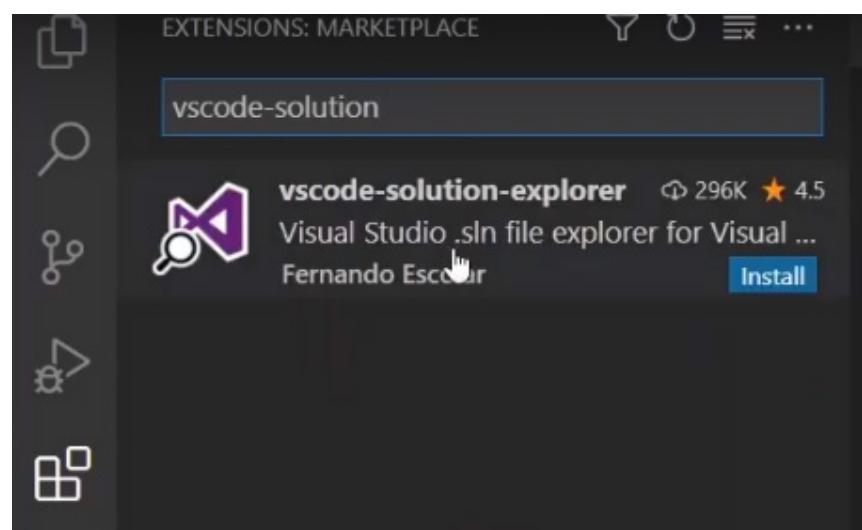
Primeiramente precisamos fazer um backup do nosso projeto atual. Clicamos numa área vazia da aba lateral de arquivos do VS Code, em seguida selecionamos **Reveal in File Explorer**.

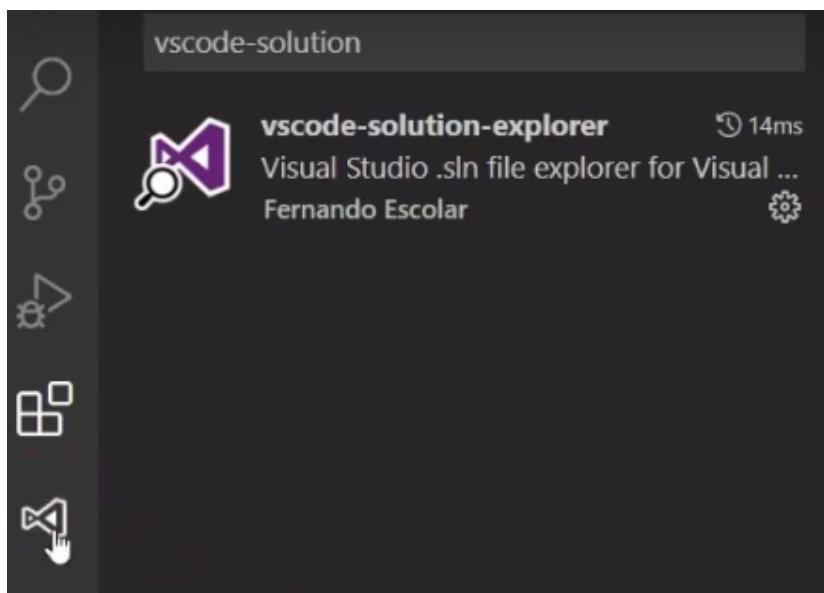
Depois basta criar uma cópia da pasta do nosso projeto:



Após isso, vamos instalar uma extensão que ajuda na criação das solutions. Na aba de extensões procure **vscode-solution**.

Depois de instalar, ative a extensão

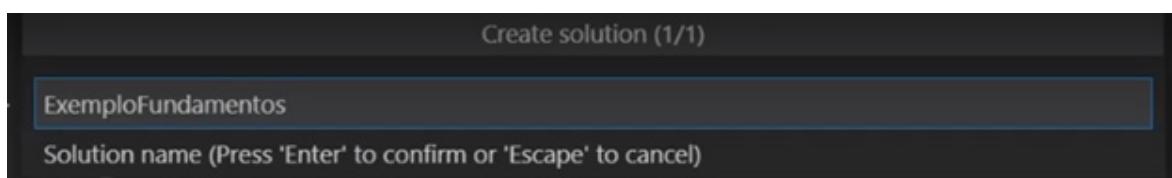




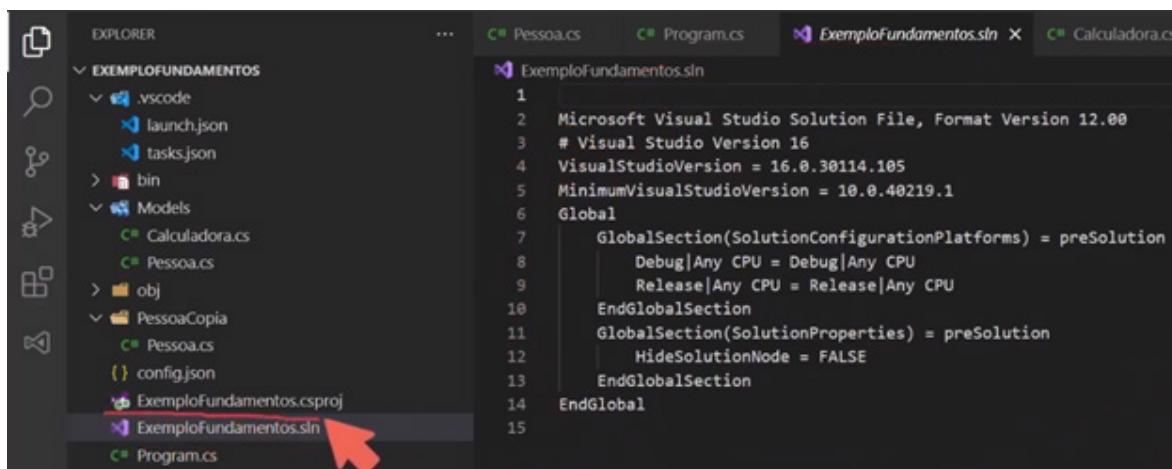
Ao clicar sobre esse ícone, veremos que ainda não temos uma solution criada.

Para criar uma nova, basta clicar com o botão direito e selecionar a opção **Create new empty solution** para iniciar a criação.

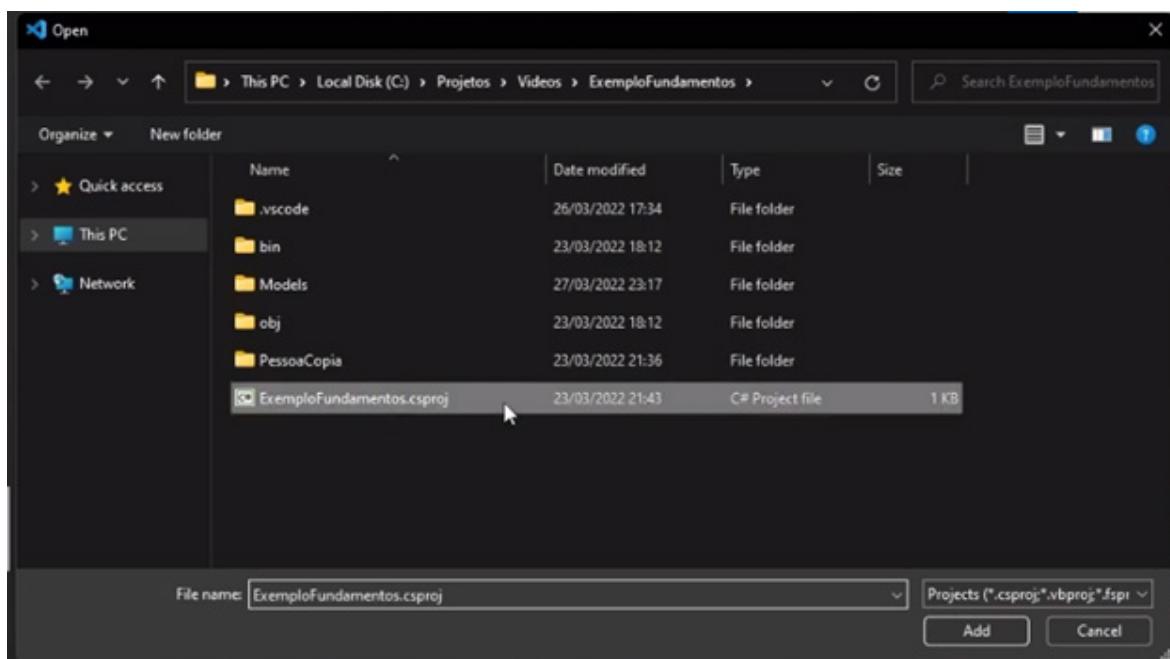
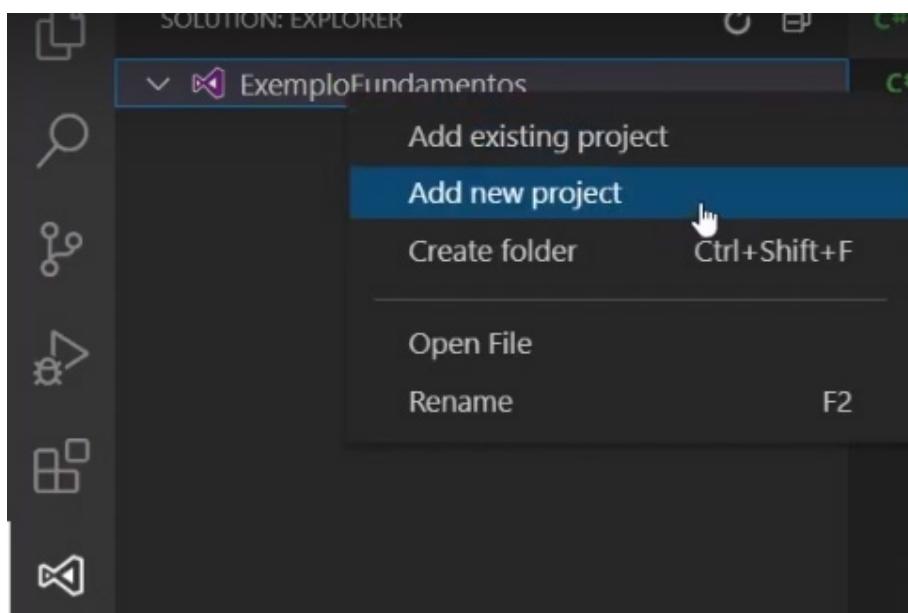
Surgirá uma caixa para darmos um nome à nossa solution:



Após isso, ao retornarmos à aba Explorer, podemos notar que apareceu um novo arquivo **.sln**:



Precisaremos adicionar uma referência do nosso projeto atual dentro desse arquivo **.sln**. Para isso, vamos voltar a aba da extensão, clicar com o botão direito do mouse sobre nosso arquivo e adicionar nosso projeto existente:



E ao retornarmos ao arquivo **.sln** da aba Explorer do nosso projeto, vamos notar algumas alterações incluindo o nome do nosso projeto:

```

EXPLORER
EXEMPLOFUNDAMENTOS
  .vscode
  bin
  Models
  obj
  PessoaCopia
  config.json
  ExemploFundamentos.csproj
  ExemploFundamentos.sln
  Program.cs

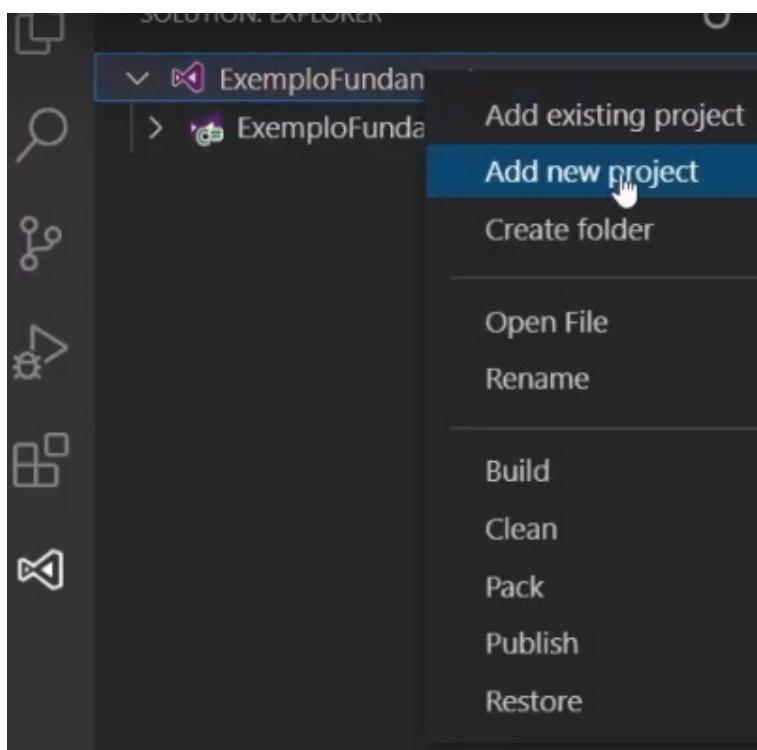
SOLUTION EXPLORER
ExemploFundamentos
  Add existing project
  Add new project
  Create folder
  Open File
  Rename
  Ctrl+Shift+F
  F2

```

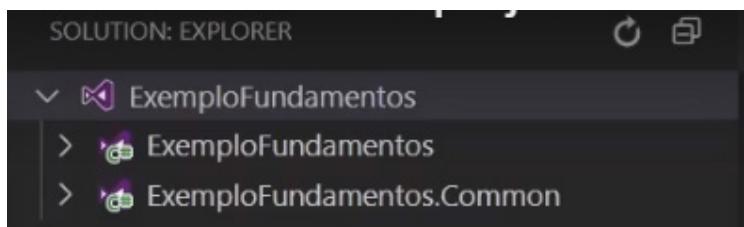
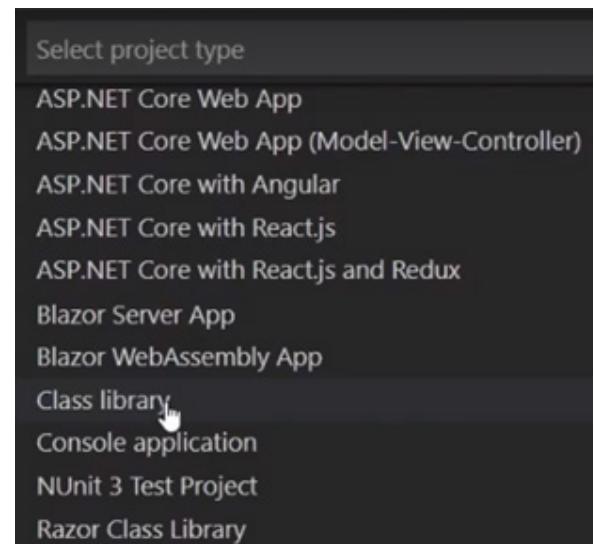
```

1
2  sat Version 12.00
3
4
5
6  !FBC") = "ExemploFundamentos", "ExemploFundamentos.csproj", "{218FC993-2
7
8  :forms) = preSolution
9
10
11
12  !$olution
13
14
15
16  Forms) = postSolution
17  !ca1_DebugAnyCPU_ActiveCfg = DebugAnyCPU

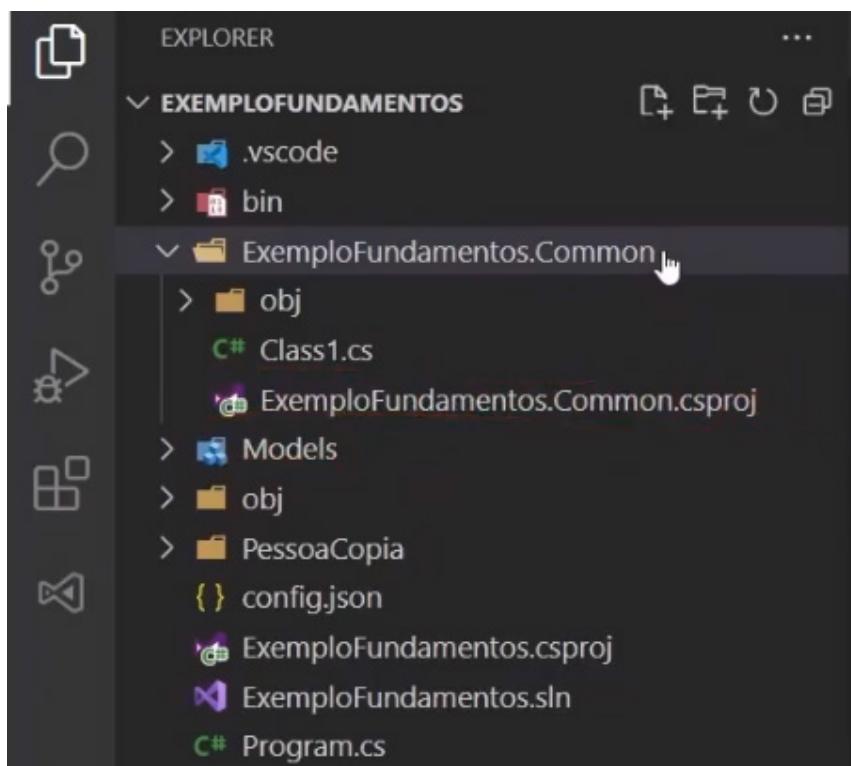
```



Agora, vamos criar um novo projeto para as nossas classes de modelo, chamado **Class Library**.

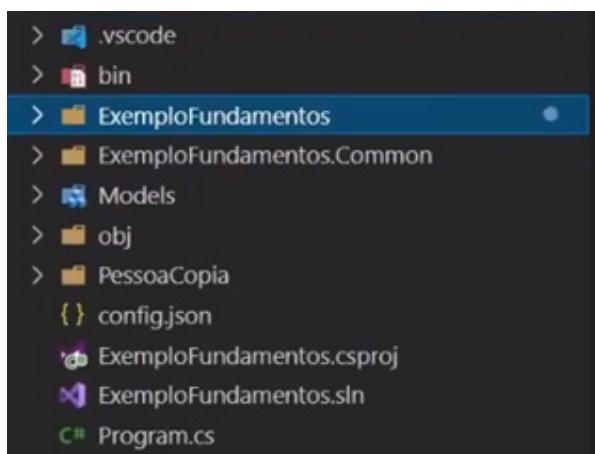


Ao criarmos esse projeto através da nova extensão, ele automaticamente já adiciona-o à nossa solution.



E ao voltarmos ao Explorer, podemos perceber que temos uma pasta de projetos dentro de outra pasta de projeto.

Organizando e referenciando projetos



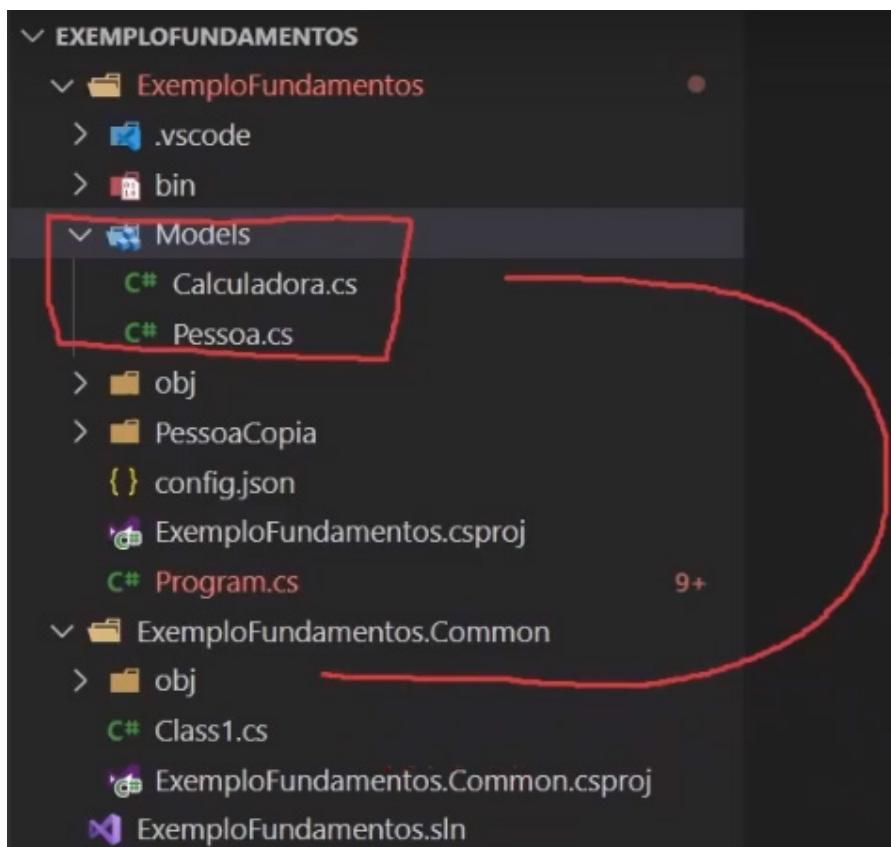
Vamos refatorar essa estrutura de pastas para que cada projeto fique numa pasta separada, de modo mais organizado.

Primeiro vamos criar uma nova pasta no diretório do nosso projeto.

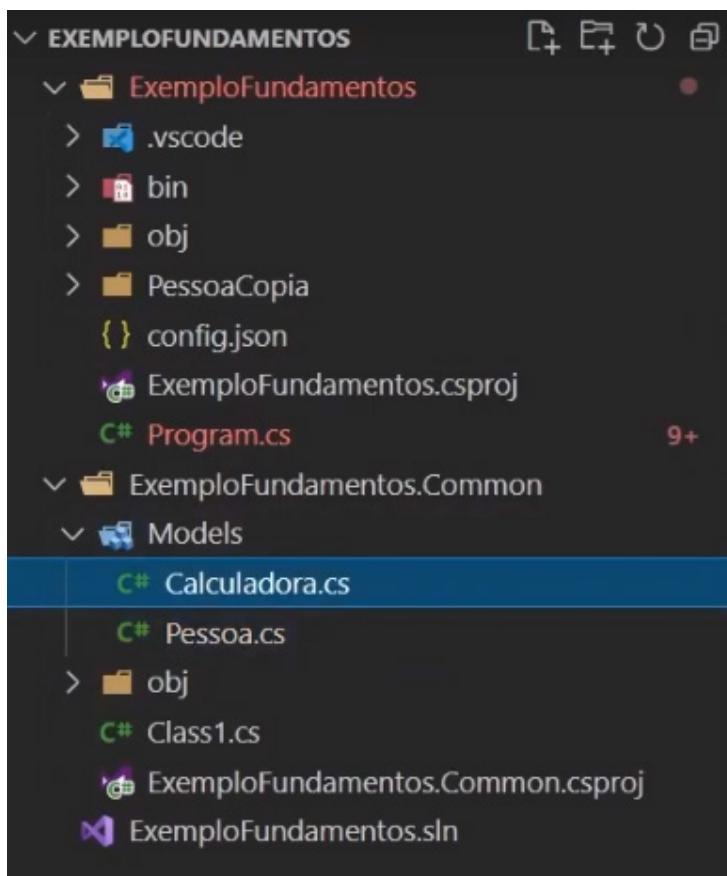


Com exceção dos arquivos **ExemploFundamentos.sln** e da pasta **ExemploFundamentos.Common**, vamos mover o restante para dentro da nova pasta.

Agora, precisaremos mover a pasta **Models** para dentro do projeto **Common**:



Resultado final após a movimentação:



Agora, por uma questão de organização, vamos renomear os namespace das nossas classes:

```
C# Calculadora.cs X

ExemploFundamentos.Common > Models > C# Calculadora.cs

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5
6 namespace ExemploFundamentos.Common.Models
```

```
C# Calculadora.cs C# Pessoa.cs X

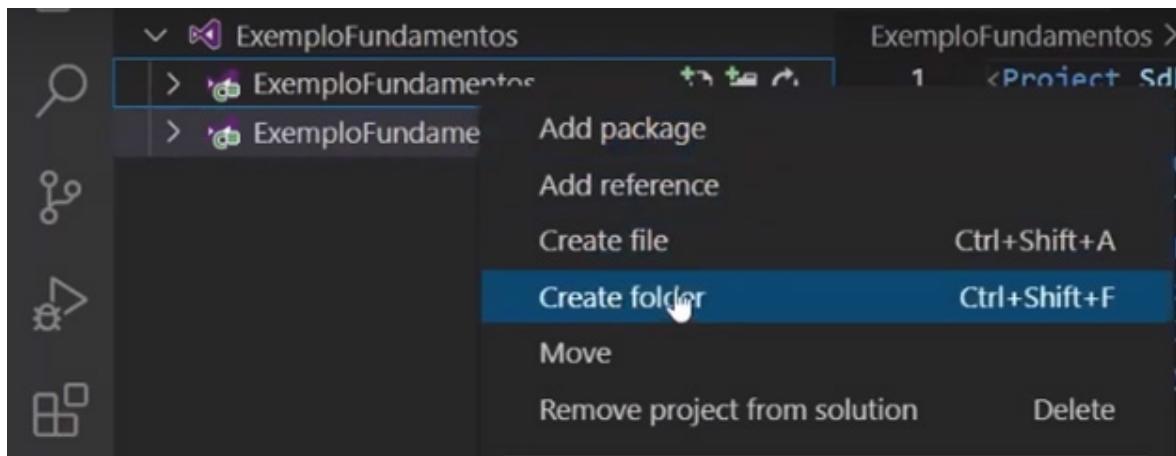
ExemploFundamentos.Common > Models > C# Pessoa.cs > ...

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5
6 namespace ExemploFundamentos.Common.Models
```

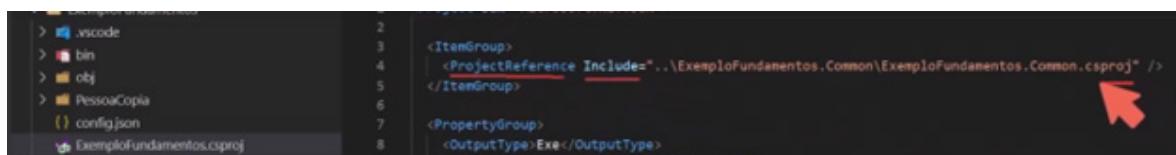
Depois, vamos corrigir a referência às classes do nosso projeto **ExemploFundamentos**. Para isso, vamos abrir nosso arquivo **.sln** e alterar a referência do nosso projeto que agora está dentro de uma pasta. Para isso, basta adicionar o nome da pasta seguido de uma barra (\):

```
1  mat Version 12.00
2
3
4
5
6  EFBC}") = "ExemploFundamentos", "ExemploFundamentos\ExemploFundamentos.csproj", "
```

E agora, vamos adicionar a referência do projeto **Common** ao nosso projeto principal:



E então, podemos perceber que a referência foi inserida ao arquivo **.csproj** do projeto principal:



```
C# Program.cs X
ExemploFundamentos > C# Program.cs
1  using ExemploFundamentos.Common.Models;
2
3  Pessoa pessoa1 = new Pessoa();
4  pessoa1.Nome = "Buta";
5  pessoa1.Idade = 20;
6  pessoa1.Apresentar();
7
```

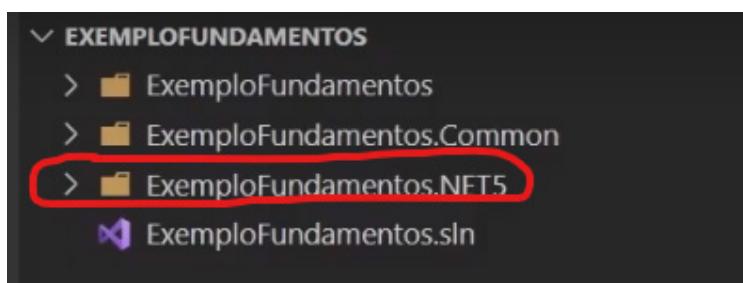
Agora, voltando ao arquivo **Program.cs** do nosso projeto principal, podemos ter acesso à classe Pessoa que está em outro projeto.

Ao executar **dotnet run**:

Olá, meu nome é Buta, e tenho 20 anos

Criando um projeto em NET5 e NET6

A partir da versão 6 do .NET, o método principal (**main**) se tornou implícito. Nas versões antigas esse método sempre está explícito.



Vamos criar um projeto com .NET 5 para exemplificar o uso do método principal.

Primeiro criamos uma nova pasta para esse projeto.

Feito isso, precisamos navegar pelo terminal para dentro da pasta criada. Atualmente estamos na pasta **ExemploFundamentos** dentro de outra pasta **ExemploFundamentos**. Então primeiro precisamos voltar um nível com o comando **cd ..** :

```
PS C:\Projetos\Videos\ExemploFundamentos\ExemploFundamentos> cd ..
PS C:\Projetos\Videos\ExemploFundamentos>
```

Agora, para entrarmos na nova pasta, basta utilizarmos o mesmo comando com o nome da pasta para onde queremos ir:

```
PS C:\Projetos\Videos\ExemploFundamentos> cd ExemploFundamentos.NET5
PS C:\Projetos\Videos\ExemploFundamentos\ExemploFundamentos.NET5>
```

Para podermos criar um novo projeto na versão 5 do .NET, precisamos ter instalado essa versão na nossa máquina. Caso já tenha instalado, então escrevemos o seguinte comando:

```
dotnet new console --framework net5.0
```

Para fins de comparação, vamos criar um outro projeto com o .NET na versão 6. Primeiro criamos a pasta e navegamos até ela pelo terminal:

```
PS C:\Projetos\Videos\ExemploFundamentos> cd ..  
PS C:\Projetos\Videos> cd ExemploFundamentos.NET6  
PS C:\Projetos\Videos\ExemploFundamentos.NET6> [red arrow pointing right]
```

E depois criamos um novo projeto do .NET. Repare que se não explicitarmos

a versão, então a que será usada será a mais recente que a máquina possuir instalada:

```
dotnet new console
```

Comparando NET5 e NET6

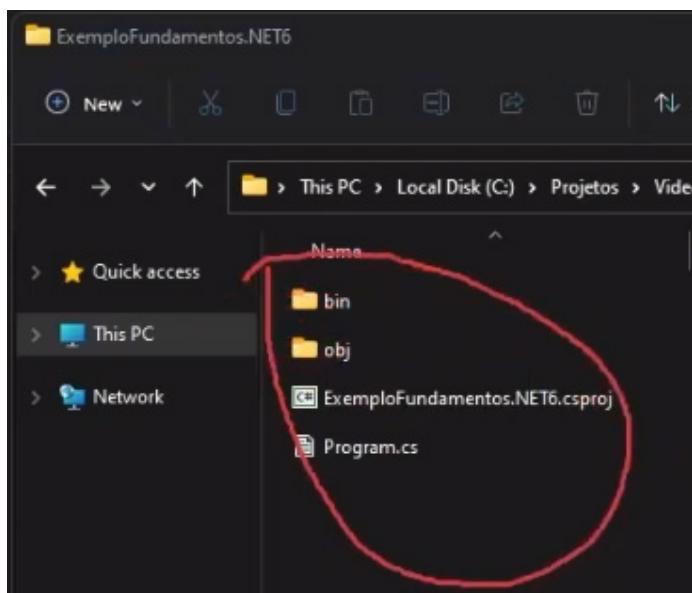
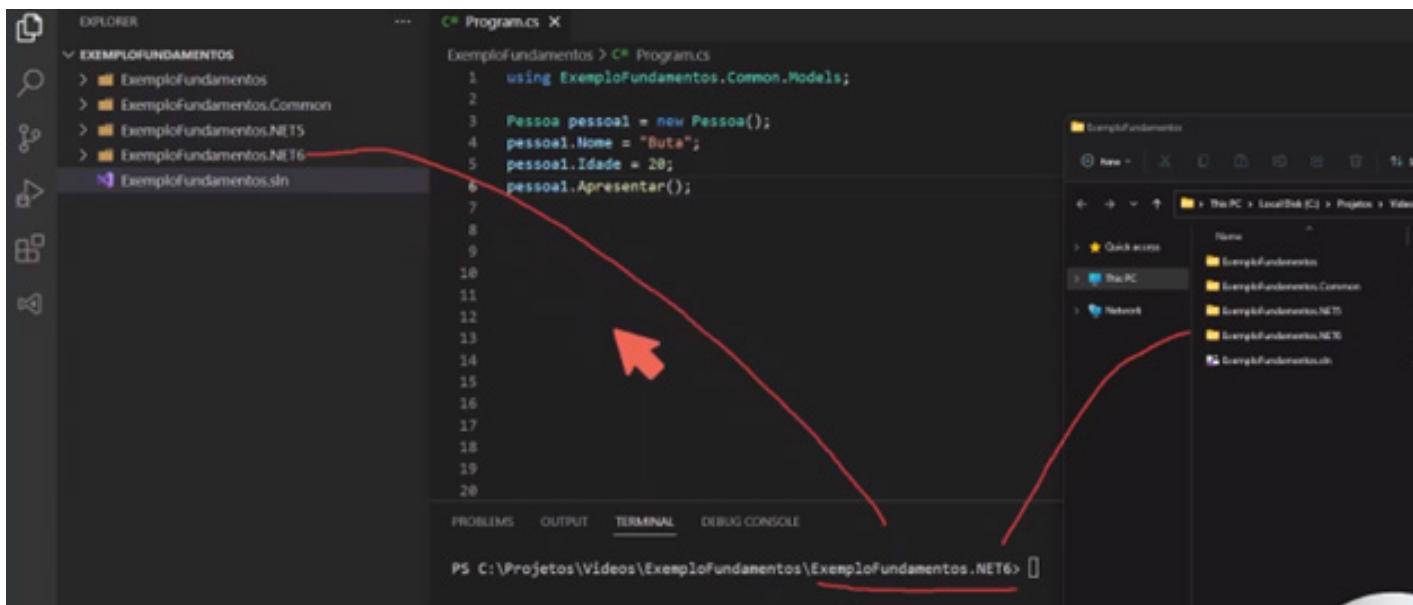
É importante sabermos da existência do método **main** porque muitos projetos ainda não estão na versão 6 ou mais recentes do .NET.

Na imagem a seguir, do lado esquerdo temos a versão .NET5 e do lado direito a versão .NET6 da classe **Program.cs**:

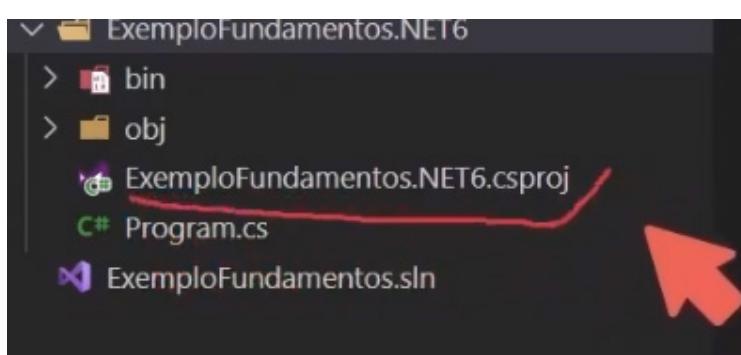
```
ExemploFundamentos.NET5 > C# Program.cs  
1 using System;  
2  
3 namespace ExemploFundamentos.NET5  
4 {  
5     class Program  
6     {  
7         static void Main(string[] args)  
8         {  
9             Console.WriteLine("Hello World!");  
10        }  
11    }  
12 }  
13  
  
ExemploFundamentos.NET6 > C# Program.cs  
1 // See https://aka.ms/new-console-template for more information  
2 Console.WriteLine("Hello, World!");  
3
```

Entendendo o caminho no terminal

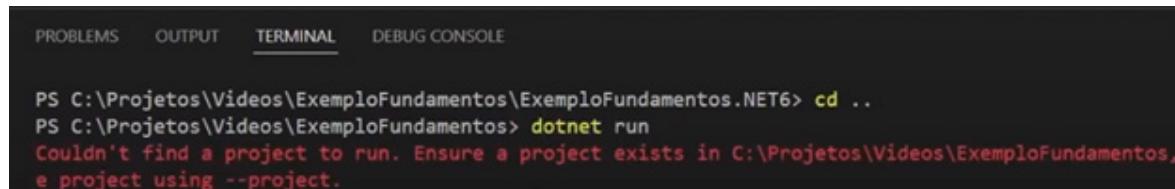
É muito importante manter sempre a atenção quanto ao caminho do projeto no nosso terminal. Olhando para o caminho no nosso terminal, podemos identificar em qual pasta estamos referenciando:



Então, nesse caso quando rodarmos o código **dotnet run**, vamos executar o projeto referente à pasta que estamos que é a de nome **ExemploFundamentos.NET6**.



O comando **dotnet run** vai sempre procurar executar o arquivo **.csproj** da pasta do projeto. Caso não encontre o arquivo, dará erro. Por exemplo, se voltarmos um nível até a pasta principal do nosso programa e executarmos o comando, veja o que acontece:



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Projetos\Videos\ExemploFundamentos\ExemploFundamentos.NET6> cd ..
PS C:\Projetos\Videos\ExemploFundamentos> dotnet run
Couldn't find a project to run. Ensure a project exists in C:\Projetos\Videos\ExemploFundamentos,
or specify one using --project.
```

III Array e Listas

Introdução Array

Array é uma estrutura de dados que armazena valores do mesmo tipo, com um tamanho fixo.

```
int[] array = new int[4];
int[] array = new int[]{42, 75, 74, 61};
string[] nomes = {"Jan", "Fev"};
```

Valores	42	75	74	61
Posição	0	1	2	3

O índice é a posição de um determinado valor de um array, sempre começando com zero. Se tentarmos acessar uma posição que não existe num determinado array, o programa será encerrado com um erro.

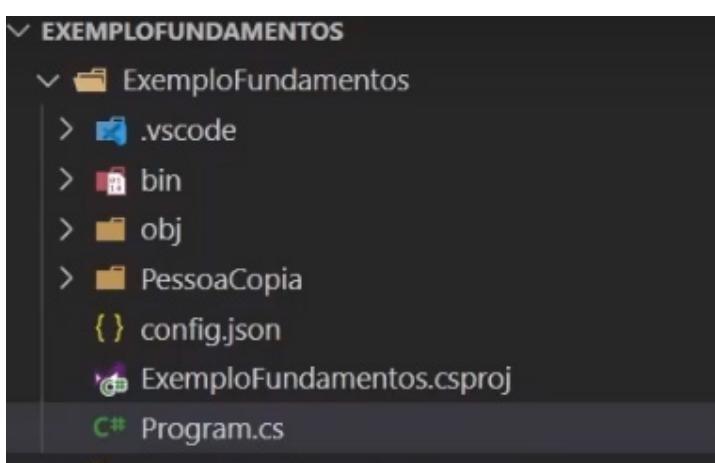
Para atribuir à uma variável o valor de um item do array:

```
int elemento = array[0];
```

E para atribuir um novo valor para uma posição específica dentro de um array:

```
array[0] = 42;
```

Implementando um array de inteiros



Vamos continuar trabalhando a classe do projeto **ExemploFundamentos**.

Vamos iniciar uma variável do tipo array de inteiros com 3 posições:

```
int[] arrayInteiros = new int[3];
```

Agora, vamos atribuir valores para as posições do array:

```
arrayInteiros[0] = 72;  
arrayInteiros[1] = 64;  
arrayInteiros[2] = 50;
```

Para acessar o valor do array, podemos utilizar o laço de repetição for:

```
11  for(int contador = 0; contador < arrayInteiros.Length; contador++)  
12  {  
13      Console.WriteLine($"Posição Nº {contador} - {arrayInteiros[contador]}");  
14  }
```

A propriedade **Length** do array serve para retornar a quantidade de posições que o array tem, nesse caso 3.

Precisamos agora navegar através do terminal para a pasta correta do nosso projeto e rodar nosso projeto:

```
PS C:\Projetos\Videos\ExemploFundamentos\ExemploFundamentos.NET6> cd ..  
PS C:\Projetos\Videos\ExemploFundamentos> cd .\ExemploFundamentos\
```

Ao executar **dotnet run**:

```
Posição Nº 0 - 72  
Posição Nº 1 - 64  
Posição Nº 2 - 50
```

Acessando um índice inválido

```
4  int[] arrayInteiros = new int[3];  
5  
6  arrayInteiros[0] = 72;  
7  arrayInteiros[1] = 64;  
8  arrayInteiros[2] = 50;  
9  arrayInteiros[3] = 1;
```

Vamos ver o que acontece ao tentar acessar uma posição inexistente do array, neste caso a posição 3.

Ao executar **dotnet run**:

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Projetos\Videos\ExemploFundamentos\ExemploFundamentos> dotnet run
Posição N° 0 - 72
Posição N° 1 - 64
Posição N° 2 - 50
PS C:\Projetos\Videos\ExemploFundamentos\ExemploFundamentos> dotnet run
Unhandled exception. System.IndexOutOfRangeException: Index was outside the bounds of the array.
```

Para solucionar este problema em específico, poderíamos aumentar o range do array:

```
int[] arrayInteiros = new int[4];
```

Ao executar **dotnet run**:

```
PS C:\Projetos\Videos\ExemploFundamentos\ExemploFundamentos> dotnet run
Posição N° 0 - 72
Posição N° 1 - 64
Posição N° 2 - 50
Posição N° 3 - 1
```

Percorrendo um array com FOREACH

Quando estamos trabalhando com um array, podemos usar um outro operador para percorrê-lo, o **foreach**. Com ele, podemos iterar sobre um array sem a necessidade de uma variável de contador:

```
foreach(int valor in arrayInteiros)
{
    Console.WriteLine(valor);
}
```

Ao executar **dotnet run**:

```
72
64
50
1
```

E se quisermos ter um contador, podemos declarar do modo convencional:

```
17 Console.WriteLine("Percorrendo o Array com o FOREACH");
18
19 int contadorForeach = 0;
20 foreach(int valor in arrayInteiros)
21 {
22     Console.WriteLine($"Posição N° {contadorForeach} - {valor}");
23     contadorForeach++;
24 }
```

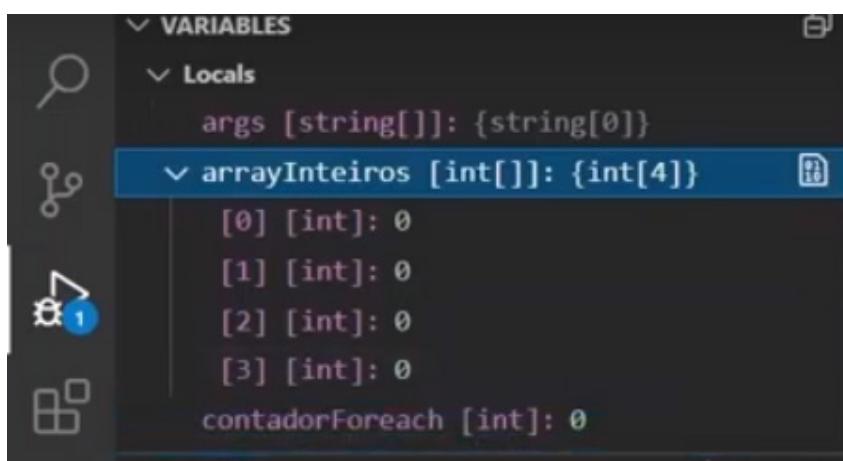
Debugando um Array

Select environment

- .NET 5+ and .NET Core
- .NET Framework 4.x (Windows only)
- Install an extension for C#...

```
4 int[] arrayInteiros = new int[4];
5
6 arrayInteiros[0] = 72;
7 arrayInteiros[1] = 64;
8 arrayInteiros[2] = 50;
9 arrayInteiros[3] = 1;
```

Vamos iniciar setando um breakpoint na linha 6, apertando a tecla F5 (Windows) e selecionando a opção .NET5+ and .NET Core e após isso, apertar o F5 novamente.



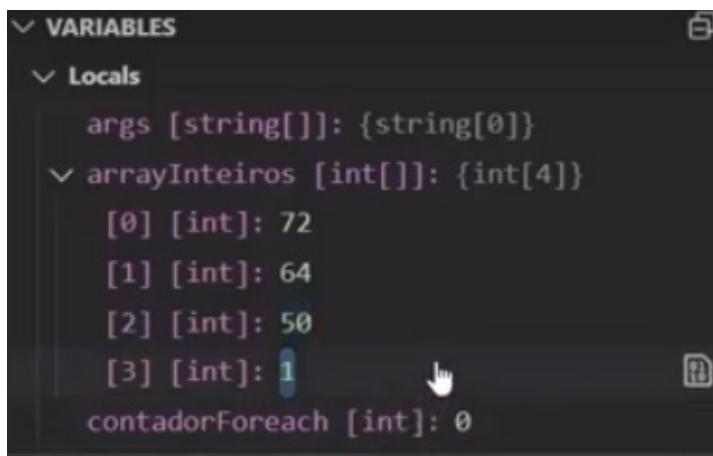
Podemos conferir que nosso array foi criado com seus valores padrão como 0, isso porque zero é o valor padrão do tipo inteiro.

Ao avançarmos com a execução do debugging, vemos que os valores são atribuídos às posições do array:

```

● 6   arrayInteiros[0] = 72;
7   arrayInteiros[1] = 64;
8   arrayInteiros[2] = 50;
9   arrayInteiros[3] = 1;
10
D 11 Console.WriteLine("Percorrendo o Array com o FOR");

```



Depois que nossa execução passar da condição do primeiro loop for, a execução partirá para o início do próximo código. Depois de avançar alguns passos, notamos que o valor da primeira posição do array é atribuído à variável **valor** do loop foreach:

```

ExemploFundamentos > C# Program.cs
● 6   arrayInteiros[0] = 72;
7   arrayInteiros[1] = 64;
8   arrayInteiros[2] = 50;
9   arrayInteiros[3] = 1;
10
11 Console.WriteLine("Percorrendo o Array com o FOR");
12 for(int contador = 0; contador < arrayInteiros.Length; contador++)
13 {
14     Console.WriteLine($"Posição Nº {contador} - {arrayInteiros[contador]}");
15 }
16
17 Console.WriteLine("Percorrendo o Array com o FOREACH");
18
19 int contadorForeach = 0;
20 foreach(int valor in arrayInteiros)
D 21 {
22     Console.WriteLine($"Posição Nº {contadorForeach} - {valor}");
23     contadorForeach++;
24 }

```

Concluindo, tanto o for quanto o foreach poderão ser usados para iterar sobre as posições de um array. A diferença é que o for usa uma variável auxiliar como contador.

Redimensionando um Array

Quando declaramos um array, declaramos também o número de posições que ele terá. Esse número de posições do array não pode ser alterado em tempo de execução do código pelo modo convencional, nem para adicionar nem diminuir uma posição.

Para podermos modificar o tamanho de um array, podemos utilizar o método **.Resize()**. Esse método espera receber 2 argumentos, o primeiro é o endereço da memória do nosso array, acessado pela palavra **ref** e o segundo argumento é o valor para o qual queremos alterar o nosso array:

```
Array.Resize(ref arrayInteiros,
arrayInteiros.Length * 2);
```

Através da ferramenta de debug, podemos conferir o funcionamento desse método. Repare na quantidade de posições antes e depois da execução do método:

```
ExemploFundamentos > C# Program.cs
1  using ExemploFundamentos.Common.Models;
2
3
4  int[] arrayInteiros = new int[4];
5
6  arrayInteiros[0] = 72;
7  arrayInteiros[1] = 64;
8  arrayInteiros[2] = 50;
9  arrayInteiros[3] = 1;
```

```
ExemploFundamentos > C# Program.cs
1  using ExemploFundamentos.Common.Models;
2
3
4  int[] arrayInteiros = new int[4];
5
6  arrayInteiros[0] = 72;
7  arrayInteiros[1] = 64;
8  arrayInteiros[2] = 50;
9  arrayInteiros[3] = 1;
10
11
12  Array.Resize(ref arrayInteiros, arrayInteiros.Length * 2);
13
14
15  Console.WriteLine("Percorrendo o Array com o FOR");
16  for(int contador = 0; contador < arrayInteiros.Length; contador++)
17  {
18      Console.WriteLine($"Posição N° {contador} - {arrayInteiros[contador]}");
19 }
```

Um array nasce e morre com a quantidade de posições no início da declaração. O que o método **Resize()** faz é, através de um processo interno, criar uma cópia do nosso array com a capacidade diferente e copiar os elementos do array antigo para o novo.

Copiando um Array para outro

Vamos criar um novo array com o dobro da capacidade do array antigo, e com o método **Copy()** vamos copiar os elementos do array antigo para o novo.

```
Array.Copy(arrayInteiros, arrayInteirosDobrado,
          arrayInteiros.Length);
```

Repare que ao avançarmos com a execução do debugging até a etapa anterior a execução do método, nosso novo array está com todas as posições valendo zero:

```
ExemploFundamentos > C# Program.cs
1  using ExemploFundamentos.Common.Models;
2
3
4  int[] arrayInteiros = new int[4];
5
6  arrayInteiros[0] = 72;
7  arrayInteiros[1] = 64;
8  arrayInteiros[2] = 50;
9  arrayInteiros[3] = 1;
10
11
12 int[] arrayInteirosDobrado = new int[arrayInteiros.Length * 2];
13 Array.Copy(arrayInteiros, arrayInteirosDobrado, arrayInteiros.Length);
14
15
16 //Array.Resize(ref arrayInteiros, arrayInteiros.Length * 2);
17
18
19 Console.WriteLine("Percorrendo o Array com o FOR");
```

Agora, após avançarmos com a execução mais uma vez, veremos o método em ação:

```
ExemploFundamentos > C# Program.cs
1  using ExemploFundamentos.Common.Models;
2
3
4  int[] arrayInteiros = new int[4];
5
6  arrayInteiros[0] = 72;
7  arrayInteiros[1] = 64;
8  arrayInteiros[2] = 50;
9  arrayInteiros[3] = 1;
10
11
12 int[] arrayInteirosDobrado = new int[arrayInteiros.Length * 2];
13 Array.Copy(arrayInteiros, arrayInteirosDobrado, arrayInteiros.Length);
14
15
16 //Array.Resize(ref arrayInteiros, arrayInteiros.Length * 2);
17
18
19 Console.WriteLine("Percorrendo o Array com o FOR");
```

Trabalhando com listas

Uma lista é um array “melhorado”. Assim como o array, ela também é capaz de representar coleções de um objeto do mesmo tipo, porém sem toda a complexidade de um array. Com a lista, não precisamos nos preocupar em gerenciar o tamanho e temos facilidades para adicionar e remover valores.

```
List<string> listaString = new List<string>();
```

<string> é o tipo de dados da lista dentro dos símbolos de menor que (<) e maior que (>).

Para adicionar elementos dentro da lista usamos o método **Add()**:

```
listaString.Add("SP");
listaString.Add("BA");
listaString.Add("MG");
```

E para poder iterar sobre os elementos da lista, podemos utilizar tanto o método for quanto o foreach da mesma forma como fizemos com o array:

```
9  for(int contador = 0; contador < listaString.Count; contador++)
10 {
11     Console.WriteLine($"Posição N° {contador} - {listaString[contador]}");
12 }
```

```
15 int contadorForeach = 0;
16 foreach(string item in listaString)
17 {
18     Console.WriteLine($"Posição N° {contadorForeach} - {item}");
19     contadorForeach++;
20 }
```

Para podermos entender como que uma lista funciona, vamos fazer algumas alterações no código e executar a ferramenta de debug:

```

ExemploFundamentos > C# Program.cs
1  using ExemploFundamentos.Common.Models;
2
3  List<string> listaString = new List<string>();
4
5  listaString.Add("SP");
6  listaString.Add("BA");
7  listaString.Add("MO");
8  listaString.Add("RJ");
9
10 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
11
12 listaString.Add("SC");
13
14 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
15
16 listaString.Remove("MO");
17
18 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");

```

Existem 2 propriedades importantes, **Capacity** e **Count**.

- Capacity
 - Refere-se a capacidade do array. Não é a capacidade da lista em si.
- Count
 - Quantidade de elementos dentro da lista.

Avançando com a execução, o valor será inserido dentro da lista e podemos notar algumas mudanças como **Capacity** valendo 4 e **Count** valendo 1:

```

ExemploFundamentos > C# Program.cs
1  using ExemploFundamentos.Common.Models;
2
3  List<string> listaString = new List<string>();
4
5  listaString.Add("SP");
6  listaString.Add("BA");
7  listaString.Add("MO");
8  listaString.Add("RJ");
9
10 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
11
12 listaString.Add("SC");
13
14 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
15
16 listaString.Remove("MO");
17
18 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");

```

Continuando, chegará um ponto onde vamos querer adicionar mais um elemento à lista, mas já temos 4 elementos sendo que a capacidade total também são 4 valores. Se fosse um array teríamos um problema, mas a lista consegue fazer esse gerenciamento automático:

VARIABLES

- Locals


```
System.Collections.Generic.List<T>.Count
System.Collections.Generic.List<T>.Capacity
System.Runtime.CompilerServices.DefaultI
args [string[]]: {string[0]}
```
- listString [List]: Count = 4


```
[0] [string]: "SP"
[1] [string]: "BA"
[2] [string]: "MG"
[3] [string]: "RJ"
```
- Raw View


```
Capacity [int]: 4
Count [int]: 4
```
- Static members
- Non-Public members

```
ExemploFundamentos > C# Programs.cs
1  using ExemploFundamentos.Common.Models;
2
3  List<string> listaString = new List<string>();
4
5  listaString.Add("SP");
6  listaString.Add("BA");
7  listaString.Add("MG");
8  listaString.Add("RJ");
9
10 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
11
12 listaString.Add("SC");
13
14 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
15
16 listaString.Remove("MG");
17
18 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
```

VARIABLES

- Locals


```
args [string[]]: {string[0]}
```
- listString [List]: Count = 5


```
[0] [string]: "SP"
[1] [string]: "BA"
[2] [string]: "MG"
[3] [string]: "RJ"
[4] [string]: "SC"
```
- Raw View


```
Capacity [int]: 8
Count [int]: 5
```
- Static members

```
ExemploFundamentos > C# Programs.cs
1  using ExemploFundamentos.Common.Models;
2
3  List<string> listaString = new List<string>();
4
5  listaString.Add("SP");
6  listaString.Add("BA");
7  listaString.Add("MG");
8  listaString.Add("RJ");
9
10 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
11
12 listaString.Add("SC");
13
14 Console.WriteLine($"Itens na minha lista: {listaString.Count} - Capacidade: {listaString.Capacity}");
```

Quando queremos remover um elemento, utilizamos o método **Remove()** e passamos o valor do elemento que queremos remover.

`listString.Remove("MG");`

VARIABLES

- Locals


```
System.Collections.Generic.List<T>.Remov
args [string[]]: {string[0]}
```
- listString [List]: Count = 4


```
[0] [string]: "SP"
[1] [string]: "BA"
[2] [string]: "RJ"
[3] [string]: "SC"
```
- Raw View


```
Capacity [int]: 8
Count [int]: 4
```
- Static members
- Non-Public members

O elemento será removido e os itens serão reorganizados, passando próximo a ocupar o lugar do valor que foi removido e assim por diante.

III Comentários e boas práticas

Introdução aos comentários

Os comentários são um recurso muito importante em qualquer linguagem de programação. Eles servem para documentar seu código, explicando determinado método ou execução. Também auxiliam outros programadores a entender o que está acontecendo.

Introdução aos comentários

Comentário com:

//: É representado em uma única linha

/* comentário */: Permite escrever com várias linhas

<summary>: Permite documentar classes, métodos, parâmetros, etc

Os comentários são completamente ignorados na hora da compilação do código, pois eles não fazem sentido para o computador.

Comentários de linha única:

```
5 // Instancia da classe pessoa
6 Pessoa p = new Pessoa();
7
8 // Atribui o nome e idade para pessoa
9 p.Nome = "Buta";
10 p.Idade = 20;
11
12 // Faz a pessoa se apresentar
13 p.Apresentar();
```

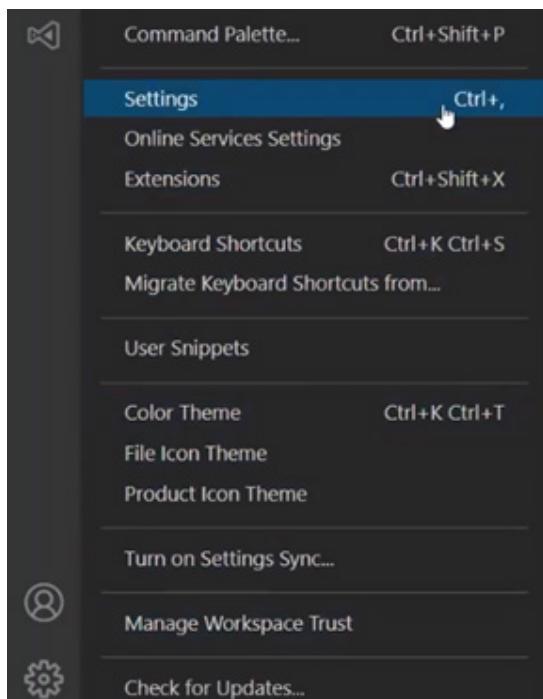
Comentários de múltiplas linhas:

```
8 /*
9  *     Atribui o nome e idade para pessoa
10    passando o nome Buta
11    e passando a idade 20
12 */
```

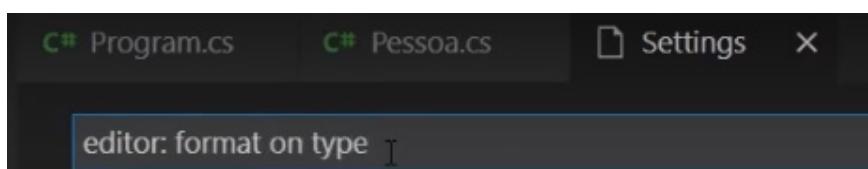
Comentando nossas classes

Ao passarmos o mouse em cima da classe Pessoa, notamos que nos é informada apenas o **namespace**:

```
4 class ExemploFundamentos.Common.Models.Pessoa
5
6 Pessoa p = new Pessoa();
```

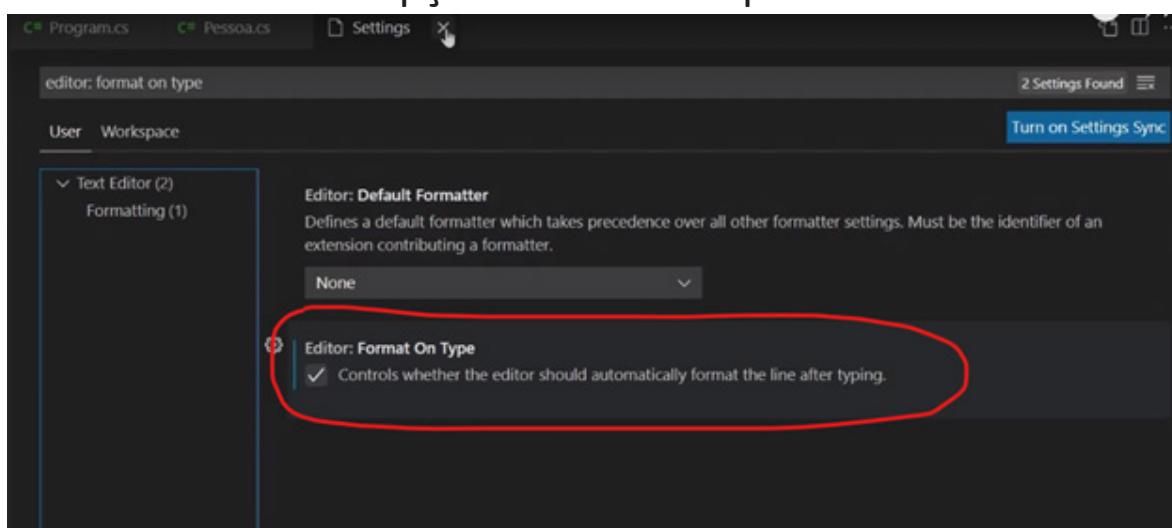


Para poder adicionar um comentário que vá aparecer ao passar o mouse. Antes disso, precisamos habilitar uma opção do VS Code para facilitar esse processo. Então, entraremos nos Settings do VS Code.



Agora vamos procurar pela opção editor:
format on type

Vamos deixar essa opção habilitada e podemos fechar o Settings:



```

8   /// <summary>
9   ///
10  /// </summary>

```

Feito isso, ao retornarmos ao código e digitarmos 3 barras seguidas (///), o VS Code irá auto completar com o <summary>.

Agora podemos escrever uma descrição para nossa classe:

```

/// <summary>
/// Representa uma pessoa física
/// <summary>

```

```

6 namespace ExemploFundamentos.Common.Models
7 {
8     /// <summary>
9     /// Representa uma pessoa física
10    /// </summary>
11    public class Pessoa
12    {
13        public string Nome { get; set; }
14        public int Idade { get; set; }
15        public string NomeRepresentanteLegalDaPessoaFisica { get; set; }
16
17        public void Apresentar()
18        {
19            Console.WriteLine($"Olá, meu nome é {Nome}, e tenho {Idade} anos");
20            //Console.WriteLine($"Olá, meu nome é {Nome} \n e tenho {Idade} anos");
21        }
22    }
23 }

```

Comentários nos métodos

```

16 // void Pessoa.Apresentar()
17 p.Apresentar();

```

Ao passarmos o mouse no método **Apresentar()**, nenhuma informação relevante será exibida.

Da mesma forma que fizemos na classe Pessoa, vamos digitar três barras (///) na linha anterior ao método apresentar para que o <summary> apareça:

```

17     /// <summary>
18     /// Faz a pessoa se apresentar, dizendo seu nome e idade
19     /// </summary>
20     public void Apresentar()
21     {

```

Voltando ao **Program.cs**, agora ao passar o mouse em cima do método nos é mostrado a descrição que escrevemos:

```
P. void Pessoa.Apresentar()
// Faz a pessoa se apresentar, dizendo seu nome e idade
p.Apresentar();
```

```
/// <summary>
///
/// </summary>
/// <param name="x"></param>
/// <param name="y"></param>
public void Somar(int x, int y)
```

```
/// <summary>
/// Realiza uma soma de dois números
/// </summary>
/// <param name="x">O primeiro número inteiro para somar</param>
/// <param name="y">O segundo número inteiro para somar</param>
public void Somar(int x, int y)
{
    Console.WriteLine($"{x} + {y} = {x + y}");
}
```

Também é possível escrevermos comentários sobre os parâmetros do método. Vamos utilizar como exemplo a classe Calculadora. Ao digitarmos as três barras no método **Somar()**, notamos

algumas diferenças.

Ao retornarmos a classe **Program.cs**, vamos instanciar a classe Calculadora e utilizar o método **Somar()**:

```
Calculadora c = new Calculadora();
```

```
15      void Calculadora.Somar(int x, int y)
16 // Faz a p.Apresa x: O primeiro número inteiro para somar
17 p.Apresa x: O primeiro número inteiro para somar
18 Calculad Realiza uma soma de dois números
20 c.Somar()
```

```
15      void Calculadora.Somar(int x, int y)
16 // Faz a p.Apresenta y: O segundo número inteiro para somar
17 p.Apresenta y: O segundo número inteiro para somar
18 Calculadora Realiza uma soma de dois números
20 c.Somar[3, ]
```

Além disso, também podemos documentar o que o método irá retornar:

<returns>Retorna a subtração de x e y</returns>

```
20     /// <summary>
21     /// Realiza uma subtração de dois números
22     /// </summary>
23     /// <param name="x">O primeiro número inteiro para subtrair</param>
24     /// <param name="y">O segundo número inteiro para subtrair</param>
25     /// <returns>Retorna a subtração de x e y</returns>
26     public int Subtrair(int x, int y)
27     {
28         Console.WriteLine($"{x} - {y} = {x - y}");
29         return x - y;
30     }
```

```
15     int Calculadora.Subtrair(int x, int y)
16 // p. Realiza uma subtração de dois números
17 Returns:
18 Ca
19 & Retorno a subtração de x e y
20 c.Subtrair(3, 5);
```

III Construindo um Sistema para um Estacionamento

Apresentação do Projeto

Objetivos:

- Adicionar um veículo
- Remover um veículo
- Dar o preço pelo uso do estacionamento
- Listar os veículos estacionados

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

PS C:\Projetos\trilha-net-fundamentos\DesafioFundamentos> dotnet run
Seja bem vindo ao sistema de estacionamento!
Digite o preço inicial:
5
Agora digite o preço por hora:
2
```

```
Digite a sua opção:
1 - Cadastrar veículo
2 - Remover veículo
3 - Listar veículos
4 - Encerrar
1
Digite a placa do veículo para estacionar:
XYZ-1234
Pressione uma tecla para continuar
```

```
Digite a sua opção:
1 - Cadastrar veículo
2 - Remover veículo
3 - Listar veículos
4 - Encerrar
3
Os veículos estacionados são:
ABC-1234
XYZ-1234
Pressione uma tecla para continuar
```

```
Digite a sua opção:
1 - Cadastrar veículo
2 - Remover veículo
3 - Listar veículos
4 - Encerrar
4
Pressione uma tecla para continuar
O programa se encerrou
```

```

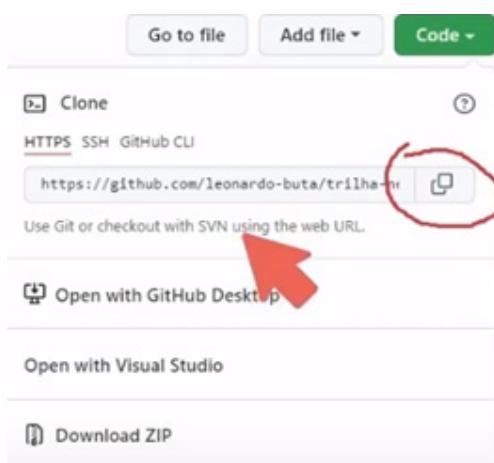
Digite a sua opção:
1 - Cadastrar veículo
2 - Remover veículo
3 - Listar veículos
4 - Encerrar
2

Digite a placa do veículo para remover:
ABC-1234
Digite a quantidade de horas que o veículo permaneceu estacionado:
4
O veículo ABC-1234 foi removido e o preço total foi de: R$ 13
Pressione uma tecla para continuar

```

Acessando o Repositório do Desafio

- Instalar o Git
- Acessar o repositório do projeto e realizar o fork para seu repositório
 - <https://github.com/digitalinnovationone/trilha-net-fundamentos-desafio>
- Baixar o projeto remoto para sua máquina local através do Git bash



Para baixar o projeto, primeiro devemos copiar o link na página do nosso projeto que foi copiado.

Depois utilizar o Git Bash para baixar o projeto para nossa máquina:

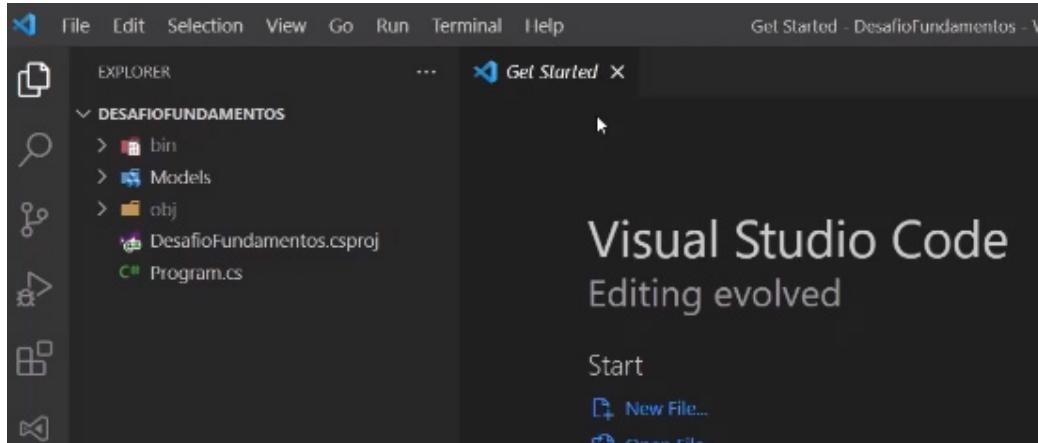
<https://github.com/leonardo-buta/trilha-net-fundamentos-desafio.git>

```

MINGW64:/c/projetos
Buta@DESKTOP-CMG8KVR MINGW64 ~
$ cd c:/projetos
Buta@DESKTOP-CMG8KVR MINGW64 /c/projetos
$ git clone https://github.com/leonardo-buta/trilha-net-fundamentos-desafio.git

```

Após isso, basta abrir a pasta do projeto no VS Code:



Hands-On: Entendendo o Desafio

Para este desafio, iremos trabalhar na classe **Estacionamento.cs**, onde estão faltando trechos de código. Iremos trabalhar nos métodos **AdicionarVeiculo()**, **RemoverVeiculo()**, e **ListarVeiculos()**.

Onde está escrito **//TODO** e **// *IMPLEMENTE AQUI*** é onde devemos trabalhar para chegar à solução do desafio.



Obrigado pela leitura!

Você não chegou aqui pulando páginas, né? 😊

Brincadeiras à parte, realmente nós da DIO esperamos que esteja curtindo sua jornada de aprendizado aqui conosco e desejamos seu sucesso sempre!

Vamos em frente!

