



C# Fundamentos

Público Alvo

Desenvolvedores e futuros desenvolvedores interessados em ingressar seus conhecimentos na linguagem de programação orientada a objetos da *Microsoft*.

Pré-Requisitos

Conhecimentos de Lógica de Programação e Banco de Dados.

Índice

C# FUNDAMENTOS	1
PÚBLICO ALVO	1
PRÉ-REQUISITOS	1
ÍNDICE	1
COPYRIGHT	5
EQUIPE.....	5
HISTÓRICO DAS EDIÇÕES	5
CAPÍTULO 01 – INTRODUÇÃO AO MICROSOFT .NET CORE	7
PROPOSTA E METODOLOGIA DO CURSO.....	7
O QUE É o .NET CORE	7
PROCESSO DE COMPILAÇÃO COM <i>MICROSOFT .NET CORE</i>	8
CAPÍTULO 02 – VISUAL STUDIO 2019 COMMUNITY EDITION	10
INSTALAÇÃO BÁSICA.....	10
TIPOS DE PROJETOS	10
PROJETO: HELLO WORLD.....	13
CAPÍTULO 03 – TIPOS DE DADOS, VARIÁVEIS E CONSTANTES	17
TIPOS DE DADOS	17
VARIÁVEIS.....	18
CONSTANTES	18
CAPÍTULO 04 – OPERADORES	19
ARITMÉTICOS.....	19
COMPARAÇÃO	20
LÓGICOS.....	20
ATRIBUIÇÃO	21
CONCATENAÇÃO	22
OPERADORES TERNÁRIOS.....	22
EXERCÍCIOS	23
CAPÍTULO 05 – CONSOLE APPLICATION	24
ESTRUTURA SEQUENCIAL	24
LINHAS DE COMENTÁRIO.....	25
INPUT E OUTPUT	25
COMANDOS DE SAÍDA (OUTPUT).....	26
COMANDOS DE ENTRADA (INPUT)	26
criando MÉTODOS	28
EXERCÍCIOS	32
CAPÍTULO 06 – FUNÇÕES DO .NET CORE	33
FUNÇÕES DE MANIPULAÇÃO DE TEXTO	33
FUNÇÕES MATEMÁTICAS E TRIGONOMÉTRICAS	34

EXERCÍCIOS.....	36
CAPÍTULO 07 – COMANDOS CONDICIONAIS	37
ESTRUTURAS DE DECISÃO	37
COMANDOS IF / ELSE / ELSE IF	37
COMANDO SWITCH.....	39
EXERCÍCIOS.....	40
CAPÍTULO 08 – ESTRUTURAS / LAÇOS DE REPETIÇÃO	41
COMANDO WHILE / DO WHILE	41
COMANDO FOR.....	42
COMANDO FOREACH.....	44
EXERCÍCIOS.....	44
CAPÍTULO 09 – VARIÁVEIS INDEXADAS.....	45
VARIÁVEIS INDEXADAS UNIDIMENSIONAIS (VETORES/ARRAYS).....	45
VARIÁVEIS INDEXADAS MULTIDIMENSIONAIS (MATRIZES).....	48
EXERCÍCIOS.....	49
CAPÍTULO 10 – DEPURANDO / DEBUGANDO CÓDIGOS	50
DEPURAR CÓDIGOS NO <i>VISUAL STUDIO 2019 COMMUNITY</i>	50
CAPÍTULO 11 – ORIENTAÇÃO À OBJETOS	55
C# E A PROGRAMAÇÃO ORIENTADA-A-OBJETOS	55
NAMESPACES.....	56
CLASSES E OBJETOS.....	57
ENCAPSULAMENTO	62
HERANÇA.....	65
POLIMORFISMO.....	67
ABSTRAÇÃO.....	71
INTERFACES	73
EXERCÍCIOS.....	75
CAPÍTULO 12 – EXERCÍCIOS DE REVISÃO – PARTE I	76
CAPÍTULO 13 – TRATAMENTO DE ERROS / EXCEÇÕES.....	82
TRY / CATCH / FINALLY.....	82
SYSTEM.EXCEPTION	83
CRIAR SUAS PRÓPRIAS EXCEPTIONS.....	85
EXERCÍCIOS.....	86
CAPÍTULO 14 – TRABALHANDO COM ARQUIVOS DE TEXTO.....	87
SYSTEM.IO (INPUT OUTPUT)	87
LER UM ARQUIVO COM STREAMREADER.....	87
ESCREVER UM ARQUIVO COM STREAMWRITER	88
EXERCÍCIOS.....	89
CAPÍTULO 15 – CONECTANDO-SE COM O BANCO DE DADOS SQL SERVER	90
CONNECTIONSTRING	90
UTILIZANDO A BIBLIOTECA SQLCONNECTION	90

Anotações

ABRINDO A CONEXÃO COM O BANCO DE DADOS	91
CAPÍTULO 16 – REALIZANDO CONSULTAS NO BANCO DE DADOS E PREENCHENDO OBJETO.....	92
MONTANDO QUERY PARA EXECUTAR NO BANCO DE DADOS	92
UTILIZANDO A BIBLIOTECA SQLCOMMAND	92
LENDOS DADOS COM O SQLREADER.....	93
CAPÍTULO 17 – INSERT, UPDATE, DELETE COM O C# NO SQLSERVER	95
EXECUTANDO COMANDO INSERT.....	95
EXECUTANDO COMANDO UPDATE	95
EXECUTANDO COMANDO DELETE	96
CAPÍTULO 18 – INTRODUÇÃO À INTERFACES GRÁFICAS (FORMULÁRIOS)	97
O QUE SÃO COMPONENTES VISUAIS.....	97
O QUE É O WINDOWS FORMS.....	97
CRIANDO UM PROJETO EM WINDOWS FORMS	97
CAPÍTULO 19 – ENTENDENDO E CRIANDO COMPONENTES DE FORMULÁRIOS	99
COMPONENTES DO WINDOWS FORMS	99
COMPONENTE LABEL E SUAS PROPRIEDADES	99
COMPONENTE TEXTBOX E SUAS PROPRIEDADES.....	100
COMPONENTE BUTTON E SUAS PROPRIEDADES.....	101
COMPONENTE PANEL E SUAS PROPRIEDADES	102
COMPONENTE NUMERICUPDOWN E SUAS PROPRIEDADES.....	103
COMPONENTE DATETIMEPICKER E SUAS PROPRIEDADES	104
COMPONENTE COMBOBOX E SUAS PROPRIEDADES.....	105
COMPONENTE DATAGRIDVIEW E SUAS PROPRIEDADES	106
EXERCÍCIOS	107
CAPÍTULO 20 – EXERCÍCIOS DE REVISÃO – PARTE II	108
CAPÍTULO 21 – EVENTOS.....	110
O QUE SÃO E PARA QUE SERVEM OS EVENTOS?	110
EVENTO CLICK DO BUTTON.....	110
EVENTO TEXTCHANGED DO TEXTBOX.....	110
EVENTO LOAD DO FORMULÁRIO.....	110
EVENTO SELECTEDINDEXCHANGED DO COMBOBOX	111
EVENTO CELLCCLICK DO DATAGRIDVIEW	111
EXERCÍCIOS	111
CAPÍTULO 22 – ESTRUTURA DOS FORMULÁRIOS	112
OBJETOS FORMULÁRIOS	112
HERANÇA DE FORMULÁRIOS	113
PARTIAL CLASSES.....	116
EXERCÍCIOS	116
CAPÍTULO 23 – VALIDAÇÕES	117
VALIDANDO COMPONENTES DE TEXTO	117
VALIDANDO COMPONENTE COMBOBOX	118
VALIDANDO COMPONENTE DATETIMEPICKER	120

VALIDANDO COMPONENTE NUMERICUPDOWN	121
EXERCÍCIOS.....	122
CAPÍTULO 24 – IMPLEMENTAÇÃO DAS REGRAS DE NEGÓCIO.....	123
EXEMPLOS DE REGRAS DE NEGÓCIO	124
EXERCÍCIO	124
CAPÍTULO 25 – RELATÓRIOS	125
criando uma tabela.....	125
criando um novo projeto	126

	Anotações

Copyright

As informações contidas neste material se referem ao curso de **C# Fundamentos** e estão sujeitas as alterações sem comunicação prévia, não representando um compromisso por parte do autor em atualização automática de futuras versões.

A Apex não será responsável por quaisquer erros ou por danos acidentais ou consequenciais relacionados com o fornecimento, desempenho, ou uso desta apostila ou os exemplos contidos aqui.

Os exemplos de empresas, organizações, produtos, nomes de domínio, endereços de e-mail, logotipos, pessoas, lugares e eventos aqui representados são fictícios. Nenhuma associação a empresas, organizações, produtos, nomes de domínio, endereços de e-mail, logotipos, pessoas, lugares ou eventos reais é intencional ou deve ser inferida.

A reprodução, adaptação, ou tradução deste manual mesmo que parcial, para qualquer finalidade é proibida sem autorização prévia por escrito da **Apex**, exceto as permitidas sob as leis de direito autoral.

Equipe

Conteúdos	Diagramação	Revisão
■ Thiago Vieira ■ Gustavo Rosauro	■ Fernanda Pereira	■ Fernanda Pereira

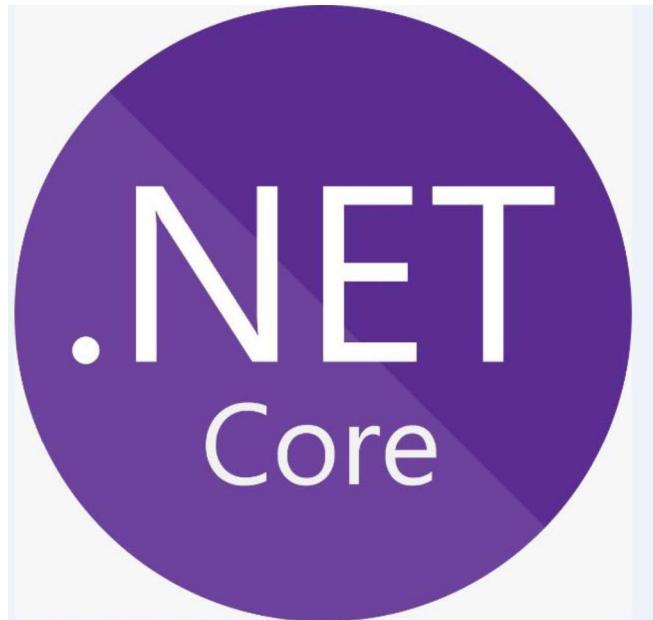
Histórico das Edições

Edição	Idioma	Edição
1ª	Português	Abril de 2017
2ª	Português	Março de 2018
3ª	Português	Julho de 2019

	Anotações

Anotações

Capítulo 01 – Introdução ao Microsoft .NET Core



Proposta e Metodologia do Curso

O Curso C# Fundamentos tem como proposta capacitar o aluno para uma entrada no mercado de trabalho. O foco será o desenvolvimento de todas as habilidades necessárias para desenvolver aplicações comerciais baseadas no .NET Core e .NET Framework para aplicações Desktop, utilizando Windows Forms.

Seguiremos a metodologia da Apex que visa Conteúdos > Demonstrações > Exercícios. Cada módulo formará-se dessa maneira, assim como os cursos anteriores aqui realizados. Porém nesse curso, como o foco é desenvolver um software e prepararmo-nos para o mercado de trabalho, teremos diversos módulos mais práticos do que teóricos, utilizando todo o conhecimento teórico e já praticado até o momento.

Ao término desse curso o aluno será capaz de desenvolver aplicações com interfaces gráficas, lógicas de negócios, e acesso a bancos de dados. Esse conteúdo forma o que diversas empresas nacionais e multinacionais consideram o necessário para um Analista e Desenvolvedor C# em nível introdutório.

O que é o .NET Core

O .NET Core é uma estrutura de software de computador gerenciada, gratuita e de código aberto para os sistemas operacionais Windows, Linux e MacOS. É um sucessor de plataforma aberta do .NET Framework. O projeto é desenvolvido principalmente pela Microsoft e lançado sob a licença MIT.

	Anotações

Processo de Compilação com Microsoft .NET Core

O .NET Core é um ambiente de execução gerenciado que fornece uma variedade de serviços para os aplicativos em execução. Ele consiste em dois componentes principais: o common language runtime (CLR), que é o mecanismo de execução que gerencia a execução de aplicativos; e a biblioteca de classes do .NET Core, que fornece uma biblioteca de testados, código reutilizável que os desenvolvedores possam chamar de seus próprios aplicativos. Os serviços que o .NET Core fornece a execução de aplicativos incluem o seguinte:

Gerenciamento de memória. Em muitas linguagens de programação, os programadores são responsáveis por alocar e liberar memória e para lidar com tempos de vida do objeto. Em aplicativos do .NET Core, o CLR fornece esses serviços em nome do aplicativo.

Um common type system. Em linguagens de programação tradicionais, tipos básicos são definidos pelo compilador, o que complica a interoperabilidade entre linguagens. No .NET Core, os tipos básicos são definidos pelo sistema de tipo do .NET Core são comuns a todas as linguagens que visam o .NET Core.

Uma biblioteca de classes abrangente. Em vez de gravar grandes quantidades de código para manipular operações comuns de programação de nível inferior, os programadores podem usar uma biblioteca prontamente acessível de tipos e membros da biblioteca de classes .NET Core.

Estruturas de desenvolvimento e tecnologias. O .NET Core inclui bibliotecas para áreas específicas de desenvolvimento de aplicativos, como ASP.NET para aplicativos web, o ADO.NET para acesso a dados e o Windows Communication Foundation para aplicativos orientados a serviços.

Interoperabilidade de linguagem. Compiladores de linguagem que visam o .NET Core emitem um código intermediário chamado linguagem CIL (Common Intermediate), que, por sua vez, é compilado em tempo de execução, o common language runtime. Com esse recurso, rotinas escritas em uma linguagem são acessíveis a outros idiomas e programadores podem se concentrar na criação de aplicativos no seu idioma preferencial ou idiomas.

Compatibilidade de versão. Com raras exceções, aplicativos que são desenvolvidos usando uma versão específica do .NET Core podem ser executados sem modificação em uma versão posterior.

Execução lado a lado. O .NET Core ajuda a resolver conflitos de versão, permitindo que várias versões do common language runtime existir no mesmo computador. Isso significa que várias versões de aplicativos também podem coexistir e que um aplicativo pode ser executado na versão do .NET Core com o qual ele foi criado.

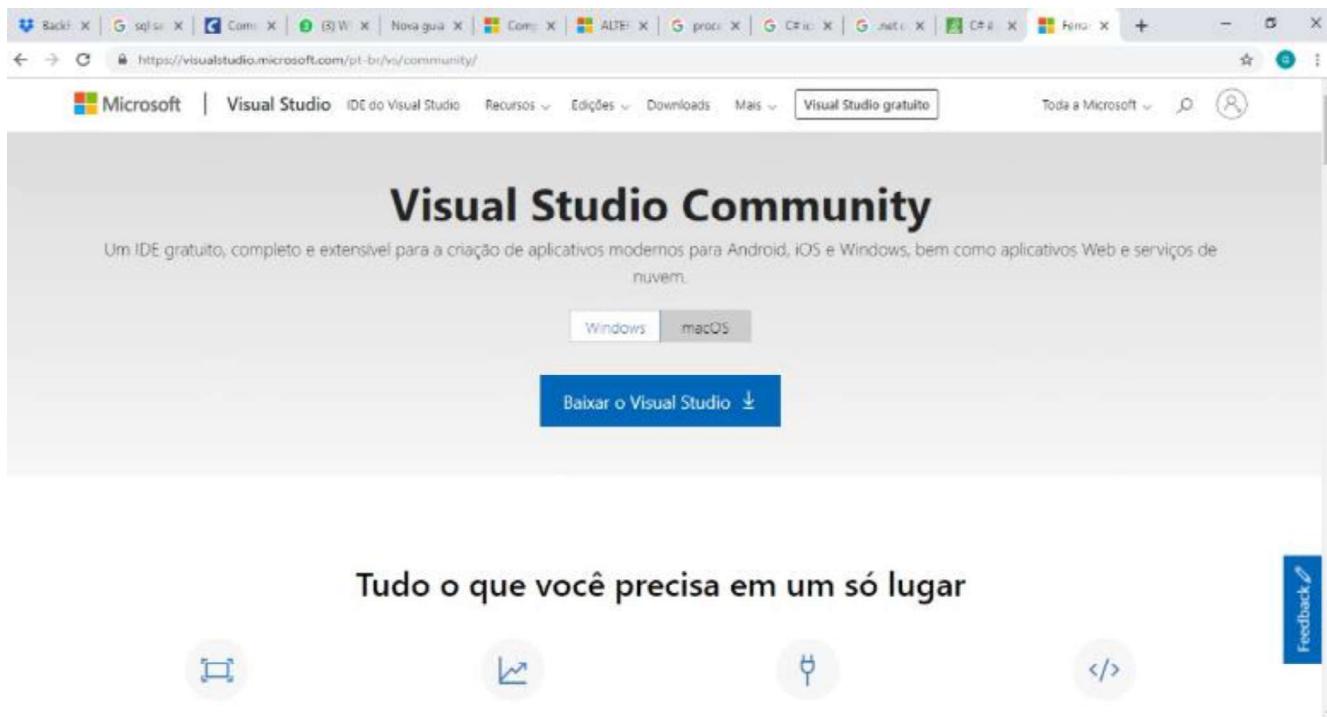
Multiplataforma. Segmentando a biblioteca de classes portátil do .NET Core, os desenvolvedores podem criar assemblies que funcionam em várias plataformas do .NET Core, como o Windows 7, Windows 8, Windows 8.1, Windows 10, Windows Phone e Xbox 360. Para obter mais informações, consulte Desenvolvimento entre plataformas com a Biblioteca de Classes Portátil.

Entre as inúmeras ferramentas fornecidas pelo CLR, uma que difere o C# de uma linguagem como o C++, por exemplo, é o *Garbage Collector*, o responsável por gerenciar o uso de variáveis sem a necessidade de tratamento via código para a maioria das aplicações que não necessitam de uma super performance, porém podem ter perdas.

	Anotações

Anotações

Capítulo 02 – Visual Studio 2019 Community Edition



O *Microsoft Visual Studio 2019 Community* é o principal *IDE (Integrated Development Environment / ambiente de desenvolvimento integrado)* com foco em desenvolvimento para e com o *.NET Core*. Em outras palavras é o Ambiente de desenvolvimento dos produtos *Microsoft*.

Existem diversas versões do *Visual Studio*, variando em funcionalidades e possibilidades de projetos. Para esse curso utilizaremos o *Visual Studio 2019 Community*.

Instalação Básica

Para trabalhar com o VS (*Visual Studio 2019 Community*) basta fazer o download de seu instalador do site oficial do produto - <https://www.visualstudio.com/pt-br/vs/community/>

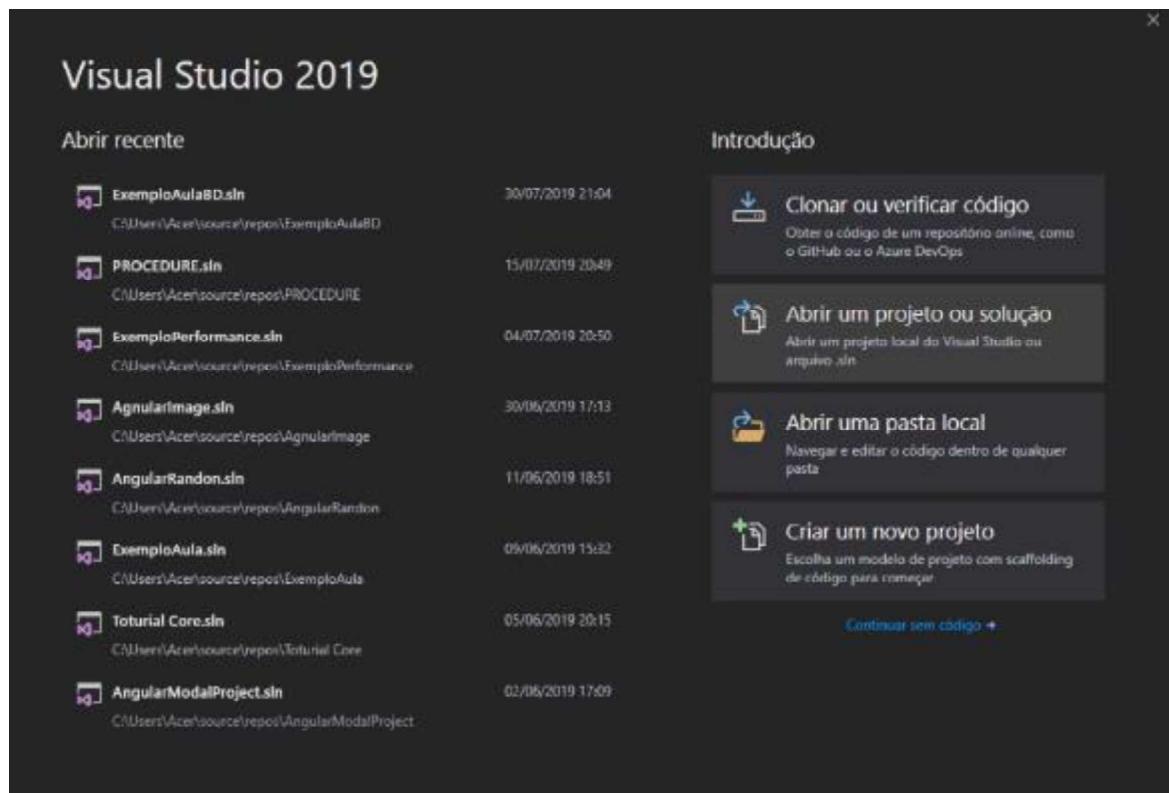
Escolha qual a versão que deseja baixar (lembrando que utilizaremos a versão *Visual Studio 2019 Community*) e instale.

Tipos de Projetos

Após a teoria, download e instalação chegou a hora de conhecermos a ferramenta e o que se diferencia dentro dela.

	Anotações

Assim que abrimos o VS ele faz as configurações básicas, o que dependendo do processamento da máquina, pode demorar um pouco. Assim que concluídas as configurações, teremos alguns itens em aberto, conforme a imagem abaixo.

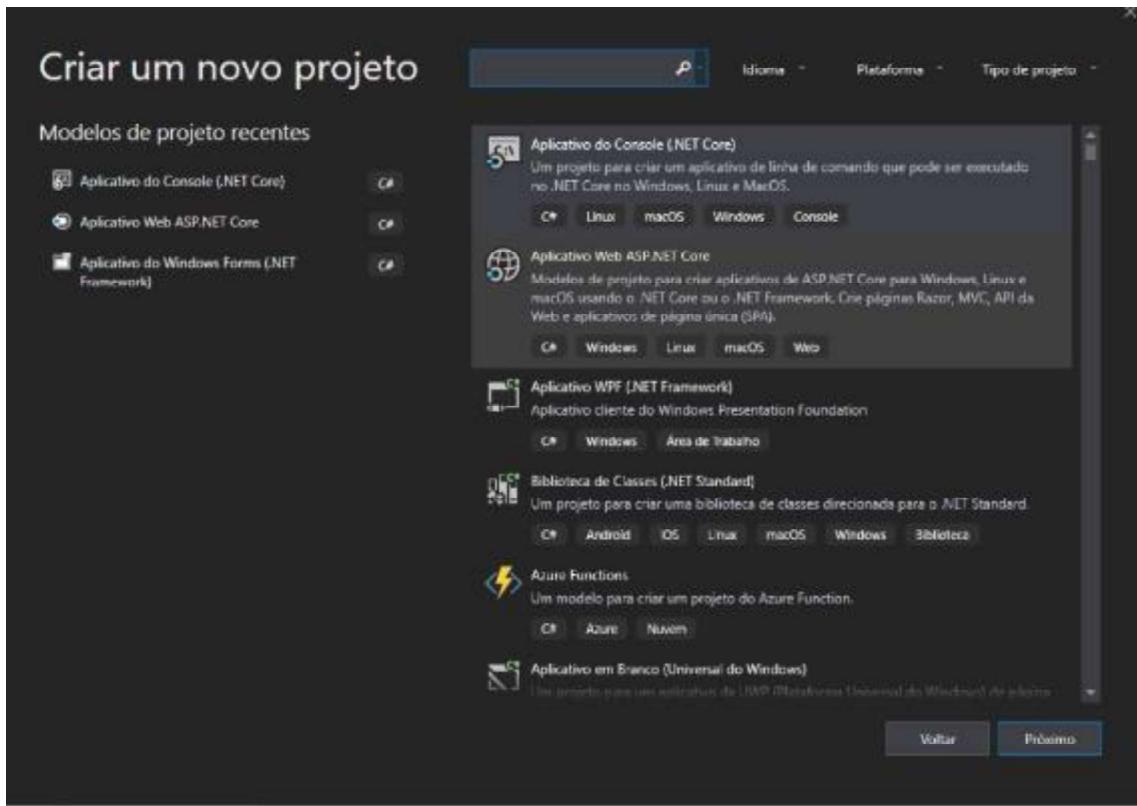


A *Start Page* é a página oficial de entrada do VS seja qual for sua versão. Ela trará as últimas novidades sobre a versão corrente e outras tecnologias disponíveis.

Na própria *Start Page* é possível visualizar o tópico “*Start*” que contém os links “*New Project*”, “*Open Project*” e “*Open from Source Control*”. Esses links são basicamente atalhos para criar um novo projeto, abrir um projeto existente e abrir um projeto usando um controlador/administrador de códigos respectivamente.

Após termos criado projetos, podemos clicar diretamente no arquivo “.sln” ou abrir ele pela opção “*Open Project*”, porém nesse momento procederemos clicando em “*New Project*”. Ao fazer alguma dessas opções teremos as seguintes possibilidades:

Anotações



- *Templates:*

Você vai encontrar diversas opções de Templates. Cada uma das opções tem foco em um *template* de projeto distinto, sendo possível criar um projeto em branco.

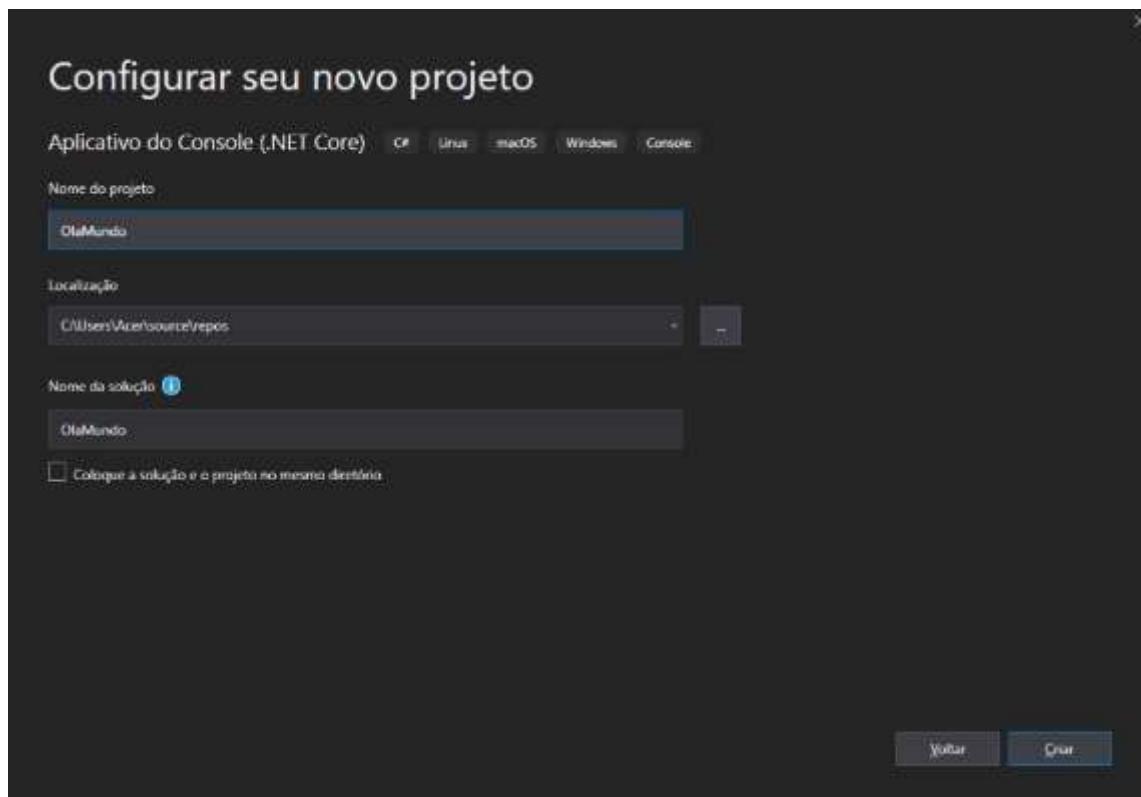
Nesse momento utilizaremos o **Template -> Visual C# -> Console Application**. Esse *template* nos possibilitará a criação de aplicações que rodam no console.

	Anotações

Projeto: Hello World

Chegou a hora de construirmos nosso primeiro programa com o *Microsoft C#.NET.Core*

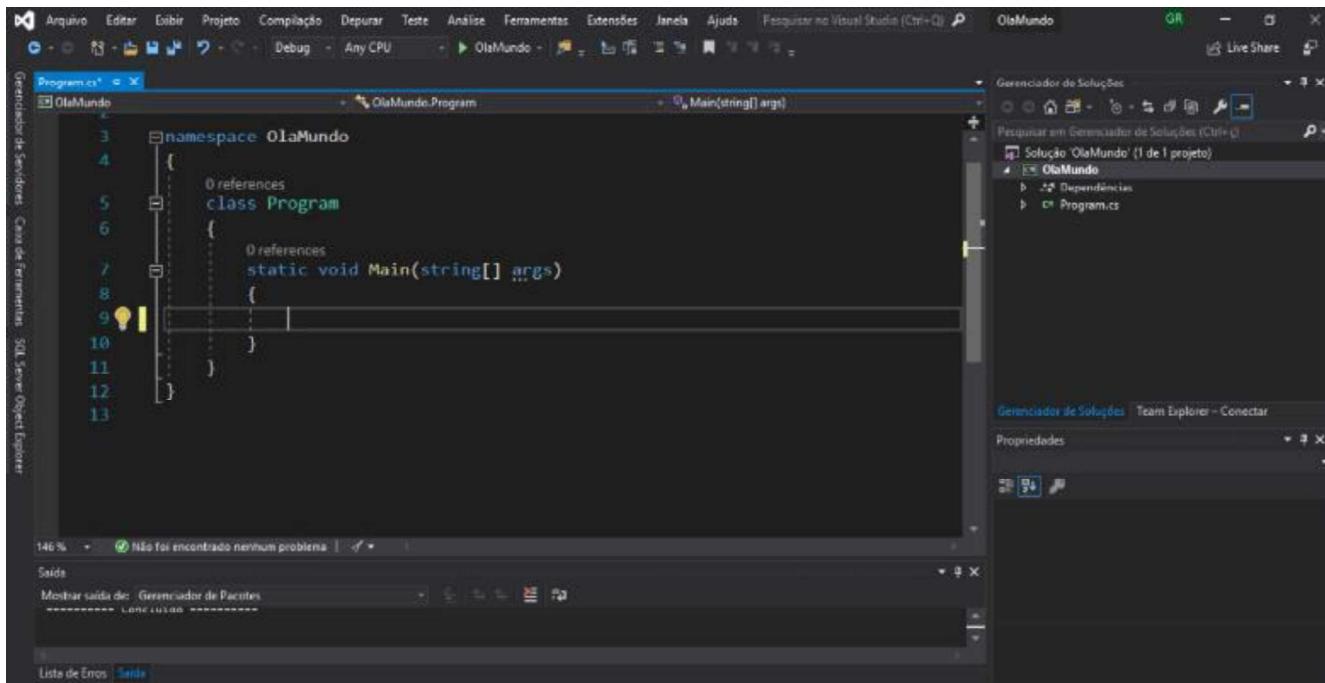
Crie um projeto *Console Application* e nomeie-o de “OlaMundo”, conforme o exemplo abaixo:



Perceba que ele salvará por padrão seu projeto dentro da pasta `C:\Documents\Visual Studio 2019\Projects`, podendo ser alterado para o destino que desejar através do botão de 3 pontinhos localizado ao lado.

Após o VS conseguir construir o projeto básico para você, aparecerão diversas novas informações, conforme a imagem abaixo:

	Anotações



Aba Solution Explorer (Explorador de Soluções)

Na aba à direita chamada “*Solution Explorer*” conseguimos visualizar que foi criado um *Solution* chamado “OlaMundo”, contendo dentro dele um projeto com o Símbolo “C#” com o mesmo nome, ou seja, no nosso conjunto de projetos (solução) temos somente um projeto, que leva o mesmo nome. Posteriormente veremos que podemos ter diversos projetos dentro da mesma “*Solution*”.

Para abrir a aba *Solution* caso tenha fechado, vá para o menu, clique em **View -> Solution Explorer** que ele retornará para onde estava.

Dentro de nosso projeto podemos ver duas ferramentas:

As Dependences – que são bibliotecas do .NET Core e servem para interpretar o nosso código.

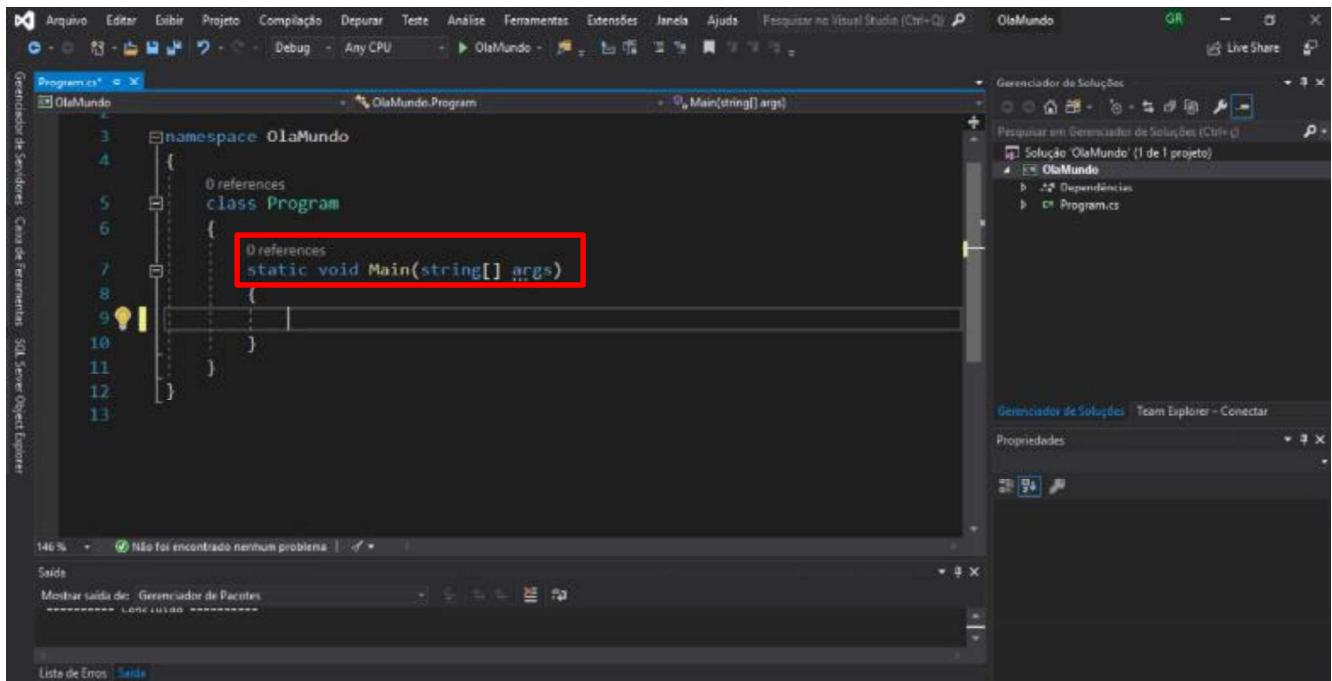
E um arquivo chamado Program.cs. Não entraremos em configurações de projetos nesse momento.

Todos os arquivos de códigos escritos em C# levam a extensão “.cs” ao final. Dando um duplo click nesse arquivo Program.cs teremos a janela Program.cs aberta, demonstrando nosso código.

Program.cs

O arquivo Program.cs é o arquivo que contém o método principal, que será executado assim que executarmos nosso programa.

	Anotações



Nesse momento não nos preocuparemos com a estrutura dele, que será tratada posteriormente. Porém o que deve ser claro é: o que colocarmos dentro desse método, é o que irá acontecer em nosso programa.

Agora que já criamos a estrutura necessária em nosso programa vamos fazer com que ele faça alguma coisa. Adicione as linhas a seguir dentro do método principal:

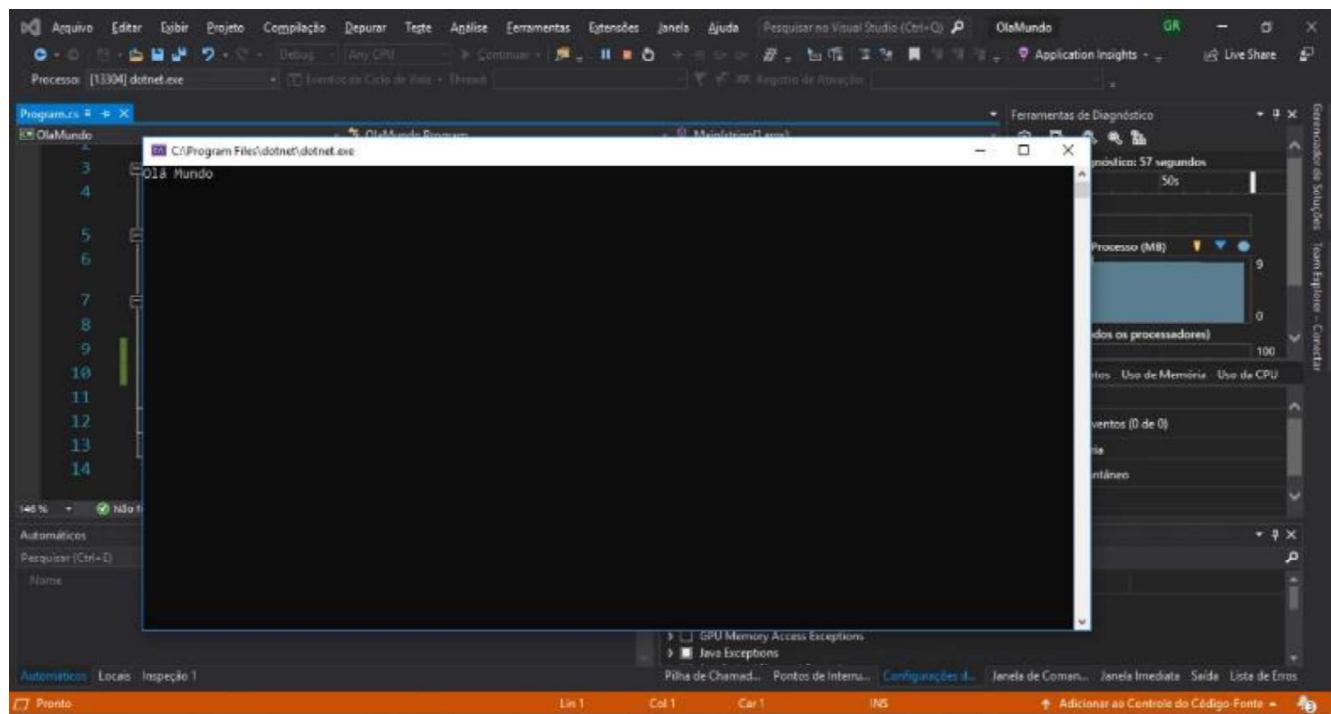
```
Console.WriteLine("Olá Mundo"); //Exibirá a informação no console
```

```
Console.ReadLine(); //Fará que o console espere algum comando antes de desligar
```

Após inserir o código necessário clique em “Start”¹ para ver seu programa funcionar.

¹ Ao clicar em “Start” o projeto é compilado e executado.

Anotações



Você construiu sua primeira aplicação em C#.NET. Para sair pressione Enter para fechar esta tela.

Anotações

Capítulo 03 – Tipos de Dados, Variáveis e Constantes

Tipos de Dados

Assim como os interessados já devem ter ouvido falar, C# é uma linguagem “fortemente tipada”, ou seja , suas variáveis devem ser bem definidas em sua criação.

Em outras palavras, cada variável tem um tipo, podendo ser int (inteiro), char (caractere), boolean (verdadeiro ou falso), entre outros. Para os mais novos na programação isso é comum, para outros isso pode ser bastante diferente, por exemplo programadores de PHP ou Javascript, onde não é necessário dizer explicitamente se uma variável receberá inteiros ou caracteres.

Existem diversos tipos de dados no C#, para entendermos de uma maneira mais simples, separaremos por três tipos, o Tipo por Valor, o Tipo por Referência e o Tipo String, que é um tipo especial do .NET.

Tipo por Valor

Tipos por valor tem como características:

- São alocados diretamente na memória *Stack* (pilha).
 - Não precisam ser inicializados com o operador new.
 - A variável armazena o valor diretamente.
 - A atribuição de uma variável a outra copia o conteúdo, criando efetivamente outra cópia da variável.
 - Cada variável tem a definição de seu tamanho mínimo e máximo.

Exemplo de tipos por Valor:

Tipo	Implementação
byte	Inteiro de 8 bits sem sinal (0 a 255).
sbyte	Inteiro de 8 bits com sinal (-127 a 128).
ushort	Inteiro de 16 bits sem sinal (0 a 65 535).
short	Inteiro de 16 bits com sinal (-32 768 a 32 767).
uint	Inteiro de 32 bits sem sinal (0 a 4 294 967 295).
int	Inteiro de 32 bits com sinal (-2 147 483 648 a 2 147 483 647).
ulong	Inteiro de 64 bits sem sinal (0 a 18 446 744 073 709 551 615).
long	Inteiro de 64 bits com sinal (-9 223 372 036 854 775 808 a 9 223 372 036 854 775 807).
double	Ponto flutuante binário IEEE de 8 bytes ($\pm 5.0 \times 10^{-324}$ a $\pm 1.7 \times 10^{308}$), 15 dígitos decimais de precisão.
float	Ponto flutuante binário IEEE de 4 bytes ($\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$), 7 dígitos decimais de precisão.
decimal	Ponto flutuante decimal de 128 bits. (1.0×10^{-28} a 7.9×10^{28}), 28 dígitos decimais de precisão.
bool	Pode ter os valores true ou false. Não é compatível com inteiro.
char	Um único caractere Unicode de 16 bits. Não é compatível com inteiro.

	Anotações

Tipo por Referência

Tipos por Referência são tipos de dados que não armazenam valores em si, somente contém um ponteiro que direciona para um espaço específico. Mais de uma variável pode indicar um mesmo espaço de memória.

Todos os tipos por Referência são obrigatoriamente inicializados pela palavra “*new*”, que é responsável pelo apontamento.

Exemplo

Tipos de dados por referência em C#: Class, Interface e Delegate.

Falaremos mais sobre Classes no módulo de Orientação a Objetos com C#.

Variações

O tipo String, é uma variação de possibilidades do C#. Ele apesar de ser um tipo por Referência, nos permite utilizá-lo como um tipo por valor. Devemos lembrar que tipos de “texto” são cadeias de caracteres, ou seja, podemos entender um String como um vetor de Char.

Variáveis

Uma variável representa um valor numérico ou cadeia de caracteres ou um objeto de uma classe. O valor que armazena a variável poderá ser alterado, mas o nome permanece o mesmo. Uma variável é um tipo de campo que serve para manipular as informações durante o fluxo do programa.

Exemplo

```
int velocidadeAgora = 75;  
bool vontadeDeEstudar = true;
```

Constantes

Uma constante é um tipo diferente de campo. Seu funcionamento é semelhante ao da criação de variáveis, porém seu valor após compilado o código, jamais será alterado.

Para definir um valor constante basta adicionar a palavra **const** antes da declaração da variável.

Exemplo

```
const int velMaxMargTiete = 90;  
const double pi = 3.14159265358979323846264338327950;
```

Anotações

Capítulo 04 – Operadores

Aritméticos

Quando pensamos em C# os operadores aritméticos podem ser entendidos basicamente como:

Operadores	Função
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto/Módulo

Adição

```
int a, b, c;  
a = 3;  
b = 4;  
c = a + b;  
//Resultado c = 7.
```

Subtração

```
int a, b, c;  
a = 3;  
b = 4;  
c = a - b;  
//Resultado c = -1.
```

Multiplicação

```
int a, b, c;  
a = 3;  
b = 4;  
c = a * b;  
//Resultado c = 12.
```

Divisão

```
int a, b, c;  
a = 11;  
b = 2;  
c = a / b;  
//Resultado c = 5. - Não pega o resto da divisão (no caso de inteiros).
```

	Anotações

Resto/Módulo

```
int a, b, c;  
a = 11;  
b = 2;  
c = a / b;  
//Resultado c = 1. - Só pega o resto da divisão.
```

Comparação

Operadores de Comparação:

Operador	Significado
==	Igualdade
>	Maior
<	Menor
<=	Menor Igual
>=	Maior igual
!=	Diferente

Exemplo

```
int A = 5; int B = 3;  
A == B // Falso  
A > B // Verdadeiro  
A < B // Falso  
A <= B // Falso  
A >= B // Verdadeiro  
A != B // Verdadeiro
```

Lógicos

Operadores lógicos:

Operador	Significado
&&	E
	OU

Exemplo && (E)

```
int num1 = 1, num2 = 3;  
bool PrimeiroTeste = num1 > 3 && num2 < 10  
//PrimeiroTeste == false (falso)
```

Para que um teste lógico seja verdadeiro perante dois testes, ambos devem ser verdadeiros.

	Anotações

Exemplo // (OU)

```
int num1 = 1, num2 = 3;  
bool SegundoTeste = num1 > 3 || num2 < 10  
//SegundoTeste == true (verdadeiro)
```

Atribuição

Operadores de Atribuição:

Operador	Significado
=	Atribuição Simples
+=	Atribuição Aditiva
-=	Atribuição Subtrativa
*=	Atribuição Multiplicativa
/=	Atribuição de divisão
%=	Atribuição Modular
++	Adição de 1 unidade
--	Subtração de 1 unidade

Simples

```
string simp = "simples";
```

Aditiva

```
int a = 3; a += 1; //a = 4
```

Subtrativa

```
int a = 3; a -= 1; //a = 2
```

Multiplicativa

```
int a = 3; a *= 4; //a = 12
```

Divisão

```
int a = 8; a /= 2; //a = 4
```

Modular

```
int a = 11; a %= 2; //a = 1
```

Adição de uma Unidade

```
int a = 11; a++; //a=12
```

Subtração de uma Unidade

```
int a = 11; a--; //a=10
```

	Anotações

Concatenação

Operador de concatenação

Operador	Significado
+	Concatena caracteres e cadeias de caracteres

O operador de concatenação tem como objetivo juntar, concatenar, dois ou mais caracteres.

Exemplo

```
string algumaVariavel = "olá, meu nome é ";
algumaVariavel = algumaVariavel + "Thiago";
// ou utilizando de uma maneira mais simples:
string algumaVariavel = "olá, meu nome é ";
algumaVariavel += "Thiago";
```

Ambos exemplos resultarão na mesma frase.

Operadores Ternários

Já vimos anteriormente a existência de operadores lógicos, aritméticos, de comparação, de atribuição e concatenação. Todos esses operadores funcionam semelhantemente a outras linguagens, porém os Operadores Ternários são restritos a algumas linguagens de programação, felizmente o C# é uma delas.

Os operadores ternários são formados por três **Operandos** - por isso o nome, e são separados por dois caracteres em especial, o "?" e ":" conforme o exemplo abaixo:

```
condição ? valor_se_verdadeiro : valor_se_falso
```

Dessa maneira, fazemos uma pergunta da qual retorne um valor Verdadeiro ou Falso. Caso o valor seja verdadeiro, retornamos o valor que estará na posição "valor_se_verdadeiro" - posicionado logo após o "?". Caso valor seja falso, retornamos o "valor_se_falso" - posicionado após o ":".

Os Operadores Ternários têm como objetivo resolver de uma maneira mais simples uma expressão lógica.

Exemplo

```
int valor1 = 10;           int valor2 = 15;
bool resultado = valor1 == valor2 ? true : false;
//resultado == false

string nome1 = "Lucas"; int string = "Henrique";
bool resultado = nome1 != nome2 ? true : false;
//resultado == true
```

Anotações

Exercícios

- 1) Preencha as lacunas com **Int32 ; String ; Boolean ; Double; Char.**

()	"Blumenau"
()	false
()	1992
()	'1'
()	"1"
()	2790.32
()	555
()	true

	Anotações

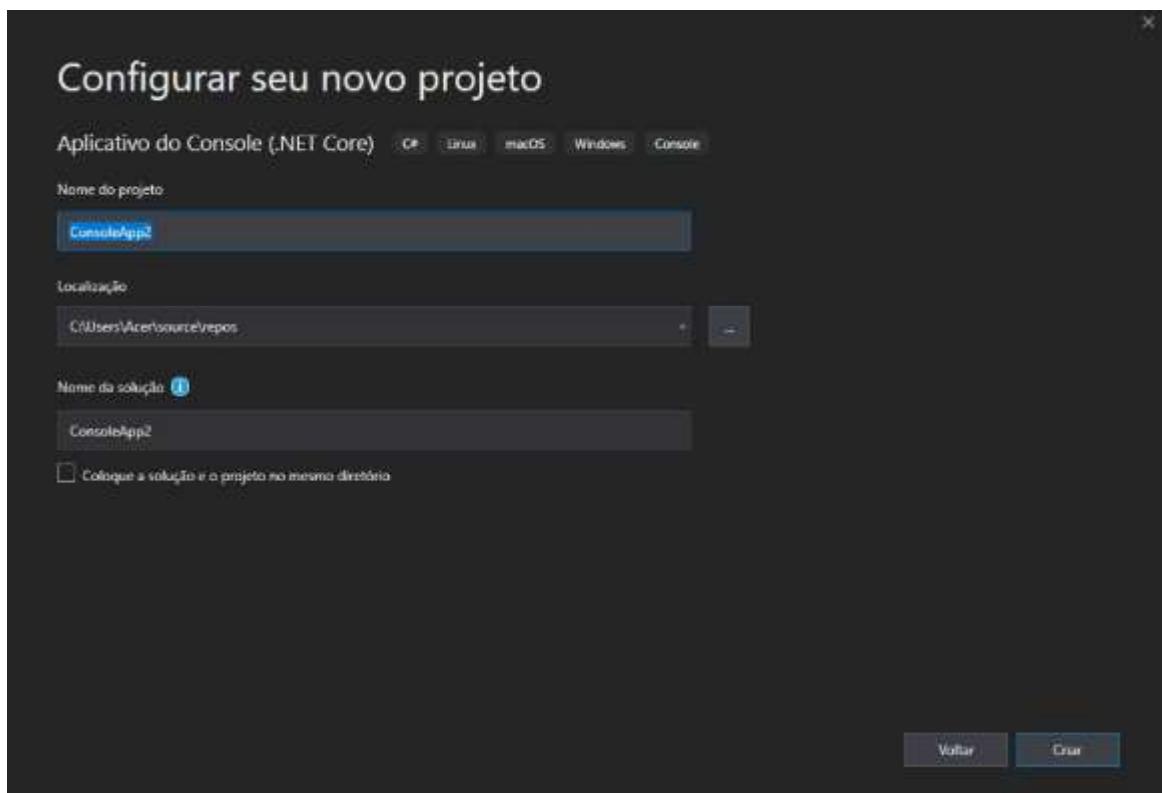
Capítulo 05 – Console Application

Estrutura Sequencial

Até esse momento, vimos como realizar operações simples, quais os operadores, tipos de dados e como funciona o *.NET Core* e o *Visual Studio*. Agora vamos entender um pouco melhor como é a estrutura sequencial de um projeto *Console Application*.

Projeto

Para entendermos melhor como é estruturado esse projeto, vamos criar um novo projeto *Console Application* chamado “EntendendoEstrutura” conforme o exemplo abaixo.



Lembre-se da pasta em que salvou o projeto.

Método principal

Quando criamos nosso projeto, como já vimos anteriormente, foi criado um arquivo “.cs” chamado *Program.cs*. Esse arquivo é o arquivo principal do programa, é ele quem diz o que acontecerá quando executarmos o código.

Ele diz isso por um simples motivo - é nele que consta o método Principal (*Main*). Esse método que é o responsável por fazer seu código acontecer. Experimente criar outro arquivo e colocar o método, ou até renomear o arquivo existente. O código que irá acontecer é o código que está dentro do método principal.

	Anotações

Esse método é composto pela seguinte sintaxe:

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

Cada palavra dessa assinatura tem um significado. Sendo eles:

Static

Define que o método pertence à Classe e não ao Objeto (instância dessa classe) e pode ser acessado diretamente, sem ser necessário criar um objeto para isso.

Void

Significa “vazio”, ou seja, o retorno desse método será vazio. Podendo ser entendido como um **procedimento** - como exemplificado em lógica de programação.

Main

É o nome do método, nesse caso chamado de Main (Principal) o que não faz diferença real em outros momentos, porém para um projeto ConsoleApplication é necessário que o nome continue como Main e que receba como parâmetros um array de string - o que veremos melhor no módulo pertinente a Variáveis Indexadas.

Linhas de Comentário

Comentários em C# podem ser feitos de diversas maneiras, sendo elas:

Caracteres	Exemplo	Explicação
//	//Isso é um comentário Isso não.	Comentários de linha única. Nada a não ser o que vem à direita dos caracteres “//” será comentado.
/* texto */	/* Sobre o código abaixo Temos X e Y Dessa maneira.... */	Comentários de múltiplas linhas. Tudo entre os caracteres “/*” e “*/” será comentado e ignorado pelo compilador.
/// <summary> /// ///</summary></summary>	/// <summary> /// Esta classe é importante... /// </summary> public class MyClass{}	Comentários de Documentação. Tem como objetivo descrever um método, classe ou namespace. Usado para documentar códigos.

Input e Output

Assim como em lógica vimos comandos que nos fornecem informações, seja escrevendo na tela ou em um arquivo, seja pedindo para que passemos algum parâmetro para o programa.

	Anotações

Neste primeiro contato com C#, utilizaremos do Namespace “Console” - que fornece ferramentas de IO (input e output), para conseguir interagir com nosso programa quando ele é feito em ConsoleApplication.

Comandos de Saída (Output)

Quando pensamos em comandos de saída, lembramos em um comando que nos dirá alguma coisa. Sendo ele responsável por exibir a informação que estamos querendo que seja exibida. Para realizar tal tarefa em C# utilizaremos o método “WriteLine()” do namespace “Console”, conforme já visto anteriormente no segundo módulo.

Exemplo

```
namespace OlaMundo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Olá Mundo Novamente");//Exibira a informação
        }
    }
}
```

Saída
Mostrar saída de: Depuração
"dotnet.exe" (CoreCLR: C:\host): Carregado - C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App\v2.1.11\System.Runtime.Extensions.dll
O programa "[13304] dotnet.exe" foi Fechado com o código -1 (0xffffffff).

Assim como quando executamos nosso projeto, o método exibiu na tela um texto exatamente como colocamos dentro desse método. Porém ele foi tão rápido que não conseguimos ver acontecer.

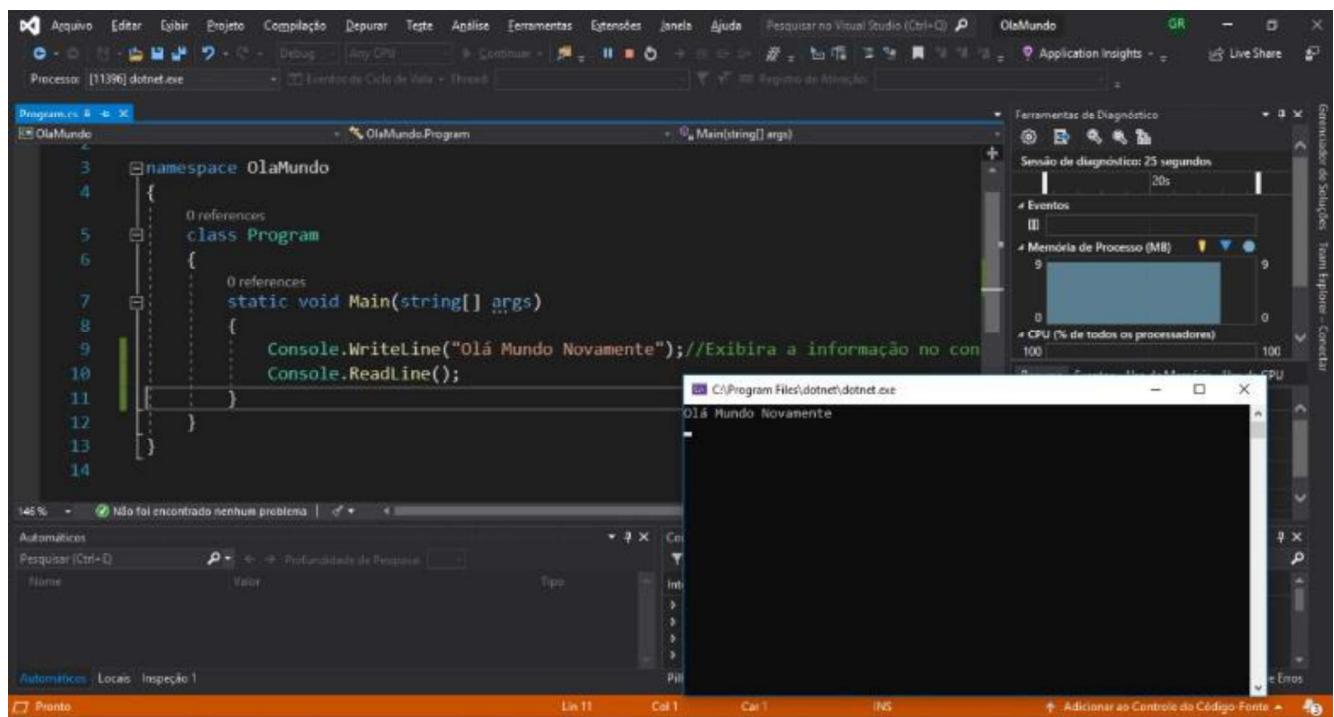
Isso aconteceu, pois, o programa não está esperando nenhuma interação com o usuário para terminar aquele bloco de código. Simplesmente escreveu no console o que você pediu e terminou o programa, exatamente como seu código sugere.

Comandos de Entrada (Input)

Seja para fazer com que o código espere um parâmetro para terminar de executar o programa, ou seja para realmente passar como parâmetro algum valor ao programa, devemos utilizar um método de Input, que fará com que o programa espere receber algum valor, conforme o exemplo utilizando o comando “ReadLine()”, que lê tudo que você digitar e só termina quando você pressiona a tecla ENTER.

	Anotações

Exemplo



Repare que dessa vez ele faz com que o programa aguarde até pressionarmos a tecla ENTER.

Os comandos de leitura do Namespace Console, sempre entendem o que foi digitado como caracteres, ou seja, texto. Podemos atribui-los diretamente às variáveis do tipo string ou convertermos o valor para o tipo que desejarmos, conforme os exemplos abaixo.

Exemplo

```
//string  
Console.WriteLine("Digite algum texto qualquer.");  
string texto = Console.ReadLine();  
Console.WriteLine("Texto digitado: " + texto);  
//int  
Console.WriteLine("Digite algum número inteiro.");  
string textoNumerico = Console.ReadLine();  
int numero = Convert.ToInt32(textoNumerico);  
Console.WriteLine("Número Digitado: " + numero);  
Console.ReadKey();
```

Todos os tipos de dados do C# permitem conversão para outros tipos. Lembrando que devem conter o conteúdo correto. Não será possível converter um texto “Nome de Alguém” em um tipo int, por exemplo.

	Anotações

Criando Métodos

Toda linguagem de Programação que se preze permite a criação de métodos e funções pelo programador.

Com o c#, esta tarefa é extremamente fácil, graças aos recursos de criação automática de métodos.

Criando Métodos com o Resolve

Vamos criar um método que vai exibir uma mensagem ao usuário.

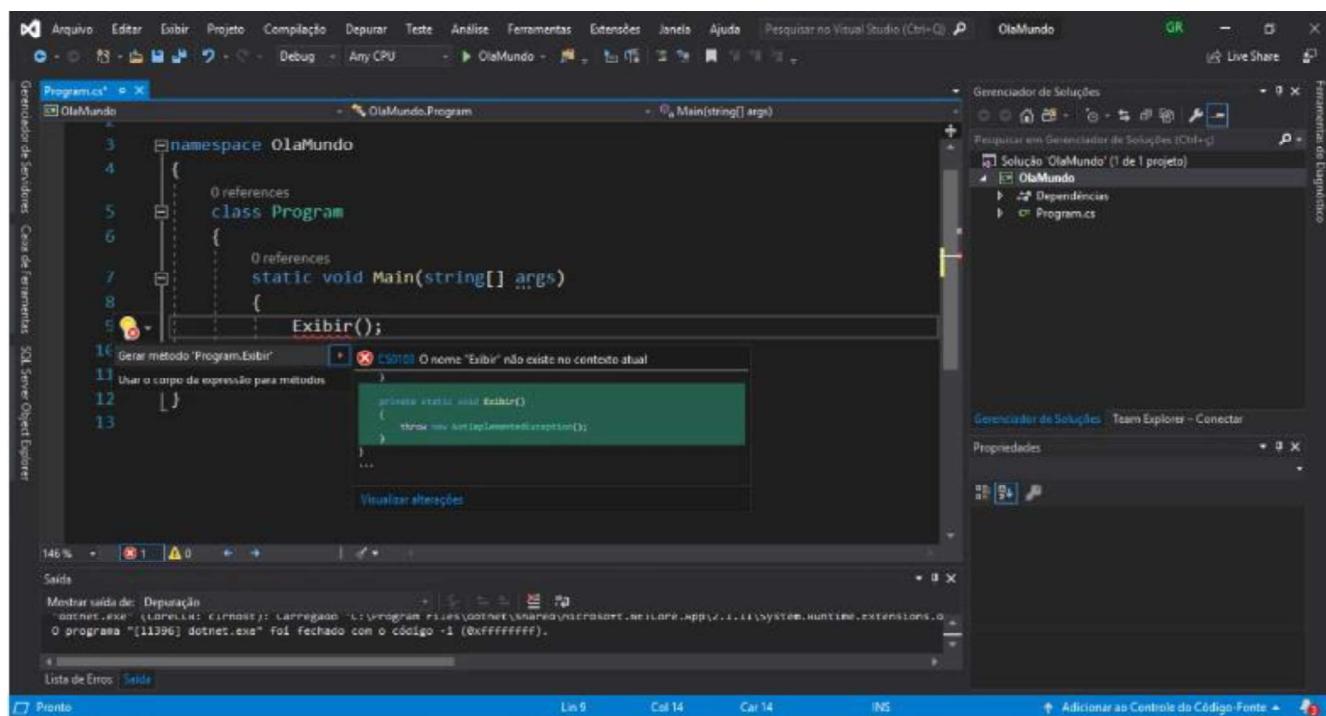
Para isso, no seu método MAIN, digite:

```
Exibir();
```

Note que esta chamada ficará sublinhada em vermelho, pois este método ainda não foi criado.

Clique, com o botão direito do mouse, sobre a chamada do método.

Note que as seguintes opções serão exibidas:



Clique em “Generate” e em “Method Stub”.

Quando seu método é criado, a estrutura abaixo é mostrada:

```
private static void Exibir()
{
    throw new NotImplementedException();
}
```

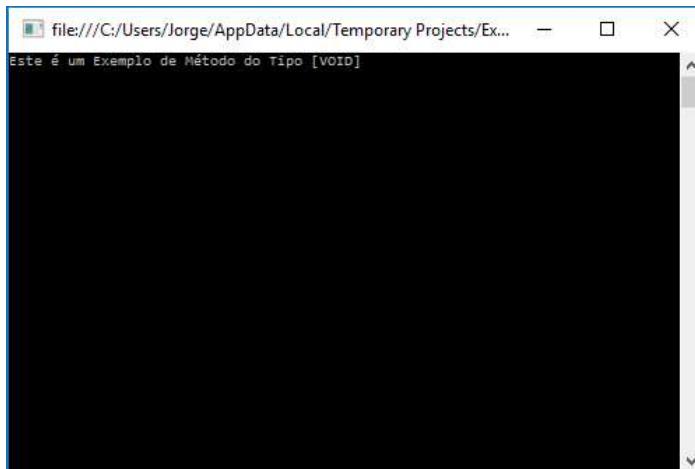
Como você pode observar, esta sequência de comandos simplesmente lançará uma exceção de método não implementado, pois o Visual Studio não consegue imaginar o que você quer que seja feito neste método.

		Anotações

Vamos preencher com os comandos necessários para que seja exibida uma mensagem ao usuário:

```
private static void Exibir()
{
    Console.WriteLine("Este é um Exemplo de Método do Tipo [VOID]");
    Console.ReadLine();
}
```

Ao rodarmos nossa aplicação, teremos o seguinte resultado:

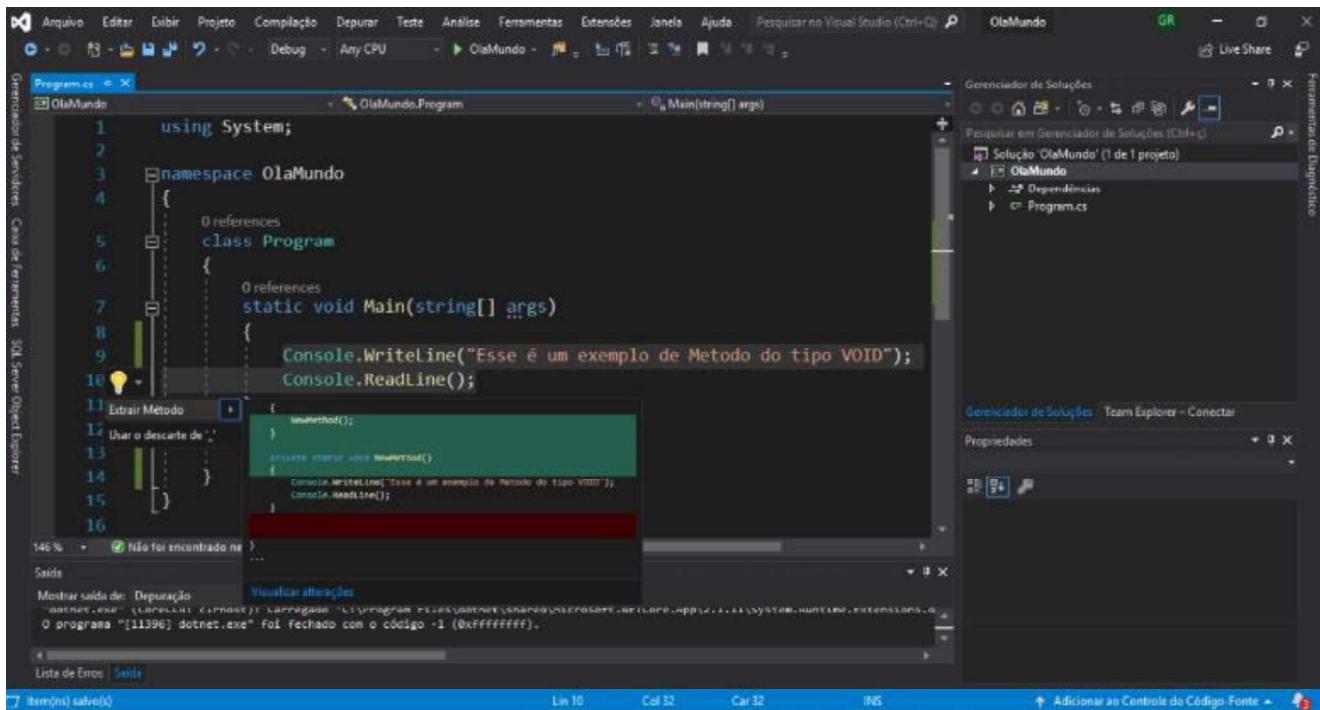


Uma outra maneira de criar métodos é digitar todas as linhas de comando no método principal, depois selecionar as linhas digitadas. Em seguida, clique com o botão direito do mouse sobre a opção “Refactor – Extract Method”. Feito isso, será exibida uma tela para que você identifique o seu método, ou seja, dê um nome para ele, e pronto. Seu método será automaticamente criado, como pode ser visto nas imagens abaixo:

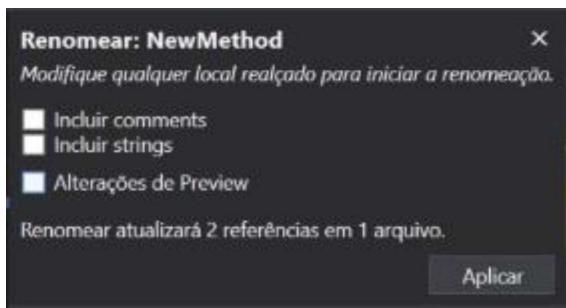
```
//Digitação dos Comandos para Extração do Método
//=====
Console.WriteLine("Este Método foi Gerado utilizando o Refactor");
Console.ReadLine();
```

Digitação dos comandos do método

	Anotações



Seleção do método de extração.



Tela para identificação do método

```
private static void ExibeMetodo()
{
    //Digitação dos Comandos para Extração do Método
    //=====
    Console.WriteLine("Este Método foi Gerado utilizando o Refactor");
    Console.ReadLine();
}
```

Método criado

Há ainda uma terceira opção, que é a criação manual de métodos.

Nesta forma você precisa digitar manualmente as linhas de comando, assim como a assinatura do método.

	Anotações

No exemplo abaixo, está a criação de um método que retorna a média entre 4 notas:

```
private double CalculaMedia(int Nota1, int Nota2, int Nota3, int Nota4)
{
    return (Nota1 + Nota2 + Nota3 + Nota4) / 4;
}
```

Anotações

Exercícios

- 1) Complete:
 - a. Método _____ é o método em que tudo ocorrerá dentro de um programa feito com o template de ConsoleApplication.
 - b. A palavra _____ define que o método não pertence a objetos daquela determinada classe, e sim à própria classe.
 - c. A palavra _____ significa que não haverá retorno daquele método, ele executa e retorna vazio.
 - d. Console é uma _____ do Namespace System.
 - e. O método ReadLine() da classe System.Console é um método de _____.
 - f. O método Write() da classe System.Console é um método de _____.
- 2) Porque e quando devemos/podemos comentar um código?
- 3) Quando desejamos exibir informações ao usuário do sistema, utilizamos de um comando de **Input** ou **Output**? Qual é o comando referente que utilizamos com a classe Console do namespace System? Exemplifique.
- 4) Crie uma aplicação que leia do teclado um peso e uma altura, calcule o IMC e exiba na tela o resultado.

	Anotações

Capítulo 06 – Funções do .NET Core

Funções de Manipulação de Texto

Funções de manipulação de texto tem como objetivo preparar o texto para uma determinada ação ou exibição. Dentro do *.NET Core* existem inúmeras funções de manipulação de texto. Nesse capítulo veremos quais são as mais utilizadas no dia-a-dia do programador.

Replace (string-Antigo, string-Novo)

A função Replace da classe *string* é provavelmente a função de texto mais utilizada na programação. Tem como objetivo substituir um determinado caractere ou texto contido dentro de uma variável *string* por outro texto ou caractere.

Exemplo

```
string textoAntigo = "Esse é um texto ANTIGO";
string textoNovo = textoAntigo.Replace("ANTIGO", "NOVO");
Console.WriteLine(textoNovo);
Console.ReadKey();
```

Substring (int-Posição-Inicial, int-Quantidade-De-Caracteres)

A Função Substring tem como objetivo conseguir separar somente uma determinada parte do texto contido em outra variável. Para essa função ocorrer da maneira certa, devemos passar o index (posição do caractere) que gostaríamos de começar a copiar e por quantos caracteres. **Lembre-se** que no C# os Indexes começam em 0 e não em 1.

Exemplo

```
string textoAntigo = "0123456";
string textoNovo = textoAntigo.Substring(1, 3);
Console.WriteLine(textoNovo);
Console.ReadKey();
```

ToLower (string)

A função ToLower, assim como o nome já diz, tem como objetivo diminuir uma string, isso significa tornar as letras que estão em **MAIÚSCULAS** para letras **minúsculas**.

Exemplo

```
string textoAntigo = "ESTOU APRENDENDO C#";
string textoNovo = textoAntigo.ToLower().Replace("maiúsculas", "minusculas");
Console.WriteLine(textoNovo);
Console.ReadKey();
```

Anotações

Repare que nesse exemplo utilizamos um método seguido de outro. O que é totalmente **válido!** Entretanto tome cuidado com a sequência em que coloca sua expressão, caso tente substituir letras minúsculas antes de pedir para que elas fiquem minúsculas, nada acontecerá, pois os caracteres não foram encontrados.

ToUpper (string)

A função `ToUpper`, faz exatamente o contrário da função `ToLower (string)`. Ela recebe uma `string` como parâmetro e faz com que todas as letras **minúsculas** dela sejam transformadas em **MAIÚSCULAS**.

Exemplo

```
string textoAntigo = "um texto escrito em letras minúsculas";
string textoNovo = textoAntigo.Replace("minúsculas",
"MAIÚSCULAS").ToUpper();
Console.WriteLine(textoNovo);
Console.ReadKey();
```

Funções Matemáticas e Trigonométricas

Funções matemáticas foram criadas com objetivo de diminuir a quantidade de cálculos necessários em um código, otimizando puramente sua escrita e possivelmente sua performance. Assim como o `.NET Core` contém inúmeras funções de manipulação de texto, as matemáticas não seriam diferentes. Existe dentro do `.NET Core` linguagens mais focadas em resolver problemas matemáticos com uma performance melhor do que o C#, porém para o convencional o C# é mais do que o suficiente.

Listaremos abaixo as funções mais utilizadas e como utiliza-las.

`Math.Pi`

Essa não é exatamente uma função e sim uma propriedade constante da classe `Math`, porém devido a sua utilidade acredito ser útil conhecermos.

Exemplo

```
double pi = Math.PI;
Console.WriteLine(pi);
Console.ReadKey();
```

`Math.Sin(x) / Math.Cos(x) / Math.Tan(x)`

Essas funções são para trabalhar com Senos, Cossenos e Tangentes respectivamente, de um determinado número.

Exemplo

```
double seno = Math.Sin(2.20);
double coseno = Math.Cos(2.20);
```

Anotações

```
double tangente = Math.Tan(2.20);  
Console.WriteLine(String.Format("{0} - {1} - {2}", seno, coseno, tangente));  
Console.ReadKey();
```

Math.Pow (x, y)

Essa função tem como objetivo retornar um número elevado a potência do outro número.

Exemplo

```
double potencia = Math.Pow(2, 8);  
Console.WriteLine(potencia);  
Console.ReadKey();
```

Math.Round (X, Y)

Essa função arredonda o número X para que ele fique com Y casas decimais. Quando o valor de Y não for passado, considera-se 0 casas decimais, ou seja, um valor inteiro.

Exemplo

```
double arredondado = Math.Round(50.1234543, 2 );  
Console.WriteLine(arredondado);  
Console.ReadKey();
```

Math.Sqrt (X)

Função responsável por retornar a raiz quadrada de um número.

Exemplo

```
double raiz = Math.Sqrt(100);  
Console.WriteLine(arredondado);  
Console.ReadKey();
```

	Anotações

Exercícios

- 1) Crie um programa que realize as seguintes funções:
 - a. Receba três parâmetros, um Nome (nome e sobrenome), uma Idade e um Telefone (somente números).
 - b. Faça uma rotina para padronizar o formato do telefone (xx)99999-9999 e exiba na tela.
 - c. Faça uma rotina que calcule a raiz quadrada da idade do usuário e retorne para o método principal um double, que será exibido na tela.
 - d. Faça uma rotina que troque os espaços entre os nomes do usuário por ";" e exiba na tela.

	Anotações

Capítulo 07 – Comandos Condicionais

Estruturas de Decisão

Qualquer linguagem de programação necessita de estruturas de decisão. Isso significa que uma das bases da programação é a comparação. A programação em C# não seria diferente e para isso existem os comandos if (se) e else (senão) e switch (escolha).

Comandos If / Else / Else if

If - Else^{2,3}

A sintaxe dos comandos if e else estão diretamente ligadas ao uso de operadores.

```
if ( expressão lógica )
{
    //Código se expressão lógica for verdadeira
}
else
{
    //Código se falsa
}
```

Exemplo if (bool) {} else {}

```
int a, b;  
  
a = 1; b = 2;  
  
if (a == b)  
{  
    Console.WriteLine("A e B são iguais");  
}  
  
else  
{
```

² Quando iniciamos a escrita do comando `if` no Visual Studio 2015 Community, existe um “Snippet” que nos permite através da tecla TAB completar a estrutura do comando inteiro. Digite `if` e posterior a isso duas vezes a tecla TAB.

³ Quando nossos comandos **if** e **else** só tem uma linha, não é obrigatória a criação das chaves.

Anotações

```
        Console.WriteLine("A e B são diferentes");  
    }  
}
```

Lembro-vos que podemos fazer a comparação de qualquer tipo, desde que com o mesmo tipo, String com String, Int com Int e assim por diante.

Else If

Outra possibilidade com o comando if (se) é o encadeamento de decisões. Existem momentos em que nossa opção não é somente verdadeiro ou falso, conforme no exemplo abaixo.

Exemplo if (bool) {} else if () {}

```
Console.WriteLine("Digite o dia da semana em que estamos");  
string diaSemana = Console.ReadLine().ToUpper();  
if (diaSemana == "SEGUNDA-FEIRA")  
{  
    Console.WriteLine("Odeio esse dia!");  
}  
else if (diaSemana == "TERÇA-FEIRA")  
{  
    Console.WriteLine("Falta muito pro fds?");  
}  
else if (diaSemana == "QUARTA-FEIRA")  
{  
    Console.WriteLine("Já estou me arrastando");  
}  
else if (diaSemana == "QUINTA-FEIRA")  
{  
    Console.WriteLine("Chega natal e não chega sexta");  
}  
else if (diaSemana == "SEXTA-FEIRA")  
{  
    Console.WriteLine("Sextou, bora tomar uma");  
}  
else if (diaSemana == "SABADO")  
{  
    Console.WriteLine("Almoçar na casa da vó");  
}
```


Anotações

```

}

else if (diaSemana == "DOMINGO")
{
    Console.WriteLine("Dormir o dia todo");
}

else
{
    Console.WriteLine("Você deve preencher somente os dias da semana e por extenso,
por exemplo: Quarta-Feira");

}

Console.Read();

```

Comando Switch⁴

No exemplo anterior vimos o quanto grande pode tornar-se um código para fazer uma validação consideravelmente simples, como dias da semana. O comando switch veio para minimizar a necessidade de toda essa estrutura, tornando-se mais legível e sendo considerada a melhor prática para opções com valores não-booleanos (diferentes de verdadeiro ou falso).

Exemplo

```

Console.WriteLine("Digite o dia da semana em que estamos");
string diaSemana = Console.ReadLine().ToUpper();

switch (diaSemana)
{
    case "SEGUNDA-FEIRA":
        Console.WriteLine("1");
        break;
    case "TERÇA-FEIRA":
        Console.WriteLine("2");
        break;
    case "QUARTA-FEIRA":
        Console.WriteLine("3");
        break;
}

```

⁴ Quando iniciamos a escrita do comando **switch** no Visual Studio 2015 Community, Existe um “Snippet” que nos permite através da tecla TAB completar a estrutura do comando inteiro. Digite switch e posterior a isso duas vezes a tecla TAB.

	Anotações

```

case "QUINTA-FEIRA":
    Console.WriteLine("4");
    break;

case "SEXTA-FEIRA":
    Console.WriteLine("5");
    break;

case "SÁBADO":
    Console.WriteLine("6");
    break;

case "DOMINGO":
    Console.WriteLine("7");
    break;

default:
    Console.WriteLine("Tá errado, preencha direito isso aí");
    break;
}
Console.Read();

```

Como observamos, o switch traz uma estrutura muito mais legível e compreensível. Sua sintaxe permite que criemos um valor default (ou não), cabendo a ele tratar as opções que não forem as conhecidas.

Exercícios

- 1) Crie um novo projeto ConsoleApplication chamado ExerciciosCondicionais. Em sua classe Program.cs desenvolva um algoritmo que receba a idade do usuário e:
 - a. Se a idade for maior ou igual a 13 e menor que 19 escreva “Adolescente”.
 - b. Se a idade for maior ou igual a 19 e menor ou igual a 60 escreva “Adulto”.
 - c. Se a idade for maior que 60 escreva “Idoso”.
 - d. Caso contrário escreva “Criança”.
- 2) No mesmo projeto, crie um novo procedimento em que pergunte ao usuário do sistema qual time ganhou a copa do mundo de futebol de 2014. Dê a ele 4 opções, dentro dessas uma deve ser a verdadeira. Caso o usuário acerte, escreva “Acertou” caso contrário “Errou”.
- 3) Utilize o mesmo projeto de IMC dos módulos anteriores e exiba o a classificação de acordo com o resultado (ex: abaixo do peso, sobrepeso, obesidade, etc) primeiro utilizando a estrutura if; depois troque para switch.

	Anotações

Capítulo 08 – Estruturas / Laços de Repetição

Comando While / Do While

While⁵

O comando while (enquanto) é o mais simples dos laços de repetição quando utilizamos C#. Podemos entendê-lo como um repetidor de informação devido a uma expressão lógica.

Exemplo

```
int n = 1;
while (n <= 10)
{
    Console.WriteLine(String.Format("Valor corrente de n é: {0}", n));
    n++;
}
Console.Read();
```

Do While

A única diferença entre o *While* e o *Do While* é a **primeira execução**. No caso do *While*, somente será executado o comando caso a validação desde a primeira tentativa (execução) seja verdadeira. No *Do While*, o código é executado a primeira vez e posteriormente é validado.

Exemplo

```
int n = 1;
do
{
    Console.WriteLine("Valor de n é: " + n);
    n++;
}
while (n >= 10);
Console.Read();
```

⁵ Quando iniciamos a escrita do comando **while** no Visual Studio 2019 Community, existe um “Snippet” que nos permite através da tecla TAB completar a estrutura do comando inteiro. Digite **while** e posterior a isso duas vezes a tecla TAB

	Anotações

Comando For⁶

O comando For tem as mesmas utilidades de um comando While, porém o incremento da variável cuja expressão lógica está vinculada é feito dentro do próprio comando.

Exemplo

```
int numero;

Console.WriteLine("Até qual número devo escrever de 1 em 1?");
numero = Convert.ToInt32(Console.ReadLine());
for (int i = 0; i < numero; i++)
{
    Console.WriteLine(i+1);
}
Console.Read();
```

	Anotações

Exemplo avançado

No exemplo abaixo vamos relacionar diversos ensinamentos até aqui, colocando em prática e aprendendo como lidar com situações mais complexas. Nesse exemplo relacionaremos os comandos While, For, comandos de input e output, estruturas de decisão e funções predefinidas.

O nosso objetivo é fazer um programa que leia um número e peça para o computador contar e escrever de 1 até o número digitado. Além disso o programa deverá perguntar se queremos fazer novamente a contagem, porém com um valor novo. Segue o código abaixo para análise.

```
int numero;
bool satisfeito = false;
while (satisfeito == false)
{
    Console.WriteLine("Até qual número devo escrever de 1 em 1?");
    numero = Convert.ToInt32(Console.ReadLine());

    for (int i = 0; i < numero; i++)
    {
        Console.WriteLine(i + 1);
    }
    Console.WriteLine("Satisfeito? S - para Sim / N - para Não");

    if (Console.ReadLine().Equals("S"))
    {
        satisfeito = true;
    }
}
Console.Read();
```

⁶ Quando iniciamos a escrita do comando **for** no Visual Studio 2015 Community, existe um “Snippet” que nos permite através da tecla TAB completar a estrutura do comando inteiro. Digite **for** e posterior a isso duas vezes a tecla TAB

	Anotações

Comando Foreach⁷

Em lógica de programação vimos que existem variáveis que armazenam informações em forma de vetores e matrizes. O Foreach (para cada um) é o comando mais utilizado para percorrer esse vetor/matriz valor a valor.

Quando ele faz a leitura ele realiza a conversão do vetor de maneira automática a cada passagem, fazendo com que a leitura seja mais demorada em relação ao for.

OBS: A sintaxe de vetores e matrizes serão revistas no próximo módulo.

Exemplo

```
int[] arrayDeInteiros = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };  
foreach (int inteiro in arrayDeInteiros)  
{  
    Console.WriteLine(inteiro);  
}  
Console.Read();
```

Exercícios

- 1) Crie um novo projeto ConsoleApplication chamado “ExerciciosLacosRepeticao”. Em sua classe Program.cs desenvolva um algoritmo que receba um número e escreva-o decrescendo de 1 em 1 até o valor chegar a zero.
- 2) No mesmo projeto crie um novo método que receba do usuário números e enquanto não for passado um número PAR o programa solicite novamente. Passado o número par, calcule a metade desse número e chame-o de X. Exiba na tela X vezes “Repetição” concatenado com o número atual das X vezes.
- 3) No mesmo projeto, crie um novo procedimento em que pergunte ao usuário do sistema qual time ganhou a última copa do mundo de futebol. Dê a ele 4 opções, dentro dessas uma deve ser a verdadeira. Caso o usuário acerte, escreva “Acertou” caso contrário, “Errou”. Não pare o programa até que ele acerte a resposta.

⁷ Quando iniciamos a escrita do comando **foreach** no Visual Studio 2019, existe um “Snippet” que nos permite através da tecla TAB completar a estrutura do comando inteiro. Digite **foreach** e posterior a isso duas vezes a tecla TAB

	Anotações

Capítulo 09 – Variáveis Indexadas

Variáveis Indexadas Unidimensionais (Vetores/Arrays)

Array

O C# assim como a maioria das linguagens de programação atuais suporta indexação de valores de mesmo tipo em uma estrutura única, o Vetor ou também chamado de Array.

Vetores em C# são definidos pela classe ou tipo e pelos caracteres “[]”, conforme abaixo.

Exemplo

```
int[] arrayDeInteiros = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
string[] arrayDeTextos = new string[] {"Segunda", "Terça", "Quarta"};
```

Dessa maneira conseguimos criar vetores unidimensionais de tamanho fixo. Onde não podemos alterar a quantidade de valores que existem dentro do vetor. Um vetor de 10 posições, pode ter no máximo 10 valores.

Outra maneira de desenvolvermos a mesma ideia é estipular o tamanho do vetor na sua criação e posteriormente adicionar valores a cada posição, conforme o exemplo:

```
int[] arrayDeInteiros = new int[5];
arrayDeInteiros[0] = 1;
arrayDeInteiros[1] = 2;
arrayDeInteiros[2] = 3;
arrayDeInteiros[3] = 4;
arrayDeInteiros[4] = 5;

for (int i = 0; i < arrayDeInteiros.Length; i++)
{
    Console.WriteLine(arrayDeInteiros[i]);
}
Console.Read();
```

Anotações

Ou para textos:

```
string[] arrayTextos = new string[5];
arrayTextos[0] = "Um";
arrayTextos[1] = "Dois";
arrayTextos[2] = "Três";
arrayTextos[3] = "Quatro";
arrayTextos[4] = "Cinco";

for (int i = 0; i < arrayTextos.Length; i++)
{
    Console.WriteLine(arrayTextos[i]);
}

Console.Read();
```

Listas - Collections

Em linguagens de programação mais antigas, como o C, existiam momentos em que gostaríamos de crescer o tamanho de nossos arrays, pois precisávamos de maior quantidade de dados armazenados. Fazíamos isso através da criação de arrays muito maiores do que o necessário, pois não sabíamos o que estaria por vir. No C# existem objetos que funcionam como um array, porém são flexíveis. Esses objetos são derivações do tipo `Collections.Generic`.

Esse objetos contém as propriedades de vetores, porém podemos inserir valores nele, não sendo necessário fixar seu tamanho.

Para adicionar e remover itens de uma lista (`List`) utilizamos algumas funções predefinidas: conforme os exemplos do objeto `List`.

Exemplo

Add (objeto)

Para adicionarmos a uma lista de inteiros um novo número inteiro utilizamos o método `Add(inteiro)`.

```
List<int> inteiros = new List<int>();
inteiros.Add(1);
inteiros.Add(2);
inteiros.Add(5);
inteiros.Add(7);

for (int i = 0; i < inteiros.Count; ++i)
```

	Anotações

```

{
    Console.WriteLine(inteiros[i] + " ");
}
Console.Read();

Podemos fazer o mesmo com qualquer tipo de dado, como por exemplo o tipo string:

List<string> listTextos = new List<string>();
listTextos.Add("Um");
listTextos.Add("Dois");
listTextos.Add("Três");
listTextos.Add("Quatro");

for (int i = 0; i < listTextos.Count; ++i)
{
    Console.WriteLine(listTextos[i] + " ");
}
Console.Read();

```

Remove (objeto)

Para remover um objeto específico de sua lista, utilizamos a função Remove() passando como parâmetro qual o objeto a ser removido, conforme o exemplo.

Exemplo

```

List<string> listTextos = new List<string>();
listTextos.Add("Um");
listTextos.Add("Dois");
listTextos.Add("Três");
listTextos.Add("Quatro");

listTextos.Remove("Dois");
for (int i = 0; i < listTextos.Count; ++i)
{
    Console.WriteLine(listTextos[i] + " ");
}
Console.Read();

```

RemoveAt (posição)

Podemos remover objetos de nossa lista a partir da posição em que ele se encontra, conforme o exemplo.

	Anotações

Exemplo

```
List<string> listTextos = new List<string>();  
  
listTextos.Add("Um");  
  
listTextos.Add("Dois");  
  
listTextos.Add("Três");  
  
listTextos.Add("Quatro");  
  
listTextos.RemoveAt(2);  
  
for (int i = 0; i < listTextos.Count; ++i)  
  
{  
  
    Console.Write(listTextos[i] + " ");  
  
}  
  
Console.Read();
```

Variáveis Indexadas Multidimensionais (Matrizes)

Entendemos como funcionam os arrays/vetores em qualquer linguagem de programação e em específico em C#. Outra possibilidade de armazenamento de informações indexadas em variáveis é a Matriz. Matrizes são espécies de Arrays de Arrays. Podemos armazenar estruturas indexadas com mais de uma dimensão, conforme os exemplos abaixo.

Exemplo

```
int[,] arrayBidimensional = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Dessa maneira declaramos uma matriz de inteiros que terá duas dimensões (identificadas pelos separadores “,” encontrados dentro da definição do array) e definimos implicitamente quais os valores para cada um dos arrays internos.

Podemos declarar a mesma matriz, sem passar valor algum para ela, da seguinte maneira:

```
int [,] arrayBidimensional = new int [4,2];
```

Dessa maneira temos a matriz estruturada, porém sem valores definidos, tendo essa 4 linhas e 2 colunas.

Para conseguirmos acessar os parâmetros de uma matriz, é necessário que digamos qual a posição exata. Em uma matriz de 2 colunas e 4 linhas, conforme acima citado, devemos acessar os valores da seguinte maneira:

```
Console.WriteLine(array2D[0, 0]);
```

Isso nos trará o valor referente a primeira linha e primeira coluna. Para vetores de mais dimensões podemos acessá-los igualmente.

	Anotações

Matrizes tem maiores utilidades dentro do mundo matemático. Com linguagens de programação como C#, podemos criar Objetos e criar listas desses objetos, sendo uma maneira muito mais simples de administrar a informação indexada. Veremos exemplos no módulo de Orientação-a-Objetos.

Exercícios

- 1) Complete:
 - a. Quando sabemos exatamente o tamanho que terá nossa coleção utilizamos _____. E quando não sabemos ao certo quantos valores existirão em nossa coleção utilizamos _____.
b. _____ ou _____ são conjuntos de variáveis do mesmo tipo.
c. _____ são conjuntos de arrays ou também chamados de _____.
 - 2) Crie um novo projeto ConsoleApplication chamado “ExerciciosVariaveisIndexadas”. Em seu método principal crie um vetor de 10 posições do tipo inteiro. Preencha esses valores “hard coded”. Para este vetor crie dinamicamente uma Lista do tipo inteiro que receba a multiplicação de cada valor do vetor por dez. Ao final exiba todos os valores de ambos na tela.
 - 3) Crie uma matriz que receba como parâmetros o Primeiro Nome de 5 pessoas. Posterior ao preenchimento dos Primeiros Nomes, receba o Segundo Nome dessas mesmas pessoas, dizendo “Qual é o sobrenome do ”, concatenado com o Primeiro Nome. No final exiba todos os Nomes e Sobrenomes na tela.

Capítulo 10 – Depurando / Debugando Códigos

Depurar códigos no Visual Studio 2019 Community

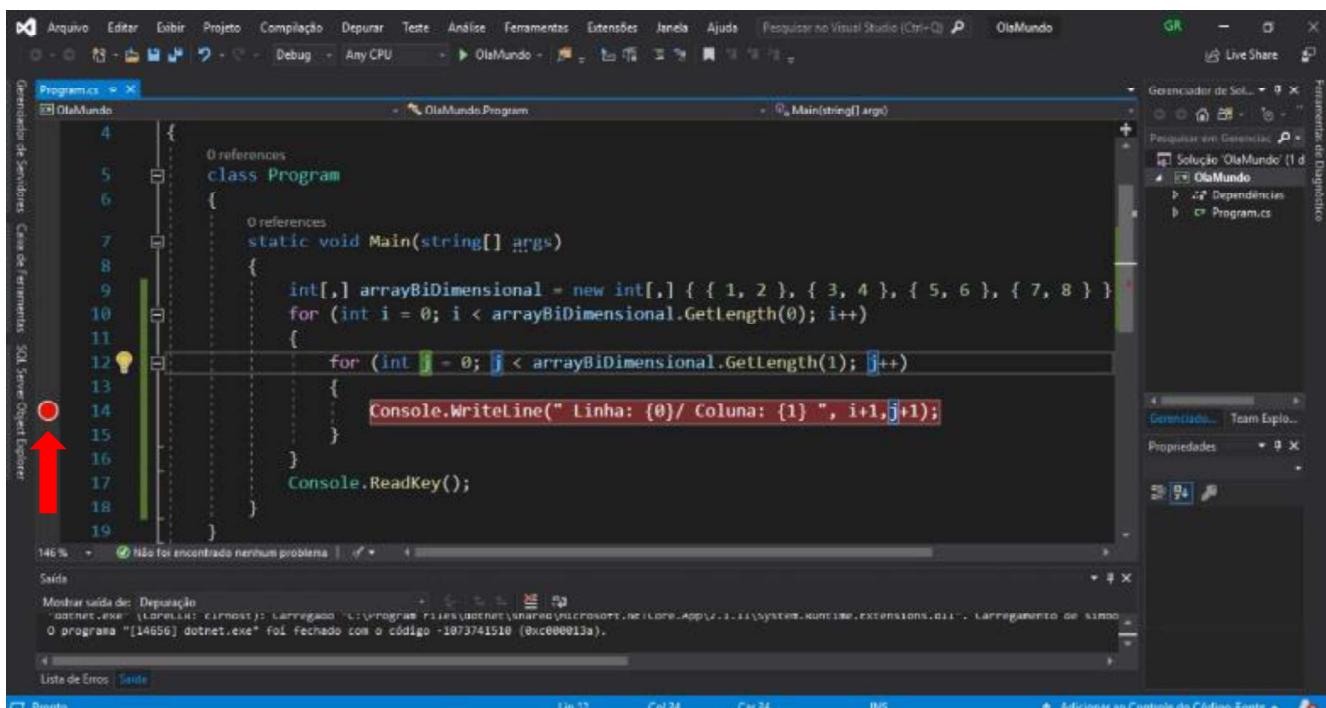
O Visual Studio 2019 Community, assim como qualquer versão do VS contém ferramentas poderosíssimas de depuração. Comandos de teclado que facilitam a depuração e visualizadores de variáveis em tempo real muito úteis. Dentre todas as funcionalidades de depuração as seguintes são destacadas:

Breakpoints

O uso de breakpoints (pontos de parada) durante o desenvolvimento é fundamental. Quando sabemos que o código passará por determinada linha de código, podemos inserir um breakpoint para que, em tempo de execução, nós possamos visualizar o valor atual de cada uma das variáveis.

Para que isso seja possível, basta clicar ao lado esquerdo da linha onde desejar a parada de execução do código, conforme exibe a imagem abaixo.

Exemplo



Visualizador de Variáveis

Quando paramos nosso código com um Breakpoint é possível visualizar o valor das variáveis simplesmente colocando o mouse em cima da variável. Dessa maneira podemos ver todos os valores, inclusive os valores contidos em cada coluna de uma matriz, conforme o exemplo abaixo.

	Anotações

```
4    {
5        0 references
6        class Program
7        {
8            0 references
9            static void Main(string[] args)
10           {
11               int[,] arrayBiDimensional = new int[3, 3] { { 1, 2 }, { 3, 4 }, { 5, 6 } };
12               for (int i = 0; i < arrayBiDimensional.GetLength(0); i++)
13               {
14                   for (int j = 0; j < arrayBiDimensional.GetLength(1); j++)
15                   {
16                       Console.WriteLine(" Linha: {0}/ Coluna: {1} ", i+1,j+1);
17                   }
18               }
19           }
20       }
21   }
```

95ms decorridos

Comandos de Continuação

Quando estamos desenvolvendo qualquer tipo de código, queremos saber qual o andamento de nosso programa, por onde está passando sua execução. Seja para prevenir Bugs ou por simples desconfiança de algum valor, é interessante que existam ferramentas para seguir seu programa passo a passo.

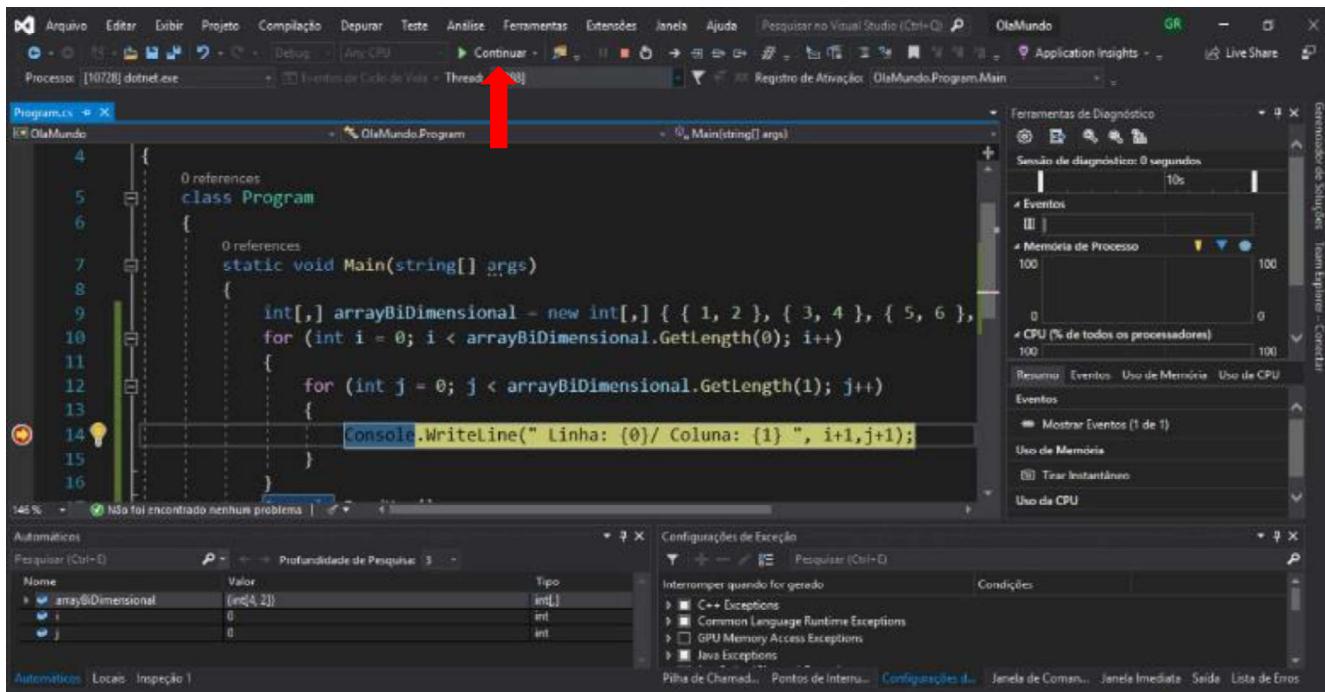
Em C# temos algumas possibilidades, sendo elas:

Continue

Ao pedir para que um programa Continue (clicando no botão continue), o seu programa seguirá em frente, parando somente onde houver outro Breakpoint.

Exemplo

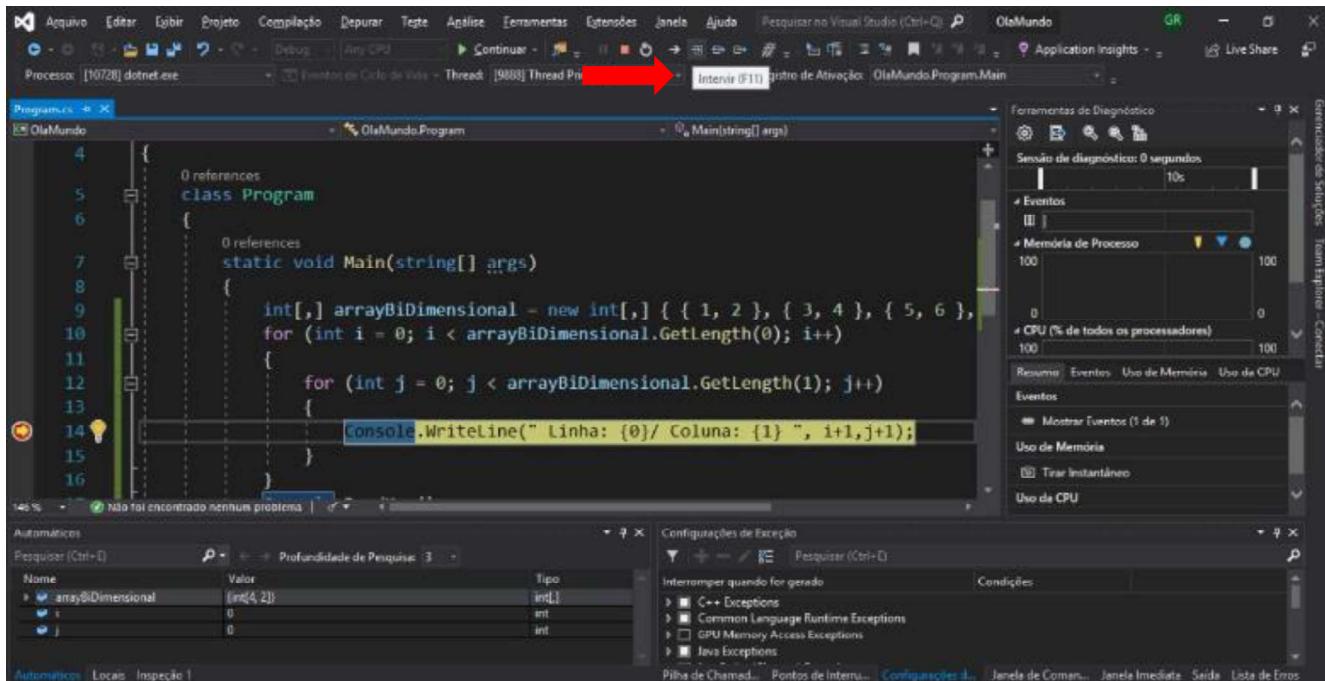
	Anotações



Step Into (Atalho - F11)

Esse comando faz com que o seu programa vá para a próxima linha de código. Esse é o passo-a-passo. Se dentro de nosso programa utilizarmos de procedimentos criados por nós mesmos (somente os criados por nós mesmos), nosso código entrará nesses procedimentos e terminado voltará para o código principal.

Exemplo

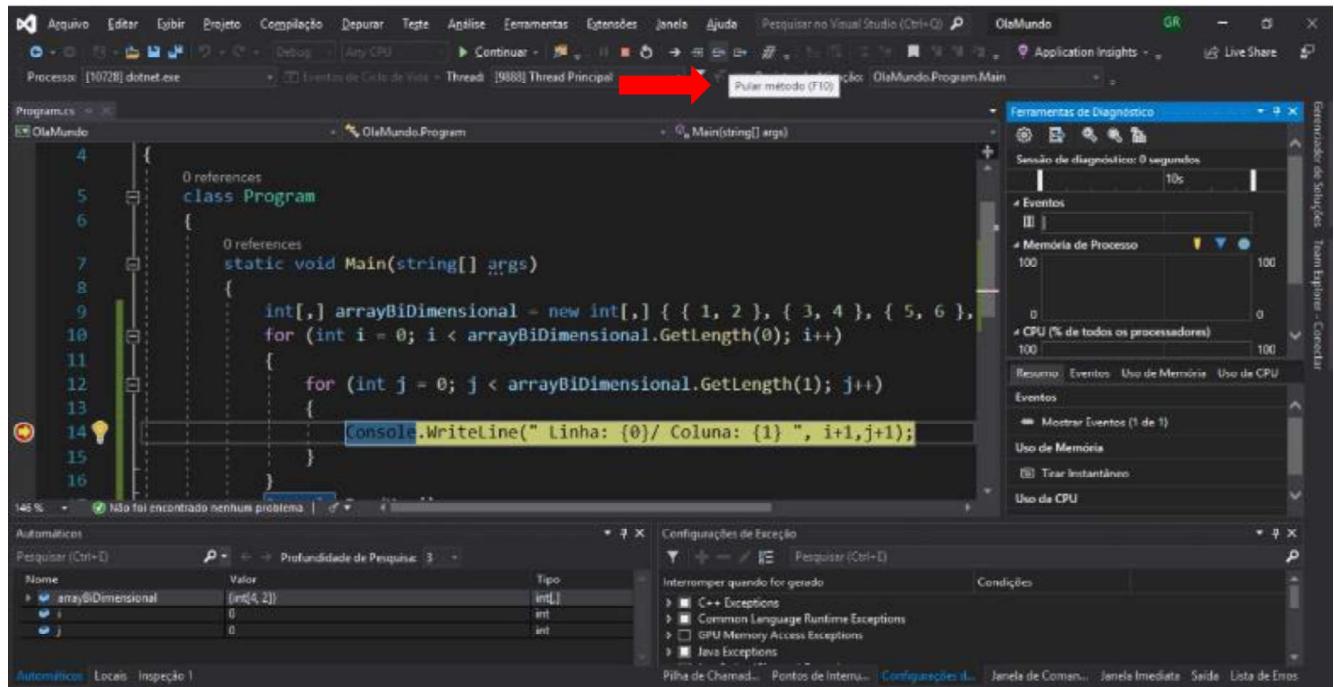


Anotações

Step Over (Atalho - F10)

Esse comando faz basicamente o mesmo passo a passo do comando Step Into, porém ele não entra em procedimentos que desenvolvemos. Ele pressupõe que nós desenvolvemos da maneira correta e se não houverem breakpoints dentro de nosso procedimento, ele executa e para na próxima linha de nosso método principal.

Exemplo

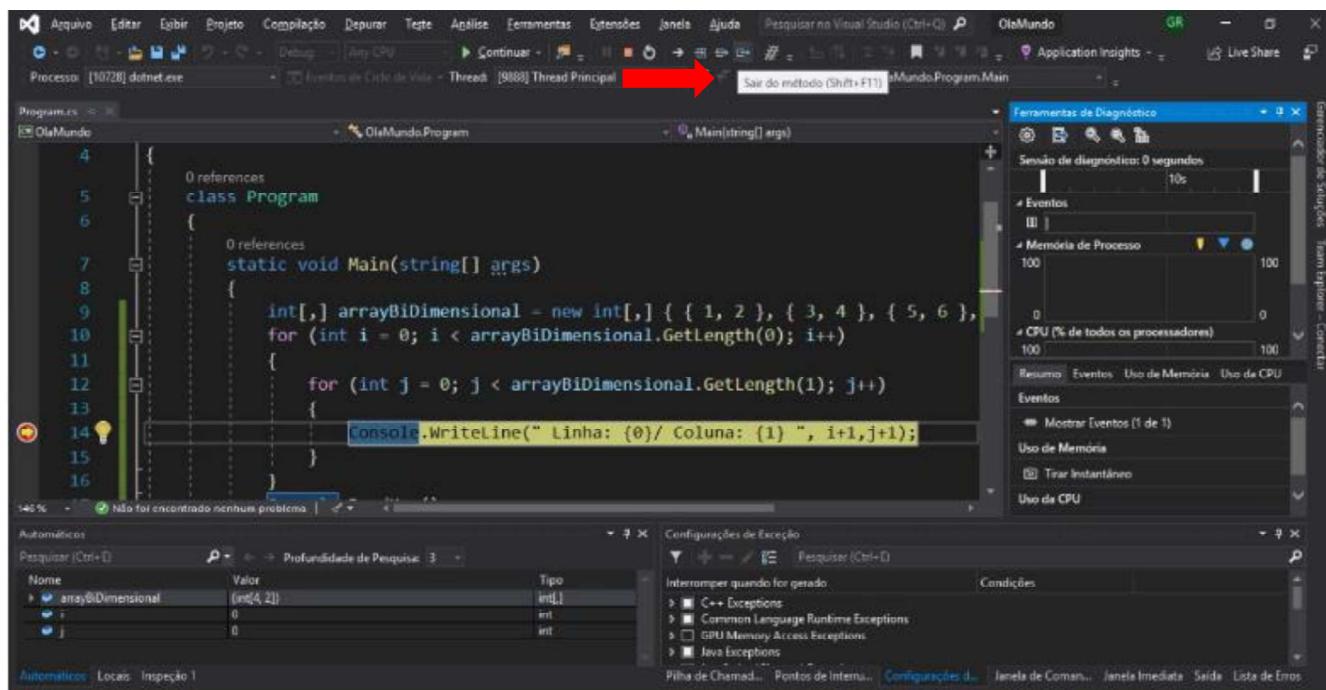


Step Out (Atalho – Shift+F11)

Ao selecionar Step Out saímos da depuração do método que estamos voltando para a próxima linha do método originário. Caso estivermos no método principal (Main) a depuração terminará.

	Anotações

Exemplo



Anotações

Capítulo 11 – Orientação à Objetos

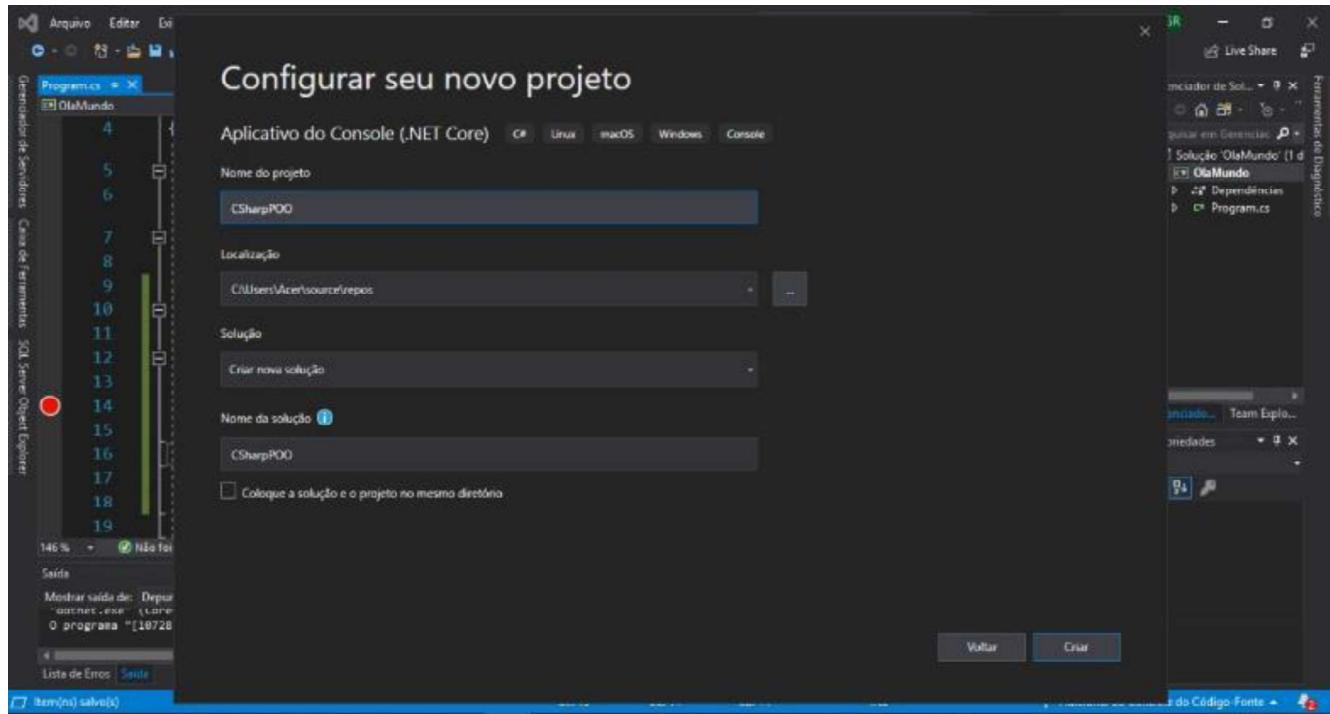
C# e a Programação Orientada-a-Objetos

Até esse momento de nosso curso, utilizamos as estruturas básicas da linguagem C# e algumas ferramentas mais usuais do *.NET Framework*. Agora entraremos nas principais funcionalidades que diferem o C# de outras linguagens tradicionais — a Orientação-a-Objetos.

Devemos lembrar, antes de mais nada, que conceitos como classes, objetos, herança e mensagem já foram entendidos previamente no curso de lógica de programação.

Para o fluxo desse módulo ser mais didático, vamos criar um projeto ConsoleApplication chamado CSharpPOO, conforme o exemplo.

Exemplo



Anotações

Após a criação de nosso projeto renomearemos nossa classe Program.cs para Principal.cs clicando com o botão direito sobre o arquivo Program.cs e renomeando, seu projeto deve estar conforme o exemplo.

```
1 using System;
2
3 namespace CSharpPOO
4 {
5     class Principal
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12}
13
```

Namespaces

Os Namespaces estão presentes em nosso código desde o começo do curso. Quando criamos nosso projeto CSharpPOO obtivemos uma série de arquivos gerados automaticamente, e um deles é o Program.cs - que mudamos para Principal.cs. Esse arquivo É uma classe. Toda classe deve ser construída dentro de um namespace, pois ele é quem separa nosso código dos códigos de outros sistemas e até do próprio .NET Framework. Sendo assim ele é **uma divisão física que delimita um conjunto de ferramentas (classes)**.

Para exemplificar de uma maneira mais simples vamos lembrar das ferramentas já utilizadas até esse momento, como por exemplo os Métodos - ReadLine () e WriteLine ().

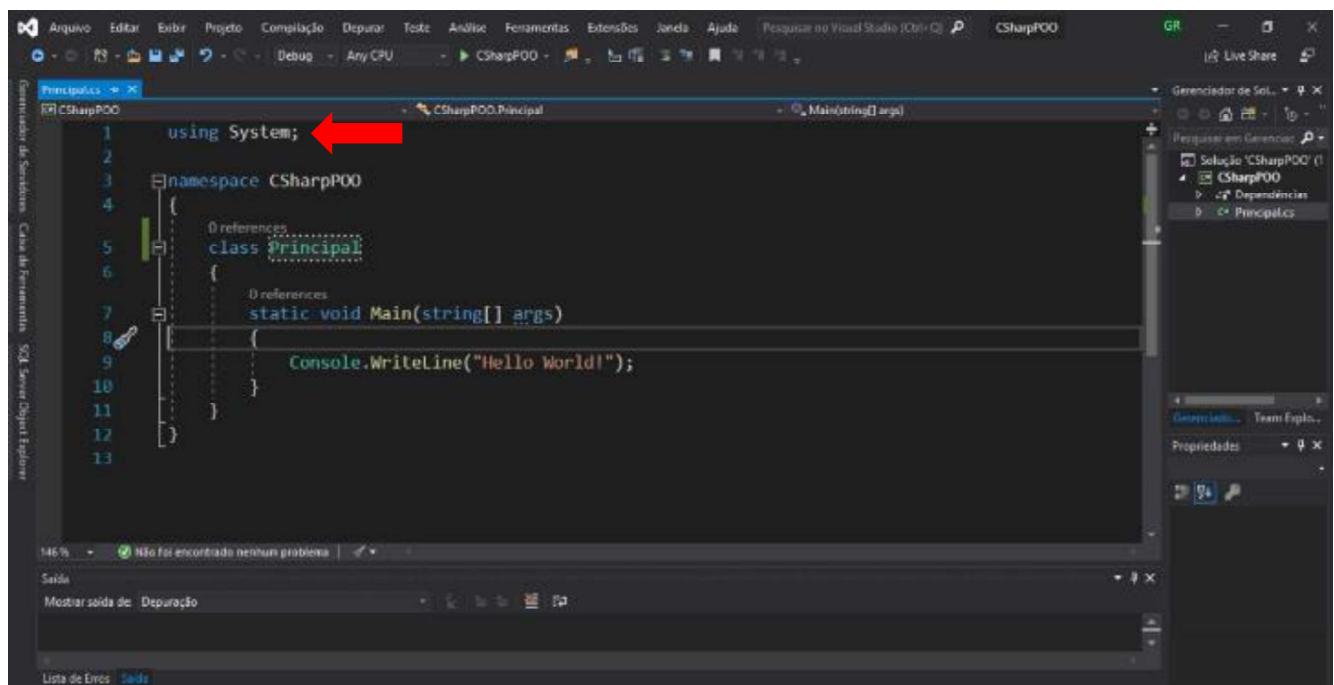
Ambos métodos fazem parte da classe Console. Essa classe faz parte do Namespace System, que é responsável por diversas funcionalidades relacionadas ao sistema. Esse agrupamento de informações que são pertinentes ao sistema, é o namespace sistema.

Sempre que utilizarmo-nos de uma funcionalidade específica do .NET Framework ou de nossos próprios projetos que não estão no nosso projeto atual, devemos adicionar essa referência desse Namespace utilizando a palavra reservada “**using**”⁸, conforme é feito no exemplo abaixo.

⁸ Podemos remover todas as referenciais que não estamos mais utilizando clicando em qualquer local do editor de texto (onde fica o código) e selecionando “Organize Usings” e clicando na opção “Remove Unused Usings”.

Anotações

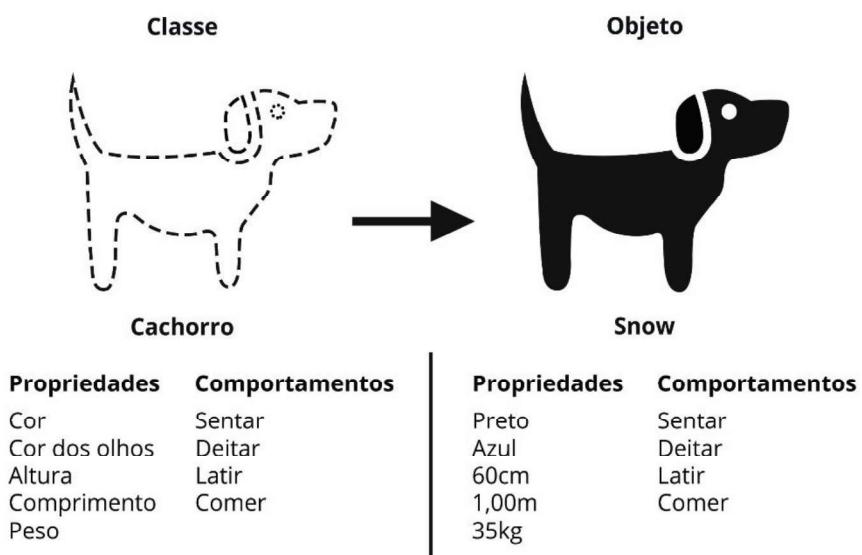
Exemplo



```
1 using System; ←
2
3 namespace CSharpPOO
4 {
5     class Principal
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12}
13
```

Veremos outros exemplos ao seguir desse módulo.

Classes e Objetos



As classes são as estruturas dos futuros objetos. São baseadas em propriedades e comportamentos que esse determinado objeto poderá ou não realizar.

Cada classe, em C#, tem como objetivo agrupar funcionalidades e informações que pertençam somente ao objeto que irá ser criado a partir dessa classe. Ou seja, uma classe Pessoa, não deveria ter informações sobre Sala de Aula.

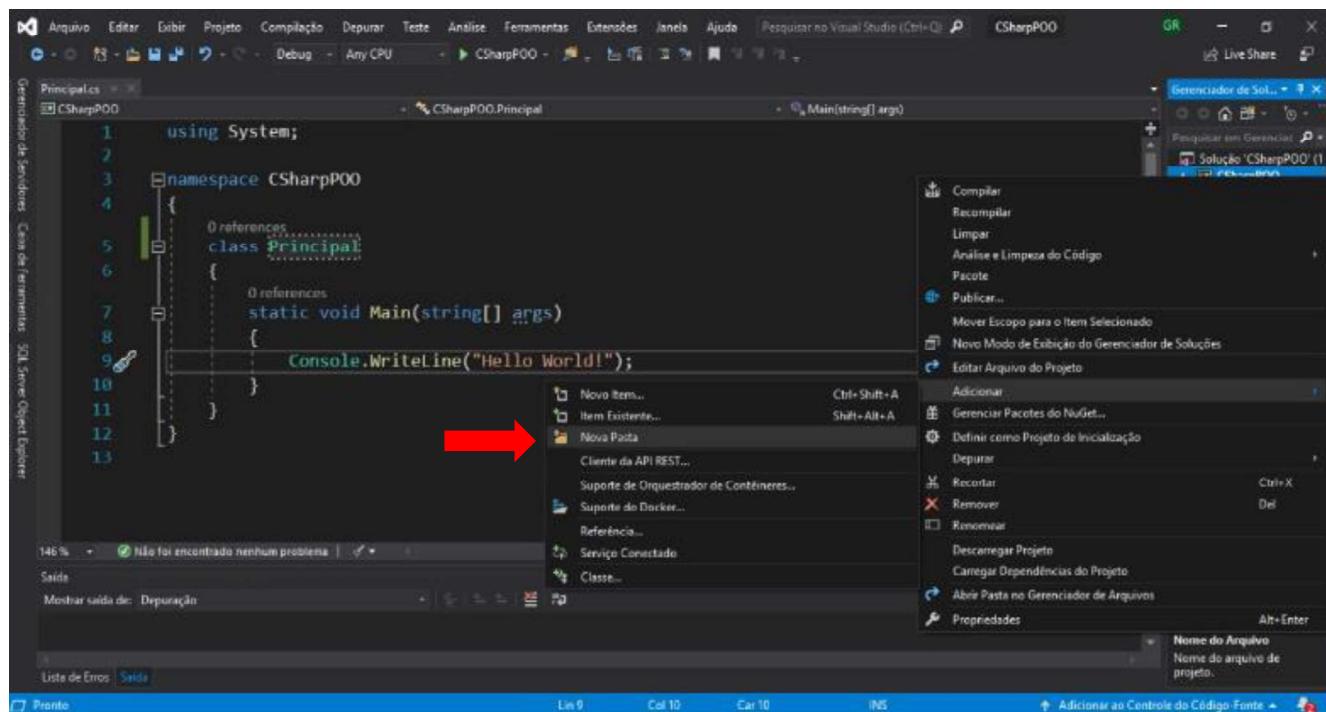
	Anotações

Agora iremos criar nossas próprias classes e fazer com que o nosso projeto se torne Orientado-a-Objetos.

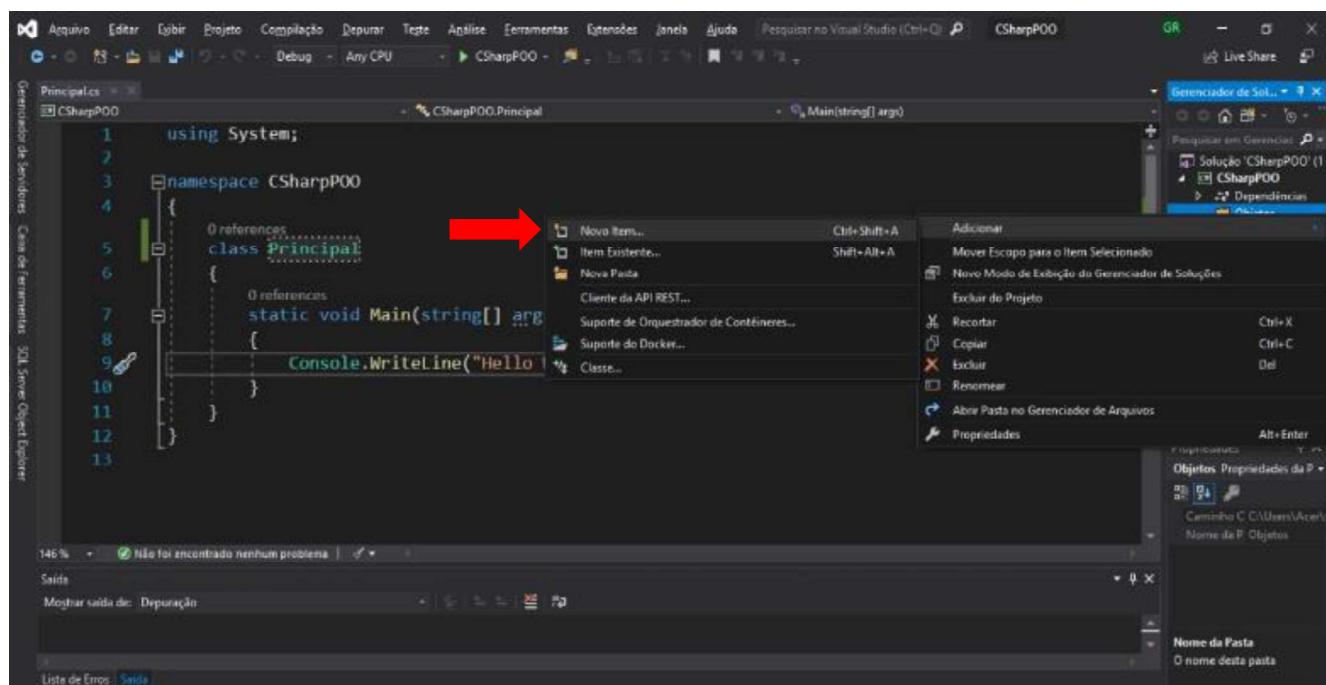
Exemplo

Passo 1 - Criação de Pastas Organizacionais

Com o botão direito clique no projeto **CSharpPOO** -> Add -> New Folder, conforme abaixo.



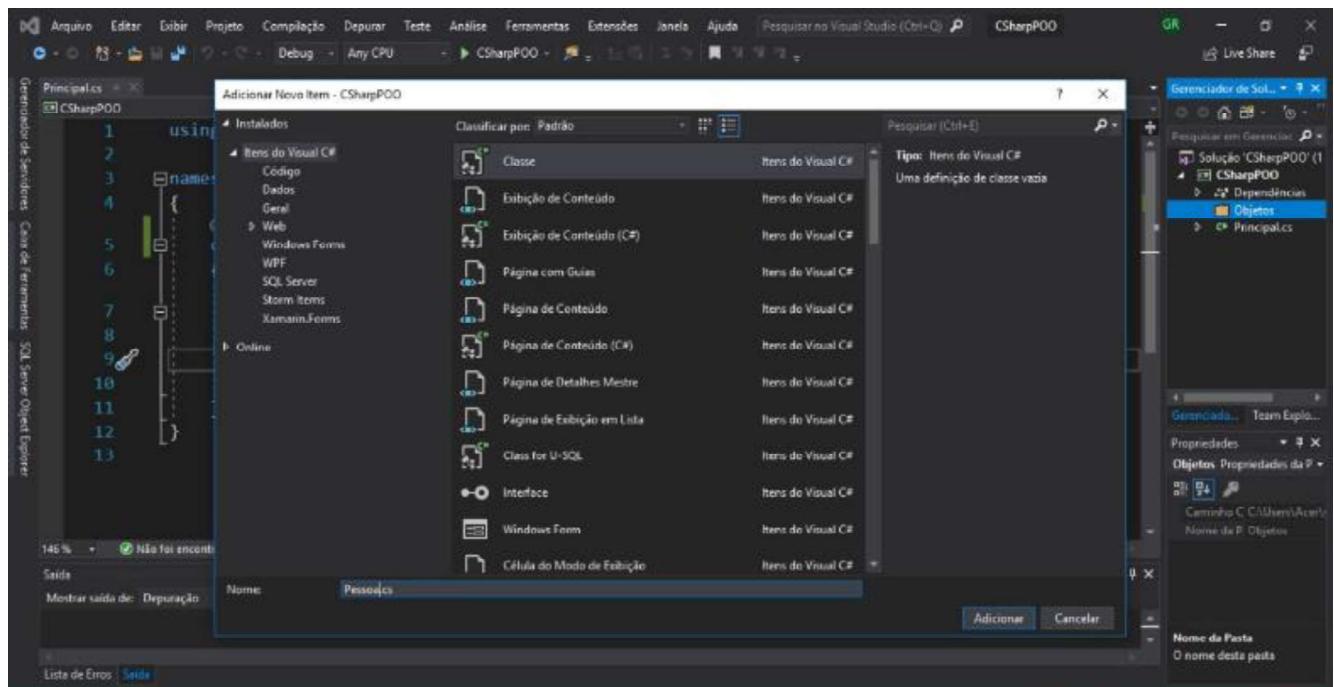
Nomeie-a de Objetos e confirme.



Anotações

Passo 2 - Adição de Classes

Clicando com o botão direito na pasta **Objetos** -> **Add** -> **Class**⁹, nomeie-a de Pessoa.cs, conforme o exemplo.



Passo 3 - Estrutura da Classe Pessoa

Após termos criado nossa pasta de Objetos e criado uma classe Pessoa, vamos entender como fazer com que o arquivo Pessoa.cs torne-se uma estrutura de um objeto Pessoa.

Como já discutido em lógica de programação, objetos tem suas características físicas - seja, suas propriedades, e tem seus comportamentos - suas ações. Podemos representar algumas dessas características e propriedades. Para esse exemplo vamos considerar as seguintes características de uma Pessoa da seguinte maneira.

Classe Pessoa

- Propriedades:
 - Nome
 - Endereço
 - Telefone
 - Cpf
 - Rg
- Comportamentos
 - Beber

⁹ Podemos criar classes através do comando Control+Shift+A e selecionando “Class”.

Anotações

E representamos isso em C# da seguinte maneira:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpPOO.Objetos
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public string Endereco { get; set; }
        public int Idade { get; set; }
        public DateTime DiaNascimento { get; set; }

        public void Beber()
        { }

        public void Comer()
        { }

        public void Andar()
        { }
    }
}
```

Feito isso, parabéns! Você criou sua primeira classe utilizando a orientação-a-objetos. Agora veremos como podemos utilizá-la em nossos programas.

Passo 4 - Instanciando Classes

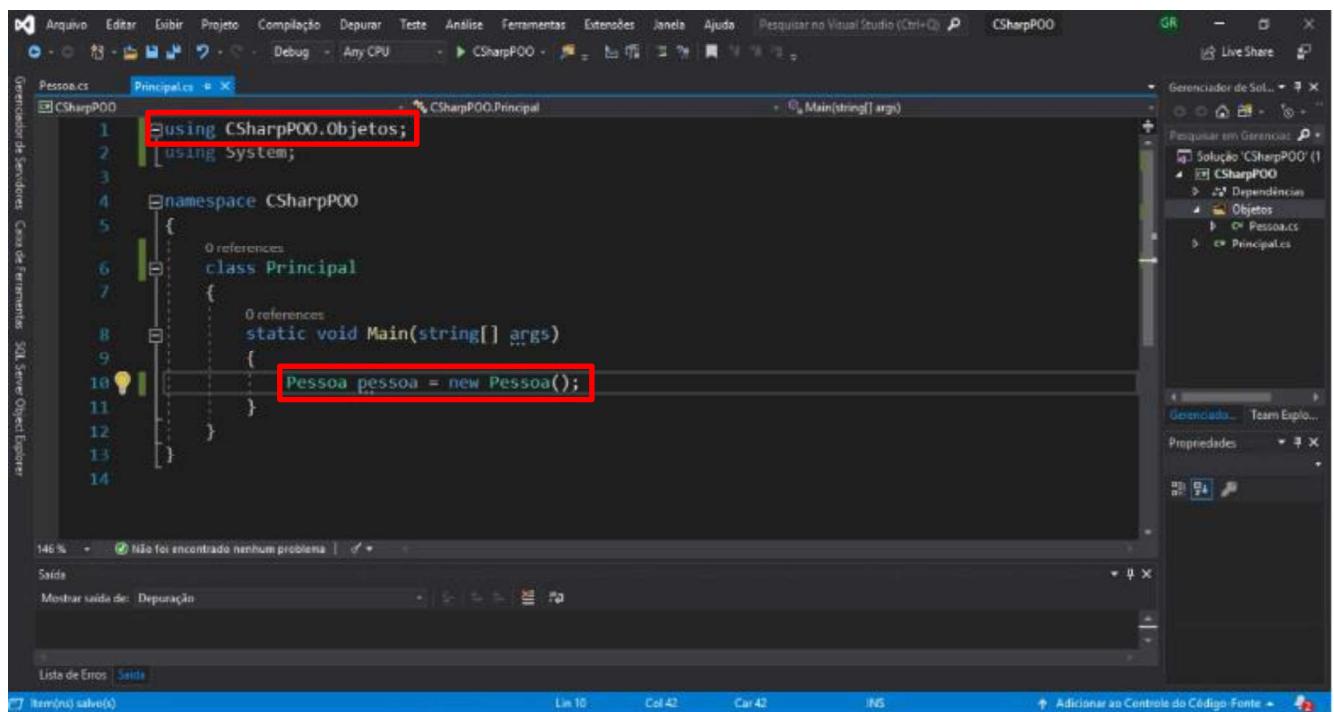
Para que isso seja possível, vamos ao nosso método *Main ()* em nossa classe Principal - quem define o que irá executar em nosso programa - e escreva o seguinte comando dentro do método principal:

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

Perceba que ele acusou não conhecer essa classe *Pessoa*. Isso acontece porque não adicionamos a Referência de nosso Namespace correto. O Namespace de Objetos ainda não é conhecido dentro dessa classe Principal. Fazemos isso rapidamente pelos comandos de teclado CONTROL+PONTO ou escrevemos acima de nossa

	Anotações

classe e namespace, junto às outras referências. De ambos os modos devemos ter a seguinte estrutura em nossa classe Principal:



```
1 using CSharpPOO.Objetos;
2 using System;
3
4 namespace CSharpPOO
5 {
6     class Principal
7     {
8         static void Main(string[] args)
9         {
10            Pessoa pessoa = new Pessoa();
11        }
12    }
13 }
14
```

Esse bloco de código define o que é uma **Instância de uma Classe**, ou seja, nosso primeiro **Objeto Pessoa**.

Passo 5 - Uso de propriedades e comportamentos

Dissemos que uma pessoa tem Nome, Email, Telefone, Endereço e Data de Nascimento. Agora iremos fazer com que nosso objeto Pessoa tenha valores para essas propriedades.

Nosso objeto pessoa passou a ser manipulável como uma variável, ou como um Tipo, assim como já trabalhamos com outros objetos. Para acessar e modificar informações sobre esse objeto utilizaremos do “DOT Notation” que é a acessibilidade de propriedades e comportamentos através do caractere “.” (Ponto).

Veja um exemplo de preenchimento de propriedades de um objeto Pessoa:

Anotações

```
1  using CSharpPOO.Objetos;
2  using System;
3
4  namespace CSharpPOO
5  {
6      class Principal
7      {
8          static void Main(string[] args)
9          {
10             Pessoa pessoa = new Pessoa();
11             pessoa.DiaNascimento = new DateTime(1992,02,23);
12             pessoa.Endereco = "Rua Lauro Muller 370";
13             pessoa.Nome = "Gustavo Rosauro";
14             pessoa.Idade = 27;
15         }
16     }
17 }
```

Parabéns novamente! Nesse momento seu Objeto Pessoa realmente existe e tem dados que o tornam único.

Encapsulamento

O encapsulamento é uma ferramenta das linguagens de programação orientadas-a-objetos que nos permite tornar o acesso as propriedades de um objeto algo mais controlado. Isso nos dá uma ferramenta poderosa de segurança das informações de nossos objetos, permitindo somente quem deve alterar aquele objeto.

Lembrando de nossa classe Pessoa, que contém propriedades como Nome, Endereço, Telefone, Email, etc. Essas informações são pertinentes ao cadastro dessa pessoa. Sendo assim, não seria interessante que outras funcionalidades sem ser a de cadastro tivessem possibilidade de alterar essa informação.

Para que isso seja possível, devemos entender como funcionam algumas palavras reservadas do C#, como public, private e protected.

Public

Quando declaramos variáveis publicas estamos querendo dizer que qualquer instância daquela classe poderá alterar essa informação. Ou seja, nada importa, qualquer um faz qualquer coisa com aquela propriedade.

Private

Variáveis private (privadas) são acessíveis somente dentro das próprias classes. Instâncias de classes (objetos) não conseguirão visualizar elas.

Protected

Variáveis protected (protegidas) são acessíveis em somente duas ocasiões, em sua própria classe e em instâncias de classes derivadas (lembrando classes derivadas são as classes que herdaram dessa classe).

Acesso de propriedades - Get / Set

Quando criamos nossa classe Pessoa, criamos junto a elas um conjunto de código conforme o exemplo:

Anotações

```
{ get; set; }
```

A palavra `get` tem como objetivo dizer o que deve ser feito na leitura dessa propriedade. E a palavra `set` diz o que deve ser feito na escrita.

Ambas ações estão feitas da maneira simplificada de declarações de propriedades do C#. Até o Framework 3.0 as propriedades deveriam ser escritas da seguinte maneira:

```
public string Nome  
{  
    get  
    {  
        // código para ler a propriedade  
    }  
  
    set  
    {  
        // código para escrever na propriedade  
    }  
}
```

Ainda é possível declarar dessa maneira propriedades, principalmente quando é necessário realizar alguma ação antes de retornar a propriedade. Essa ação deve ser realizada dentro do código contido nos blocos de código `Get` e `Set`.

Agora vamos supor que em nossa classe `Pessoa`, exista um método de nome `Cadastrar` e que recebe como parâmetros todas as propriedades de nosso futuro Objeto `Pessoa`. Seria interessante que ele fosse o único ponto em que meu código pudesse alterar as informações essas propriedades e não permitir que mais ninguém altere.

Podemos fazer isso da seguinte maneira:

```
public string Nome { get; private set; }  
public string Endereco { get; private set; }  
public int Idade { get; private set; }  
public DateTime DiaNascimento { get; private set; }
```

Adicionando o `PRIVATE` antes do sub-método `SET`, nós estamos relacionando esse sub-método e dizendo a ele que somente a própria classe poderá alterar informações. Pelo fato de não termos declarado a propriedade como `private`, o sub-método `GET` ainda é público, podendo qualquer pessoa Ler esse objeto.

O modo de utilização em nosso método `Main ()` ficará da seguinte maneira:

```
class Principal  
{
```

Anotações

```

static void Main(string[] args)
{
    Pessoa pessoa = new Pessoa();
    pessoa.Cadastrar(
        "Gustavo Rosauro",
        "Rua Lauro Muller 370",
        27,
        new DateTime(1992, 02, 23)
    );

    //Já sendo cadastrada as informações no objeto, podemos acessa-las
    Console.WriteLine(pessoa.DiaNascimento);
    Console.WriteLine(pessoa.Nome);
    Console.WriteLine(pessoa.Idade);
    Console.WriteLine(pessoa.Endereco);

    Console.Read();
}

}

```

E nossa classe `pessoa`:

```

class Pessoa
{
    public string Nome { get; private set; }
    public string Endereco { get; private set; }
    public int Telefone { get; private set; }
    public DateTime DiaNascimento { get; private set; }

    public Pessoa Cadastrar(string _nome, string _endereco, int _idade, DateTime
    _diaNascimento)
    {
        this.DiaNascimento = _diaNascimento;
        this.Endereco = _endereco;
        this.Nome = _nome;
        this.Idade = _idade;
        return this;
    }
}

```

Anotações

```

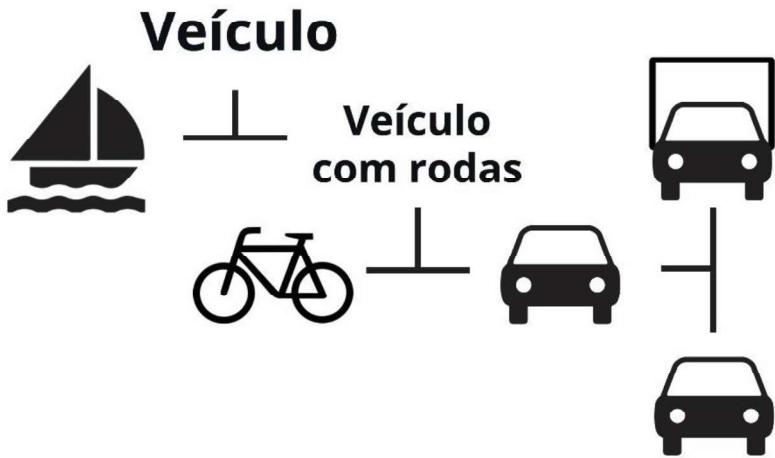
public void Beber()
{
}

public void Comer()
{
}

public void Andar()
{
}

```

Herança



Herança é o processo de especialização de classes. Em C# podemos especializar uma classe utilizando, após o nome de nossa classe, o caractere ":" seguido pelo nome da classe que desejamos especializar. Para demonstrar criaremos uma nova classe dentro de nossa pasta Objetos chamada Aluno, conforme o exemplo.

Exemplo

	Anotações

The screenshot shows the Visual Studio IDE interface. In the center, there is a code editor window displaying the file `Aluno.cs` which contains the following C# code:

```
using System;
using System.Text;

namespace CSharpPOO.Objetos
{
    public class Aluno : Pessoa
    {
        public int NumeroMatricula { get; set; }
        public string CursoMatriculado { get; set; }
        public string Escola { get; set; }
    }
}
```

To the right of the code editor is the Solution Explorer, which shows the project structure for 'CSharpPOO' (1). It includes a 'Dependências' node and an 'Objetos' node containing files `Aluno.cs`, `Pessoa.cs`, and `Principais.cs`. The status bar at the bottom indicates that 1 item(s) were saved.

Dessa maneira estamos dizendo que um Objeto Aluno é uma Pessoa e um aluno.

Para utilizarmos esse objeto, a manipulação das informações funcionará da seguinte maneira:

The screenshot shows the Visual Studio IDE interface again. This time, the code editor displays the file `Principais.cs` with the following code:

```
class Program
{
    static void Main(string[] args)
    {
        Aluno aluno = new Aluno();
        aluno.DiaNascimento = new DateTime(1992, 02, 23);
        aluno.Endereco = "Rua Lauro Muller 370";
        aluno.Nome = "Gustavo Rosario";
        aluno.Idade = 27;
        aluno.NumeroMatricula = 232423434;
        aluno.CursoMatriculado = "C# Fundamentos";
        aluno.Escola = "Apex";
        Console.WriteLine(aluno.DiaNascimento);
        Console.WriteLine(aluno.Endereco);
        Console.WriteLine(aluno.Nome);
        Console.WriteLine(aluno.Idade);
        Console.WriteLine(aluno.NumeroMatricula);
        Console.WriteLine(aluno.CursoMatriculado);
        Console.WriteLine(aluno.Escola);
        Console.Read();
    }
}
```

The Solution Explorer on the right shows the same project structure as before. The status bar at the bottom indicates 1 error, 0 warnings, and 0 messages.

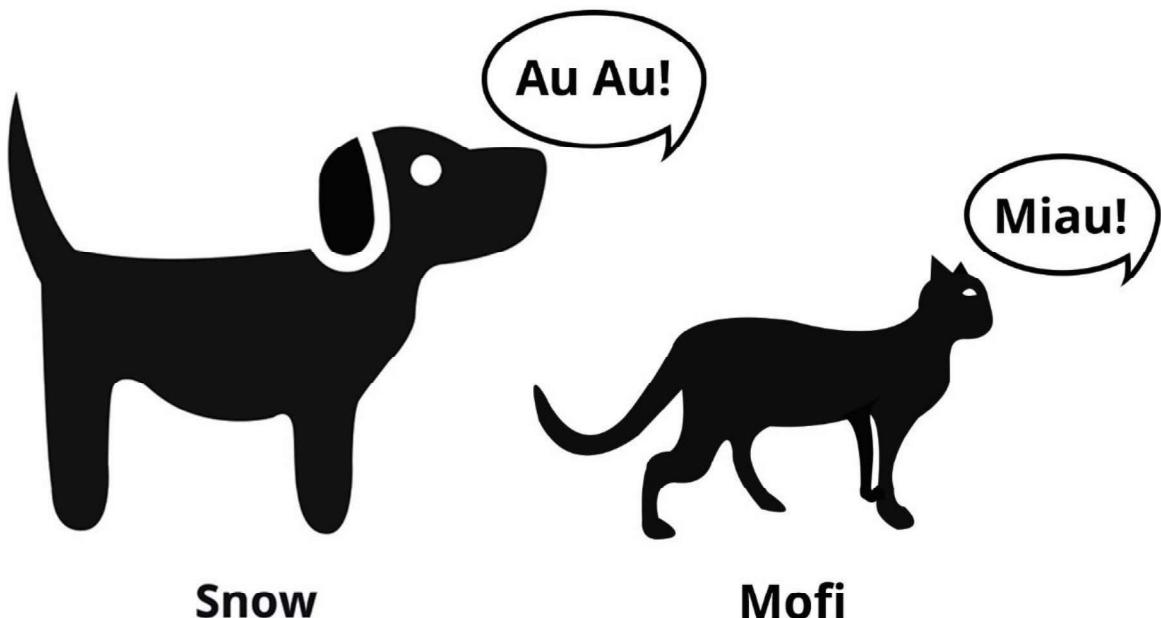
O funcionamento é semelhante ao de uso tradicional das propriedades, ficando praticamente imperceptível a distinção entre propriedades da classe pai e filha.

Utilizando o mesmo exemplo, as propriedades da classe Pessoa foram acessíveis para o objeto Aluno. Os comportamentos também podem ser acessados, basta utilizar conforme feito com as propriedades.

		Anotações

```
aluno.Comer();  
aluno.Bebêr();
```

Polimorfismo



Polimorfismo está totalmente relacionada ao processo de Herança. É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para uma referência a um objeto do tipo da superclasse.

Para exemplificar esse conceito, criaremos uma pasta chamada Polimorfismo e três classes, Veiculo, Automóvel e Barco, sendo Automóvel e Barco especializações de Veículo. A estrutura de suas classes ficará dessa maneira:

```
class Veiculo  
{  
    public string TipoModelo { get; private set; }  
  
    public Veiculo(string tipoModelo)  
    {  
        this.TipoModelo = tipoModelo;  
    }  
    public virtual void Mover()  
    { }  
  
    public virtual void Parar()
```

Anotações

```

    {
}

public class Barco : Veiculo
{
    public Barco(string tipoModelo)
        : base(tipoModelo)
    {
    }

    public override void Mover()
    {
        Console.WriteLine("Acelerando barco");
    }

    public override void Parar()
    {
        Console.WriteLine("Atracando barco no porto.");
    }
}

public Automovel(string tipoModelo)
    : base(tipoModelo)
{
}

public override void Mover()
{
    Console.WriteLine("Acelerando o automovel");
}

public override void Parar()
{
    Console.WriteLine("Parando o automovel.");
}

```

Anotações

Nossa classe Principal deve estar da seguinte maneira:

```
class Principal
{
    static void Main(string[] args)
    {
        //Cria array de veiculos, podendo ser Automovel, Barco ou qualquer
        que herde de Veiculo

        Veiculo[] veiculo = new Veiculo[2];

        veiculo[0] = new Automovel("BMW");
        veiculo[1] = new Barco("Phantom");
        MovimentarVeiculo(veiculo[0]);
        MovimentarVeiculo(veiculo[1]);

        Console.WriteLine("Digite Enter para os veículos pararem. ");
        Console.ReadLine();
        PararVeiculo(veiculo[0]);
        PararVeiculo(veiculo[1]);
        Console.Read();

    }

    public static void MovimentarVeiculo(Veiculo veiculo)
    {
        Console.WriteLine(veiculo.TipoModelo);
        veiculo.Mover();
    }

    public static void PararVeiculo(Veiculo veiculo)
    {
        Console.WriteLine(veiculo.TipoModelo);
        veiculo.Parar();
    }
}
```

Agora que já montamos nosso código, vamos entende-lo.

Criamos três classes, Veículo, Automóvel e Barco. Automóvel e Barco são especializações da classe Veículo. Dentro das classes criamos métodos com o mesmo nome dos métodos existentes na classe Veículo.

- A diferença entre a assinatura dos métodos das classes filhas em relação a classe pai é a palavra “**Override**” que significa **Sobrescrever**. Quando sobrescrevemos um método da classe pai, é necessário que isso seja possível. Para que isso torne-se possível adicionamos a palavra “**virtual**” na assinatura da classe pai.
 - Quando sobrescrevemos um método significa que o método válido é o da classe filha e não o da classe pai.

Outra possibilidade do polimorfismo é a capacidade de **OverLoads** de métodos, o que significa Sobrecarga de métodos. Consiste em métodos com o mesmo nome, porém assinaturas diferentes (quantidade de parâmetros e retornos possíveis). Overloads tem como objetivo definir outra maneira de realizar a mesma operação. Como por exemplo:

Nosso Automóvel chama o método Parar () cujo não leva parâmetros e retorna vazio (void). Podemos criar um método chamado Parar do qual recebe uma string como parâmetro. Isso seria um Overload do método Parar, conforme o exemplo.

Exemplo

```
public class Automovel : Veiculo
{
    public Automovel(string tipoModelo)
        : base(tipoModelo)
    {
    }

    public override void Mover()
    {
        Console.WriteLine("Acelerando o automovel");
    }

    public override void Parar()
    {
        Console.WriteLine("Parando o automovel.");
    }

    public void Parar(string comoParar)
    {
        Console.WriteLine("Parando o Automovel: " + comoParar);
    }
}
```

Anotações

E nossa classe principal poderíamos chamar os métodos da seguinte maneira:

```
class Principal
{
    static void Main(string[] args)
    {
        //Cria array de veiculos, pondendo ser Automovel, Barco ou qualquer que herde de
        Veiculo

        Automovel veiculo = new Automovel("BMW");
        //Movimenta
        MovimentarVeiculo(veiculo);
        Console.WriteLine("Digite o modo de parada do Automovel.");
        string modo = Console.ReadLine();//Por exemplo "Muito depressa.".

        //Parar com overload
        veiculo.Parar(modo);

        Console.Read();
    }

    public static void MovimentarVeiculo(Veiculo veiculo)
    {
        Console.WriteLine(veiculo.TipoModelo);
        veiculo.Mover();
    }

    public static void PararVeiculo(Veiculo veiculo)
    {
        Console.WriteLine(veiculo.TipoModelo);
        veiculo.Parar();
    }
}
```

Abstração

Classes abstratas vieram para fornecer um padrão ao desenvolvimento de um sistema.

Utilizando nosso exercício de polimorfismo podemos entender que nós nunca teremos um Objeto do tipo Veículo puramente, pois ele sempre será Automóvel ou Barco. Entretanto se nós quiséssemos seria possível construir um objeto veículo.

Anotações

Para que nós não permitíssemos essa possibilidade, nós teríamos que dizer que veículo é somente um modelo, uma **Abstração** de um Veículo.

Exemplo

Passo 1 - Copie a pasta Polimorfismo

Copie a Pasta Polimorfismo e renomeie sua cópia para “Abstracao” e renomeie os namespaces de suas classes para **CSharpPOO.Abstracao**.

- namespace CSharpPOO.Abstracao

Passo 2 - Torne a classe Veiculo Abstrata

- public abstract class Veiculo

Passo 3 - Substitua Virtual por Abstract nos métodos e retire seu corpo (chaves - {})

Sua classe Veículo deve tornar-se como essa:

```
Veiculo.cs  ✘ Barco.cs      Automovel.cs    Barco.cs      Automovel.cs    Veiculo.cs      Principal.cs
CSharpPOO.Abstracao.Veiculo

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpPOO.Abstracao
{
    public abstract class Veiculo
    {
        public string TipoModelo { get; private set; }

        public Veiculo(string tipoModelo)
        {
            this.TipoModelo = tipoModelo;
        }
        public abstract void Mover();
        public abstract void Parar();
    }
}
```

Devido a já termos nossas classes filhas com os métodos implementados, nosso código está pronto. Basta compilar e testar.

Veja que a diferença entre uma Classe e uma Classe Abstrata é:

- Não se pode instanciar uma classe Abstrata (criar um objeto abstrato não existe).
- Classes abstratas podem ser herdadas e esse é seu propósito.
- Uma classe abstrata pode conter métodos comuns, **somente serão obrigatórios às classes filhas os métodos abstratos**.
- Um método abstract ele já é virtual, não podendo explicitar.
- Por ser um método virtual é obrigatório o uso de overrides nos métodos das classes filhas.

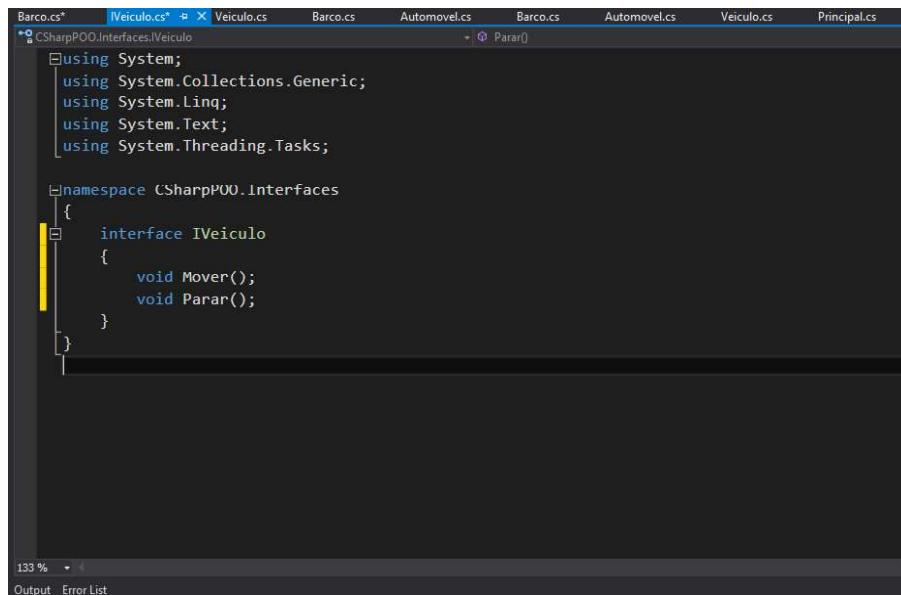
Anotações

Interfaces

A interface¹⁰ é mais uma possibilidade organizacional de seu sistema. Uma interface é um tipo de classe que contém apenas as assinaturas de métodos, propriedades, eventos e indexadores. Cabe a uma classe a implementação concreta.

Diferente de uma classe abstrata que pode conter métodos próprios, somente podemos definir a estrutura de Interfaces, conforme o exemplo.

Exemplo



The screenshot shows a Visual Studio code editor window with several tabs at the top: Barco.cs*, IVeiculo.cs*, Veiculo.cs, Barco.cs, Automovel.cs, Barco.cs, Automovel.cs, Veiculo.cs, and Principal.cs. The IVeiculo.cs* tab is active. The code in the editor is:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpPOO.Interfaces
{
    interface IVeiculo
    {
        void Mover();
        void Parar();
    }
}
```

Quando criarmos classes que definirão como deve ser o procedimento de Mover o veículo e o procedimento para Parar o veículo, faremos elas Implementarem a classe Veículo, conforme abaixo com a classe Barco.

¹⁰ Interfaces tem o entendimento em muitas literaturas como os Contratos do software. Especialmente em WebServices.

		Anotações

The screenshot shows a code editor window with several tabs at the top: Barco.cs*, IVeiculo.cs*, Veiculo.cs, Barco.cs, Automovel.cs, Barco.cs, Automovel.cs, Veiculo.cs, and Principal.cs. The active tab is Barco.cs*. The code in the editor is:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpPOO.Interfaces
{
    class Barco : IVeiculo
    {
        void IVeiculo.Mover()
        {
            Console.WriteLine("Movendo barco");
        }

        void IVeiculo.Parar()
        {
            Console.WriteLine("Atracando barco no porto.");
        }
    }
}
```

Uma classe pode implementar mais de uma interface, porém todos os métodos de todas interfaces devem estar implementados, causando um erro de compilação se contrário.

Anotações

Exercícios

1) Relacione:

- | | |
|---|------------------|
| ■ Proteção e segurança de acesso. | ■ Herança |
| ■ Estrutura | ■ Polimorfismo |
| ■ Conjunto de classes do mesmo tema | ■ Abstração |
| ■ Especialização | ■ Interfaces |
| ■ Obrigatoriedade do padrão | ■ Classes |
| ■ Classes que jamais serão objetos, porém objetos herdarão dela | ■ Namespaces |
| ■ Especialização e sobrecarga de funções | ■ Encapsulamento |

2) Crie um projeto ConsoleApplication e nomeie-o de “ExerciciosPOO”. Seu projeto deverá:

- a. Conter classes de animais. (Animais sugeridos, Pato, Gato, Cachorro)
- b. Cada animal deve obrigatoriamente herdar de uma superclasse, sendo essa abstrata.
- c. A superclasse deve conter o método abstrato EmitirSom () .
- d. Cada um dos animais deve ser capaz de emitir som de seu próprio jeito.
- e. O método principal deve ser capaz de visualizar as propriedades dos objetos, porém somente através do método CadastrarAnimal () que as subclasses sobrescreverão (como sobrecarga) da classe pai, será possível cadastrá-los.
- f. O método principal deve cadastrar cada um dos animais criados e atribui-los a um array da classe pai.
- g. O método principal deve emitir o som de cada animal.

	Anotações

Capítulo 12 – Exercícios de Revisão – Parte I



1. Antes do racionamento de energia ser decretado, quase ninguém falava em quilowatts; mas, agora, todos incorporaram essa palavra em seu vocabulário. Sabendo-se que 100 quilowatts de energia custa um sétimo do salário mínimo, fazer um algoritmo que receba o valor do salário mínimo e a quantidade de quilowatts gasta por uma residência e calcule. Imprima:

- O valor em reais de cada quilowatt
- O valor em reais a ser pago
- O novo valor a ser pago por essa residência com um desconto de 10%

2. Em época de pouco dinheiro, os comerciantes estão procurando aumentar suas vendas oferecendo desconto. Faça um algoritmo que possa entrar com o valor de um produto e imprima o novo valor tendo em vista que o desconto foi de 9%.

3. Criar um algoritmo que efetue o cálculo do salário líquido de um professor. Os dados fornecidos serão: valor da hora aula, numero de aulas dadas no mês e percentual de desconto do INSS.

	Anotações

4. Todo restaurante, embora por lei não possa obrigar o cliente a pagar, cobra 10% para o garçom. Fazer um algoritmo que leia o valor gasto com despesas realizadas em um restaurante e imprima o valor total com gorjeta.

5. Uma pessoa resolveu fazer uma aplicação em uma poupança programada. Para calcular seu rendimento, ela deverá fornecer o valor constante da aplicação mensal, a taxa e o número de meses. Sabendo-se que a fórmula usada para este cálculo é:

Valor acumulado = $P * ((1+i)^n - 1)/i$

i = taxa

P = aplicação mensal

n = número de meses (obs. $(1+i)$ elevado a n)

6. Entrar com um número e imprimi-lo caso seja maior que 20.

7. Construir um algoritmo que leia dois valores numéricos inteiros e efetue a adição; caso o resultado seja maior que 10, apresentá-lo.

8. Construir um algoritmo que leia dois números e efetue a adição. Caso o valor somado seja maior que 20, este deverá ser apresentado somando-se a ele mais 8; caso o valor somado seja menor ou igual a 20, este deverá ser apresentado subtraindo-se 5.

9. A prefeitura do Rio de Janeiro abriu uma linha de crédito para os funcionários estatutários. O valor Máximo da prestação não poderá ultrapassar 30% do salário bruto. Fazer um programa que permita entrar com o salário bruto e o valor da prestação e informa se o empréstimo pode ou não ser concedido.

10. Construir um algoritmo que indique se o número digitado está compreendido entre 20 e 90 ou não.

11. Entrar com um número e imprimir uma das mensagens: maior do que 20, igual a 20 ou menor do que 20.

12. Entrar com nome, sexo e idade de uma pessoa. Se a pessoa for do sexo feminino e tiver menos que 25 anos, imprimir nome e a mensagem: ACEITA. Caso contrário, imprimir nome e a mensagem: NÃO ACEITA.

	Anotações

13. Entrar com a sigla do estado de uma pessoa imprimir uma das mensagens:

- Carioca
- Paulista
- Mineiro
- Outros estados

14. Criar um algoritmo que leia dois números e imprimir uma mensagem dizendo se são iguais ou diferentes.

15. Entrar com dois números e imprimir o maior numero (suponha números diferentes).

16. Entrar com dois números e imprimir o menor numero (suponha números diferentes).

17. Entrar com dois números e imprimi-los em ordem crescente (suponha números diferentes).

18. Entrar com dois números e imprimi-los em ordem decrescente (suponha números diferentes).

19. Entrar com três números e imprimir o maior numero (suponha números diferentes).

20. Entrar com três números e imprimi-los em ordem crescente (suponha números diferentes).

21. Entrar com três números e imprimi-los em ordem decrescente (suponha números diferentes).

22. Entrar com três números e armazená-los em três variáveis com seguintes nomes: maior, intermediário e menor (suponha números diferentes).

23. Efetuar a leitura de cinco números inteiros diferentes e identificar o maior e o menor valor.

	Anotações

24. Entrar com a idade de uma pessoa e informar:

- Se é maior de idade
- Se é menor de idade
- Se é maior de 65 anos

25. Entrar com nome, nota da PR1 e nota da PR2 de um aluno, imprimir nome, nota da PR1, nota da PR2, média e uma das mensagens: APROVADO, REPROVADO ou EXAME (a média é 7 para aprovação, menor que 3 para reprovação e as demais em exame).

26. Um comerciante comprou um produto e quer vende-lo com um lucro de 45% se o valor da compra for menor que R\$ 20,00; caso contrario, o lucro será de 30%. Entrar com o valor do produto e imprimir o valor da venda.

27. Um restaurante esta fazendo uma promoção semanal de descontos para clientes de acordo com as iniciar do nome da pessoa. Criar um algoritmo que leia o primeiro nome do cliente, o valor de sua conta e se o nome iniciar com as letras A, D, M ou S, dar um desconto de 30%. Para o cliente cujo nome não se inicia por nenhuma dessas letras, exibir a mensagem “Que pena. Nesta semana o desconto não é para seu nome; mas continue nos prestigiando que sua vez chegara”.

28. A policia rodoviária resolveu fazer cumprir a lei e cobrar dos motoristas a DUT. Sabendo-se que o mês em que o emplacamento do carro deve ser renovado é determinado pelo último número da placa do mesmo, criar um algoritmo que, a partir da leitura da placa do carro, informe o mês em que o emplacamento deve ser renovado.

29. Faça um programa que verifique e mostre os números entre 1.000 e 2.000 (inclusive) que, quando divididos por 11 produzam resto igual a 5.

30. Faça um programa que leia um valor n. inteiro e positivo, calcule e mostre a seguinte soma:

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

	Anotações

31. Faça um programa que mostre as tabuadas dos números de 1 a 10.

32. Faça um programa que leia cinco grupos de quatro valores (A, B, C, D) e mostre-os na ordem lida. Em seguida, mostre-os em ordem crescente e decrescente.

33. Uma loja tem 15 clientes cadastrados e deseja enviar uma correspondência a cada um deles anunciando um bônus especial. Faça um programa que leia o nome do cliente e o valor das suas compras no ano passado. Calcule e mostre um bônus de 10% se o valor das compras for menor que R\$1.000,00 e de 15% caso seja maior do que R\$ 1.000,00.

34. Uma companhia de teatro deseja dar uma série de espetáculos. A direção calcula que a R\$45,00 o ingresso serão vendidos 120 ingressos, e que as despesas serão de R\$200,00. Diminuindo-se R\$0,50 o preço dos ingressos espera-se que as vendas aumentem em 26 ingressos.

Faça um programa que escreva uma tabela de valores de lucros esperados em função do preço do ingresso, fazendo-se variar esse preço de R\$ 5,00 a R\$ 1,00 de R\$0,50 em R\$0,50. Escreva ainda o lucro máximo esperado, o preço do ingresso e a quantidade de ingressos vendidos para a obtenção desse lucro.

35. Faça um programa que receba a idade de dez pessoas e que calcule e mostre a quantidade de pessoas com idade maior ou igual a 18 anos.

36. Faça um programa que receba idade de 15 pessoas e que calcule e mostre:

- a quantidade de pessoas em cada faixa etária
- a percentagem de pessoas na primeira e na última faixa etária, com relação ao total de pessoas.

Faixa etária	idade
1	Até 15 anos
2	De 16 a 30 anos
3	De 31 a 45 anos
4	De 46 a 60 anos
5	Acima de 61 anos

37. Faça um programa que mostre a tabuada dos números de 1 a 10.

	Anotações

38. Uma loja utiliza um código V para transações à vista e P para transações à prazo. Faça um programa que receba o código e o valor de 15 transações. Calcule e mostre:

- valor total das compras à vista
- valor total das compras à prazo
- valor total das compras efetuadas
- valor da primeira prestação das compras à prazo, sabendo-se que essa serão pagas em três vezes.

39. Faça um programa que receba a idade, altura e peso de 25 pessoas e calcule e mostre:

- quantidade de pessoas com idade superior a 50 anos
- média das alturas das pessoas com idade entre 10 e 20 anos
- percentagem de pessoas com peso inferior a 40 quilos entre todas as pessoas analisadas

40. Faça um programa que receba a idade e o peso de sete pessoas. Calcule e mostre:

- a quantidade de pessoas com mais de 90 quilos
- a média das idades das sete pessoas

	Anotações

Capítulo 13 – Tratamento de Erros / Exceções

Try / Catch / Finally

As linguagens de programação orientadas a objetos, como o C# e o Java, contam com uma estrutura que nos permite tratar erros de um modo muito eficaz e simples.

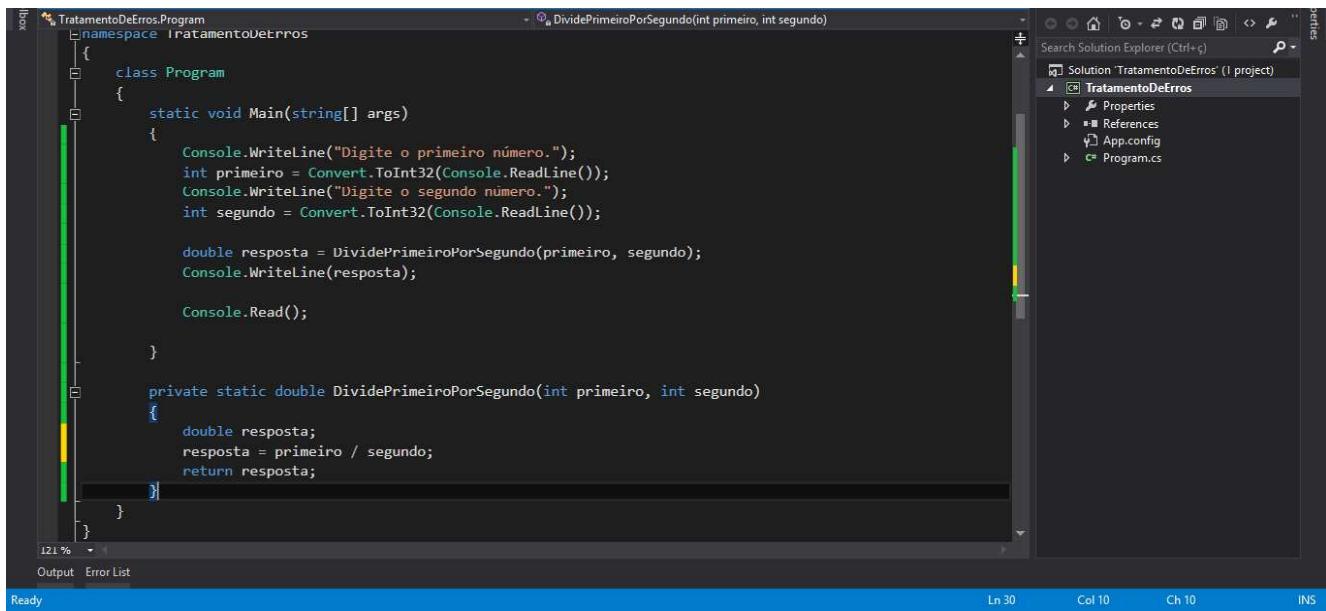
Foi dividido em três estruturas o que deveremos utilizar como prevenção de erros, sendo eles:

1. **Try¹¹**: Tudo pertencente ao bloco de código Try é o que possivelmente causará um erro, portanto prevemos que esse ponto é crítico e temos que nos precaver.
2. **Catch¹²**: Caso nossa previsão acerte e um erro aconteça, o que devemos fazer? É nesse bloco de código que você deve programar a solução do problema ou exibição ao usuário da maneira mais agradável.
3. **Finally**: Caso nosso código funcione ou não, esse bloco irá rodar. Esse bloco de código não é obrigatório.

O modo de usarmos esse conceito em algum momento de nosso código é simples, crie um projeto e nomeie-o de “TratamentoDeErros”.

Exemplo

Supondo que precisamos que o usuário nos informe dois números, e nós devolvamos a ele a divisão do primeiro número pelo segundo. Nossa código ficaria da seguinte maneira:



```
using System;
namespace TratamentoDeErros
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Digite o primeiro número.");
            int primeiro = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Digite o segundo número.");
            int segundo = Convert.ToInt32(Console.ReadLine());

            double resposta = DividePrimeiroPorSegundo(primeiro, segundo);
            Console.WriteLine(resposta);

            Console.Read();
        }

        private static double DividePrimeiroPorSegundo(int primeiro, int segundo)
        {
            double resposta;
            resposta = primeiro / segundo;
            return resposta;
        }
    }
}
```

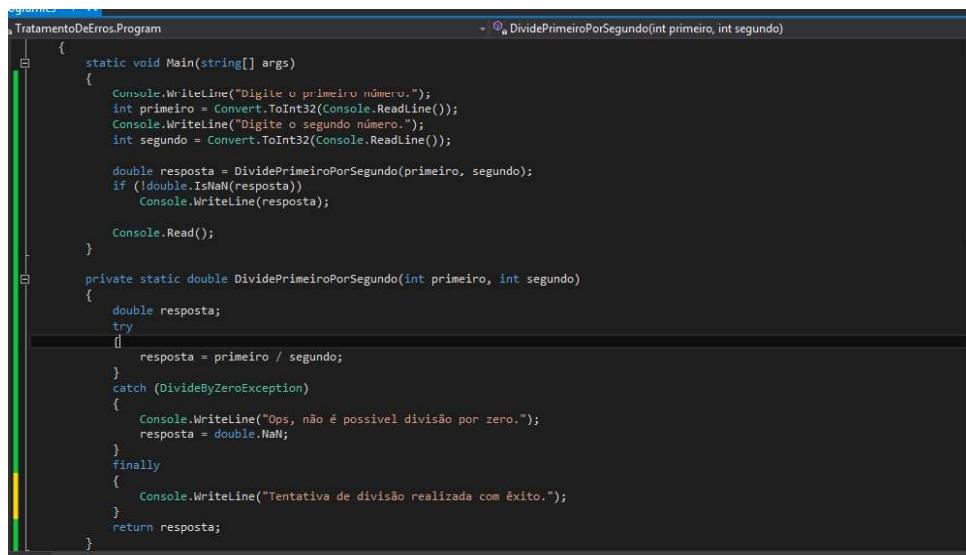
Em um primeiro instante, não teremos problemas em utilizar nosso programa, porém não supomos que alguém digitaria Zero (“0”) como segundo parâmetro. Como todos sabem a divisão por zero não é possível.

¹¹ Quando desejarmos, podemos criar o comando completo através do “Snippet” try+TAB+TAB.

¹² O Comando **catch(Exception)** não é possível de ser criado sem o “try”, por isso nada acontecerá através do snippet catch+TAB+TAB.

Anotações

Isso causará uma Exceção e nosso programa não funcionará da maneira que desejávamos. O modo de tratar esse problema é adicionarmos o ponto do nosso código que causará possivelmente essa exceção a um Try & Catch, conforme feito abaixo:



```
TratamentoDeErros.Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Digite o primeiro número.");
        int primeiro = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Digite o segundo número.");
        int segundo = Convert.ToInt32(Console.ReadLine());

        double resposta = DividePrimeiroPorSegundo(primeiro, segundo);
        if (!double.IsNaN(resposta))
            Console.WriteLine(resposta);

        Console.Read();
    }

    private static double DividePrimeiroPorSegundo(int primeiro, int segundo)
    {
        double resposta;
        try
        {
            resposta = primeiro / segundo;
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Ops, não é possível divisão por zero.");
            resposta = double.NaN;
        }
        finally
        {
            Console.WriteLine("Tentativa de divisão realizada com êxito.");
        }
        return resposta;
    }
}
```

Dessa maneira nosso código pode executar mais de uma instrução, dependendo de nossa entrada. Seja qual for estará seguro de erros de divisão por zero.

Assim como divisão por erro pode ser um problema, passar caracteres também pode. Devemos tentar prever ao máximo as possíveis ameaças ao funcionamento de nosso software. Pode parecer maçante no começo, porém após algum tempo isso torna-se muito útil e previne muitos problemas futuros.

System.Exception

Cada uma das exceções de um possível problema contém suas características, porém podemos não querer ser tão exatos em determinados momentos. Sendo assim existe uma SuperClasse da qual todas as exceções herdam, a classe do namespace **System** chamada **Exception**.

Exemplo

Qualquer exceção tem o seu tipo, e todos eles herdam de System.Exception. Para tratar de uma maneira bem genérica (e não recomendável) seu sistema, faça da seguinte maneira:

Anotações

```
TratamentoDeErros.Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Digite o primeiro número.");
        int primeiro = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Digite o segundo número.");
        int segundo = Convert.ToInt32(Console.ReadLine());

        double resposta = DividePrimeiroPorSegundo(primeiro, segundo);
        if (!double.IsNaN(resposta))
            Console.WriteLine(resposta);

        Console.Read();
    }

    private static double DividePrimeiroPorSegundo(int primeiro, int segundo)
    {
        double resposta;
        try
        {
            resposta = primeiro / segundo;
        }
        catch (Exception)
        {
            Console.WriteLine("Ops, não é possível divisão por zero.");
            resposta = double.NaN;
        }
        finally
        {
            Console.WriteLine("Tentativa de divisão realizada com êxito.");
        }
        return resposta;
    }
}
```

Qualquer erro, seja divisão por zero ou qualquer outro tipo de problema que pudesse ocorrer, como por exemplo estouro de tamanho da variável, cairia nessa exceção. Sendo genérica, porém tratada. Não poderemos dizer qual a causa exata, mas podemos prevenir a situação.

Uma maneira de prevenirmos nosso código de uma maneira bem inteligente, é tratar os erros conhecidos como algo conhecido, e os genéricos ficarem como segundo plano, conforme o exemplo:

```
TratamentoDeErros.Program
{
    int segundo = Convert.ToInt32(Console.ReadLine());

    double resposta = DividePrimeiroPorSegundo(primeiro, segundo);
    if (!double.IsNaN(resposta))
        Console.WriteLine(resposta);

    Console.Read();
}

private static double DividePrimeiroPorSegundo(int primeiro, int segundo)
{
    double resposta;
    try
    {
        resposta = primeiro / segundo;
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("Ops, não é possível divisão por zero.");
        resposta = double.NaN;
    }
    catch (Exception)
    {
        Console.WriteLine("Ops, ocorreu algo além do esperado.");
        resposta = double.NaN;
    }
    finally
    {
        Console.WriteLine("Tentativa de divisão realizada com êxito.");
    }
    return resposta;
}
```

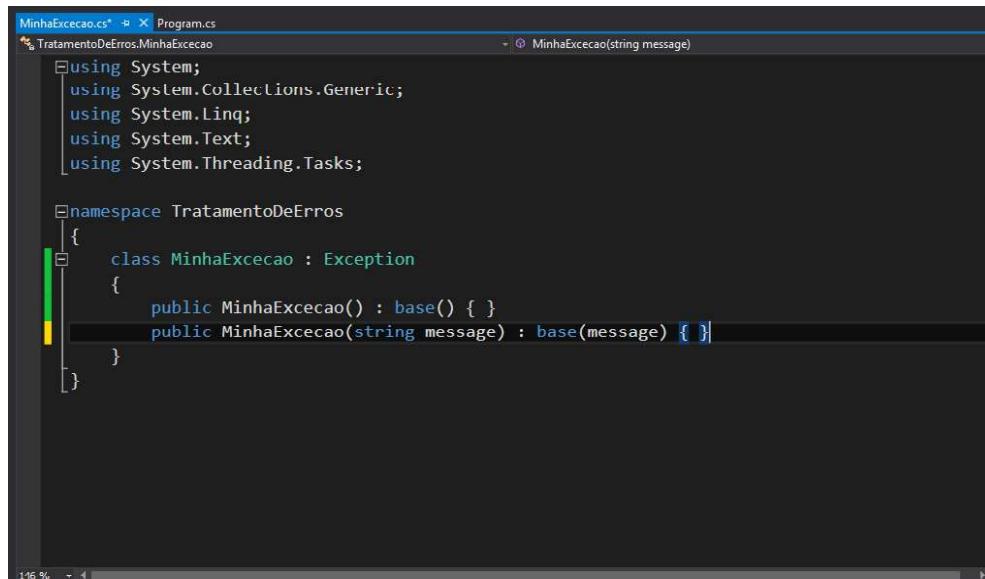
A encadeação de possíveis exceções é possível, basta ter um catch para cada uma.

Criar suas próprias Exceptions

Para criarmos nossas Exceptions, devemos criar uma Classe que **Herde de Exception**. Dessa maneira as propriedades necessárias serão levadas.

Exemplo

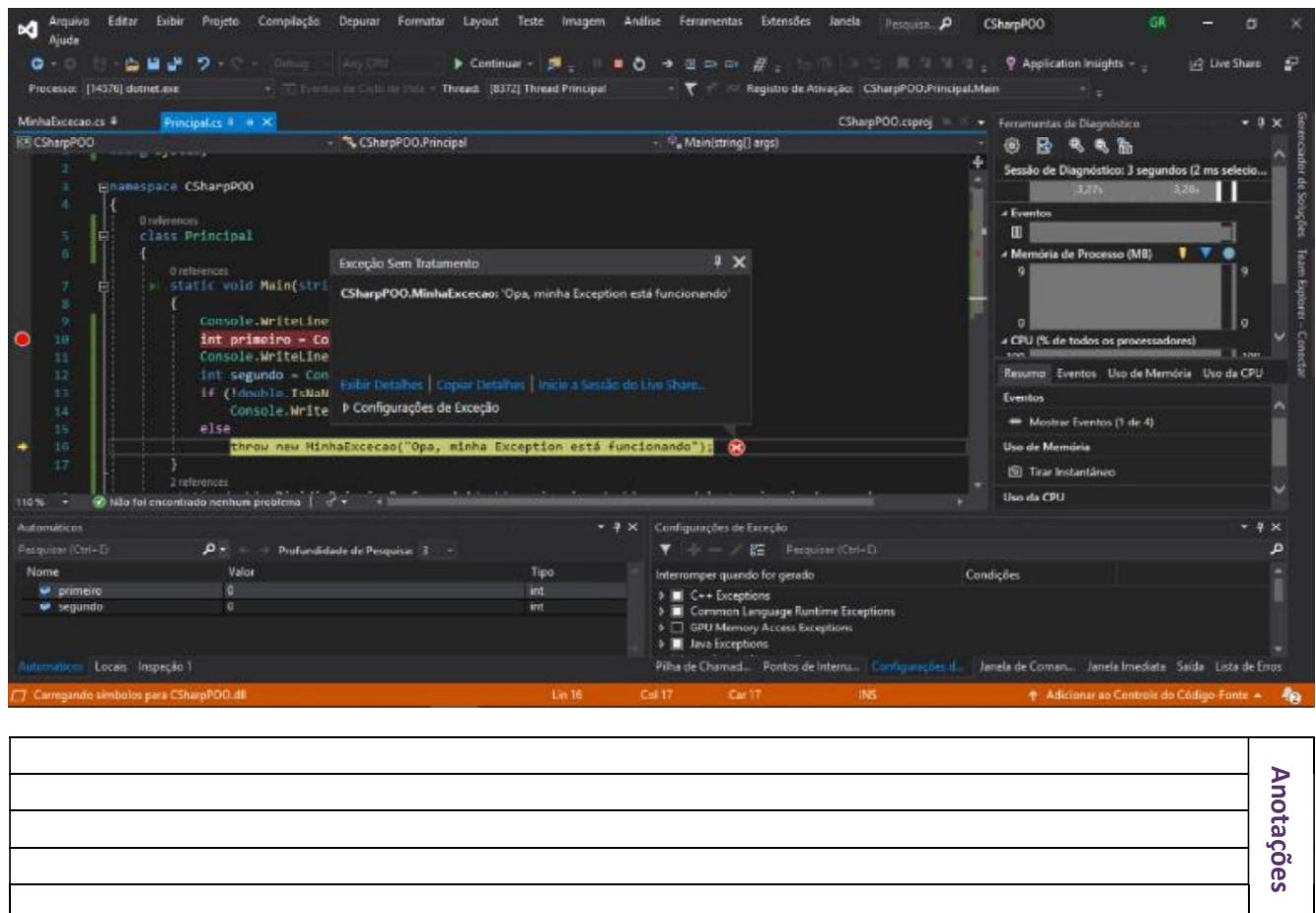
Vamos criar uma classe chamada MinhaExcecao e definir os métodos construtores dela da seguinte maneira:



```
MinhaExcecao.cs  Program.cs
TratamentoDeErros.MinhaExcecao
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TratamentoDeErros
{
    class MinhaExcecao : Exception
    {
        public MinhaExcecao() : base() { }
        public MinhaExcecao(string message) : base(message) { }
    }
}
```

Vamos supor que queremos exibir essa nossa Exception caso ocorra um erro em nosso método de dividir o primeiro número pelo segundo. O modo de uso e retorno disso será:



The screenshot shows the Visual Studio IDE interface during a debugging session. A tooltip is displayed over the code in the `Main` method of `Principal.cs`:

Exceção Sem Tratamento
CSharpPOO.MinhaExcecao: 'Opa, minha Exception está funcionando'

The tooltip also includes options: Exibir Detalhes, Copiar Detalhes, Iniciar Sessão do Live Share, and Configurações de Exceção.

The code in `Principal.cs` is as follows:

```
namespace CSharpPOO
{
    class Principal
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Int primeiro = " + primeiro);
            Console.WriteLine("Int segundo = " + segundo);
            if (segundo != 0)
                Console.WriteLine(primeiro / segundo);
            else
                throw new MinhaExcecao("Opa, minha Exception está funcionando");
        }
    }
}
```

The `primeiro` variable is set to 0 and the `segundo` variable is set to 0. The `primeiro` variable has a tooltip: "Nome: primeiro, Valor: 0, Tipo: int".

The status bar at the bottom shows: Carregando símbolos para CSharpPOO.dll, Lin 16, Col 17, Car 17, IN5, Anotações.

Dessa maneira nosso código será parado quando essa exceção ocorrer.

Exercícios

- 1) Complete:
 - a. Os comandos _____, _____ e _____ tem como objetivo o tratamento de possíveis erros que podem acontecer em nosso código.
 - b. O comando _____ não é obrigatório.
 - c. Todas Exceptions que criemos ou utilizemos em nosso comando catch devem obrigatoriamente ser uma classe _____ ou _____ da classe System.Exception.
- 2) Crie um novo projeto e nomeie-o de ExerciciosTratamentoErros que:
 - a. Em seu método principal faça um loop que vai de zero até um número digitado pelo usuário.
 - b. Se o número passado for menor que 10, exiba uma exception criada por você chamada “ValorMuitoBaixoException” com a mensagem “Ops, assim não é divertido”.
 - c. Se o valor for maior que 10.000 exiba “ValorMuitoAltoException”, com a mensagem “Ops, muita coisa - vai com calma”.
 - d. Caso não esteja nessa validação, exiba uma contagem de zero até o valor passado.
 - e. Caso valor passado não seja um inteiro exiba uma exceção genérica.

	Anotações