

[illegible]

## SUMÁRIO

Apresentação	4
<b>UNIDADE 1</b> - Ferramentas e preparação do ambiente	5
Objetivos	5
Ferramentas necessárias	5
Como uma aplicação .Net é compilada e executada	6
Opção 1: Visual Studio Code + Dotnet Cli	6
Passo 1 - Instalando o dotnet cli	7
Passo 2 - Instalando o Visual Studio Code	10
Opção 2: Visual Studio Community	12
Considerações Finais	15
<b>UNIDADE 2</b> - Conhecendo do .Net Framework	17
Objetivos	17
Porque eu preciso do .Net Framework?	17
.Net Framework	18
.Net Core	18
.Net Standard	18
Entendendo os tipos de projetos	19
Aplicação Console	20
Criando pelo Dotnet Cli	20
Criando pelo Visual Studio Community 2017	21
Explorando a estrutura gerada	22
Aplicação Class Library	23
Criando pelo Dotnet Cli	24
Criando pelo Visual Studio Community 2017	24
Aplicação Web	25
Criando pelo Dotnet Cli	25
Criando pelo Visual Studio Community	26

Explorando a estrutura de arquivos	27
Criando uma solução	28
<b>UNIDADE 3 - Conhecendo a linguagem</b>	30
Objetivos	30
Classes	30
Atributos	31
Propriedades	31
Métodos	33
Constantes	35
Enumerações	36
Namespaces	38
Arrays	39
<b>UNIDADE 4 - Avançando na Orientação a Objetos com C#</b>	42
Objetivos	42
Membros estáticos	42
Coleções	44
Herança	45
Sealed	48
Cuidados ao utilizar Herança	48
Interfaces	49
Classes Abstratas	50
Polimorfismo	51
<b>UNIDADE 5 - Extras</b>	54
Objetivos	54
Tratamentos de Exceções	54
Extension Methods	55
Programação assíncrona	56
Generics	59
Expressões Lambda e Linq	61

## Apresentação

Olá Aluno, meu nome é Alan Lira, sou formado em Sistemas de Informação, atualmente curso especialização em Arquitetura de Software Distribuído e trabalho há aproximadamente 6 anos com desenvolvimento de software, destes 6 anos aproximadamente 5 deles foram trabalhando com .Net, já trabalhei com outras tecnologias porém a plataforma .Net e C# ainda são as tecnologias que mais tenho aptidão.

Nesta disciplina busco passar uma visão geral de programação orientada a objetos com C#, utilizando como base o que o mercado atualmente está requisitando de um desenvolvedor C# e .Net, muitos conteúdos não estarão presentes neste material, porém acredito que com o que este material reuniu, seja possível ter a base teórica necessária para que você avance em outros conceitos mais avançados, o que falta em grande parte dos profissionais atualmente é o domínio de conceitos básicos que permeiam o desenvolvimento de software, e o objeto deste material e da disciplina é apresentar os recursos, sua finalidade e algumas vezes seus riscos. Espero que essa matéria atinja suas expectativas e caso tenha alguma dúvida, sugestão ou crítica fique a vontade para entrar em contato comigo:

- Email: [alan.lira1992@gmail.com](mailto:alan.lira1992@gmail.com)
- LinkedIn: <https://www.linkedin.com/in/alanlira92/>
- Github: <https://github.com/lira92>

## UNIDADE 1 - Ferramentas e preparação do ambiente

Neste capítulo veremos como instalar as ferramentas necessárias para que você comece a desenvolver aplicações utilizando C# e orientação a objetos, você precisará de um editor e da sdk do .net framework que é o framework utilizado para desenvolver em C#, ele provê classes básicas para a construção de suas aplicações, mas não se preocupe, veremos mais detalhes sobre ele no capítulo 2.

### Objetivos

Ao final deste capítulo é esperado que os seguintes conteúdos sejam absorvidos:

- Saber quais ferramentas são necessárias para começar a desenvolver com C#.
- Aprender o papel de cada ferramenta no seu dia-a-dia como desenvolvedor.
- Aprender minimamente como uma aplicação C# é compilada e executada.
- Aprender como é feita a instalação das ferramentas.

### Ferramentas necessárias

Durante muito tempo desenvolvedores C# precisavam desenvolver suas aplicações e executá-las em um ambiente Windows, porém, atualmente isso não é mais necessário, hoje podemos desenvolver e executar aplicações C# em várias plataformas, como por exemplo: Windows, MacOS e Linux, isso se dá ao fato da Microsoft (desenvolvedora e mantenedora do .Net) ter reescrito o .Net para ser multiplataforma e o nomeou de .Net Core, e não é só isso, o .net core atualmente é open source, o que significa que qualquer desenvolvedor pode ver e propor alterações no código do .net. Você pode ter acesso aos artefatos e códigos fonte do .net a partir desse repositório: <https://github.com/dotnet/core>.

As ferramentas necessárias para que você comece a desenvolver suas aplicações também ficaram bem mais flexíveis que no passado, onde você utilizaria o Visual Studio convencional quase que obrigatoriamente. Basicamente as ferramentas que você vai precisar será um editor, a sdk do .net e o runtime que é instalado junto com o sdk, porém como eu mencionei anteriormente as ferramentas necessárias atualmente estão mais flexíveis, então vou passar para você duas opções de conjunto de ferramentas: **Dotnet Cli + Visual Studio Code** e **Visual Studio 2017 Community**, mesmo que eu irei demonstrar a instalação do Visual Studio Code, na opção 1, você pode instalar outros editores e utilizá-los com o dotnet cli. Não se preocupe em escolher ou entender a fundo cada um das opções, vamos entender mais sobre as opções para que você chegue na conclusão de qual é melhor escolha para você.

## Como uma aplicação .Net é compilada e executada

É importante que você entenda como uma aplicação .Net é compilada e executada, pois, isso pode lhe ajudar a compreender benefícios e limitações da plataforma e pode te auxiliar no dia-a-dia como desenvolvedor.

Uma aplicação .net core para ser executada precisa de um runtime e as bibliotecas do framework, estes artefatos são fornecidos através do .net core runtime. Para executar uma aplicação web usando .net core também é necessário o runtime do ASP.NET, que é o componente do .net usado para desenvolver e executar aplicação web. o runtime do ASP.NET já inclui também o runtime do .net core. Por fim, para que aplicações .net sejam desenvolvidas é necessária uma sdk de desenvolvimento chamada de .net core sdk, ela já inclui o runtime do .net core, runtime do aspnet e ferramentas que são utilizadas durante o desenvolvimento.

A linguagem C# é compilada, isso significa que para que sua aplicação seja executada ela precisa passar por um processo de compilação, esse processo aparenta ser complexo e chato para nós desenvolvedores não é? Porém há inúmeros benefícios num processo de compilação, como por exemplo, o fato dele gerar artefatos otimizados para a execução e muitas vezes nesse processo de compilação são identificados problemas no código, já nos dando feedback imediato, diminuindo erros em tempo de execução, que são aqueles ocasionados no momento que a aplicação está executando uma determinada parte do código com defeito.

No caso do .net core, o processo de compilação é realizado pela sdk e produz arquivos resultantes da compilação do código fonte, o arquivo que geralmente contém seus artefatos de código tem a extensão *.dll*.

### Opção 1: Visual Studio Code + Dotnet Cli

Depois de ter entendido um pouco sobre como as coisas funcionam, vamos aprender como instalar as ferramentas Visual Studio Code e o dotnet cli, lembrando que essa não é única forma de desenvolver usando .net, porém possui características específicas que poderão fazer com que você acredite que esta é ou não a melhor escolha pra você.

Algumas características principais:

- Você pode instalar o dotnet cli e o visual studio code em várias plataformas, seja windows, linux ou mac.
- O tempo de instalação é menor em relação a segunda opção.
- Essas ferramentas ocuparão menos espaço no seu computador em relação a segunda opção.
- Algumas tarefas podem se tornar um pouco mais manuais do que na segunda opção.

Considerando essas características específicas, você conseguirá desenvolver, executar e debugar normalmente suas aplicações escolhendo essa opção, então sem mais delongas, vamos a processo de instalação.

## Passo 1 - Instalando o dotnet cli

Para que você consiga criar seus projetos e executá-los é necessário que você instale o dotnet cli, pois bem, vamos acessar o seguinte link:

<https://www.microsoft.com/net/download/dotnet-core/2.1>

A última versão do .net cli disponível é a 2.1 no momento em que este material foi produzido, porém pode ser que quando você esteja lendo isso, a última versão não seja mais esta, então você poderá acessar este link que lhe mostrará a versão atualizada:

<https://www.microsoft.com/net/download>

Após acessar o link de acordo com a versão, será exibida uma página semelhante a esta da imagem a seguir:

## .NET Core 2.1 downloads

Not sure what to download? Head to our main [Downloads page](#).

To use .NET Core 2.1 with Visual Studio, you'll need Visual Studio 2017 15.7 or newer. Make sure you've got the [latest version of Visual Studio](#).

	Build apps - SDK	Run apps - Runtime
<b>v2.1.5</b> <b>Current</b> LTS Released 2018-10-02 <a href="#">Release notes</a> Supports C# 7.3 Supports F# 4.5 Supports Visual Basic 15.5 Included in Visual Studio 15.8.6 ASP.NET Core IIS Module 8.2.1991.0	<b>SDK 2.1.403</b> <b>Windows</b> <ul style="list-style-type: none"><li>.NET Core Installer: <a href="#">x64</a>   <a href="#">x86</a></li><li>.NET Core Binaries: <a href="#">x64</a>   <a href="#">x86</a></li></ul> <b>macOS</b> <ul style="list-style-type: none"><li>.NET Core Installer: <a href="#">x64</a></li><li>.NET Core Binaries: <a href="#">x64</a></li></ul> <b>Linux</b> <ul style="list-style-type: none"><li>Package Manager Instructions: <a href="#">x64</a></li><li>.NET Core Binaries: <a href="#">x64</a>   <a href="#">ARM32</a>   <a href="#">ARM64</a>   <a href="#">x64 Alpine</a></li></ul> <b>Other</b> <ul style="list-style-type: none"><li>Checksums: <a href="#">Checksums</a></li></ul>	<b>Runtime 2.1.5</b> <b>Windows</b> <ul style="list-style-type: none"><li>ASP.NET Core/.NET Core: <a href="#">Runtime &amp; Hosting Bundle</a></li><li>ASP.NET Core Installer: <a href="#">x64</a>   <a href="#">x86</a></li><li>ASP.NET Core Binaries: <a href="#">x64</a>   <a href="#">x86</a></li><li>.NET Core Installer: <a href="#">x64</a>   <a href="#">x86</a></li><li>.NET Core Binaries: <a href="#">x64</a>   <a href="#">x86</a></li></ul> <b>macOS</b> <ul style="list-style-type: none"><li>ASP.NET Core Binaries: <a href="#">x64</a></li><li>.NET Core Installer: <a href="#">x64</a></li><li>.NET Core Binaries: <a href="#">x64</a></li></ul> <b>Linux</b> <ul style="list-style-type: none"><li>Package Manager Instructions: <a href="#">x64</a></li><li>ASP.NET Core Binaries: <a href="#">x64</a>   <a href="#">ARM32</a>   <a href="#">x64 Alpine</a></li><li>.NET Core Binaries: <a href="#">x64</a>   <a href="#">ARM32</a>   <a href="#">ARM64</a>   <a href="#">x64 Alpine</a></li></ul> <b>Other</b> <ul style="list-style-type: none"><li>Checksums: <a href="#">Checksums</a></li></ul>

Na coluna do meio, escolha a sua plataforma, e clique no link em frente a opção “.NET Core Installer”, se atente a arquitetura do seu computador também (x64 ou x86), caso você esteja usando linux acesse a opção “Package Manager Instructions” para consultar qual o processo de instalação nas diferentes distribuições suportadas.

Eu irei demonstrar a instalação no Windows, porém mesmo possuindo características bem específicas acredito que você não terá dificuldades em instalar o dotnet cli caso seu sistema operacional seja outro.

Clicando sobre a opção “.NET Core Installer” na seção windows, será feito o download do instalador na minha máquina:



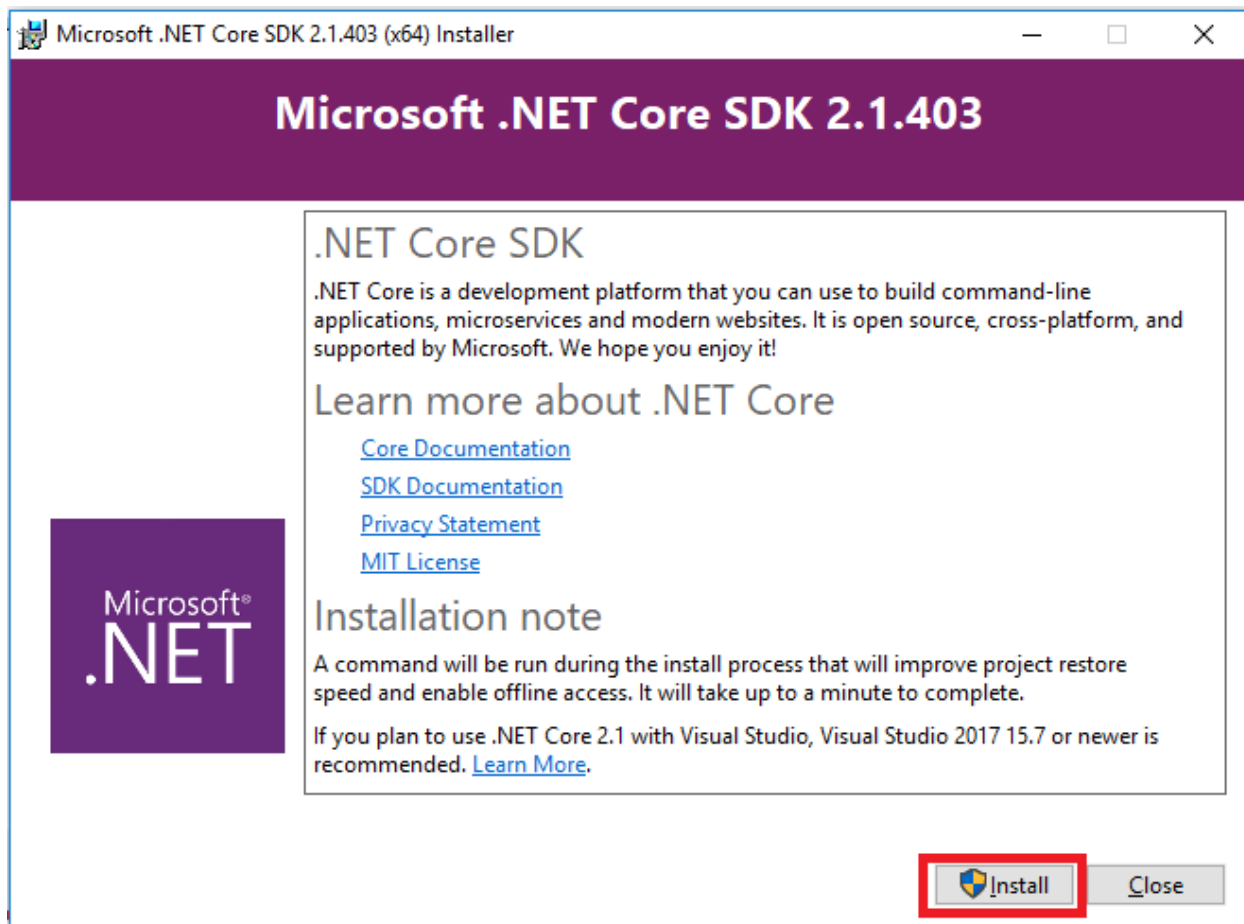
## Step 1: Run Installer

When your download completes, run the installer and the steps to install .NET on your machine.

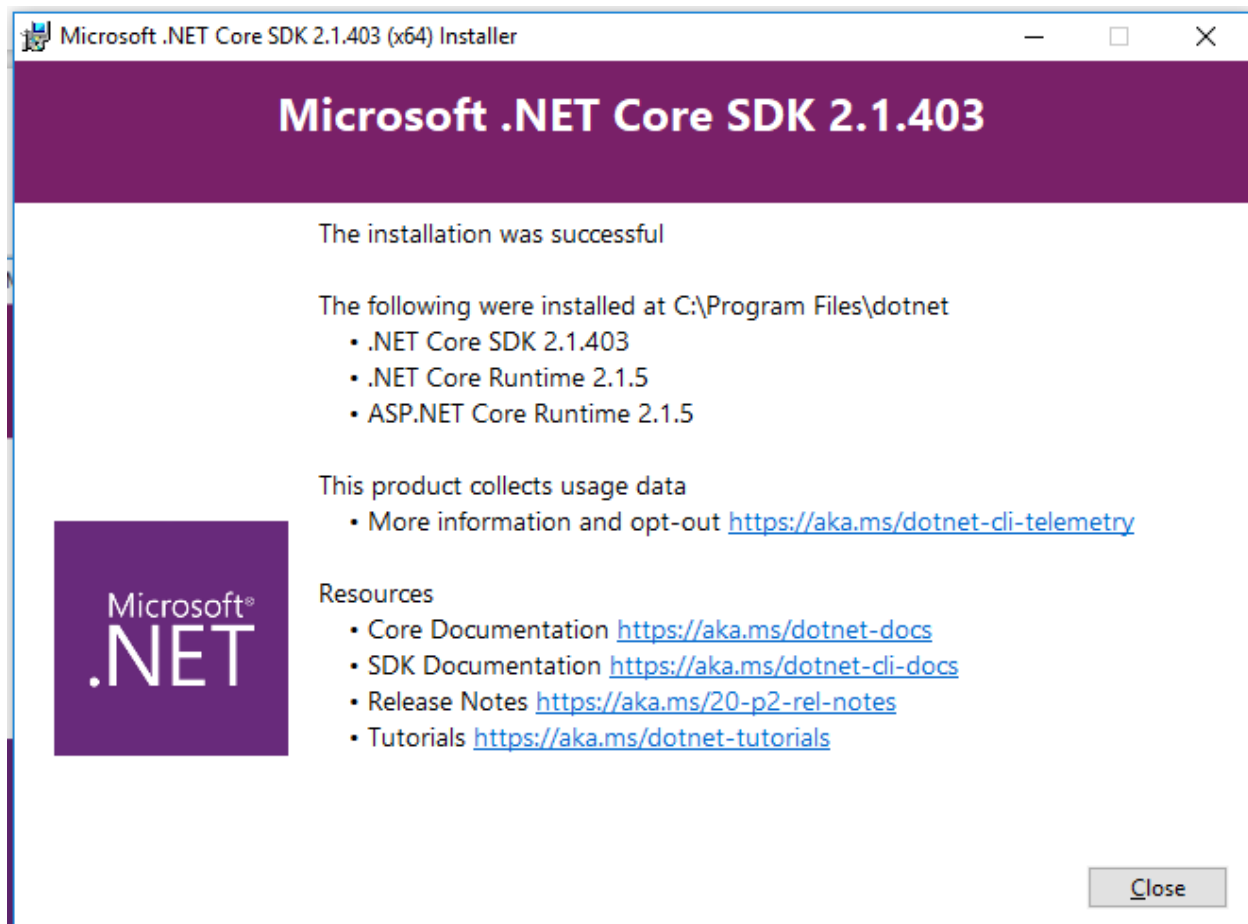


Abra o instalador e após iniciado, você deverá clicar em “Install” conforme a imagem a seguir:





A clicar sobre a opção o processo de instalação iniciará e após concluído deverá ser apresentada uma janela semelhante a essa:



Isso significa que o dotnet cli foi instalado com sucesso, para verificarmos isso abra uma janela cmd ou powershell e execute: dotnet --version

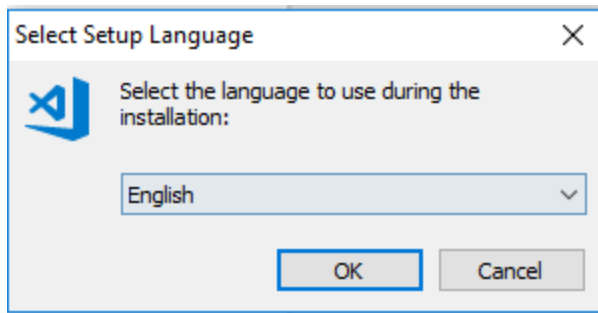
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

PS C:\Users\alan_> dotnet --version
2.1.403
PS C:\Users\alan_>
```

Se foi apresentada a versão que você instalou, então tudo ocorreu bem.

## Passo 2 - Instalando o Visual Studio Code

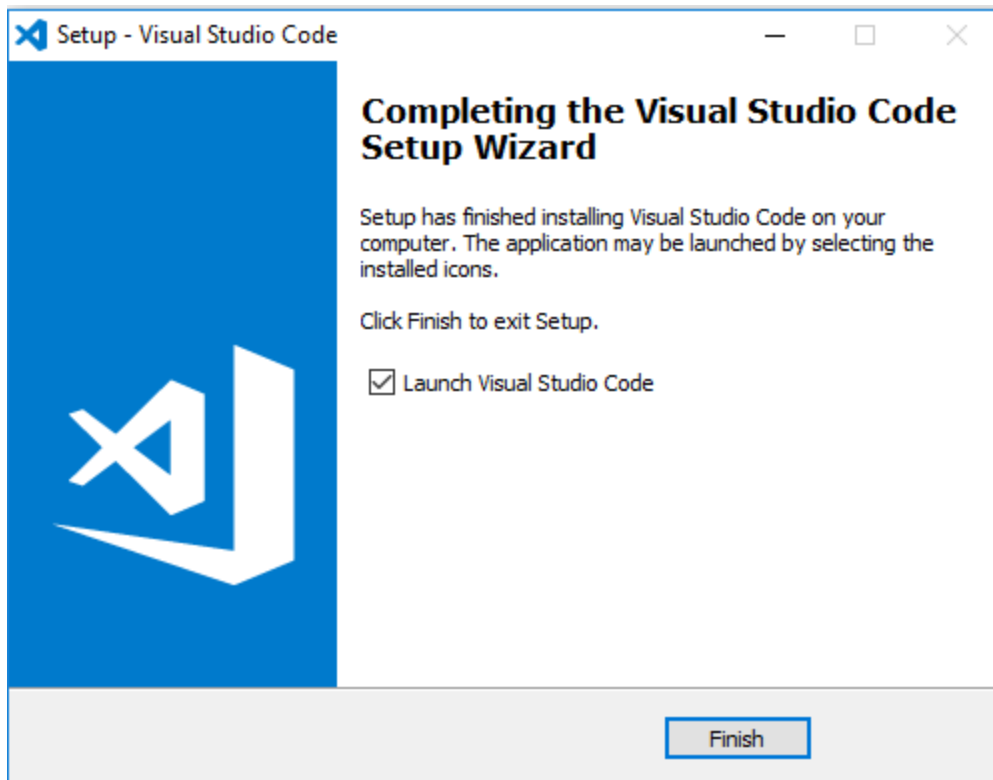
Agora que temos o dotnet cli, já podemos criar e rodar nossos projetos, porém vamos precisar de um editor para que possamos editar nossos arquivos com uma ótima experiência. para isso vamos acessar o link <https://code.visualstudio.com/download> para instalar o Visual Studio Code. Selecione a versão do seu sistema operacional e baixe o instalador, após o download execute o arquivo e deve ser requisitado que você selecione o idioma de sua preferência:



Selecione o idioma desejado e clique em ok. Após a opção ok ser clicada serão apresentadas algumas janelas de instruções de acordo com a sequência abaixo:

1. O instalador abrirá uma janela de boas-vindas e você deve clicar em “Next”.
2. Agora o instalador lhe pedirá para aceitar a licença, Selecione “I accept the agreement” e clique na opção “Next”.
3. O instalador lhe questionará sobre onde quer instalar o Visual Studio Code, selecione a pasta desejada e clique em “Next”.
4. Nesta etapa o instalador lhe informará que adicionará um item no menu de início do windows, altere o nome dele caso julgue necessário, ou selecione “Don’t create a Start Menu Folder” para que o menu não seja criado, ao final clique na opção “Next”.
5. Nesta etapa o instalador lhe questionará sobre algumas tarefas adicionais que ele pode executar, algumas opções se mostram interessantes, como por exemplo, adicionar opções no menu de contexto do windows explorer (menu apresentado quando clicado com o botão direito do mouse) para que facilite abrir pastas e arquivo com o visual studio code, marque as desejadas e clique em “Next”.
6. Agora o instalador lhe mostrará as ações que irá efetuar, se estiver tudo certo, clique em “Install” para iniciar o processo de instalação.
7. Uma barra de progresso será exibida e ao final você terá o Visual Studio Code instalado.

Se todas as etapas forem concluídas com sucesso a seguinte janela será exibida para você:



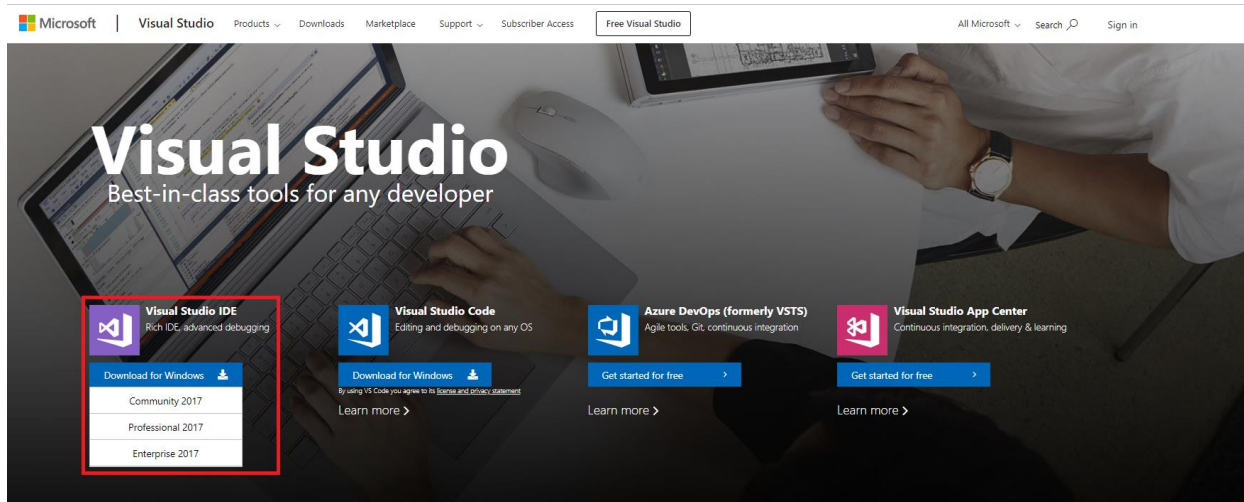
Agora você já possui as ferramentas necessárias para começar a desenvolver com C#.

### Opção 2: Visual Studio Community

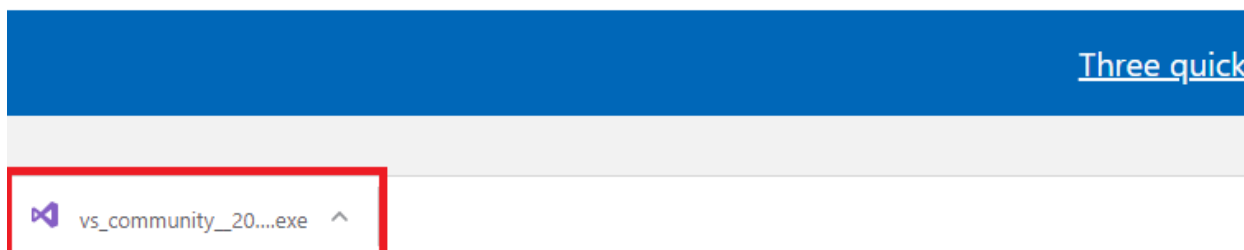
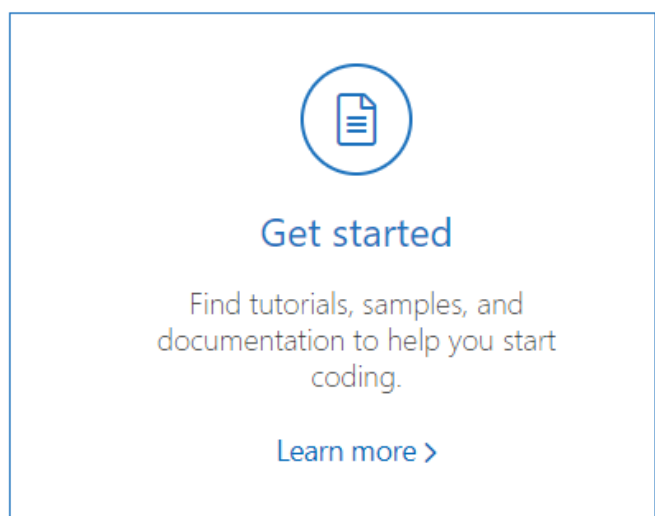
Essa opção possui como requisito a instalação de apenas uma ferramenta, porém, não se engane, todos os componentes necessários serão instalados junto com ela, ela também possui características específicas que poderão ajudar na sua escolha, veja algumas delas:

- Atualmente somente possui instalação para Windows.
- Você terá uma produtividade maior com ela.
- Ocupará mais espaço no seu computador do que a opção anterior.
- Consumirá mais recursos da sua máquina do que a opção anterior.

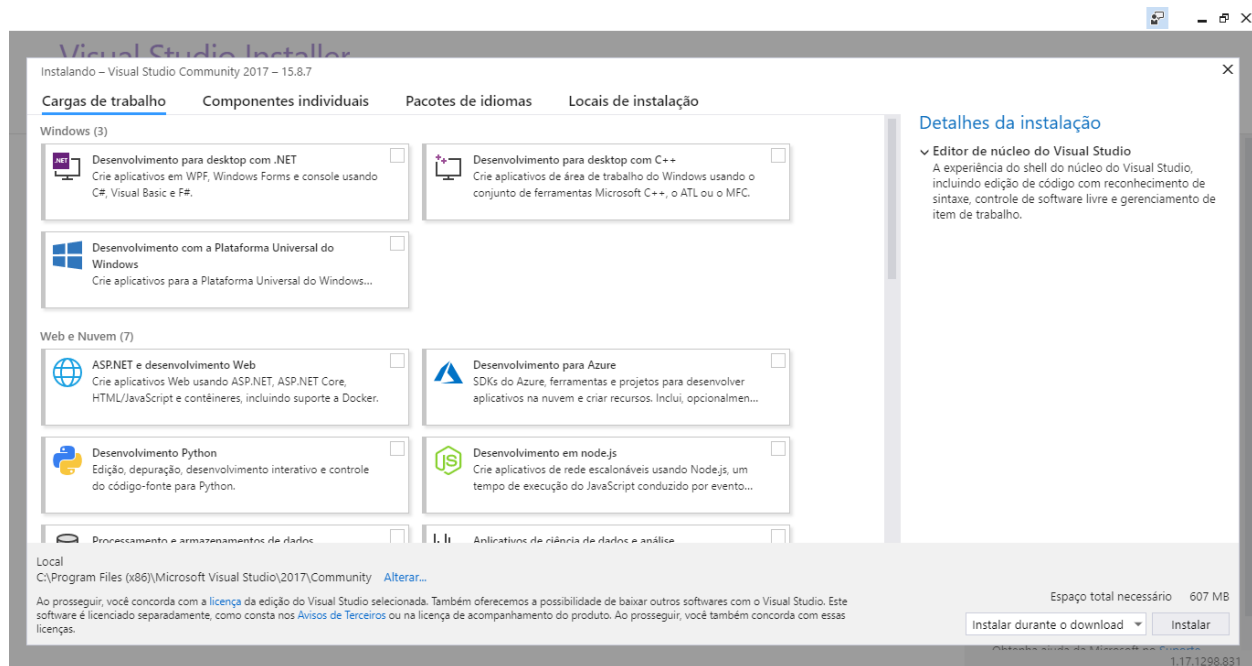
Após apresentar algumas características dessa opção, vamos a instalação do Visual Studio Community 2017, para isso acesse o link: <https://visualstudio.microsoft.com/> após a página ser exibida passe o mouse sobre o botão "Download for Windows" abaixo da opção "Visual Studio IDE" conforme a imagem baixo:



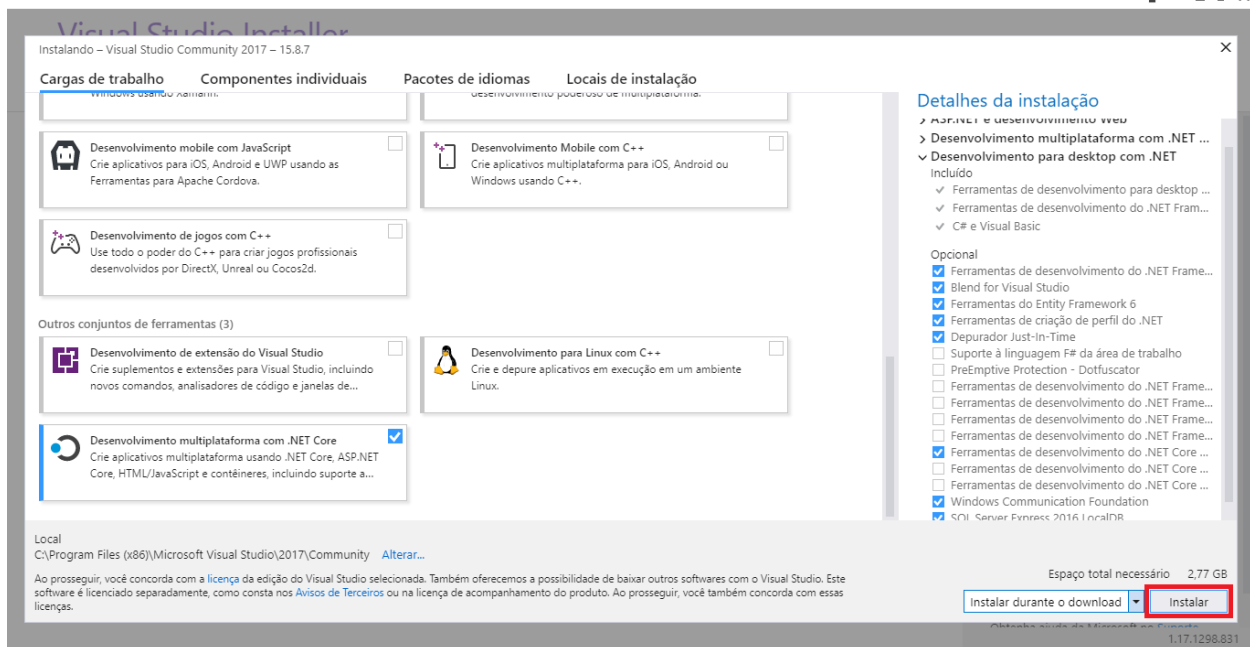
Clique sobre a opção Community 2017, uma nova página será carregada e o download será iniciado:



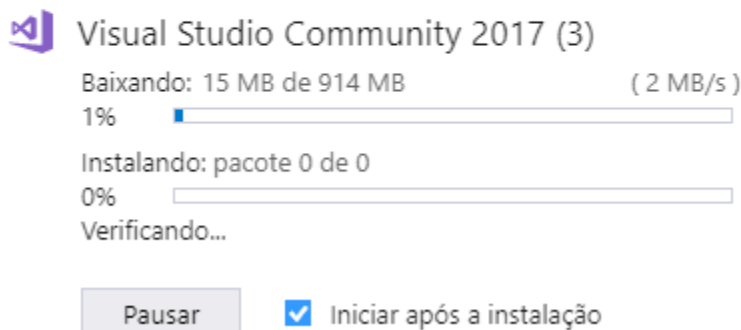
Após o término do download, abra o arquivo do instalador e ele irá fazer o download de algumas dependências, além de fazer algumas configurações. Ao final lhe apresentará a seguinte tela:



O Visual Studio é uma IDE poderosa que pode ser utilizada para desenvolvimento de diversos tipos de aplicações, seja em linguagens ou plataformas diferentes, por isso a instalação é modular, onde podemos escolher os recursos que serão necessários para o nosso cenário, o tamanho em disco ocupado dependerá de quais módulos você selecionar. Vamos selecionar 3 módulos, porém fique a vontade para selecionar outros se for do seu interesse, e caso não queira selecionar no momento, você poderá abrir a qualquer momento o instalador após a instalação para adicionar ou remover módulos. Os módulos que iremos selecionar serão: “Desenvolvimento para desktop com .NET”, “ASP.NET e desenvolvimento Web” e “Desenvolvimento multiplataforma com .NET Core”, sendo o último o principal, os demais vamos instalar pois poderá nos ser útil para abrir projetos criados em versões anteriores ou desenvolver aplicações desktop. Após selecionar os módulos desejados clique em instalar, conforme a imagem a seguir:



Uma janela de progresso será exibida:



Após a conclusão da instalação você terá instalado o Visual Studio Community 2017 e se a opção “Iniciar após a instalação” estiver marcada ele já será iniciado, agora você já está apto para nosso capítulo 2, fique a vontade para explorar suas funcionalidades.

## Considerações Finais

Neste capítulo, vimos como funciona basicamente o desenvolvimento C# atualmente, quais são as ferramentas necessárias e como o desenvolvimento se dá nessa linguagem. Tivemos uma

visão geral de como uma aplicação nessa plataforma é desenvolvida, preparada (compilação) e como ocorre a execução da mesma. Por fim, passamos por duas opções de preparação do ambiente que vamos utilizar durante os próximos capítulos, você pode escolher a que deseja de acordo com seu equipamento e necessidade, caso você esteja numa máquina windows, você pode instalar normalmente as duas opções caso queira experimentar qual mais lhe agrada.



## UNIDADE 2 - Conhecendo do .Net Framework

Neste capítulo entenderemos um pouco sobre o .net framework, pois é ele quem é responsável por prover a plataforma para que nossas aplicações escritas em C# sejam desenvolvidas e executadas. Acredito que você já esteja enjoado de tanta teoria e queira ir direto para o código, detesto desapontá-lo, pois, ainda veremos pouco código, porém é de extrema importância que você tenha uma visão geral sobre a situação atual do .net, as diferentes versões que ele possui e que tipos de aplicações podemos criar com ele, isso vai fazer com que você saiba qual versão usar dependendo da aplicação que você precisa.

### Objetivos

Ao final deste capítulo, espero que estes pontos estejam um pouco mais claros pra você:

- Porque eu preciso do .net framework
- Saber quais as versões disponíveis do .net framework
- Conhecer o .Net Framework (4.7)
- Conhecer o .Net Core (2.1)
- Conhecer o .Net Standard
- Conhecer os principais tipos de projetos que conseguimos criar com .net e C#
- Conhecer alguns arquivos importantes dentro de nossos projetos

Espero que você não tenha se assustado com a quantidade de objetivos deste capítulo, porém, após vermos estes itens, estaremos aptos para enfim entrar no universo C# conhecendo o poder proporcionado pelo .net framework.

Uma coisa interessante que talvez lhe chame a atenção caso ainda não saiba é que o C# não é a única linguagem suportada pelo .net framework, achou confuso? O .net framework provê a plataforma e bibliotecas básicas para nossas aplicações C# e outras linguagens, isso mesmo, o .net pode compilar e executar aplicações VB .Net e F# também, linguagens bem diferentes em relação à sintaxe e paradigma, fique a vontade para pesquisar sobre elas, pois nosso foco será desvendar um pouco da linguagem C#.

### Porque eu preciso do .Net Framework?

O .net framework proporciona bibliotecas básicas para usarmos em nossos projetos para que possamos nos focar em resolver realmente nossos problemas de negócio, isso inclui bibliotecas para manipular arquivos, conectar em uma base de dados, enviar um e-mail, consumir uma API e mais uma infinidade de outras tarefas, além disso ele possui o “motor” que fará com que nossa aplicação depois de compilada seja executada em um computador.

Outro conhecimento importante para aprendermos, é saber escolher qual versão do .net framework eu devo utilizar, sim, isso mesmo, o .net framework possui várias versões com diferentes características e limitações, isso pode ser muito confuso no início, porém é algo que foi necessário para que ele pudesse evoluir e ao mesmo tempo, aplicações já desenvolvidas continuarem funcionando.

## **.Net Framework**

Durante um bom tempo, tínhamos apenas uma linha de versões do .net, essa versão atualmente encontra-se na versão **4.7.2**, e ela possui suporte para desenvolver e executar aplicações somente em um ambiente Windows, essa versão ainda é mantida atualmente, principalmente pelo fato da existência de aplicações que já estão no mercado a um certo tempo. Vale ressaltar que caso você precise criar uma aplicação .net framework **4.7.2** ou alguma versão anterior você precisará utilizar o **Visual Studio** convencional, que no Capítulo 1 está representado pela versão **Community 2017**.

## **.Net Core**

Em 2016, a microsoft lançou uma versão reescrita do .net, que possui suporte para outras plataformas além de **windows**, como **linux** e **MacOS**, isso foi algo muito importante para os desenvolvedores .net, ainda mais que essa nova versão é totalmente **open-source**. Porém, como é uma versão reescrita, algumas limitações ainda existem, como por exemplo, no momento em que esse material é produzido ainda não é possível construir aplicações **desktop** com .net core, mas em compensação podemos criar aplicações **Web** e **Console** e executá-las em diferentes versões sem qualquer mudança em nosso código, legal, não é mesmo?

## **.Net Standard**

Agora você deve estar se perguntando porque uma terceira versão do .net foi necessária, mas na verdade o **.net standard** é um pouco diferente do que as versões apresentadas anteriormente, pois seu papel é proporcionar uma espécie de **“ponte”** entre essas versões e mais algumas plataformas.

Em nossas aplicações é muito comum compartilharmos e re-utilizarmos pedaços de códigos que servem a um propósito comum e nos ajudam a não ter que escrever o mesmo código novamente, criado por nós mesmos ou por outra pessoa, geralmente chamamos esses pedaços reusáveis de **bibliotecas de classes** (Em um universo orientado a objetos) ou simplesmente **bibliotecas**. Quando o .net core foi lançado, bibliotecas criadas usando o .net framework não poderiam ser usadas em projetos que usavam .net core e vice-versa, isso prejudicou muito a

**flexibilidade** de bibliotecas, pois agora elas deveriam ser escritas para uma versão específica, e por isso foi necessário criar o **.net standard**, que serve para criarmos bibliotecas que podem ser usadas tanto no **.net core** quanto no **.net framework**, isso significa que se você escrever uma biblioteca usando .net standard ela estará apta para ser **referenciada** em mais de uma versão do .net sem precisar sofrer qualquer alteração, isso aumenta a **interoperabilidade** de um artefato do código.

No link <https://docs.microsoft.com/pt-br/dotnet/standard/net-standard> você pode encontrar mais informações sobre o .net standard e consultar a tabela abaixo que informa quais versões do .net standard podem ser usadas em quais versões de outras plataformas:

## Suporte à implementação do .NET

A tabela a seguir lista as versões mínimas de plataforma compatíveis com cada versão do .NET Standard.

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework <sup>1</sup>	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
Plataforma Universal do Windows	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299
Windows	8.0	8.0	8.1					
Windows Phone	8.1	8.1	8.1					
Windows Phone Silverlight	8.0							

Fonte: <https://docs.microsoft.com/pt-br/dotnet/standard/net-standard>

Para te ajudar a entender essa tabela, vou pegar como exemplo a versão **1.3** do .net standard, caso você crie uma biblioteca com essa versão, ela poderá ser referenciada em qualquer projeto .net core com a versão **1.0** ou **superior** além de qualquer projeto .Net Framework **4.6** ou **superior**, outras plataformas também são listadas nas quais o .net standard é suportado.

## Entendendo os tipos de projetos

Agora veremos os tipos de projetos que podemos criar usando o .net e C#, estarei sempre dando prioridade para exibir projetos criados com **.net core** ou **.net standard** (No caso de bibliotecas), que é uma versão mais nova e que pode ser executada em várias plataformas. Atualmente

podemos criar uma infinidade de aplicações para os mais diversos cenários que possamos imaginar, podemos desenvolver aplicações **web**, aplicações **desktop**, aplicações **console** (úteis para desempenhar tarefas que não precisem de uma interface gráfica), aplicações **mobile** (através do Xamarin) entre outras. Alguns desses tipos de aplicações possuem limitações de ambientes e ferramentas específicas, dependendo geralmente do **Visual Studio** convencional em um ambiente **Windows**.

## Aplicação Console

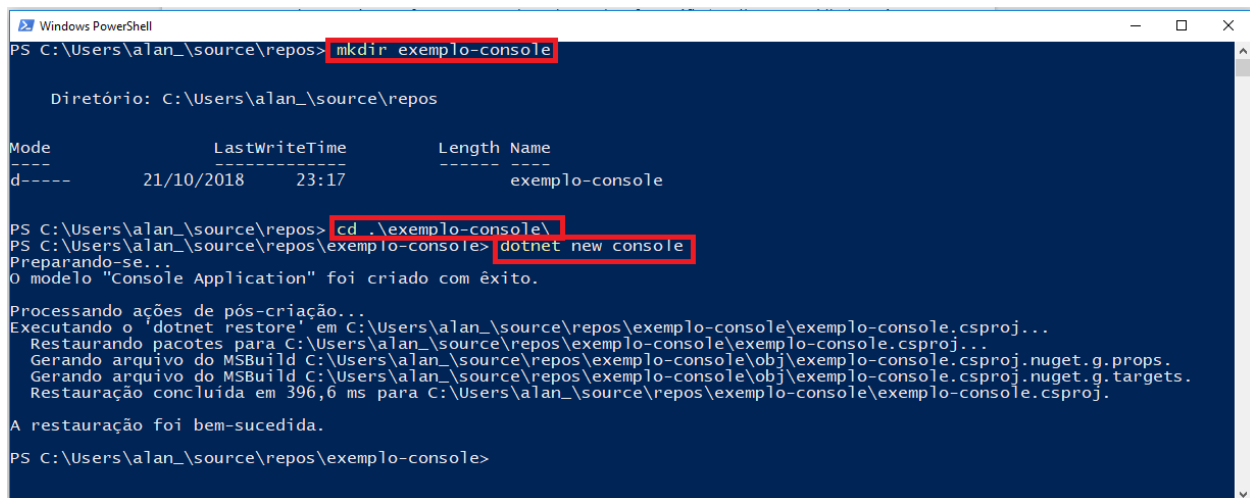
Vamos começar por um tipo de aplicação que possui uma estrutura muito simples e apresenta uma característica interessante de não possuir interface gráfica, ele é caracterizado por abrir uma janela de comando onde pode interagir com o usuário da aplicação. Esse tipo de aplicação pode ser usada para **automatizar** pequenas tarefas, como converter arquivos, enviar e-mails de forma automática entre outros cenários.

## Criando pelo Dotnet Cli

Pois bem, considerando que estaremos utilizando o .net core, caso você queira criar um projeto console usando o dotnet cli, você deve abrir uma janela de comando (cmd ou powershell), navegar até a pasta onde deseja que o projeto seja gerado e execute o seguinte comando:

```
dotnet new console
```

Abaixo uma imagem mostrando o uso desse comando:



```
Windows PowerShell
PS C:\Users\alan_\source\repos> mkdir exemplo-console

Diretório: C:\Users\alan_\source\repos

Mode                LastWriteTime         Length Name
----                -
d-----         21/10/2018   23:17             exemplo-console

PS C:\Users\alan_\source\repos> cd ..\exemplo-console\
PS C:\Users\alan_\source\repos\exemplo-console> dotnet new console
Preparando-se...
O modelo "Console Application" foi criado com êxito.
Processando ações de pós-criação...
Executando o 'dotnet restore' em C:\Users\alan_\source\repos\exemplo-console\exemplo-console.csproj...
Restaurando pacotes para C:\Users\alan_\source\repos\exemplo-console\exemplo-console.csproj...
Gerando arquivo do MSBuild C:\Users\alan_\source\repos\exemplo-console\obj\exemplo-console.csproj.nuget.g.props.
Gerando arquivo do MSBuild C:\Users\alan_\source\repos\exemplo-console\obj\exemplo-console.csproj.nuget.g.targets.
Restauração concluída em 396,6 ms para C:\Users\alan_\source\repos\exemplo-console\exemplo-console.csproj.
A restauração foi bem-sucedida.
PS C:\Users\alan_\source\repos\exemplo-console>
```

No meu caso, naveguei até a pasta **repos**, criei uma nova pasta chamada **exemplo-console**, naveguei até ela e executei o comando para criar o projeto, o meu projeto terá o nome de **exemplo-console**, pois o dotnet cli **herdou** o nome da pasta onde o comando foi executado, porém, opcionalmente eu posso definir um parâmetro **-n** indicando o nome do projeto que eu quero que seja gerado, por exemplo:

```
dotnet new console -n ExemploConsole
```

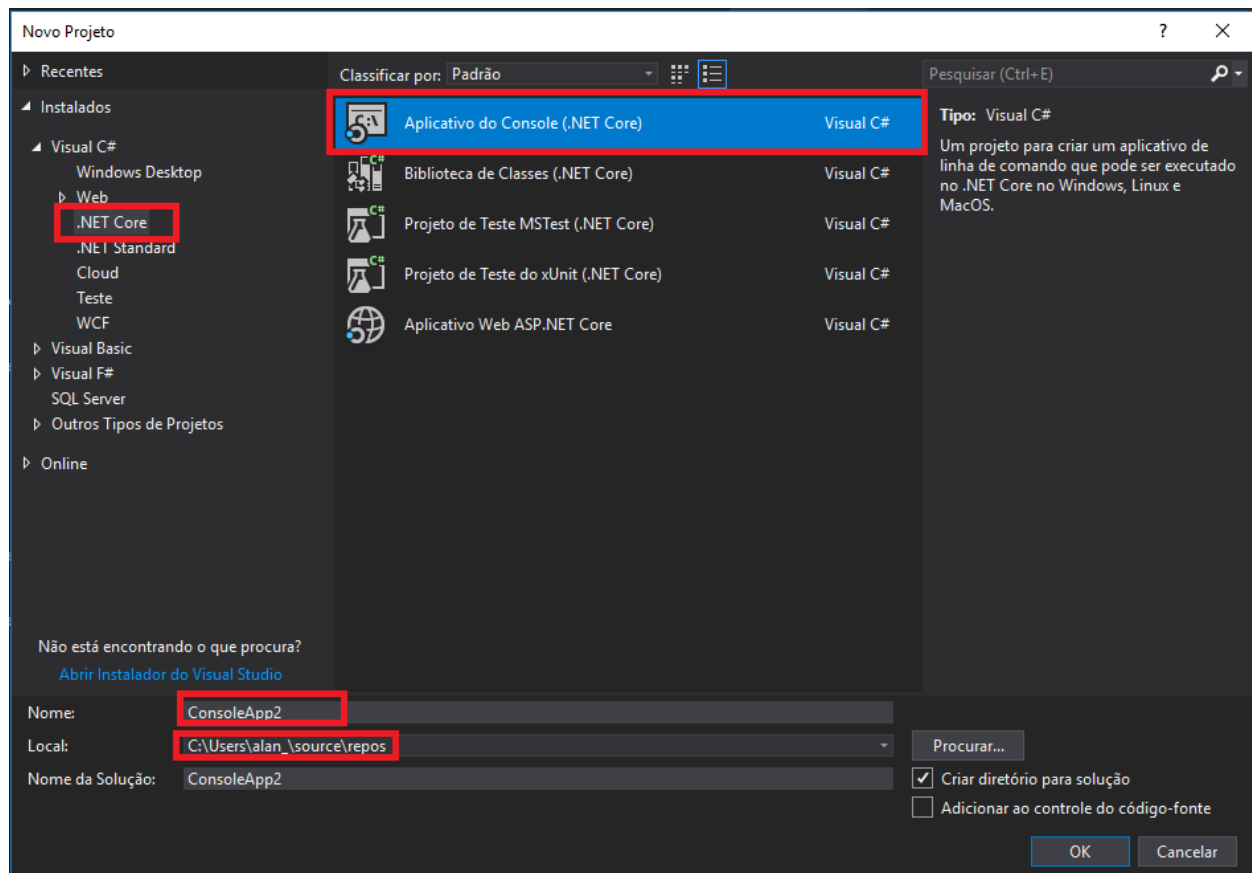
Além da informação que o projeto foi criado com sucesso, perceba que o dotnet cli executou algumas tarefas adicionais, ele me disse que executou um comando chamado “dotnet restore”, esse comando serve para restaurar bibliotecas de classes necessárias para nosso projeto, o dotnet restore faz o download dessas bibliotecas usando o **Nuget**, que é o gerenciador de pacotes utilizado no .net, ele possui um repositório de bibliotecas oficial chamado de **nuget.org**, e é de lá que o download é efetuado. Se você desejar explorar os arquivos gerados com o **Visual Studio Code** execute:

```
code .
```

Obs: Instale as extensões e complementos que o Visual Studio Code solicitar, para que você consiga executar o projeto

### Criando pelo Visual Studio Community 2017

Se você utilizar o **Visual Studio Community 2017** para criar a aplicação, o processo é ainda mais facilitado e visual, com o **Visual Studio Community 2017** aberto, acesse a opção **Arquivo > Novo > Projeto** ou utilize as teclas de atalho **Ctrl + Shift + N**, uma janela muito parecida com essa será exibida:

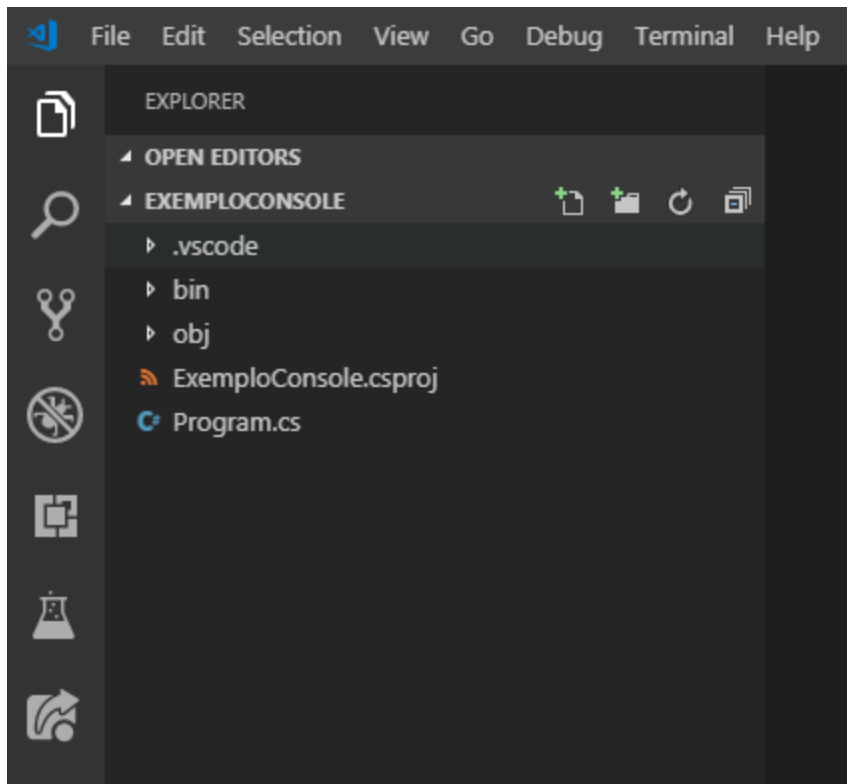


Você deverá selecionar a categoria **.NET Core** no menu à esquerda e depois **“Aplicativo do Console (.NET Core)”**, logo abaixo você tem a opção de alterar o nome do projeto e o local onde o mesmo será salvo, depois disso clique em **“OK”** e o projeto será criado.

Em ambos os cenários (Visual Studio Community 2017 e Visual Studio Code) apertando a tecla **F5** você executa o projeto, que ao ser executado imprimirá no console **“Hello World”** e encerrará.

### Explorando a estrutura gerada

Os arquivos gerados após a criação do projetos serão estes:



Caso você tenha criado usando o **Visual Studio Community**, os arquivos serão muito parecidos, temos um arquivo C# chamado **Program.cs** (.cs é a extensão dos arquivos c#) que é o primeiro arquivo C# a ser executado e no momento responsável por imprimir “Hello World” no console, além disso vemos um arquivo também muito importante que é o **ExemploConsole.csproj** arquivos com a extensão **.csproj** são arquivos que contém informações sobre o seu projeto, você terá apenas um arquivo **.csproj** por projeto, ele é responsável por dizer qual versão do .net estamos usando, quais bibliotecas estamos referenciando, entre outras informações. As demais pastas criadas com exceção da pasta **.vscode** (que não existirá, caso você esteja usando Visual Studio Community) são oriundas do processo de restauração e execução do dotnet, que no momento não precisamos nos atentar ao conteúdo delas.

### Aplicação Class Library

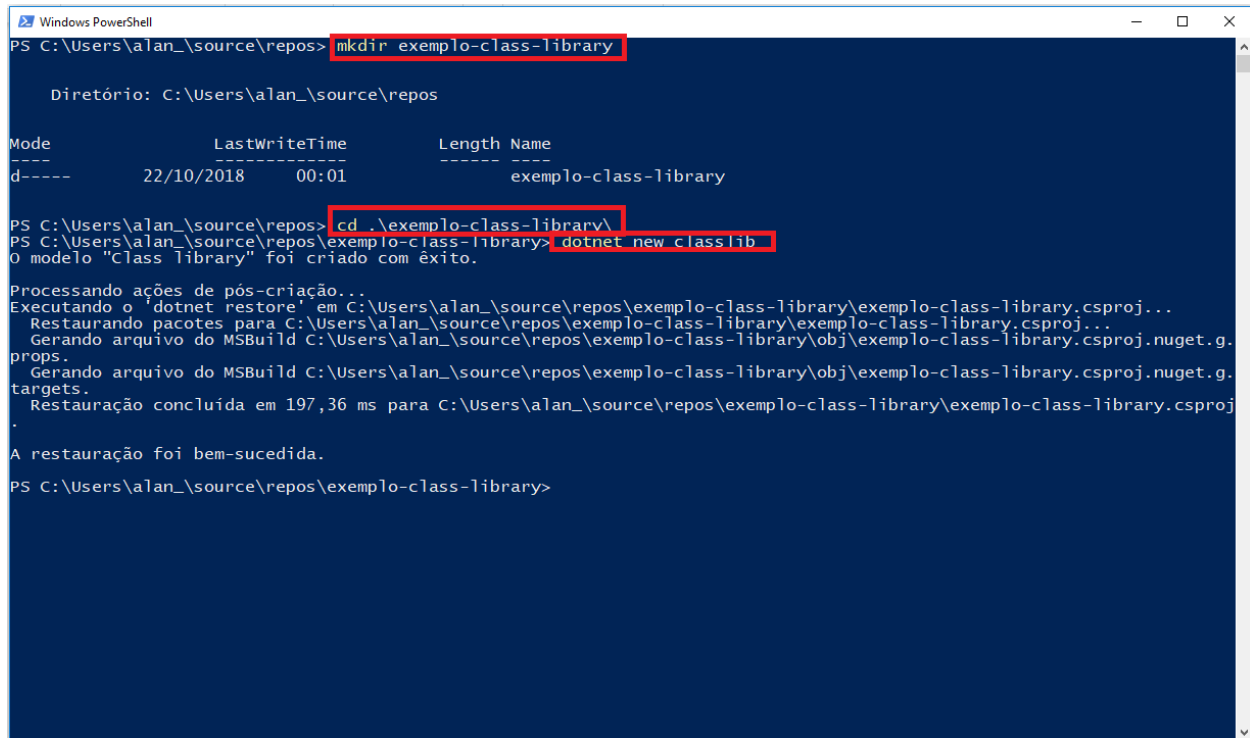
Uma aplicação do tipo **Biblioteca de classes** pode ser útil para que você **reutilize** pedaços de código que você já escreveu em outro momento e tem um comportamento comum, utilizando em mais de uma aplicação, por exemplo, imagine que você criou um algoritmo para validar se um telefone é válido, você pode ter a necessidade de validar telefone em vários dos seus projetos e vai achar desnecessário escrever o mesmo algoritmo em todos os projetos que deseja utilizá-lo, a solução para isso é criar uma **Biblioteca de classes** e referenciar em todos os projetos que será necessário utilizar. Vamos ver como criar um projeto de biblioteca de classes?

## Criando pelo Dotnet Cli

Para criar usando o dotnet cli, execute:

```
dotnet new classlib
```

Então você verá uma janela parecida com essa:



```
Windows PowerShell
PS C:\Users\alan_\source\repos> mkdir exemplo-class-library

Diretório: C:\Users\alan_\source\repos

Mode                LastWriteTime         Length Name
----                -
d-----          22/10/2018     00:01         exemplo-class-library

PS C:\Users\alan_\source\repos> cd .\exemplo-class-library\
PS C:\Users\alan_\source\repos\exemplo-class-library> dotnet new classlib
O modelo "Class Library" foi criado com êxito.

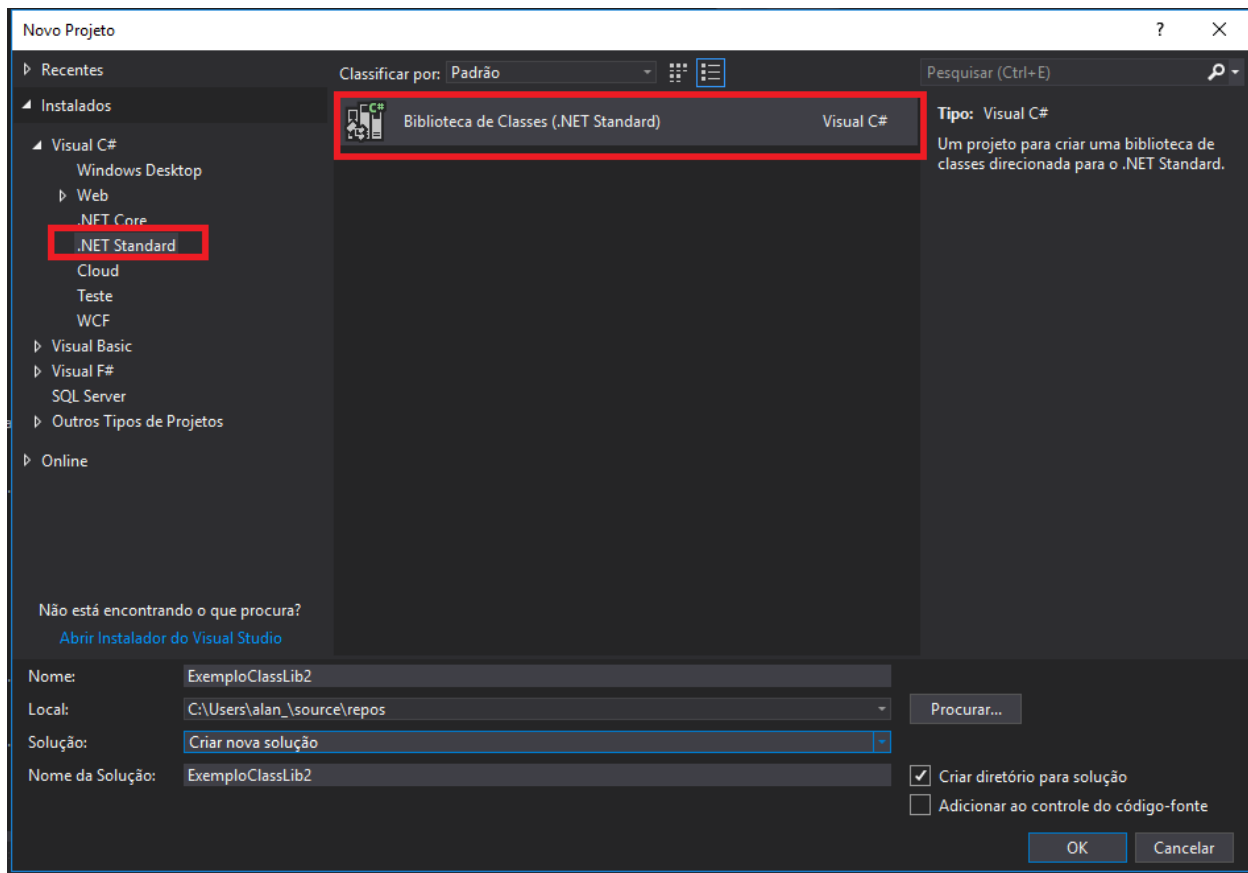
Processando ações de pós-criação...
Executando o 'dotnet restore' em C:\Users\alan_\source\repos\exemplo-class-library\exemplo-class-library.csproj...
Restaurando pacotes para C:\Users\alan_\source\repos\exemplo-class-library\exemplo-class-library.csproj...
Gerando arquivo do MSBuild C:\Users\alan_\source\repos\exemplo-class-library\obj\exemplo-class-library.csproj.nuget.g.props.
Gerando arquivo do MSBuild C:\Users\alan_\source\repos\exemplo-class-library\obj\exemplo-class-library.csproj.nuget.g.targets.
Restauração concluída em 197,36 ms para C:\Users\alan_\source\repos\exemplo-class-library\exemplo-class-library.csproj.
A restauração foi bem-sucedida.
PS C:\Users\alan_\source\repos\exemplo-class-library>
```

Algo que é importante ressaltar é que mesmo você não dizendo explicitamente ao dotnet cli, ele irá criar sua biblioteca de classes usando o **.Net Standard**, que é a versão que lhe proporcionará maior compatibilidade com as demais versões do .net.

## Criando pelo Visual Studio Community 2017

Para criar usando o **Visual Studio Community 2017**, acesse a opção **Arquivo > Novo > Projeto** ou aperte as teclas de atalho **Ctrl + Shift + N** e selecione o tipo de projeto igual ao da imagem:





Perceba que nesse caso estamos dizendo explicitamente para o Visual Studio criar uma biblioteca de classes usando o **.Net Standard**, mais adiante veremos como utilizar uma biblioteca de classes em outro projeto.

## Aplicação Web

Para criar uma aplicação web usando .net, é necessário a presença de classes e pacotes do **AspNet**, componente responsável por prover recursos para criação de aplicações web no ambiente .net. Há diversos templates disponíveis no **Dotnet cli** e no **Visual Studio Community 2017** para criação de aplicações web, porém todos basicamente tem uma estrutura parecida, com a adição de arquivos específicos para determinados tipo de aplicação web, veremos aqui como criar uma aplicação web usando o template mais simples disponível.

## Criando pelo Dotnet Cli

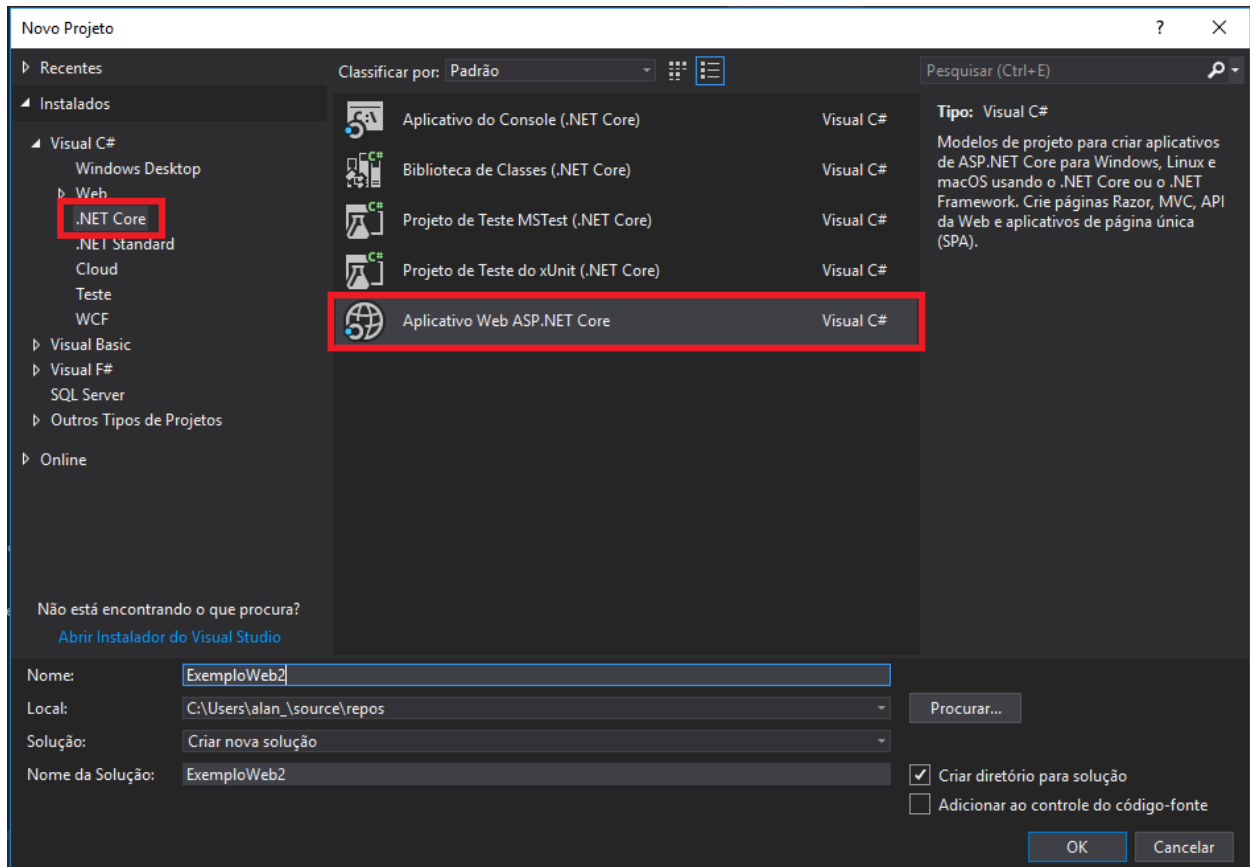
Para criar um projeto usando o dotnet cli execute o comando:

```
dotnet new web
```

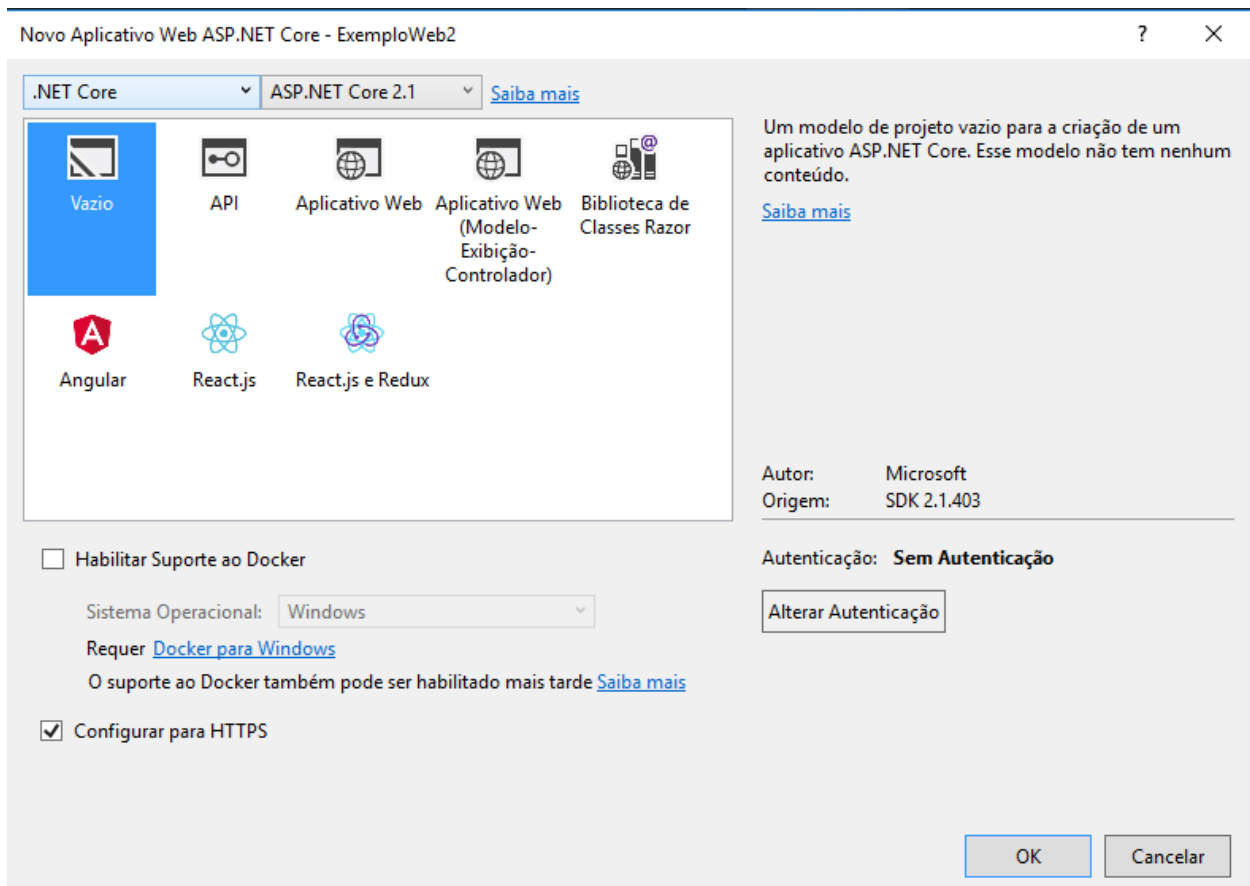
Serão exibidas algumas informações muito parecidas com o dos outros templates vistos anteriormente.

### Criando pelo Visual Studio Community

Para criar um projeto web pelo **Visual Studio Community 2017** acesse a opção **Arquivo > Novo > Projeto** ou aperte as teclas de atalho **Ctrl + Shift + N**, selecione o tipo de projeto conforme imagem abaixo:



Ao clicar em **OK**, diferente dos outros templates vistos anteriormente, você será questionado sobre qual tipo de projeto web você deseja criar, selecione o tipo **Vazio** conforme imagem abaixo:

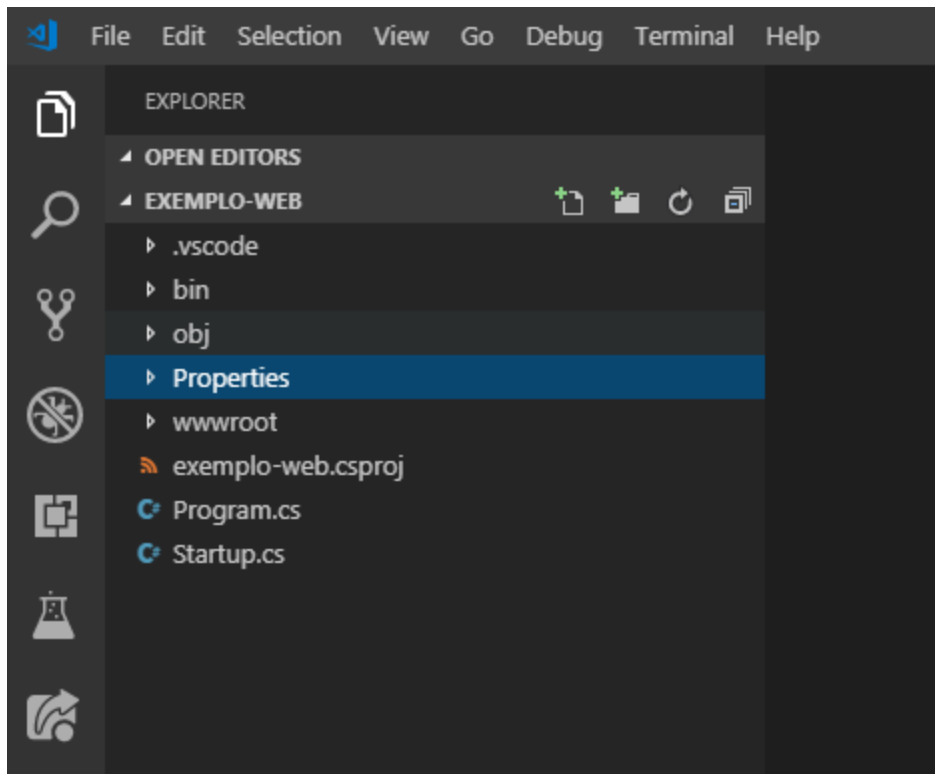


Clique em **OK** e seu projeto será criado, fique a vontade para explorar os demais tipos de projeto web.

Nos dois cenários de criação do projeto, ao apertar **F5** uma janela do seu navegador será aberta e estará escrito “Hello World”.

### Explorando a estrutura de arquivos

Os arquivos gerados basicamente serão estes (com algumas variações quando criados pelo visual studio community).



Aqui temos alguns arquivos novos em relação a nossa aplicação console, temos agora outro arquivo C# chamado **Startup**, que é executado logo após o **Program.cs** e que tem o papel de preparar e configurar algumas coisas da nossa aplicação web, além disso temos uma pasta **Properties** e uma pasta **wwwroot**, a pasta **Properties**, tem um arquivo chamado **launchSettings.json** responsável por configurar como nossa aplicação será executada, qual porta e qual endereço são algumas dessas informações, já a pasta **wwwroot** é uma pasta vazia que tem o intuito de receber arquivos estáticos de uma aplicação web, como por exemplo, imagens, fontes, arquivos de estilos, entre outros.

## Criando uma solução

Em muitas situações, quando desenvolvemos aplicações mais robustas, não teremos somente um projeto, correto? Imagine que você possui um projeto **web** que registra clientes e uma aplicação **console** que envia um e-mail para esses clientes assim que eles são cadastrados, além disso você compartilha pedaços de código nessas duas aplicações por meio de uma **biblioteca de classes**, seria complexo trabalhar com esses projetos em pastas e até janelas do editor separadas, não é? Para isso existe a possibilidade de você criar uma **solução de projetos**, que agrupa vários projetos em uma estrutura de pastas e é caracterizada por um arquivo na raiz da pasta do seu projeto com a extensão **.sln**. Se você utilizou o **Visual Studio Community** nos exemplos anteriores, você deve ter percebido que esse arquivo já é criado de forma padrão, porém você pode criar uma solução vazia, para depois ir adicionando os projetos que desejar,

usando o Visual Studio Community ao acessar a janela de criação de um novo projeto, existe a categoria **“Outros Tipos de Projetos”** que dentro possui a opção **“Solução em branco”** você pode iniciar seu projeto a partir da solução e ir adicionando seus projeto conforme a necessidade. Caso você queira criar uma solução usando o Dotnet cli, navegue até uma pasta e execute o comando:

```
dotnet new sln
```

Depois disso, crie seu projeto usando o comando `dotnet new [tipo do projeto]` e adicione na solução usando o comando:

```
dotnet sln add projeto/projeto.csproj
```

No exemplo que eu citei, num cenário hipotético, teríamos uma biblioteca de classes que teríamos que adicionar referência nos demais projetos, para fazer isso no dotnet cli você executaria:

```
dotnet add projeto/projeto.csproj
```

No Visual Studio o processo é mais **visual**, no nome do projeto clique com o botão direito, e então navegue até a opção **Adicionar > Referência**, e então escolha o projeto que contém sua biblioteca de classes. Mas o que adicionar uma referência significa? Você estará dizendo explicitamente que os projetos que referenciam outros, terão acesso a classes, métodos, entre outros artefatos públicos que o projeto referenciado expõe, criando assim uma relação de dependência.

## Considerações Finais

Neste capítulo vimos bastante teoria, porém conseguimos explorar alguns tipos de projetos que podemos ter usando .net e C#, além disso, vimos as versões que estão disponíveis do .net framework, e algumas características das mesmas que nos orientarão no momento de escolher qual versão utilizar. Em cada exemplo de projeto que passamos aprendemos um pouco sobre alguns arquivos que são muito importantes dentro dos nossos projetos e agora acredito que estamos aptos para finalmente começar a ver código C#.

## UNIDADE 3 - Conhecendo a linguagem

Neste capítulo começaremos de fato a explorar a programação utilizando a linguagem C#, veremos como criar classes, adicionar propriedades, atributos e métodos a elas, além de verificar algumas estruturas comuns da linguagem, como estrutura de decisão e repetição. Por fim veremos o que são namespaces e Enumerações.

### Objetivos

Ao final deste capítulo, espero você esteja mais familiarizado com a linguagem C# e que compreenda os seguintes temas:

- O que são e como criar classes C#
- Tipos do C#
- Como declarar atributos, propriedades e constantes
- Compreender algumas estruturas de decisão e repetição
- Como criar métodos
- Compreender o uso de Namespaces e enumerações.

A linguagem C# é uma linguagem **compilada**, como já falamos sobre, **fortemente tipada** e utiliza **tipagem estática**, isso significa que ao declarar uma variável em C# devemos dizer explicitamente seu tipo (tipagem estática) e após declarada não pode ter seu tipo alterado e só podemos fazer operações entre ela e outras variáveis ou valores do mesmo tipo (tipagem forte).

A linguagem C# tem também fortes características da **Orientação a Objetos**, assim como o Java, e proporciona simplicidade na grande maioria de suas estruturas, possibilitando a escrita de código de **fácil leitura** e **manutenção**. Atualmente o C# encontra-se na versão **7** com uma infinidade de recursos que foram adicionados durante os anos de sua existência.

### Classes

Assim como toda linguagem orientada a objetos o C# utiliza em abundância estruturas chamadas de **classes**. Classes são estruturas que podem apresentar **características** e **comportamentos**, vejamos um exemplo básico de uma classe C#:

```
public class Cliente
{
}
```

Neste exemplo temos uma classe pública, chamada de **Cliente**, que irá representar um conjunto de características e comportamentos que um cliente do mundo real possui, ela possibilitará que

eu crie vários **objetos** que representem diferentes clientes, cada um com suas características, fazendo uso de um **construtor** que neste caso existe implicitamente. Para que eu crie um novo cliente eu devo escrever:

```
Cliente meucliente = new Cliente();
```

## Atributos

Até o momento não há nada que diferencie um cliente de outro, vamos adicionar características a ele, utilizando nesse primeiro momento, os atributos:

```
public class Cliente
{
    public int codigo;
    public string nome;
    public decimal rendaMensal;
    public bool clientePremium;
}
```

Agora cada cliente que eu criar, poderei definir um **código**, **nome**, uma **renda mensal** diferente e dizer que ele pode ser ou não um **cliente premium**, de acordo com uma regra do meu negócio. Para criar um cliente de exemplo, escrevemos:

```
Cliente meucliente = new Cliente();
meucliente.codigo = 1;
meucliente.nome = "João Da Silva";
meucliente.rendaMensal = 2300;
meucliente.clientePremium = true;
```

## Propriedades

Temos então um objeto cliente que representa um cliente chamado **João Da Silva**, porém perceba que criamos todos nossos atributos como **públicos**, isso quer dizer que minha classe não tem controle do que são colocados em cada um dos seus atributos e nem quando são alterados, seria um problema um cliente criado com o código 1 seja alterado para outro código, sem qualquer critério correto? Para isso teríamos que passar a dar o controle de quando alterar as informações para nossa classe e para isso vamos transformar nossos atributos em **propriedades**:

```
public class Cliente
{
    public int Codigo { get; set; }
    public string Nome { get; set; }
    public decimal RendaMensal { get; set; }
    public bool ClientePremium { get; set; }
}
```

Não parece ter alterado muito nossa classe, não é mesmo? nossos atributos agora começam com letras maiúsculas (convenção para propriedades) e possuem um { **get; set;** } na frente das declarações, isso no momento está indicando somente que todas as nossas propriedades podem ser **lidas e alteradas**, mas como ainda continuam públicas, ainda não existem restrições de quem pode fazer isso, então vamos melhorar isso:

```
public class Cliente
{
    public int Codigo { get; private set; }
    public string Nome { get; set; }
    public decimal RendaMensal { get; set; }
    public bool ClientePremium { get; private set; }
}
```

Agora a palavra **set** do nosso atributo **Codigo** e do nosso atributo **ClientePremium** receberam uma palavra *private* antes, isso indica que, agora somente dentro da minha classe **Cliente** estas propriedades podem ser alteradas, mesmo ainda sendo possível lê-las em qualquer lugar onde este cliente estiver sendo manipulado. Nossa classe agora possui **mais controle** sobre suas propriedades, pelos menos propriedades que eu considero mais sensíveis (neste cenário hipotético), seguindo nosso exemplo, já que agora somente a classe **Cliente** pode definir valores para o **Codigo** e **ClientePremium**, em que momento podemos fazer isso? Seria ideal fazermos isso somente quando um cliente é criado, não é mesmo? para isso vamos definir um **construtor** personalizado:



```

public class Cliente
{
    public Cliente(int codigo, string nome, decimal rendaMensal)
    {
        Codigo = codigo;
        Nome = nome;
        RendaMensal = rendaMensal;
        ClientePremium = RendaMensal >= 5000
    }

    public int Codigo { get; private set; }
    public string Nome { get; set; }
    public decimal RendaMensal { get; set; }
    public bool ClientePremium { get; private set; }
}

```

Neste caso, no momento que criarmos um cliente é necessário atender os requisitos dele e fornecer os dados **codigo**, **nome** e **rendaMensal** em forma de parâmetros e o construtor se responsabilizará em definir estes valores nas nossas propriedades, sendo assim, um cliente com o código 1 após ser criado não poderá receber outro código, além disso implementamos uma regra de negócio que indica que um cliente será premium se a renda mensal inicial for **maior ou igual** que R\$ 5.000, e essa definição uma vez definida no construtor **não poderá** ser alterada por intervenções externas a nossa classe. Vamos verificar com criar um cliente agora?

```

Cliente meucliente = new Cliente(1, "João da Silva", 2500);

```

## Métodos

Imagine agora que a renda não é o único fator que define quando um cliente é premium, ele pode ser promovido à cliente premium de acordo com um valor ou quantidade de compras efetuadas, mas nesse caso nosso cliente já existe e será promovido, alterando o atributo **ClientePremium** de **false** para **true**, como **ClientePremium** é um atributo privado, a classe **Cliente** deve se responsabilizar por possibilitar essa ação, você deve se lembrar que citei que, uma classe pode ter características e comportamentos e é isso que vamos definir agora para nossa classe, um comportamento utilizando um **Método**:

```

public class Cliente
{

```

```

public Cliente(int codigo, string nome, decimal rendaMensal)
{
    Codigo = codigo;
    Nome = nome;
    RendaMensal = rendaMensal;
    ClientePremium = RendaMensal >= 5000
}

public int Codigo { get; private set; }
public string Nome { get; set; }
public decimal RendaMensal { get; set; }
public bool ClientePremium { get; private set; }

public void PromoverCliente()
{
    ClientePremium = true;
}
}

```

Agora um cliente existente pode ser promovido a cliente premium após ser criado, vejamos como poderíamos consumir este método:

```

Cliente meucliente = new Cliente(1, "João da Silva", 2500);
// Aqui meucliente.ClientePremium é false
meucliente.PromoverCliente();
// Aqui meucliente.ClientePremium passa a ser true

```

Perceba que disponibilizamos a funcionalidade de promover cliente por meio de um **método público**, e sendo público qualquer lugar onde o cliente for manipulado ele poderá ser promovido sem restrições, podendo ocasionar algo inesperado, então vamos assegurar que um cliente seja promovido somente se ele atender a certos critérios definidos no nosso negócio:

```

public bool PromoverCliente(decimal valorEmCompras)
{
    if (valorEmCompras >= 8000)
    {
        ClientePremium = true;
    }
}

```

```
        return true;
    }
    return false;
}
```

O cliente agora deve ter pelo menos R\$ 8.000 em compras efetuadas para ser promovido e como feedback para quem utilizar esse método estou retornando **true** em caso de sucesso ao promover o cliente e **false** em caso de falha, utilizando a estrutura de decisão **if** e neste caso não há necessidade de um **else** (fluxo executado quando um cliente não possuir valorEmCompras de no mínimo R\$ 8.000) pois após a palavra-chave **return**, nenhum outro código deste método é executado impedindo que um cliente que foi promovido retorne **false** para seu utilizador.

## Constantes

Com nossa classe cliente já bem definida, é um momento para refatorarmos um pouco o código e definirmos valores importantes de nossas regras como **constantes**, isso facilita a leitura do código e possíveis alterações serão mais simples:

```
public class Cliente
{
    private const decimal RENDA_MINIMA = 5000;
    private const decimal VALOR_MINIMO_EM_COMPRAS = 8000;

    public Cliente(int codigo, string nome, decimal rendaMensal)
    {
        Codigo = codigo;
        Nome = nome;
        RendaMensal = rendaMensal;
        ClientePremium = RendaMensal >= RENDA_MINIMA
    }

    public int Codigo { get; private set; }
    public string Nome { get; set; }
    public decimal RendaMensal { get; set; }
    public bool ClientePremium { get; private set; }

    public bool PromoverCliente(decimal valorEmCompras)
    {
        if (valorEmCompras >= VALOR_MINIMO_EM_COMPRAS)
```

```

    {
        ClientePremium = true;
        return true;
    }
    return false;
}
}

```

Neste caso, como o próprio nome sugere o valor de **constantes** não podem ser alterados depois de definidos e se você tentar fazer isso o compilador lhe alertará e não conseguirá efetuar a compilação.

## Enumerações

Continuando com nosso modelo de negócio, nossa classe cliente agora terá que possibilitar um novo nível de classificação de clientes e não somente ser ou não um **cliente premium**, esse novo nível se chamará **MASTER**, como prever isso no nosso código? Eu poderia criar outra propriedade booleana indicando quando um cliente é master, porém se tornaria trabalhoso identificar o nível de classificação de um cliente ou até mesmo alterá-lo, e se a cada novo nível eu tiver que criar uma nova propriedade, se tornará insustentável, não é mesmo? Para isso vamos utilizar um recurso chamado **Enumerações** que será definido logo abaixo:

```

public enum EClassificacao
{
    Convencional,
    Premium,
    Master
}
public class Cliente
{
    private const decimal RENDA_MINIMA_PREMIUM = 5000;
    private const decimal RENDA_MINIMA_MASTER = 8500;
    private const decimal VALOR_MINIMO_EM_COMPRAS_PREMIUM = 8000;
    private const decimal VALOR_MINIMO_EM_COMPRAS_MASTER = 10000;

    public Cliente(int codigo, string nome, decimal rendaMensal)
    {

```

```

        Codigo = codigo;
        Nome = nome;
        RendaMensal = rendaMensal;
        Classificacao = DefinirClassificacao();
    }

    public int Codigo { get; private set; }
    public string Nome { get; set; }
    public decimal RendaMensal { get; set; }
    public EClassificacao Classificacao { get; private set; }

    private EClassificacao DefinirClassificacao()
    {
        if (RendaMensal < RENDA_MINIMA_PREMIUM )
        {
            return EClassificacao.Convencional;
        }
        if (RendaMensal < RENDA_MINIMA_MASTER)
        {
            return EClassificacao.Premium;
        }
        return EClassificacao.Master;
    }
}

```

Nossa classe está ficando extensa não é mesmo? Vou deixar para que você imagine como definir o método **PromoverCliente**, veja que agora alteramos o **ClientePremium** para **Classificacao** e o tipo dele agora se tornou uma enumeração, isso significa que essa propriedade **deve conter** somente valores definidos na nossa enumeração, além disso criei um método **privado** (somente a classe cliente pode utilizá-lo) para definir qual a classificação nosso cliente irá se enquadrar.

**Obs:** Uma dica é que você pode definir valor inteiros para os valores da coleção, caso seja mais significativo, exemplo:

```

public enum EClassificacao
{
    Convencional = 0,
    Premium = 1,
    Master = 2
}

```

```
}
```

## Namespaces

Vamos aproveitar que a classe **Cliente** ficou um pouco extensa e vamos organizá-la de forma que fique separada da enumeração, e para isso vamos utilizar **Namespaces**, eles são uma parte importante da identificação e organização não só de classes, mas também de enumerações e outras estruturas, vou utilizar como exemplo a separação da classe **Cliente** em um **namespace** **Classes** e a enumeração **EClassificacao** em um namespace **Enumeracoes**, vejamos como fica:

```
namespace Enumeracoes
{
    public enum EClassificacao
    {
        Convencional,
        Premium,
        Master
    }
}

using Enumeracoes;

namespace Classes
{
    public class Cliente
    {
        public EClassificacao Classificacao { get; private set; }
    }
}
```

Em um caso real a classe **Cliente** e a enumeração **EClassificacao** estariam em arquivos distintos além de namespaces, mas como se trata de um exemplo didático, não vamos nos atentar a isso. Perceba que agora que separamos em namespaces, estou utilizando uma nova palavra chave, chamada **using**, isso é necessário para deixar explícito de onde a enumeração **EClassificacao** se encontra, pois namespaces podem ser compostos e geralmente transmitem a localização em pastas do artefato. Comumente os namespaces são definidos de forma que o primeiro

agrupamento seja o **nome da aplicação** e depois, **separados por pontos**, os **nomes das pastas** até chegar no artefato em questão, vejamos um exemplo:

```
namespace MinhaAplicacao.Vendas.Classes
{
    public class Cliente
    {
    }
}
```

Nesse caso, nossa classe cliente faz parte de um projeto chamado **“MinhaAplicacao”** e está dentro de uma pasta **“Classes”** que por sua vez se encontra numa pasta chamada **“Vendas”**.

## Arrays

Agora vamos voltar a nossa classe cliente e imaginar que precisamos receber os e-mails que nosso cliente possui e para isso vamos utilizar um **array**:

```
namespace Classes
{
    public class Cliente
    {
        public Cliente(int codigo, string nome, decimal rendaMensal,
            string[] emails)
        {
            Codigo = codigo;
            Nome = nome;
            RendaMensal = rendaMensal;
            Classificacao = DefinirClassificacao();
            Emails = emails;
        }

        public int Codigo { get; private set; }
        public string Nome { get; set; }
        public decimal RendaMensal { get; set; }
        public EClassificacao Classificacao { get; private set; }
        public string[] Emails { get; private set; }
    }
}
```

```
}  
}
```

Para definir uma propriedade do tipo array devemos utilizar o tipo seguido de colchetes, indicando que se trata de uma estrutura de **vários valores**. Agora, para que possamos utilizar os valores desta propriedade, para gerar uma lista de e-mails do cliente por exemplo, podemos utilizar uma estrutura de **repetição**:

```
namespace Classes  
{  
    public class Cliente  
    {  
        public Cliente(int codigo, string nome, decimal rendaMensal,  
string[] emails)  
        {  
            Codigo = codigo;  
            Nome = nome;  
            RendaMensal = rendaMensal;  
            Classificacao = DefinirClassificacao();  
            Emails = emails;  
        }  
  
        public int Codigo { get; private set; }  
        public string Nome { get; set; }  
        public decimal RendaMensal { get; set; }  
        public EClassificacao Classificacao { get; private set; }  
        public string[] Emails { get; private set; }  
  
        public string GerarListaDeEmails()  
        {  
            var listaEmails = "O cliente possui " + Emails.Length +  
" e-mails:\n";  
            foreach(var email in Emails)  
            {  
                listaEmails += email + "\n";  
            }  
            return listaEmails;  
        }  
    }  
}
```



```
}  
}  
}
```

Nesse novo método você pode ver o uso de uma estrutura de repetição: **O ForEach**, ele tem a função de repetir o bloco contido dentro de suas chaves, de acordo com a quantidade de elementos dentro de um array, além disso, ele armazena o elemento corrente em uma variável que chamamos de **iterador**, nesse caso representado pelo nome `email`, perceba também que estou apresentando uma nova palavra chave, a palavra **var**, ela está sendo usada para evitar que eu tenha que definir o tipo do **iterador** e o tipo da minha variável de **retorno** (`listaEmails`) responsável por armazenar minha representação da lista de e-mails, ela é muito utilizada pois simplifica a declaração de variáveis, porém, só pode ser utilizada em escopo de métodos ou construtores, como no código acima, não sendo possível declarar propriedades ou atributos de uma classe utilizando-se dela por exemplo, nesse caso temos que definir **explicitamente** o tipo.

### Considerações Finais

Neste capítulo vimos muitos recursos interessantes da linguagem C#, começamos pelas **classes** definindo uma chamada `Cliente` e fomos a incrementando com **características** e **comportamentos**, as características representadas primeiramente na forma de **atributos** e posteriormente em **propriedades** e os comportamentos representados pelos métodos. Para suportar nossos cenários de negócio vimos **estruturas de decisão**, **constantes**, **enumerações**, **namespaces**, além de ter uma breve introdução de **estruturas de repetição** que vamos voltar a vê-las quando chegarmos na parte de Coleções, no próximo capítulo continuaremos a explorar o uso de Orientação a Objetos em C#, porém veremos conceitos mais avançados.

## UNIDADE 4 - Avançando na Orientação a Objetos com C#

Neste capítulo iremos mais a fundo nos conceitos de orientação a objetos utilizando C#, agora que temos conceitos básicos da linguagem veremos estruturas mais complexas mas que nos darão ainda mais poder para desenvolvermos nossas aplicações, veremos estruturas estáticas, coleções, interfaces, herança, classes abstratas e por fim alguns modificadores.

### Objetivos

Ao final do capítulo você estará mais familiarizado com os seguintes conteúdos:

- Membros estáticos
- Coleções
- Interfaces
- Herança
- Classes abstratas
- Modificadores readonly e virtual

Como você pode perceber teremos um grande desafio durante este capítulo, por isso, teremos que nos concentrar ainda mais durante ele do que nos anteriores, pois se conseguirmos desvendar os conceitos que estão por vir será possível resolver problemas de negócio de uma forma elegante e de alto nível. Vamos lá?

### Membros estáticos

Você deve se lembrar que no capítulo anterior tínhamos uma classe cliente que possibilitava criarmos clientes diferentes e caracterizá-los utilizando nossas propriedades, isso quer dizer que nossa classe armazenava um **estado** de cada cliente que criávamos e a cada novo cliente (**new**) um novo **estado** era criado (completamente independente), nossos métodos **promoverCliente** e **DefinirClassificacao** utilizavam esse estado para acessar e alterar informações dele. Poderão acontecer casos onde queiramos compartilhar um mesmo estado ou até não depender dele (em métodos), indiferente de onde e quando invocarmos um método, atributo, propriedade e etc. Para este uso o mais indicado será utilizar **membros estáticos**.

No caso de uma classe **estática**, ela não poderá ser instanciada, ou seja, você não poderá utilizar um **new Classe()** por exemplo, pois ela já possuirá uma instância global criada pelo runtime, dessa forma todos os membros (atributos, métodos, etc) dessa classe deverão ser estáticos.

Já **métodos estáticos** poderão estar presentes em classes não estáticas também, porém, métodos estáticos só poderão utilizar membros estáticos, como propriedades e atributos, por exemplo. Métodos estáticos são indicados principalmente onde não há dependência do estado para sua execução, onde os únicos requisitos para sua execução são seus parâmetros e o

resultado é obtido através do retorno. Vamos ver um exemplo de uso para um método estático?

```
public static class ValidadorDeCpf
{
    public static bool Validar(string numero)
    {
        return numero.Length == 11;
    }
}
```

Perceba que indicamos que se trata de uma classe e um método estático utilizando a palavra **static**, neste caso qualquer declaração de método, atributo, propriedade e etc na nossa classe **ValidadorDeCpf**, deverá possuir a palavra **static**, perceba também que o uso do método **Validar** não depende de qualquer informação de estado, tudo que ele necessita está sendo provido por parâmetro. Para utilizarmos esse método estático teríamos o seguinte código:

```
var cpfvalido = ValidadorDeCpf.Validar("12345678910");
```

Nesse caso, não há presença da criação de uma instância, e nem seria permitido algo como **new ValidadorDeCpf()**. E no caso de propriedade/atributos estáticos?

```
public static class VisualizacoesDaPagina
{
    public static int Contador { get; set; }
}
```

```
VisualizacoesDaPagina.Contador++;
```

Nesse exemplo, nosso contador será iniciado com 0, pois é o valor default de um int e logo após ele é incrementado, passando a ser 1, em qualquer local da nossa aplicação que ele for acessado, ele terá o mesmo valor, e em caso de alteração ela será propagada para todos os locais que acessam essa informação, este é um caso onde nosso estado é global sendo compartilhado com todas as demais classes.

## Coleções

Um recurso muito utilizado no desenvolvimento em geral são as coleções, elas são estruturas de dados que armazenam uma quantidade dinâmica de itens de um mesmo tipo, ao contrário de arrays, que por sua vez, possuem tamanho fixo, coleções são mais flexíveis, permitindo adição de itens, remoção de itens, alteração de itens e etc. Existem diversas classes no C# que nos possibilitam trabalhar com coleções, cada uma possui seu propósito específico, vou utilizar a classe **List** para demonstrar um exemplo na mesma classe que já usamos no capítulo anterior, usando como requisito de negócio a adição de vendas a um cliente, vejamos:

```
using System.Collections.Generic;
public class Venda
{
    public string Descricao { get; set; }
    public decimal Valor { get; set; }
}
public class Cliente
{
    /** definição das constantes **/
    public Cliente(int codigo, string nome, decimal rendaMensal)
    {
        Codigo = codigo;
        Nome = nome;
        RendaMensal = rendaMensal;
        Classificacao = DefinirClassificacao();
        Vendas = new List<Venda>();
    }

    public int Codigo { get; private set; }
    public string Nome { get; set; }
    public decimal RendaMensal { get; set; }
    public EClassificacao Classificacao { get; private set; }
    public List<Venda> Vendas { get; set; }

    ...

    public void AdicionarVenda(Venda venda)
    {
        Vendas.Add(venda);
    }
}
```

```

public decimal CalcularTotalDeVendas()
{
    decimal total = 0;
    foreach (var venda in Vendas)
    {
        total = total + venda.Valor;
    }
    return total;
}
}

```

Veja que agora a classe cliente possui uma propriedade **Vendas** que é do tipo **List<Venda>**, isso indica que cada item adicionado a nossa lista deve ser do tipo da classe **Venda** declarada logo acima da classe cliente. Sempre que o tipo **List** for utilizado deve ser importado o namespace **System.Collections.Generic** utilizando a cláusula **using**. Para criar um comportamento que fizesse sentido a operação, criei um método **AdicionarVenda** que tem a responsabilidade de adicionar novas vendas para nosso cliente, além disso o método **CalcularTotalDeVendas** pode ser utilizado para consultar o total em compras que esse cliente possui e este faz uso da estrutura de repetição **ForEach** que havíamos visto no primeiro capítulo, mas nesse caso ele está sendo usado para percorrer nossa lista de vendas que é uma coleção, e nesse caso o iterador **venda** recebe a cada iteração um objeto do tipo **Venda**.

## Herança

Talvez um dos conceitos mais conhecidos da orientação a objetos é o conceito de Herança, porém ao mesmo tempo é um dos conceitos que é facilmente mal empregado em casos onde ele não seria necessário, portanto agora vamos tentar desmistificar o conceito de Herança, para que possamos fazer bom uso dela.

A herança possibilita um grande reuso de código proporcionando que uma classe **herde** de outra, mas o que significa uma classe **herdar** de outra? Uma classe pode **incorporar** atributos, propriedades e métodos de outra classe fazendo uso de **herança**, isso é bastante utilizado em casos onde há uma **especialização** de classes, imagine que estamos desenvolvendo uma aplicação para um banco e possuímos diferentes tipos de **Contas Bancárias**, **Conta Corrente**, **Conta Poupança**, **Conta Aplicação** e etc. Cada conta possui comportamentos e características, uma conta corrente, por exemplo, possui funcionalidade de limite, que é o recurso onde o banco lhe empresta um valor superior ao seu saldo na conta, porém a conta poupança não possui este recurso, então não seria a melhor escolha definirmos na mesma classe, mas se a separarmos

teremos muitos comportamentos repetidos, certo? Ora, todas as nossas contas possuem **saldo**, funcionalidade de **saque** e **depósito**, então qual seria a abordagem para que não seja necessário repetir todas essas funcionalidade nas classes especializadas? A resposta é **Herança**! Vejamos esse caso traduzido na linguagem C#:

```
public class Conta
{
    public decimal Saldo { get; protected set; }

    public void Depositar(decimal valor)
    {
        Saldo += valor;
    }

    public virtual void Saque(decimal valor)
    {
        if (valor <= Saldo)
        {
            Saldo -= valor;
        }
    }
}

public class ContaCorrente : Conta
{
    public decimal Limite { get; private set; }

    override public void Saque(decimal valor)
    {
        if (valor <= (Saldo + Limite))
        {
            Saldo -= valor;
        }
    }
}

public class ContaPoupanca : Conta
{
}
```

Muitas coisas novas em um pequeno pedaço de código não é? Mas vamos lá, de forma resumida nessa implementação eu possuo uma classe **base** chamada Conta, nela eu possuo os comportamentos e características que todas as minhas contas possuem, ou seja, um saldo, operação de saque e depósito, e possuo duas classes especializadas que **herdam** desta classe **base**, e isso é indicado pelos dois pontos (":") ao lado do nome de cada classe especializada.

Ainda na classe base, logo na primeira definição de propriedade, vemos uma palavra nova, a palavra **protected**, que é um nível de visibilidade diferente dos que já vimos anteriormente, que até então eram **private** e **public**, o nível **protected** só faz sentido quando há herança de classes, pois, como deve se lembrar **private** indica que somente a classe possui permissão para fazer uso do recurso, já **public** indica que qualquer outro tem permissão para tal, e o **protected** indica que somente a classe em que foi declarado e classes que **herdem** desta, possuem permissão, neste caso em específico estamos dizendo que somente a classe **Conta** e classes que herdem dela poderão alterar a propriedade **Saldo**, isso foi necessário para que fosse possível a classe **ContaCorrente** implementar um comportamento diferente para o método saque, fazendo com que fosse possível realizar um saque se o valor requisitado fosse superior ao saldo somado ao limite da conta, recurso este somente disponível na conta corrente.

No método **Saque** temos duas coisas novas, na implementação da classe base temo presença da palavra **virtual** e na implementação da classe **ContaCorrente** a palavra **override**, o uso desses dois recurso estão relacionados, um método que possui **override** no início de sua declaração indica que há uma **sobrescrita** de um método herdado, no nosso caso o método **Saque**, isso dá poder a classe especializada a mudar o comportamento do método definido na classe base, passando a ser ignorado a definição na classe base quando este método for executado para a classe que o **sobrescreveu**, já a palavra **virtual** presente na definição da classe base é um **modificador** e indica que é permitido que o método seja sobrescrito.

Caso seja necessário por alguma razão, a partir da classe especializada acessar informações ou comportamentos da classe **base**, isso pode ser feito utilizando exatamente esta palavra. Vejamos:

```
public class ContaPoupanca : Conta
{
    override public void Saque(decimal valor)
    {
        base.Saque(valor);
        // Comportamentos adicionais.
    }
}
```

## Sealed

Há certos casos onde seja interessante definir que uma classe não pode ser herdada, por exemplo, digamos que a classe **ContaCorrente** já possui a especialização suficiente e uma herança mal empregada poderia comprometer a regra de negócio, dessa forma poderíamos fazer uso da palavra **sealed** em sua definição fazendo com que nenhuma classe consiga herdá-la, dessa forma o compilador indicaria erro na tentativa de fazê-lo. Vejamos como seria sua definição:

```
public sealed class ContaCorrente : Conta
{
    public decimal Limite { get; private set; }

    override public void Saque(decimal valor)
    {
        if (valor <= (Saldo + Limite))
        {
            Saldo -= valor;
        }
    }
}
```

## Cuidados ao utilizar Herança

Existem diversos textos publicados sobre problemas ao se utilizar **Herança em Orientação a Objetos** e algo que gostaria de chamar a sua atenção é para cuidados ao se utilizar Herança em suas classes, pois algo que tem o intuito de reutilização de código pode tornar seu código cheio de “pontas soltas” e de implementações sem sentido. Algo que é sempre utilizado como exemplo para explicar Herança é o uso da expressão “é um”, por exemplo, **Cachorro é um Animal**, logo uma classe **Cachorro** herdaria de um classe **Animal**, porém levar isso como uma regra é um grande risco de estar modelando seu software de maneira errada e comprometer a **manutenibilidade**, imagine um exemplo onde você possui uma classe Ave:

```
public class Ave
{
    public void Andar() { }
    public void Voar() { }
}
```

Nesse caso, qualquer classe que representar uma ave deverá herdar dessa classe, certo? Como



definiríamos uma classe Pato ou uma classe Frango? Patos e Frangos são aves, porém não voam, e quando se utiliza herança todos os comportamentos e características são herdados, fazendo com que nossa implementação não represente de forma adequada o negócio que nos propomos a representar. O exemplo foi somente didático e não representa um caso real, porém o problema pode acontecer com as classes das nossas aplicações, a herança é um recurso muito poderoso da **Orientação a Objetos**, porém seu uso inadequado pode trazer resultados indesejados.

## Interfaces

Uma estrutura muito importante da orientação a objetos assim como as classes, são as interfaces, elas não possuem **nenhuma implementação concreta**, sua função é somente definir comportamentos que serão implementados por classes que se propuserem a implementar esta interface, vejamos um exemplo:

```
public interface IBoleto
{
    string GerarCodigoDeBarras();
}
```

A interface **IBoleto** não define como gerar um código de barras, nem há possibilidade de instanciar um objeto do tipo **IBoleto**, porém essa interface servirá como um contrato para as classes que a implementarem, o compilador obrigará que todas as classes que se propuserem a implementar a interface **IBoleto** definam o método **GerarCodigoDeBarras**, vejamos como implementar essa interface:

```
public class BoletoBancoDoSul: IBoleto
{
    public string GerarCodigoDeBarras()
    {
        // Código para gerar o código de barras.
    }
}
```

Nesse caso, a classe **BoletoBancoDoSul** é obrigada a definir o método **GerarCodigoDeBarras**, caso não defina o compilador acusará um erro e não efetuará a compilação. Você pode se perguntar, mas para que isso seria útil? Uma das vantagens no uso de interfaces é exatamente a obrigatoriedade da implementação dos comportamentos, sempre que uma classe que

implementa **IBoleto** for vista, saberemos que ela terá este método implementado, o que a torna mais previsível, porém há um motivo ainda mais nobre que vamos ver quando chegarmos na parte de **polimorfismo**.

## Classes Abstratas

As **classes abstratas** se assemelham em algumas características com as **interfaces**, classes abstratas **não podem ser instanciadas**, além do que ela pode definir **métodos sem implementação** que **obrigará** classes que herdarem dela a implementá-los, porém classes abstratas podem conter implementação de métodos e atributos, diferentemente das interfaces. Você lembra do nosso exemplo da conta bancária? Poderíamos defini-lo usando uma classe abstrata, veja:

```
public abstract class Conta
{
    public decimal Saldo { get; protected set; }

    public void Depositar(decimal valor)
    {
        Saldo += valor;
    }

    public abstract void Saque(decimal valor);
}

public class ContaCorrente : Conta
{
    public decimal Limite { get; private set; }

    public override void Saque(decimal valor)
    {
        if (valor <= (Saldo + Limite))
        {
            Saldo -= valor;
        }
    }
}

public class ContaPoupanca : Conta
{

```

```

    public override void Saque(decimal valor)
    {
        if (valor <= Saldo)
        {
            Saldo -= valor;
        }
    }
}

```

Nossa classe conta agora é uma **classe abstrata**, isso significa que não podemos instanciá-la diretamente, devemos utilizar suas classes especializadas para tal, o que parece coerente não é mesmo? Além disso a classe **Conta** implementa um comportamento comum e o comportamento de saque está marcado como **abstract** fazendo que as classes que herdem dela devam implementar esse método de forma específica.

## Polimorfismo

**Poliformismo** assim como herança é tido como um dos **pilares** da **Orientação a Objetos**, mas o que é polimorfismo e porque ele é tão importante? Polimorfismo é a possibilidade que a orientação a objetos nos provê para que um tipo de objeto possa tomar várias formas. Pareceu estranho? Vamos tentar melhorar isso, quando utilizamos **herança** ou quando implementamos uma **interface**, teremos a certeza que a classe que herdou ou implementou deverá conter todos os comportamentos ou atributos (no caso de herança) definidos na classe ou interface que ela herdou, correto? Isso possibilita que possamos manipular os objetos sem saber realmente o seu tipo específico, vamos pegar o exemplo da classe conta, imagine que temos uma classe responsável por exibir o saldo na tela:

```

public class ExibirSaldo
{
    public void ExibirEmTela(Conta conta)
    {
        var saldo = conta.Saldo;
        // Lógica para exibir o saldo
    }
}

```

Se você reparar, o método **ExibirEmTela** recebe como parâmetro um parâmetro do tipo **Conta**, isso mesmo, o tipo da nossa classe **base** e **abstrata** definida anteriormente, você pode se

perguntar, mas classes abstratas não permitem instâncias? **Classes abstratas** e **interfaces** não permitem que seja criado diretamente um objeto do seu tipo, porém eu posso utilizar para identificar um **parâmetro** como no exemplo, fazendo com que qualquer tipo que implemente a classe **Conta** se torne elegível para ser passado como parâmetro, isso faz com que não seja necessário repetição de código para executar operações comuns entre as várias classes que herdam da classe **Conta**.

Eu também posso me beneficiar das facilidades do **polimorfismo** utilizando **interfaces**, na seção de interfaces, vimos como exemplo uma interface chamada **IBoleto**, ela definia um método chamado **GerarCodigoDeBarras** e todas as classes que a implementarem deverão definir este método, pois bem, vamos definir uma classe responsável por construir um boleto, juntando as informações de Boleto de cada banco e gerando o arquivo do boleto, vejamos:

```
public class GeradorBoleto
{
    private readonly IBoleto _boleto;

    public GeradorBoleto(IBoleto boleto)
    {
        _boleto = boleto;
    }

    public void MontarBoleto()
    {
        var codigoDeBarras = _boleto.GerarCodigoDeBarras();

        // Lógica para gerar o boleto.
    }
}
```

Veja que estou declarando um atributo do tipo **IBoleto**, isso mesmo, meu atributo tem o tipo de uma **interface** e da mesma forma que no exemplo anterior com classes abstratas, qualquer classe que implemente a interface **IBoleto** estará elegível para ser passada como parâmetro no momento de criar um objeto do tipo **GeradorBoleto**. Você também deve ter percebido a presença de uma outra palavra chamada **readonly** que é um **modificador** com a finalidade de indicar que um atributo após ser instanciado não poderá receber outro valor, além disso atributos **readonly** devem obrigatoriamente ser definidos no construtor da classe.

Voltando a nossa implementação de polimorfismo, imagine que novas implementações de boletos de bancos diferentes são implementados periodicamente, seria pouco produtivo ter de

alterar a classe **GeradorBoleto** toda vez, não é mesmo? Utilizando nossa implementação isso **não acontece**, pois a cada nova implementação nada mudará para a classe **GeradorBoleto**, ela ainda está esperando um parâmetro do tipo da nossa interface, e ela somente sofrerá alteração se realmente houver alteração na lógica da geração do boleto, no arquivo ou algo do gênero.

Utilizar interfaces é muito indicado para declarar dependências entre as classes, quando criamos um tipo de parâmetro, propriedade e até mesmo um retorno de uma classe específica criamos o que é conhecido como **Acoplamento**, e o que sempre buscamos para um código de qualidade é **baixo acoplamento**, quando você depende diretamente de uma classe, significa que qualquer alteração na classe você estará comprometendo o funcionamento de outra, isso dificulta **evoluções no software** e tende a ocasionar **comportamentos inesperados**.

## Considerações Finais

Neste capítulo vimos assuntos mais complexos da orientação a objetos e como isso é transcrito na linguagem C#, implementamos **classes** e **métodos estáticos**, utilizamos **coleções**, além de descobrir alguns dos principais pilares da orientação a objetos como a **Herança** e **Polimorfismo** e como isso é possível utilizar as estruturas proporcionadas pela linguagem, como **Interfaces** e **classes** sejam elas **abstratas** ou não. A partir desse módulo temos conteúdo suficiente para desenvolver diversos cenários com C# e para o último módulo nossa missão é explorar recursos poderosos providos pelas bibliotecas do .net framework e como implementá-los usando C#.

## UNIDADE 5 - Extras

Neste capítulo iremos explorar alguns recursos extras, que nos darão suporte no desenvolvimento das nossas aplicações, são conceitos que não estão totalmente ligados a orientação a objetos mas que são essenciais para nosso conhecimento como desenvolvedor C#.

### Objetivos

O objetivo deste capítulo não é aprofundar em cada assunto e explorarmos todas as opções de uso dos mesmos, mas dar uma visão geral dos recursos, saber do que se trata e qual sua finalidade. Este capítulo irá prover um ponto de partida dos seguintes temas:

- Tratamento de exceções
- Extension Methods
- Programação assíncrona (async/await)
- Generics
- Expressões Lambda e Linq

### Tratamentos de Exceções

**Exceções** são erros em **tempo de execução** do software, eles ocorrem quando não acontece algo da forma esperada, fazendo com que o programa **capture o erro** e **dispare** de forma que **interrompa** o fluxo normal do código, um exemplo de uma exceção que pode ocorrer, é quando uma instrução tenta efetuar uma divisão por zero, tenta acessar um índice de um array que não existe ou tenta escrever em um arquivo somente leitura.

Para definir tarefas executadas quando um bloco de código dispara uma exceção podemos utilizar a cláusula **try**, ela geralmente não vem declarada sozinha, pode haver a presença de um ou mais blocos **catch** e pode ou não possuir um bloco **finally**. Vejamos um exemplo:

```
try
{
    // bloco que poderá disparar uma exceção
}
catch(ArgumentNullException)
{
    throw;
}
catch(Exception)
{
    throw;
}
finally
```

```
{  
  
}
```

O **try** e o **catch** sempre estarão presentes, o **catch** poderá estar presente mais de uma vez e o **finally** é opcional.

O bloco declarado no bloco **try** é o código que será validado e caso haja uma exceção será **interrompido**, já a cláusula **catch** tem o objetivo de capturar a exceção de acordo com o tipo especificado ou caso a exceção lançada seja de um tipo **derivado** do tipo especificado, por exemplo, no bloco acima o primeiro **catch** será executado caso uma exceção do tipo **ArgumentNullException** for lançada ou um tipo que **herde** dela, o segundo bloco **catch** será executado caso qualquer outro tipo de exceção seja lançado, isso porque o tipo **Exception** é o tipo base de todas as exceções, ele captura qualquer exceção que não tenha sido capturada anteriormente. Por fim o código definido no bloco **finally** será executado de qualquer forma, tenha sido lançada ou não uma exceção.

Podemos criar nossas próprias exceções personalizadas para disparar erros específicos de nossas aplicações, basta herdar da classe **Exception** ou de outra exceção mais específica. para disparar uma exceção utilizamos a cláusula **throw** da seguinte forma:

```
throw new MinhaException("Mensagem de erro");
```

O tratamento de exceções é algo essencial para nossa aplicações, tratar os erros e exibi-los de forma amigável para o usuário é algo esperado de um desenvolvedor cuidadoso. Além de tratar os erros algo muito utilizado é capturá-los e registrar logs dos mesmos para análise futura e existem diversas bibliotecas que facilitam o uso de logs no ecossistema .net.

## Extension Methods

Um recurso muito interessante presente no **c#** e outras linguagens do .net framework é a possibilidade de criar **extension methods** ou **métodos de extensão** em português, eles são um tipo especial de método que adiciona comportamento a classes já definidas sem ser necessário que esse tipo seja **herdado** ou algo do gênero. Podemos utilizá-los para facilitar pequenas tarefas tanto em classes criadas por nós quanto classes de terceiros, como classes do .net framework por exemplo. Vejamos um exemplo:

```
public static class MyExtensions
```

```
{
    public static bool EhUmCpf(this string texto)
    {
        return texto.Length == 11;
    }
}
```

Obs: Método de extensão devem ser estáticos e definidos em uma classe estática.

Definindo esse **método de extensão** qualquer string criada, possuirá o método **EhUmCpf** definido como se estivesse presente na própria definição do tipo, e possibilita que o utilizemos da seguinte forma:

```
var ehcpf = "12345678910".EhUmCpf();
```

Interessante não é mesmo? **Extension Methods** proporcionam muitas possibilidades, principalmente quando se trata de estender funcionalidade existentes e permitem um código mais simples.

## Programação assíncrona

Algo que é muito levado em consideração para possuímos uma aplicação performática e que provê uma boa experiência ao usuário é a **programação assíncrona**, para que possamos entender do poder da **programação assíncrona** temos que primeiro entender como a **programação síncrona** funciona.

Todos os exemplos utilizados neste material até o momento utilizam a **programação síncrona**, todos os métodos que escrevemos são executados de forma linear, ou seja, o runtime executa uma instrução após a outra, e para que uma próxima instrução seja executada o runtime **aguarda** até a finalização de cada instrução, o que acontece caso o programa execute uma instrução que depende de um processo mais lento, como por exemplo **ler um arquivo**, **acessar um serviço remoto** ou algo do gênero? A **linha (thread) de execução** do seu programa fica travada aguardando o término da execução, se estivermos falando de uma aplicação com uma interface gráfica, a tela permanecerá travada e caso o usuário interaja com ela muito provavelmente veremos aquela mensagem “O programa não está respondendo” (se estivermos no windows), como resolver este problema? Utilizando **programação assíncrona**! Comumente a forma que vemos para escrever código assíncrono inclusive em outras linguagens se baseiam na chamada de outras funções que são executadas quando o processo “lento” termina, vejamos um pseudo-código de exemplo:



```

void minhaFuncao()
{

ConsumirServicoExterno().onComplete(minhaFuncaoComplete).onError(minh
aFuncaoError);
    // Executar alguma outra tarefa.
}

void minhaFuncaoComplete(resultado)
{
    // Executa algo após o término da execução do processo lento
}

void minhaFuncaoError(error)
{
    // Executa se houver algum erro no processo lento.
}

```

Não se atenha a funcionalidade deste código, o objetivo dele é apenas expressar uma ideia, pois geralmente é assim que parece um código assíncrono escrito em diversas linguagens, achou complexo? Exatamente! A escrita de código assíncrono em sua forma “pura” é complexa e difícil de realizar tarefas simples, porém o que aconteceria caso um código utilizando esses princípios fosse executado? Ao executar **minhaFuncao** o runtime chamaria o método **ConsumirServicoExterno** e não esperaria o término de sua execução, continuaria executando outras tarefas definidas após essa instrução, porém, quando essa operação de consumir um serviço externo tivesse um **retorno** ou um **erro**, chamaria a devida função com o resultado, fazendo com que, caso haja uma interface de usuário executando essa operação, ela não fique “travada” aguardando por uma operação externa a ela, isso também pode economizar poder de processamento, sabia? Enquanto uma **linha de execução (thread)** está aguardando o término de uma instrução, ela está “ociosa” e poderia estar executando outras tarefas no seu computador.

Conseguimos entender que a programação assíncrona em diversos casos é essencial não é mesmo? Mas precisamos escrever códigos complexos como no exemplo? A resposta é não! o C# a partir da versão **5** e o .net framework a partir da versão **4.5** oferecem suporte a programação assíncrona através do uso de **async** e **await**, que é uma abstração que facilita o uso deste tipo de abordagem de maneira mais simples, “por baixo dos panos” o que acontece é algo muito parecido com o exemplo acima, mas para o desenvolvedor fica muito mais simples. Veja um

exemplo:

```
async Task<int> AcessaAWebAsync()
{
    // Estamos declarando um cliente http que irá consumir um website
    HttpClient client = new HttpClient();

    // GetStringAsync retorna um Task<string>. Isso significa que
    // quando você esperar a task ser
    // executada você terá uma string contendo o conteúdo da página
    (urlContents).
    Task<string> getStringTask =
client.GetStringAsync("http://msdn.microsoft.com");

    // Aqui você pode executar outras tarefas que não dependam do
    resultado obtido
    DoIndependentWork();

    // Aqui o operador await suspende a execução do método
    AcessaAWebAsync
    // - AcessaAWebAsync não pode continuar até getStringTask
    terminar de executar
    // - Durante esse período a execução retorna para quem invocou o
    método AcessaAWebAsync
    // - A execução continua aqui quando getStringTask terminar de
    executar
    // - O operador await então recupera o string de retorno do
    getStringTask
    string urlContents = await getStringTask;

    // A clausula return especifica um inteiro como resultado
    // Qualquer método que estiver aguardando a execução de
    AcessaAWebAsync
    // terá o tamanho da resposta.
    return urlContents.Length;
}
```

Fonte: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/async/>

Precisamos utilizar a palavra **async** para indicar que se trata de um método **assíncrono**, perceba

que o retorno declarado também é um pouco diferente do convencional, para este tipo de retorno que deve-se “esperar” para obter o resultado é utilizado o tipo **Task**, pois não se trata de um valor concreto e sim a indicação que ao término da execução um tipo **int** (no caso do exemplo) será produzido. Além disso a presença do **await** indica que naquele ponto o retorno concreto é necessário para que se dê continuidade na execução, então o operador **await** se responsabiliza por “extrair” o valor de retorno da task retornada pelo método **getStringTask** e quando isso ocorrer a execução continuará. Mais simples não é mesmo? Apesar de palavras novas estarem presentes, o fluxo de execução pode ser mais facilmente compreendido nesta implementação. Algumas informações importantes:

- Você só pode fazer uso de **await** em um método marcado como **async**.
- o método **AcessaAWebAsync** poderá ser invocado e aguardado utilizando **await** por outro método assíncrono que o invocar.
- Caso não se trate de um método assíncrono que estiver invocando o método **AcessaAWebAsync** e o retorno concreto seja necessário, deverá ser utilizado:

```
var retorno = AcessaAWebAsync().GetAwaiter().GetResult();
```

- É uma convenção terminar o nome de um método assíncrono com **Async**.

## Generics

**Generics** ou **Genéricos**, em português, segundo a documentação da microsoft “é um recurso que representa o conceito de parâmetros de tipo, que possibilita a criação de classes e métodos que adiam a especificação de um ou mais tipos até que a classe ou método seja declarado e instanciado pelo código do cliente.”, mas o que isso significa? Imagine que precise criar estruturas que servirão para o mesmo propósito independente do tipo do objeto empregado, as estruturas mais comuns que permitem este uso, são as coleções, e o .net implementa este conceito na classe **List** por exemplo, quando você precisa declarar uma **List**, você deve adicionalmente especificar o tipo dessa lista, veja o exemplo:

```
List<Cliente> minhaLista = new List<Cliente>();
```

O código acima especifica que estamos criando uma lista que conterá somente objetos do tipo **Cliente**, o que está entre “< >” se trata do nosso tipo **genérico**, a implementação da classe **List** não sabe diretamente com qual tipo ela estará lidando, portanto dentro da classe **List**, não seria possível invocar um método presente na classe **Cliente** definido por nós.

Mas porque o uso de Generics é interessante? **Generics** provê a possibilidade de implementações genéricas que possam ser reutilizadas, porém sem que o cliente da classe perca benefícios de linguagens com tipos bem definidos, pois, o compilador saberá, por exemplo, que

na nossa lista acima só conterá objetos do tipo **Cliente**, indicando erro caso, acessando uma posição específica da lista, tentarmos invocar um método que não está definido na nossa classe **Cliente**.

Podemos definir nossas próprias implementações utilizando **Generics**, para que possamos prover classes reutilizáveis que possuem funcionalidade genérica, evitando termos que implementar soluções específicas **duplicadas** ou perder o benefício de **tipos bem definidos**. Está em dúvida de como o uso de generics pode facilitar nossa vida como desenvolvedor? Imagine que precisamos criar uma estrutura de árvore de objetos, porém, queremos utilizar essa estrutura de dados para várias classes, poderíamos ter uma estrutura mais ou menos assim:

```
public class Tree<T>
{
    public Tree(T item, T parent)
    {
        Parent = parent;
        Item = item;
        Children = new List<T>();
    }

    public T Parent { get; private set; }
    public T Item { get; private set; }
    public List<T> Children { get; private set; }

    public void AddChild(T child)
    {
        Children.Add(child);
    }
}
```

Perceba que usamos **T** para representar o tipo que deverá ser definido quando um objeto deste tipo for declarado, e possui propriedades definidas com este tipo genérico para representar meu item, o item pai e uma lista de itens filhos, dessa forma podemos ter a funcionalidade de uma árvore para qualquer tipo de dado, imagine que temos uma classe do tipo **Menu** e queremos utilizar nossa classe **Tree**:

```
Tree<Menu> arvoreDeMenus = new Tree<Menu>();
```

Já imaginou outros usos possíveis para **Generics**? Você também pode utilizá-los em métodos,

interfaces, entre outros. Você se recorda do nosso exemplo em que criamos uma interface **IBoleto**? Poderíamos implementar um método que gerasse o layout do nosso boleto utilizando **Generics**, ficaria assim:

```
public static void GerarArquivoBoleto<T>(T boleto) where T: IBoleto
{
    var codigoDeBarras = boleto.GerarCodigoDeBarras();
    // Restante da lógica para gerar o boleto
}
```

Aqui temos o uso de generics em um método, além disso, este bloco de código possui algo novo, o uso de **restrição do tipo genérico**. O uso de **restrições** é interessante, pois, quanto mais específico for a restrição do tipo mais a implementação que utiliza o tipo genérico poderá fazer uso de propriedades e métodos que o tipo genérico possui, perceba que no exemplo dizemos que **T** deverá ser uma implementação da interface **IBoleto** e fazemos isso através da palavra **where**, isso possibilita que dentro do nosso método **GerarArquivoBoleto** possamos invocar métodos definidos na nossa interface **IBoleto**, nesse caso o método **GerarCodigoDeBarras** está sendo invocado, também podemos definir classes **base** como restrição, tornando qualquer classe que herde delas apta para utilizar a implementação.

## Expressões Lambda e Linq

Esses dois recursos sem dúvida estarão presentes em inúmeras ocasiões no desenvolvimento de aplicações C#, isso porque são mais comumente utilizados na **manipulação de dados** e na grande maioria de nossas aplicações temos que trabalhar com dados, seja qual a for a fonte deles.

Mesmo que nesta seção **Linq** e **expressões Lambda** estejam sendo apresentados juntos cada um possui sua responsabilidade, então vamos entender cada conceito separado e depois exploramos o uso deles juntos, e você verá que é um casamento perfeito.

Vamos começar com **Linq**, na documentação da microsoft podemos extrair o seguinte conceito:

“O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#. Tradicionalmente, consultas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou no suporte a IntelliSense.”

Resumindo, o Linq é uma sintaxe de consulta de dados assim como SQL, porém utiliza classes do C# e tem a capacidade de **abstrair** a fonte de dados, podemos utilizar **Linq** para fazermos consultas em **Xml**, **Base de dados**, **Coleções** em memória entre outros. O Linq permite que façamos operações em cima dos dados, como filtros, somas, ordenações, entre outras, vamos

verificar o seguinte exemplo utilizando listas em memória:

```
static void Run()
{
    // Aqui declaramos um array que servirá como fonte de dados.
    int[] scores = new int[] { 97, 92, 81, 60 };

    // Aqui criamos uma consulta filtrando itens com valor superior a
    80.
    IEnumerable<int> scoreQuery =
        from score in scores
        where score > 80
        select score;

    // Aqui ordenamos a lista de forma ascendente
    IEnumerable<int> orderedScoreQuery =
        from score in scoreQuery
        orderby score ascending
        select score;
}
```

No exemplo acima não está destacado, mas sempre que utilizarmos Linq devemos importar o namespace **System.Linq**.

Perceba no exemplo que temos diversas palavras reservadas e a forma de utilização se assemelha muito ao SQL, porém ainda podemos simplificar muito essas operações fazendo com que o código fique mais simples e operações como as do exemplo acima fiquem extremamente pequenas, para isso vamos conhecer as **expressões Lambda**.

As expressões Lambda são pequenas instruções que executam ações, se assemelham com funções ou métodos, porém suas declarações são caracterizadas por códigos menores e comumente aparecem sendo passadas como parâmetros para métodos que irão executá-las em um momento oportuno.

Um expressão lambda consiste em duas partes separadas pelos caracteres “=>”:

1. A primeira define um ou mais argumentos que a expressão lambda manipulará.
2. A segunda define a expressão que representa a ação executada.

Podemos traduzir os exemplos definidos logo acima como expressões lambda, vejamos:

```

static void Run()
{
    int[] scores = new int[] { 97, 92, 81, 60 };

    IEnumerable<int> scoreQuery = scores.Where(score => score > 80);

    IEnumerable<int> orderedScoreQuery = scoreQuery.OrderBy(score =>
score);
}

```

Perceba que o código ficou extremamente menor, com presença de elementos expressivos que indicam de forma clara a ação realizada.

Podemos utilizar Linq e expressões lambda para trabalhar com objetos mais complexos, e é a partir daí que o uso começa a parecer mais interessante, vejamos outro exemplo:

```

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

public class ConsultaPessoas
{
    static void Run()
    {
        var listaDePessoas = new List<Pessoa>()
        {
            new Pessoa {
                Email = "maria@maria.com",
                Idade = 16,
                Nome = "Maria"
            },
            new Pessoa {
                Email = "pedro@pedro.com",
                Idade = 17,
            }
        }
    }
}

```

```

        Nome = "Pedro"
    },
    new Pessoa {
        Email = "joao@joao.com",
        Idade = 24,
        Nome = "João"
    }
};

IEnumerable<string> maioresDeIdade = listaDePessoas
    .Where(pessoa => pessoa.Idade > 18)
    .OrderBy(pessoa => pessoa.Nome)
    .Select(pessoa => pessoa.Nome);
}
}

```

Neste exemplo, a partir de uma lista de pessoas, filtramos pessoas maiores de 18 anos, ordenamos por ordem alfabética através de seus nomes e depois selecionamos somente o nome, gerando uma lista de strings com eles.

Também podemos fazer projeções em cima dos dados gerando uma lista de outro tipo. Considerando que eu tenha uma classe **PessoaMaior** com nome e email, é possível executar:

```

IEnumerable<PessoaMaior> maioresDeIdade = listaDePessoas
    .Where(pessoa => pessoa.Idade > 18)
    .Select(pessoa => new PessoaMaior
    {
        Nome = pessoa.Nome,
        Email = pessoa.Email
    });

```

Podemos iterar sobre uma coleção usando Lambda e executar uma ação para cada item na lista, no exemplo abaixo encaminhamos um e-mail para cada pessoa da lista acima:

```

static void Run()
{
    var maioresDeIdade = listaDePessoas

```



```

        .Where(pessoa => pessoa.Idade > 18)
        .Select(pessoa => new PessoaMaior
        {
            Nome = pessoa.Nome,
            Email = pessoa.Email
        }).ToList();

    maioresDeIdade.ForEach(pessoa => EncaminharEmail(pessoa));
}

private static void EncaminharEmail(PessoaMaior pessoa)
{
    // Envia e-mail para pessoas maiores de 18 anos.
}

```

Há diversas outras possibilidades com Linq e expressões Lambda, espero que você possa testar e pesquisar mais sobre seu uso.

### Considerações Finais

Neste Capítulo encerramos nosso material de programação orientada a objetos com C# explorando alguns recursos da linguagem e do .net framework que dão suporte às nossas tarefas como desenvolvedores, são recursos poderosos e muito provavelmente você irá ver a utilização deles em aplicações reais, não fique satisfeito com o que viu sobre estes recursos neste capítulo, pesquisa por conta própria, pois como mencionado na apresentação do capítulo, o principal intuito era apresentar o recurso e sua finalidade, somente através de prática e pesquisa você entenderá a fundo o funcionamento deles e os aplicará a cada dia com maior perfeição.

## Referências:

MICROSOFT. Programação assíncrona com async e await (C#). {online}. Disponível na Internet via <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/async/>. Arquivo capturado em 29 de out. 2018.

MICROSOFT. Genéricos (Guia de Programação em C#). {online}. Disponível na Internet via <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/>. Arquivo capturado em 29 de out. 2018.

MICROSOFT. Restrições a parâmetros de tipo (Guia de Programação em C#). {online}. Disponível na Internet via <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters>. Arquivo capturado em 29 de out. 2018.

MICROSOFT. Introdução aos genéricos (Guia de Programação em C#). {online}. Disponível na Internet via <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/introduction-to-generics>. Arquivo capturado em 29 de out. 2018.

MICROSOFT. Benefícios dos genéricos (Guia de Programação em C#). {online}. Disponível na Internet via <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/benefits-of-generics>. Arquivo capturado em 29 de out. 2018.

PIRES, Eduardo. C# – Para Iniciantes – Parte 3 – Expressão Lambda. {online}. Disponível na Internet via <http://www.eduardopires.net.br/2012/07/c-sharp-iniciantes-expressao-lambda/>. Arquivo capturado em 29 de out. 2018.

MOREIRA, Leonardo. Tratamento de exceções em C#. {online}. Disponível na Internet via <https://www.devmedia.com.br/space/leonardo-moreira>. Arquivo capturado em 29 de out. 2018.

ALURA. Herança. {online}. Disponível na Internet via <https://www.caelum.com.br/apostila-csharp-orientacao-objetos/heranca/>. Arquivo capturado em 29 de out. 2018.

MICROSOFT. .NET Standard. {online}. Disponível na Internet via <https://docs.microsoft.com/pt-br/dotnet/standard/net-standard>. Arquivo capturado em 29 de out. 2018.

### Referências das figuras:

MICROSOFT. .NET Standard. {online}. Disponível na Internet via <https://docs.microsoft.com/pt-br/dotnet/standard/net-standard>. Arquivo capturado em 29 de out. 2018.