

# Introdução ao .NET

O .NET é um ecossistema para o desenvolvimento de aplicativos. O termo *ecossistema* descreve as múltiplas facetas de um ambiente de desenvolvimento de aplicativos e da comunidade ao redor dele. Em outras palavras, o .NET é composto por muitas partes e pessoas que juntas formam um ambiente poderoso para a criação de aplicativos.

Para que os desenvolvedores de software possam executar o código, eles devem primeiro compilá-lo. O *compilador do .NET* é um programa que converte o código-fonte em uma linguagem especial chamada *IL (linguagem intermediária)*. O compilador do .NET salva o código de IL em um arquivo chamado *assembly do .NET*. Ao compilar o código em um formato "intermediário", você pode usar a mesma base de código, independentemente do local de execução dele, seja no Windows, no Linux ou em um hardware de computador de 32 ou 64 bits.

O *runtime do .NET* é um ambiente de execução para o seu assembly do .NET compilado. Em outras palavras, o runtime do .NET é o que executa e gerencia o seu aplicativo à medida que ele é executado em um sistema operacional.

## Dica

Às vezes, as pessoas acham erroneamente que a linguagem de programação C# é .NET. No entanto, C# e .NET são diferentes. C# é uma sintaxe de linguagem de programação. Como parte da sintaxe, você pode referenciar e chamar métodos definidos em assemblies ou bibliotecas de código do .NET. Além disso, você usa o compilador do C# que está instalado com o SDK do .NET para criar um assembly com seu código C#. O runtime do .NET executa assemblies do .NET. Ao entender essas distinções, você compreenderá conceitos importantes conforme aprende mais sobre o .NET e o C#.

## Usar bibliotecas e estruturas de aplicativos do .NET.

Todos os softwares são criados em camadas, o que significa que são executados em vários níveis de abstração em um computador:

- No nível mais baixo, o software se comunica diretamente com o hardware do computador. Ele controla o fluxo de dados na placa-mãe, nos processadores, na memória e nos discos rígidos.
- No nível seguinte, o software permite que o usuário final forneça instruções por meio de um sistema operacional.
- No próximo nível, um software como o .NET fornece uma maneira para você desenvolver e executar aplicativos.
- No nível seguinte, as estruturas de aplicativo e as bibliotecas de funcionalidade permitem criar rapidamente aplicativos avançados usando menos esforço do que os métodos de desenvolvimento mais antigos permitiam.

Uma biblioteca de códigos encapsula a funcionalidade para uma finalidade específica em um assembly. Para o .NET, milhares de bibliotecas estão disponíveis. Essas bibliotecas podem ser próprias ou de terceiros e podem ser comerciais ou de software livre. Elas fornecem uma ampla gama de funcionalidades que você pode usar em seus aplicativos. Basta fazer referência a esses assemblies e chamar os métodos necessários. Dessa forma, como desenvolvedor, você cria com base no trabalho de outros desenvolvedores de software. Você poupa tempo e energia porque não precisa criar e manter todos os recursos por conta própria.

## Quais são os principais modelos de aplicativos?

Você deve estar imaginando quais estruturas dão suporte a quais modelos de aplicativo. Use a tabela a seguir a fim de mapear um modelo de aplicativo para uma estrutura do .NET.

Modelo de aplicativo	Estrutura	Observações
Web	ASP.NET Core	A estrutura para a criação da lógica do lado do servidor.
Web	ASP.NET Core MVC	A estrutura para a criação da lógica do lado do servidor para páginas da Web ou APIs Web.
Web	Razor Pages do ASP.NET Core	A estrutura para a criação de HTML gerado pelo servidor.
Cliente Web	Blazor	O Blazor é uma parte do ASP.NET Core. Os dois modos dele permitem a manipulação de DOM (Modelo de Objeto do Documento) por meio de soquetes como um veículo de comunicação para a execução de código do lado do servidor ou como uma implementação do WebAssembly para a execução do C# compilado no navegador.
Desktop	WinForms	Uma estrutura para criar aplicativos no estilo "acinzentado" do Windows.
Desktop	Windows Presentation Foundation (WPF)	Uma estrutura para criar aplicativos da área de trabalho dinâmicos em conformidade com diferentes fatores de forma. O WPF permite que os elementos da forma realizem movimentos, esmaecimentos, deslizamentos e outros efeitos com a ajuda de uma biblioteca de animações avançada.
Móvel	Xamarin	Permite que os desenvolvedores do .NET criem aplicativos para dispositivos iOS e Android.

Além disso, o .NET capacita ambientes populares de desenvolvimento de jogos de software livre e de terceiros e mecanismos como o Unity.

## Sobre.

O .NET Framework original foi lançado no início de 2002. Desde então, muitas atualizações e diversas outras funcionalidades foram introduzidas.

Depois de 2002, a Microsoft trabalhou para criar uma versão do .NET com compatibilidade entre plataformas. O objetivo era permitir que os desenvolvedores escrevessem uma base de código e a usassem em sistemas operacionais macOS, Linux e Windows.

Por meio desses esforços, o .NET Core foi introduzido por volta de 2014. A Microsoft manteve o .NET Framework original. Contudo, novos recursos e aprimoramentos foram reservados para o .NET Core. O *Core* foi posteriormente descartado do nome. As versões principais seguintes foram .NET 5, .NET 6, .NET 7 e assim por diante. As versões geralmente são lançadas em novembro.

## Ambiente de desenvolvimento.

A primeira decisão tomada pelos desenvolvedores é selecionar as ferramentas que usarão para criar os aplicativos. Para os desenvolvedores que preferem um ambiente visual, o Visual Studio 2019 é a melhor opção.

O Instalador do Visual Studio no Visual Studio 2019 oferece opções na forma de cargas de trabalho. Uma *carga de trabalho* é uma coleção de estruturas, bibliotecas e outras ferramentas que trabalham em conjunto para criar um modelo de aplicativo específico.

A carga de trabalho **.NET Core para desenvolvimento multiplataforma** instala o SDK do .NET. O SDK do .NET contém todas as bibliotecas, ferramentas e modelos necessários para começar a escrever código.

Se você preferir um ambiente de linha de comando, baixe e instale o Visual Studio Code e o SDK do .NET separadamente.

## O Try .NET

O Try .NET oferece uma maneira fácil de experimentar o C# e o .NET. Você pode usá-lo sem instalar nenhum software no computador local.

Se quiser experimentar o aplicativo vá para <https://try.dot.net> O Try .NET é uma ótima maneira de experimentar pequenos exemplos de código sem instalar nada no computador local.

## Escrevendo o primeiro código.

Observando o seguinte código no Try .NET, Visual Studio 2019 ou Visual Studio Code.

C#

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Hello world!");
    }
}
```

Saída

Hello world!

O código **public static void Main()** e o conjunto de chaves dele definem um tipo de bloco de código chamado de **método**. Um método contém um agrupamento de código que funciona para uma única finalidade ou responsabilidade em seu sistema de software.

Os **métodos** são organizados dentro de outros blocos de código chamados de **classes**. Uma **classe** pode conter um ou mais **métodos**. A classe no código anterior chama-se **Program**.

```
C#  
  
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        Console.WriteLine("Hello world!");  
    }  
}
```

A linha de código **Console.WriteLine()** inserida no nosso método, está *chamando* ou executando o método **WriteLine()**. O método **WriteLine()** está contido na classe **Console**.

Na verdade, o nome completo dele é **System.Console.WriteLine()**. A palavra **System** foi omitida na sua chamada para **Console.WriteLine()**. Mas a primeira linha de código **using System;** instruirá o compilador do **C#** a procurar na biblioteca de classes base.

```
C#  
  
using System;
```

## Recursos

Se você quiser usar uma interface gráfica do usuário para criar aplicativos com o C#, [baixe e instale o Visual Studio 2019](#).

### 📌 Observação

Se você não é um usuário licenciado (pagante) do Visual Studio, baixe o *Community Edition*, que tem todos os recursos necessários para começar.

Se quiser começar usando uma interface de linha de comando, baixe e instale o [SDK do .NET](#) e o [Visual Studio Code](#).

[Extensão do C#](#) para Visual Studio Code.

## Adicionar pacotes ao projeto do .NET

O .NET tem muitas **bibliotecas** principais que cuidam de tudo, desde o gerenciamento de arquivos até o HTTP e a compactação de arquivos. Também há um enorme ecossistema com bibliotecas de terceiros. É possível usar o **NuGet**, o Gerenciador de Pacotes do .NET, para instalar essas bibliotecas e usá-las no aplicativo.

Uma **dependência** de pacote é uma biblioteca de terceiros. Trata-se de um trecho de código reutilizável que faz algo e que pode ser adicionado ao seu aplicativo. É algo de que seu aplicativo *depende* para funcionar.

Um **pacote** é composto por uma ou mais bibliotecas que podem ser adicionadas ao aplicativo para que você possa aproveitar os recursos delas.

## Instalar um pacote.

Para saber mais sobre um pacote antes de instalá-lo acesse

**<https://www.nuget.org/packages/<package name>>**. Essa URL levará você até uma página detalhada sobre o pacote. Selecione a lista suspensa **Dependências** para ver de quais pacotes ele depende para funcionar.

Você pode adicionar um pacote ao projeto do .NET invocando um comando no terminal CLI do .NET Core. Um comando de instalação típico se parece com este: ***dotnet add package <name of package>***. Quando você executa o comando ***add package***, a ferramenta de linha de comando se conecta a um registro global, busca o pacote e o armazena em um local de pasta em cache que todos os projetos podem usar.

Após a instalação e a compilação do projeto, as referências são adicionadas às suas pastas de depuração ou versão. O diretório do projeto se parece com este:

Bash

```
-| bin/  
---| Debug/  
-----| net3.1  
-----| <files included in the dependency>
```

## Registro do NuGet e a ferramenta dotnet.

Quando você executa ***dotnet add package <name of dependency>***, o .NET vai para um registro global chamado registro NuGet.org e procura o código a ser baixado. Ele fica localizado em ***https://nuget.org***. Você também poderá procurar pacotes nesta página se usar um navegador. Cada pacote tem um site dedicado que você pode acessar.



Versão	Downloads	Última atualização
2.8.26	172.780	Há quatro meses

## Comandos .NET

A CLI do .NET tem uma quantidade razoável de comandos. Os comandos ajudam você com tarefas como instalar pacotes, criar pacotes e inicializar projetos do .NET.

Para se lembrar do que os comandos fazem, é útil considerar que eles se dividem em categorias:

- **Gerenciar dependências.** Há comandos que cuidam da instalação, remoção e limpeza após as instalações de pacotes. Também há comandos para atualizar pacotes.
- **Execute programas.** A ferramenta .Net Core pode ajudar você a gerenciar os fluxos no desenvolvimento do aplicativo. Exemplos de fluxos de aplicativo são a execução de testes, a compilação de código e a execução de comandos de migração para atualizar projetos.
- **Criar e publicar pacotes.** Diversos comandos podem ajudar você com tarefas como criar um pacote compactado e efetuar push do pacote para um registro.

Para ver uma lista detalhada com todos os comandos, insira ***dotnet --help*** no terminal.

Os pacotes instalados são listados na seção **dependencies** do arquivo **.csproj**. Para ver exatamente quais pacotes estão na pasta, digite **dotnet list package**.

```
Saída

Project 'DotNetDependencies' has the following package references
[net5.0]:
Top-level Package      Requested   Resolved
> Humanizer            2.7.9      2.7.9
```

Esse comando lista apenas os pacotes de nível superior e não as dependências deles, que chamamos de *pacotes transitivos*.

A inclusão de transitivos permite ver dependências junto com todos os pacotes que você instalou. Se executar **dotnet list package --include-transitive**, você poderá ver esta saída:

```
Saída

Project 'DotNetDependencies' has the following package references
[net5.0]:
Top-level Package      Requested   Resolved
> Humanizer            2.7.9      2.7.9

Transitive Package      Resolved
> Humanizer.Core        2.7.9
> Humanizer.Core.af     2.7.9
> Humanizer.Core.ar     2.7.9
> Humanizer.Core.bg     2.7.9
> Humanizer.Core.bn-BD  2.7.9
> Humanizer.Core.cs     2.7.9
...
```

## Restaurar dependências.

Quando você cria ou clona um projeto, as dependências incluídas não são baixadas nem instaladas até que você compile o projeto. Você pode restaurar manualmente as dependências, bem como ferramentas específicas do projeto que são especificadas no arquivo de projeto executando o comando **dotnet restore**. Na maioria dos casos, não é preciso usar o comando explicitamente. A restauração do NuGet é executada implicitamente, se necessário, quando você executa comandos como **new**, **build** e **run**.

## Limpar dependências.

É provável que, cedo ou tarde, você se dê conta de que não precisa mais de um pacote. Ou você poderá perceber que instalou um pacote do qual não precisa. Talvez você encontre outro pacote que realizará melhor uma determinada tarefa.

Para remover um pacote do projeto, use o comando **remove** da seguinte maneira: **dotnet remove <name of dependency>**. Esse comando removerá o pacote do arquivo **.csproj** do projeto.

## Criar um projeto .NET de exemplo

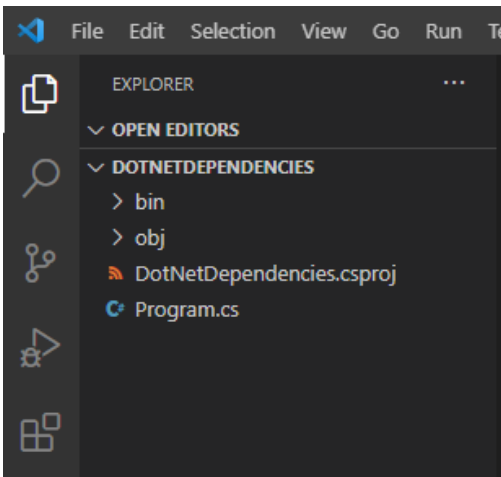
Para configurar o projeto do .NET a fim de trabalhar com dependências, usaremos o **Visual Studio Code** e seu terminal integrado.

1. No Visual Studio Code, escolha **Arquivo > Abrir Pasta**.
2. Crie uma pasta chamada **DotNetDependencies** no local de sua escolha e clique em **Selecionar Pasta**.
3. No Visual Studio Code, abra o terminal integrado selecionando **Exibir > Terminal** no menu principal.
4. Na janela do terminal, digite o comando **dotnet new console**.

```
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies> dotnet new console

Bem-vindo(a) ao .NET 5.0.
-----
Versão do SDK: 5.0.201
```

Esse comando cria um arquivo **Program.cs** em sua pasta com um programa básico "Olá, Mundo" já escrito, junto com um arquivo de projeto em C# chamado **DotNetDependencies.csproj**.



5. Para executar o programa "Olá, Mundo" digite o comando **dotnet run** no terminal.

```
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies> dotnet run
Hello World!
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies> |
```

A janela do terminal exibe "Olá, Mundo!" como saída.



# Adicionar um pacote do NuGet usando a ferramenta .Net Core

1. Abra **Program.cs**. Ele deve ser assim.

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

2. Instale a biblioteca do Humanizer executando o comando **dotnet add package Humanizer --version 2.7.9**

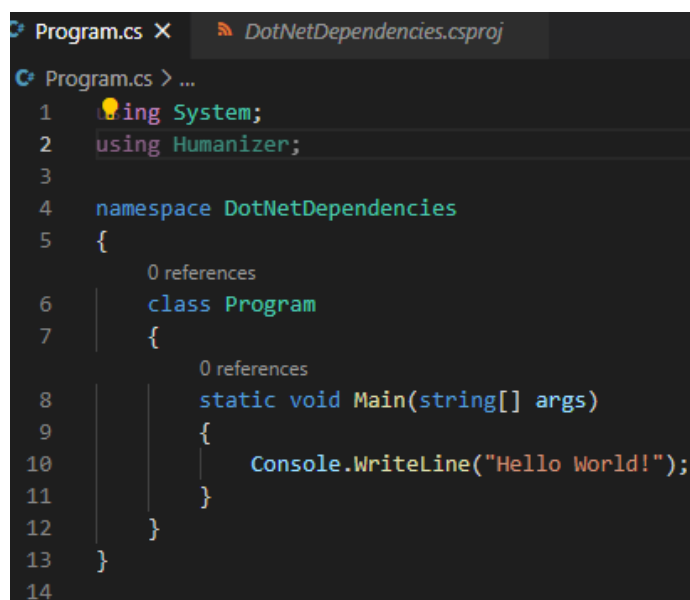
```
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies> dotnet add package Humanizer --version 2.7.9
Determinando os projetos a serem restaurados...
Writing C:\Users\Diogo\AppData\Local\Temp\tmp2EFF.tmp
info : Adicionando PackageReference do pacote 'Humanizer' ao projeto 'C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies\DotNetDependencies.csproj'.
info : Restaurando pacotes para C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies\DotNetDependencies.csproj...
info : GET https://api.nuget.org/v3-flatcontainer/humanizer/index.json
info : OK https://api.nuget.org/v3-flatcontainer/humanizer/index.json 581ms
info : GET https://api.nuget.org/v3-flatcontainer/humanizer/2.7.9/humanizer.2.7.9.nupkg
info : OK https://api.nuget.org/v3-flatcontainer/humanizer/2.7.9/humanizer.2.7.9.nupkg 60ms
info : GET https://api.nuget.org/v3-flatcontainer/humanizer.core.af/index.json
info : GET https://api.nuget.org/v3-flatcontainer/humanizer.core.ar/index.json
```

```
info : Humanizer.Core.bg 2.7.9 instalado de https://api.nuget.org/v3/index.json com o hash de conteúdo kpRp7agDo5WaMSGbHgybQaQUhUy2EPe7p6KoIe020XF3TCckMPFvmIcm6m4sVwdXjaIwfkWhkPBmVwKNfcmovg==.
info : Humanizer.Core.af 2.7.9 instalado de https://api.nuget.org/v3/index.json com o hash de conteúdo 7NMBpe6YeZ6G1NVT2mgwB4RdZn9g9zbUtssae47TFuS UxCfE8SdSiiwXu6QG+twjuaa15B833HnqKMJejlNxcA==.
info : O pacote 'Humanizer' é compatível com todas as estruturas especificadas no projeto 'C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies\DotNetDependencies.csproj'.
info : PackageReference do pacote 'Humanizer' versão '2.7.9' adicionada ao arquivo 'C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies\DotNetDependencies.csproj'.
info : Confirmando restauração...
info : Gravando o arquivo de ativos no disco. Caminho: C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies\obj\project.assets.json
log : C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies\DotNetDependencies.csproj restaurado (em 11,33 sec).
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies>
```

3. Abra o arquivo **DotNetDependencies.csproj** e localize a seção **ItemGroup**. Você deverá ter uma entrada parecida com esta.

```
1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <PropertyGroup>
4     <OutputType>Exe</OutputType>
5     <TargetFramework>net5.0</TargetFramework>
6   </PropertyGroup>
7
8   <ItemGroup>
9     <PackageReference Include="Humanizer" Version="2.7.9" />
10  </ItemGroup>
11
12 </Project>
13
```

4. Adicione o conteúdo **using Humanizer**; na parte superior do arquivo **Program.cs** a fim de inicializar o Humanizer.



```
Program.cs X DotNetDependencies.csproj
Program.cs > ...
1 using System;
2 using Humanizer;
3
4 namespace DotNetDependencies
5 {
6     0 references
7     class Program
8     {
9         0 references
10        static void Main(string[] args)
11        {
12            Console.WriteLine("Hello World!");
13        }
14    }
15 }
```

5. Adicione os métodos **HumanizeQuantities()** e **HumanizeDates()** a classe **program**.

```
static void HumanizeQuantities()
{
    Console.WriteLine("case".ToQuantity(0));
    Console.WriteLine("case".ToQuantity(1));
    Console.WriteLine("case".ToQuantity(5));
}

static void HumanizeDates()
{
    Console.WriteLine(DateTime.UtcNow.AddHours(-24).Humanize());
    Console.WriteLine(DateTime.UtcNow.AddHours(-2).Humanize());
    Console.WriteLine(TimeSpan.FromDays(1).Humanize());
    Console.WriteLine(TimeSpan.FromDays(16).Humanize());
}
```

6. Atualize o método **Main** para chamar os novos métodos.

```
C#

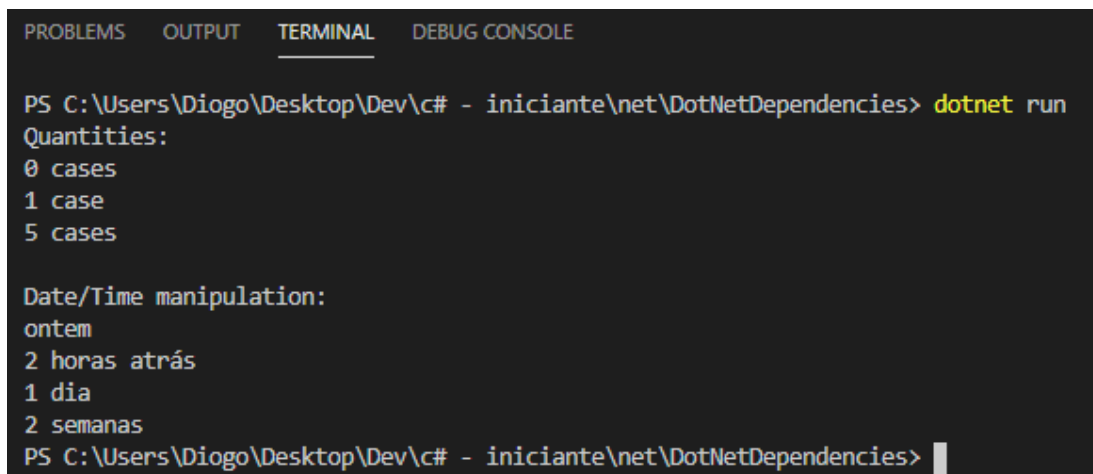
static void Main(string[] args)
{
    Console.WriteLine("Quantities:");
    HumanizeQuantities();

    Console.WriteLine("\nDate/Time Manipulation:");
    HumanizeDates();
}
```

7. Execute o aplicativo por meio do comando a seguir no terminal.

**dotnet run**

A saída a seguir será exibida.



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies> dotnet run
Quantities:
0 cases
1 case
5 cases

Date/Time manipulation:
ontem
2 horas atrás
1 dia
2 semanas
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies> █
```

Desta forma você instalou o **Humanizer** com sucesso como uma dependência e escreveu lógica para o código do aplicativo a fim de tornar os dados mais legíveis por pessoas.

## Gerenciar atualizações de dependência no projeto do .NET

Mais cedo ou mais tarde, você desejará atualizar para uma nova versão de uma biblioteca. Talvez uma função passe a ser preterida. Ou talvez haja um novo recurso em uma versão mais recente de um pacote que você está usando.

Leve estas considerações em conta antes de tentar atualizar uma biblioteca:

- **O tipo de atualização.** Que tipo de atualização está disponível? Trata-se de uma pequena correção de bug? Ela adicionará um novo recurso de que você precisa? Isso interromperá seu código? Você pode comunicar o tipo de atualização usando um sistema chamado *controle de versão semântico*.
- **Se o projeto está configurado corretamente.** Você pode configurar o projeto do .NET de maneira a receber somente os tipos de atualização desejados.
- **Problemas de segurança.** Gerenciar as dependências do projeto ao longo do tempo envolve estar ciente dos problemas que podem ocorrer. Problemas surgem conforme vulnerabilidades são detectadas.

## Controle de versão semântico.

O controle de versão semântico é a forma como você expressa o tipo de alteração que você ou algum outro desenvolvedor está introduzindo em uma biblioteca.

- **Versão principal.** O número na extremidade esquerda. Por exemplo, é o 1 em 1.0.0. Uma alteração nesse número significa que você pode esperar alterações da falha no código. Talvez seja necessário reescrever parte do código.
- **Versão secundária.** O número do meio. Por exemplo, é o 2 em 1.2.0. Uma alteração nesse número significa que recursos foram adicionados. O código deve continuar funcionando. De modo geral, é seguro aceitar a atualização.
- **Versão de patch.** O número na extremidade direita. Por exemplo, é o 3 em 1.2.3. Uma alteração nesse número significa que foi aplicada uma alteração que corrige algo no código que deveria ter funcionado. Deve ser seguro aceitar a atualização.

Tipo	O que acontece
Versão principal	1.0.0 muda para 2.0.0
Versão secundária	1.1.1 muda para 1.2.0
Versão de patch	1.0.1 muda para 1.0.2

## Localizar e atualizar pacotes desatualizados.

O comando **dotnet list package --outdated** lista pacotes desatualizados. Esse comando pode ajudar você a descobrir quando versões mais novas de pacotes estão disponíveis. Veja uma saída típica do comando:

Saída			
Top-level Package	Requested	Resolved	Latest
> Humanizer	2.7.*	2.7.9	2.8.26

Estes são os significados dos nomes das colunas na saída:

- **Requested.** A versão ou o intervalo de versão que você especificou.
- **Resolved.** A versão real que foi baixada para o projeto, que corresponde à versão especificada.
- **Latest.** A última versão disponível para atualização do NuGet.

O fluxo de trabalho recomendado é executar estes comandos, nesta ordem:

- Execute **dotnet list package**. Esse comando lista todos os pacotes desatualizados. Ele fornece informações nas colunas **Requested**, **Resolved** e **Latest**.
- Execute **dotnet add package <package name>**. Se você executar esse comando, ele tentará atualizar para a última versão. Opcionalmente, passe **--version=<version number/range>**.

## Atualizar as dependências do aplicativo.

Vamos atualizar as dependências do nosso aplicativo que criamos anteriormente.

1. No arquivo **DotNetDependencies.csproj**, examine `dependencies`. Ele deve ser semelhante a este código.

```
XML

<ItemGroup>
  <PackageReference Include="Humanizer" Version="2.7.9" />
</ItemGroup>
```

2. Para ver as dependências instaladas, execute este comando.

**dotnet list package**

Isso deve gerar como saída a versão solicitada e a versão final resolvida (instalada).

```
> dotnet list package
O projeto 'DotNetDependencies' tem as seguintes referências de pacote
[net5.0]:
Pacote de Nível Superior      Solicitado  Resolvido
> Humanizer                  2.7.9      2.7.9
```

3. Para ver quais dependências estão desatualizadas, execute este comando.

**dotnet list package --outdated**

A saída deverá ter uma aparência semelhante à saída a seguir.

```
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDependencies> dotnet list package --outdated

As fontes a seguir foram usadas:
https://api.nuget.org/v3/index.json

O projeto 'DotNetDependencies' tem as seguintes atualizações para seus pacotes
[net5.0]:
Pacote de Nível Superior      Solicitado  Resolvido  Mais Recente
> Humanizer                  2.7.9      2.7.9      2.8.26
```

Por padrão, esse comando verificará a última versão estável. Acrescente as informações abaixo ao comando para verificar pacotes de pré-lançamento.

**--include-prerelease**

4. Para instalar a versão mais recente, execute o comando a seguir.

**dotnet add package humanizer**

Se quiser atualizar para uma versão específica da dependência, você poderá fazer um acréscimo ao parâmetro **--version** e especificar a versão desejada.

**dotnet add package Humanizer --version 2.8.26**

Por fim, você também pode instalar o último pacote de pré-lançamento fazendo um acréscimo ao parâmetro **--prerelease**.

**dotnet add package Humanizer --prerelease**