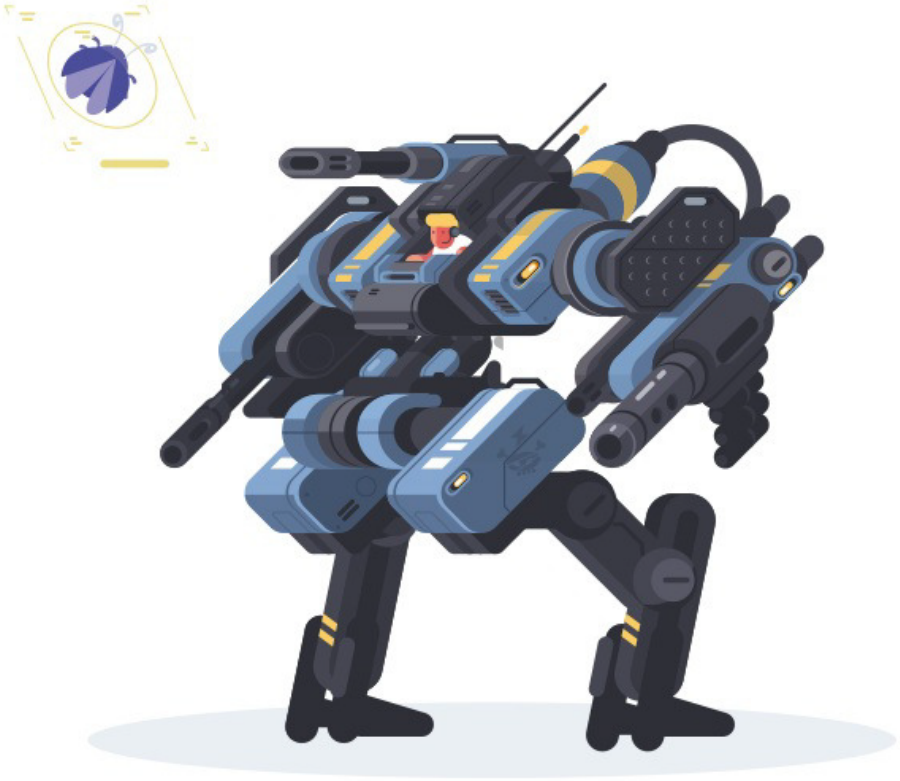
Produtividade em C#

Obtenha mais resultado com menos esforço



Casa do Código

CLÁUDIO RALHA

ISBN

Impresso: 978-65-86110-70-8

Digital: 978-65-86110-69-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

DEDICATÓRIA

*Para minha filha **Iara**,*
*o melhor presente que **Deus** me deu.*

*Para meus pais **Lígia** e **Vidal** (em memória),*
por todos os ensinamentos que me deram em vida.

AGRADECIMENTOS

Escrever um livro como este é uma tarefa quase insana, pois o C# é uma das dez linguagens mais importantes do mundo e que está no mercado há pelo menos 20 anos, o que, em termos práticos, se traduz em um volume imenso de material a ser estudado. Sem o apoio da minha amada Flávia, da minha filha Iara, da minha editora Vivian Matsui e da minha revisora Sabrina Barbosa, este livro não teria nascido. Meus irmãos Cristiane e Renan e meus amigos Marco Aurélio Coelho, Francisco Gaspar, Gustavo Pinto, André Costa, Ana Paula Oliveira e Fábio Peluso também foram de suma importância na estrada que percorri até aqui.

Meu contato com a linguagem C# se iniciou assim que ela deixou de ser apenas um projeto interno da Microsoft em janeiro de 2001. Nesse ano, chegaram ao Brasil os primeiros CDs com o que se transformaria pouco depois no Visual Studio 2002. Eu trabalhava para a consultoria Accenture nessa época desenvolvendo em Visual Basic 6 e iniciei meus estudos nas linguagens C#, lendo o máximo de livros e revistas disponíveis e acompanhando o trabalho de desenvolvedores brilhantes que tanto me ajudaram e me inspiraram, dentre os quais posso destacar Fábio Gallupo, Mauro Sant´Anna, Renato Degiovani, Israel Aeceo, Fábio Câmara, Ramon Durães, Carlos dos Santos, Renato Haddad e John Sharp.

Era um tempo em que a documentação on-line da Microsoft ainda não era tão boa e não existiam sites como o *Stack Overflow*. Os livros de programação em C# eram escritos em sua maioria por

americanos e depois traduzidos para o português, e boa parte do conteúdo novo chegava através de revistas impressas ou de guias preparatórios para as certificações da plataforma .NET, que obtive enquanto me preparava para a minha primeira aventura ousada.

Ignorando o fato de que as coisas normalmente aconteciam fora do Brasil, condensei o máximo de informações relevantes sobre a plataforma .NET obtidas através de leitura, experimentação e troca de experiências com outros profissionais, e publiquei em 2004, pela editora Digerati, o meu primeiro livro: *Segredos do Visual Studio.NET*. Era uma obra voltada para o Visual Studio 2003 que descia no detalhe sobre como tirar o máximo possível da IDE e do framework .NET para ser mais produtivo e que serviu de referência para a escrita deste *Produtividade em C#*. Segredos do Visual Studio.NET me ajudou a conquistar excelentes oportunidades no mercado de trabalho e a conhecer pessoas fantásticas. O livro estava tão à frente do seu tempo que a própria Microsoft Press só foi publicar um livro com abordagem semelhante em 2008 (4 anos depois) com o título de *Microsoft Visual Studio Tips* da autora Sara Ford, uma ex-funcionária da Microsoft que foi responsável pelo projeto do site Codeplex.com.

No início dos anos 2000, outro profissional que considero o mais brilhante desenvolvedor que o Brasil já teve havia iniciado o seu site de programação voltado para VB.NET e C#, um portal que tanto ajudou a comunidade técnica brasileira ao longo dos últimos 20 anos com uma infinidade de artigos voltados para as plataformas .NET e .NET Core. Esse autor incansável e apaixonado por programação se chama José Carlos Macoratti e o site em questão é o Macoratti.NET que dispensa apresentações. Foi com muito orgulho que recebi a notícia de que o mestre Macoratti

havia aceitado o meu convite para escrever o prefácio deste livro. Muito obrigado de coração por este presente!

Passei os últimos 20 anos da minha vida desenvolvendo projetos para a plataforma .NET/.NET Core e ao longo desse tempo vi surgir outros profissionais brilhantes em eventos, palestras, artigos, cursos e livros ou no próprio mercado de trabalho. Dentre eles, gostaria de nomear Ricardo Guerra, Lucas Massena, Alexandro Prado, Luciano de Almeida Reis, Ricardo Portella, Marcio Elias, Henrique Tensfield, Cláudio Menezes, Sérgio Costa, Paulo Brightmore, Ramon Silva, André Baltieri, Renato Groffe, Giovanni Bassi, Luca Gabrielli, Fernando José Pedrosa, Avedis Bochoglonian e Rubens A. Lucca.

Todos vocês de alguma forma me ajudaram a transformar em realidade o sonho de escrever um projeto tão complexo como Produtividade em C#. A todos vocês, incluindo os amigos que por algum lapso de memória eu tenha esquecido, o meu muito obrigado!

Cláudio Ralha

SOBRE O AUTOR

Cláudio Ralha é cientista de dados, arquiteto de software, autor de vários livros e treinamentos, palestrante nas áreas de inteligência investigativa e desenvolvimento de software e apaixonado por compartilhar conhecimento.

Com 35 anos de experiência em informática e várias certificações Microsoft e IBM, o autor já atuou como colunista de informática do jornal O GLOBO (na época, tinha 18 anos de idade), editor técnico da revista Infomania, colaborador de revistas como CPU PC, CPU MSX, Geek, Hackers e PC Brasil, além de articulista em grandes portais como Macoratti.NET e Linha de Código.

Ao longo de sua carreira, Cláudio também ocupou cargos de gerente de desenvolvimento, analista desenvolvedor e administrador de banco de dados. Foi consultor de clientes como Petrobras, Shell, Esso, Furnas, Brascan, CVM, ONS, Organizações Globo, Rede Globo, Jornal O GLOBO, Brasil Veículos, Elsevier e SAAB Medav (Alemanha). Além disso, atuou em parceria com grandes consultorias como Accenture, Stefanini, IBM, ATT/PS, InforMaker, Relacional, Value Team, Softtek, Provider IT, CECAM e 3S Tecnologia.

Em 2003, fundou a Halteck Consulting onde permanece como CEO e instrutor. A empresa, com foco em consultoria, treinamentos e desenvolvimento de software, tem se destacado pelos serviços prestados, tanto em território nacional quanto no exterior, nas áreas de inteligência investigativa e desenvolvimento

de software para as forças da lei, Exército Brasileiro e o setor privado.

PREFÁCIO

Quando recebi o convite para escrever o prefácio deste livro, fiquei a princípio apreensivo pela responsabilidade, mas como conheço a obra do autor e o seu trabalho como profissional, também fiquei feliz com essa distinção e empolgado em apresentar o trabalho de um autor nacional sobre um tema atual que visa compartilhar conhecimento técnico sobre a principal linguagem da plataforma .NET.

Conheço o trabalho do Cláudio, que vem atuando há tempos na área de TI, uma boa parte com a plataforma .NET, e ele sempre se mostrou um profissional esmerado em seu trabalho e preocupado em estar em constante sintonia com a evolução que é inerente à nossa área de atuação.

Sou do tempo da internet discada e acompanho a evolução da linguagem C# desde o lançamento da versão 1.0 e sua crescente aceitação desde então. Com essa constante evolução e com o advento da .NET Core, um ambiente multiplataforma, a linguagem expandiu seus horizontes e suas possibilidades de atuação para fora do mundo Windows.

Isso com certeza acabou atraindo muitos desenvolvedores e desenvolvedoras para a plataforma .NET e a linguagem C# acabou se tornando a principal ferramenta de trabalho para codificação no ambiente .NET Core. Com toda essa responsabilidade, a linguagem precisa evoluir constantemente para incorporar novos recursos e assim poder continuar oferecendo um ambiente produtivo e robusto.

Nesse contexto, estar atualizado e conhecer os novos recursos da linguagem C# e do ambiente .NET Core pode fazer a diferença tanto no quesito desempenho quanto na produtividade. Embora possamos encontrar na internet muitos artigos relacionados ao assunto, nada como um bom livro para consultar e ler a qualquer momento. Por isso, sou um apreciador de livros com conteúdos técnicos relacionados a minha área de atuação, pois sei que mesmo após muitos anos de experiência na área sempre temos algo a aprender com outros profissionais e suas contribuições.

Ao lançar mais esta obra, na qual apresenta os recursos da linguagem C# de forma simples e objetiva, o autor nos brinda com um texto em português sobre um tema bem atual, abordando assuntos de interesse de toda a comunidade .NET. O livro serve para os que atuam profissionalmente na área, para aqueles que usam C# como um meio para alcançar seu objetivo e também para quem deseja migrar para essa linguagem.

Assim, este livro traz uma visão panorâmica atual da linguagem C#, comparando a evolução do código com exemplos práticos e pontuais, e, dessa forma, você não fica obrigado a uma leitura sequencial, podendo alternar entre os capítulos conforme o tópico de seu interesse no momento e usando a obra como um manual de consulta e referência para o dia a dia.

Bom estudo!

José Carlos Macoratti (Fundador do Portal Macoratti.NET)

INTRODUÇÃO

Alta produtividade é um objetivo perseguido por todo bom profissional em qualquer área. Quando se fala em desenvolvimento de software, especificamente, para alcançá-la não basta ter foco e esforço. Isso porque a todo momento surgem novas linguagens de programação que a cada release trazem novas funcionalidades, sem falar em uma infinidade de frameworks e bibliotecas. Infelizmente, mesmo que estudemos da hora de acordar até a hora de dormir todos os dias de nossas vidas, não seremos capazes de assimilar e principalmente colocar em prática, nos projetos em desenvolvimento, todas as opções fornecidas pelos fabricantes das tecnologias que utilizamos, pelo simples fato de que elas evoluem e são trocadas em uma velocidade insana.

Por outro lado, se pararmos para analisar com calma as principais linguagens de programação utilizadas no mercado de TI, veremos que a maior parte possui funcionalidades similares, como suporte a *Orientação a Objetos*, *Generics*, *Inferência de tipos*, *Tipos anônimos*, *Expressões Regulares*, *Reflection*, *Programação Assíncrona*, dentre outras.

Em termos práticos, o que faz toda a diferença é o fato de que algumas linguagens foram pensadas para serem mais concisas e legíveis que outras, características estas que aumentam substancialmente a nossa produtividade. Junte a uma delas uma IDE poderosa como o Visual Studio e um profissional que conhece bem suas ferramentas de trabalho e você terá a fórmula perfeita para se destacar no mercado e tocar seus projetos profissionais e pessoais na velocidade que almeja.

Criada há cerca de 22 anos, tendo como base linguagens como C++ e Java, a linguagem C# tem inovado a cada novo release e revolucionado a forma como se produz software. Ela é hoje a principal linguagem de desenvolvimento da Microsoft para as plataformas desktop, web e mobile e, com frequência, inclui funcionalidades que rapidamente caem no gosto dos desenvolvedores e depois são copiadas pelos concorrentes.

Ao longo deste livro 100% prático e recheado de exemplos curtos e simples, reunimos recursos suportados pelas versões mais recentes do compilador C# que, ao serem aplicados no código produzido, vão impactar positivamente no tamanho do código gerado e no tempo envolvido, resultando em alta produtividade.

Seguindo o mantra do "menos é mais", você verá que existem novas formas de se resolver velhos problemas e que, só porque algo funciona, não significa que seja a melhor forma de concluir a tarefa. *Produtividade em C#* foca no que realmente importa para quem quer produzir um código elegante e de qualidade. Tenha em mente que o nosso código é o nosso melhor cartão de visitas e que é preciso se reinventar sempre. Venha conosco conhecer o que a linguagem C# tem de melhor a nos oferecer.

Bom estudo e sucesso em seus projetos!

Cláudio Ralha

(Novembro de 2020)

EXECUTANDO OS PROJETOS UTILIZANDO O .NET 5

A unificação dos frameworks .NET e .NET Core está se tornando uma realidade com a chegada do .NET 5. É uma mudança inevitável para a qual precisamos estar preparados e que nos brindará com vários recursos novos introduzidos no C# 9.0 e outros que já estão em desenvolvimento para o C# 10.0.

No momento em que a escrita deste livro foi finalizada (novembro de 2020), o novo framework havia acabado de ser lançado e para testar as novidades introduzidas na linguagem C# 9.0, ainda é necessário executar alguns passos que envolvem:

a) A atualização da sua cópia do Visual Studio através do instalador do Visual Studio. Note que não é mais necessário instalar o *Visual Studio Preview*, versão da IDE que contém recursos ainda não incluídos na versão de produção. A distribuição *Community* gratuita é suficiente para executar todos os exemplos deste livro.

b) A configuração da versão correta do .NET em uso após a criação de um projeto. Por enquanto, os projetos aparecem pré-configurados para o .NET 3.1 apesar de o .NET 5 já estar presente na lista de opções. Esses passos obviamente deixarão de ser necessários em futuras atualizações do Visual Studio.

Para criar uma aplicação de teste, execute os seguintes passos:

1. Execute o Visual Studio atualizado e crie um novo projeto do tipo *Aplicação de Console (.NET Core)* em C#. Atenção neste

passo para não se confundir e escolher a opção *Aplicação de Console (.NET Framework)*, que só permitirá escolher até o Framework .NET 4.8.

2. O projeto criado está configurado por padrão para usar o .NET Core 3.1 e o C# 8. Para configurá-lo para usar o .NET 5, clique com o botão direito do mouse sobre o arquivo do projeto no *Gerenciador de Soluções* e selecione *Propriedades*. A página de propriedades do projeto abrirá. Na guia *Aplicativo*, selecione em *Estrutura de Destino* a opção *.NET 5.0* e a seguir tecle *Ctrl + s* para salvar a alteração.

Pronto! A partir de agora você já pode experimentar os novos recursos incluídos no compilador C# 9.0.

USANDO UMA FUNCIONALIDADE DO C# 9.0 PARA TESTAR

O C# 9.0 introduziu vários recursos novos e alguns deles serão abordados neste livro, uma vez que aumentam a nossa produtividade e reduzem o nosso esforço. Para este teste inicial utilizaremos as *instruções de nível superior* (em inglês, *top level statements*).

Apesar do nome pomposo, este recurso é extremamente simples e agradará tanto a quem está iniciando na linguagem C# quanto aqueles que precisam fazer demonstrações rápidas de fragmentos de código sendo executados sem a necessidade de inserir o esqueleto de código clichê que vemos em linguagens como C#, Java e C++. Se você pensou no C# se comportando como Python, Ruby ou Lua, acertou!

Para ilustrar o seu uso vamos executar o seguinte roteiro:

1. No *Gerenciador de Soluções*, altere o nome do arquivo `Program.cs` para um nome qualquer, por exemplo, `Codigo.cs`. Note que este passo somente ilustra que para usar este recurso não estamos mais presos à convenção de ter um arquivo `Program.cs`, no qual normalmente é definido o ponto de entrada através do método `Main`.

2. Observe o código gerado pelo template da aplicação de console em C#. Note que há 12 linhas de código, mas só a linha que escreve a mensagem `Hello World!` possui código realmente executável. Usando o novo recurso, poderíamos reescrever o código anterior desta forma:

```
using System;

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

Ou em uma única linha usando:

```
System.Console.WriteLine("Hello World!");
```

3. Para conferir isso na prática, remova o código gerado pelo template do Visual Studio e insira em seu lugar o código a seguir:

```
using System;

Console.WriteLine("Testando o recurso de instruções de nível supe
rior do C# 9.0");
DateTime hoje = DateTime.Now;
DateTime ontem = hoje.AddDays(-1);
DateTime amanha = hoje.AddDays(1);
Console.WriteLine($"Ontem: {ontem:dd/MM/yyyy}");
Console.WriteLine($"Hoje: {hoje:dd/MM/yyyy}");
Console.WriteLine($"Amanhã: {amanha:dd/MM/yyyy}");
```

4. Tecle *Ctrl + F5* e observe que o código desse exemplo

executa sem erro.

Parabéns! Você acaba de executar o seu primeiro programa em .NET 5.0.

Caso receba a mensagem de erro a seguir, é sinal de que você esqueceu de ajustar a versão correta do .NET em uso:

```
    O recurso 'top-level statements' não está disponível em C# 8.0. Use a versão de linguagem 9.0 ou superior.
```

As instruções de nível superior dispensam o uso do código clichê, com o qual estamos acostumados, e nos permitem escrever scripts simples e concisos que podem utilizar qualquer classe do framework .NET, retornar valores para o programa chamador e até executar código assíncrono. Como restrição, só podemos ter um único arquivo de código no projeto que as utilize, caso contrário um erro será gerado.

Boa parte dos exemplos deste livro podem ser reescritos usando este novo recurso. Não fizemos desta forma para que ele possa ser útil para o maior número de leitores e leitoras, mas você está livre para experimentar à vontade, pois não vamos criar nenhum projeto de teste que se alongue por vários capítulos. Todos os exemplos que apresentaremos são curtos e diretos. *Produtividade em C#* é uma obra que foca no que o time de desenvolvimento incluiu de útil e produtivo em todas as versões já lançadas do compilador e não em uma versão específica da linguagem.

Sumário

Capítulo 1

STRINGS

Strings estão presentes em nossas vidas como desenvolvedores desde o primeiro *Hello World* que escrevemos. Não importa quantas linguagens diferentes tenhamos usado desde então, jamais esqueceremos a sensação de ver aquele código tão simples rodando com sucesso pela primeira vez. Esse momento único representa o começo de uma longa jornada.

Sem o uso de strings não seria possível ler entradas do usuário ou escrever mensagens em uma aplicação de console, bater papo em mensageiros instantâneos como o WhatsApp e o Telegram, postar opiniões em microblogs como o Twitter, interagir em redes sociais como o Facebook e o LinkedIn ou consultar ferramentas de busca como o Google e o Waze. Dada a sua importância, acreditamos que iniciar abordando a manipulação de strings de maneira inteligente seja a melhor porta de entrada para um livro focado em produtividade.

Uma *string* ou *cadeia de caracteres* é uma coleção sequencial de caracteres usada para representar texto. Nas plataformas .NET e .NET Core, o texto de uma string é representado como uma sequência de unidades de código UTF-16 e é possível armazenar em memória até 2GB (cerca de 1 bilhão de caracteres).

Ao longo deste capítulo, vamos apresentar formas mais produtivas de resolver várias tarefas diárias envolvendo a manipulação de strings. As dicas que reunimos nas próximas páginas tornarão o seu código mais conciso, elegante e legível.

1.1 CRIANDO STRINGS MAIS LEGÍVEIS

A concatenação de strings pode ser feita de forma elegante, o que garante maior legibilidade ao código-fonte, ou de forma básica sem recorrer a recursos mais recentes da linguagem, tornando o código mais difícil de entender. Nas próximas seções, veremos alguns recursos criados pelo time de desenvolvimento do C# para minimizar o efeito conhecido como "código spaghetti".

Usando interpolação de strings

No C# 6.0 foi introduzido um recurso chamado *interpolação de strings*. Antes de a linguagem suportá-lo, era necessário certo malabarismo para concatenar strings. Veja no exemplo a seguir uma das técnicas que era recomendada, baseada no método `Format` da classe `String`:

```
var livro = new Livro();
var mensagem = string.Format("{0} é o novo livro de {1}", livro.T
itulo, livro.Autor);
```

Compare o código anterior com a maneira mais simples disponível atualmente:

```
var livro = new Livro();
var mensagem = $"{livro.Titulo} é o novo livro de {livro.Autor}";
```

Note que, para usarmos a interpolação de strings, é necessário preceder a string com o caractere `$`.

Usando o caractere @ para preservar espaços, tabulações e quebras de linhas em strings

Incluir espaços, tabulações e quebras de linha em uma string é uma tarefa que costuma dar dor de cabeça. A solução adotada por muitos desenvolvedores pode ser vista no exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var mensagem = "Amigo leitor,\n\n\tobrigado por ter a
dquirido este livro. Esperamos que ele o ajude em sua jornada diá
ria de trabalho.\n\nBom estudo!\nCláudio Ralha";
            Console.WriteLine(mensagem);
        }
    }
}
```

Como você pode observar, o exemplo utiliza caracteres de controle para inserir quebra de linha (`\n`) e tabulação (`\t`) diretamente no texto da string. Ao ser executado, este código vai produzir a seguinte saída:

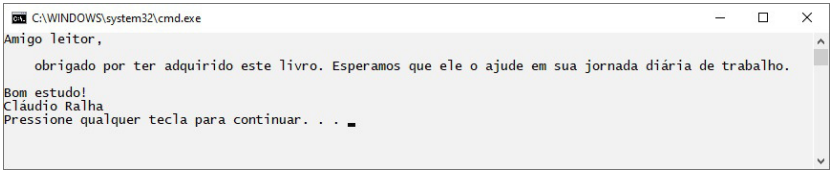


Figura 1.1: Preservando espaços, tabulações e quebras de linhas

Apesar de ser 100% funcional, este exemplo não é de fácil manutenção. Felizmente, existe uma maneira mais simples e

legível de obtermos o mesmo resultado. Basta preceder a declaração de uma string com o símbolo `@` e aplicar as quebras e tabulações como mostrado na próxima listagem:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var mensagem = @"Amigo leitor,

    obrigado por ter adquirido este livro. Esperamos que ele o aj
ude em sua jornada diária de trabalho.

Bom estudo!
Cláudio Ralha";
            Console.WriteLine(mensagem);
        }
    }
}
```

Este recurso, conhecido como *verbatim string*, é particularmente útil quando estamos criando consultas SQL e desejamos quebrar a linha para separar as cláusulas. Exemplo:

```
var sql = @"SELECT *
            FROM livros
            WHERE autor = 'Claudio Ralha'";
```

Usando o caractere @ para simplificar a representação dos caminhos de arquivos em strings

O uso do caractere `@`, mostrado no tópico anterior, anula os *caracteres de controle* da string. Por esse motivo, evita a necessidade de duplicarmos o caractere `\` usado para escapar caracteres de controle quando precisamos tê-lo representado no

texto da string. Quando não o utilizamos, somos obrigados a duplicar o caractere `\` ao especificar caminhos de pastas e arquivos do sistema operacional no Windows. Exemplo:

```
var pastaDocumentos = "C:\\Users\\ClaudioRalha\\Documents";
```

Obviamente, isso é algo que está longe de ser agradável aos olhos de um bom desenvolvedor. Para a nossa sorte, basta precedermos a string com o caractere `@` para que o compilador nos permita especificar o path para o arquivo da maneira como estamos acostumados:

```
var pastaDocumentos = @"C:\Users\ClaudioRalha\Documents";
```

Veja no próximo exemplo o mesmo recurso sendo usado para simplificar a passagem para um método de uma string de conexão para um banco de dados:

```
optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Datab
ase=Livro;Trusted_Connection=True;");
```

Esperamos que, ao atingir esse ponto, você já tenha se convencido de que o arroba e o ponto e vírgula são dois caracteres que não podem faltar no seu código.

1.2 CONVERTENDO STRINGS

A conversão de strings em arrays de bytes ou array de strings é outra tarefa que o desenvolvedor executa com frequência, principalmente quando precisa consumir métodos das classes dos frameworks .NET e .NET Core ou de bibliotecas de terceiros. Para lidar com esses cenários, vamos conhecer nas próximas seções métodos incluídos no framework para tornar a nossa vida mais simples.

Convertendo uma string em um array de bytes

Alguns métodos de classes dos frameworks .NET e .NET Core exigem que uma string seja passada como um *array de bytes*. Para fazer essa conversão com o mínimo de esforço, utilize o método estático `GetBytes`, presente na classe `Unicode` ou na classe `UTF8` do namespace `System.Text.Encoding`, para efetuar a conversão de string para `byte[]`. Exemplo:

```
string mensagem = "Produtividade em C#";
byte[] conteudo = System.Text.Encoding.Unicode.GetBytes(mensagem)
;
```

Para conferir este trecho de código em execução, execute o exemplo apresentado no próximo tópico.

Convertendo um array de bytes em uma string

Acabamos de ver como efetuar uma string em um array de bytes. Haverá casos em que precisaremos fazer o caminho contrário, ou seja, transformar um array de bytes em uma string. Para executar essa tarefa, utilize o método `GetString` presente nas classes `Unicode` ou `UTF8` do namespace `System.Text.Encoding`. Confira a seguir um exemplo completo de como efetuar esta conversão:

```
using System;
using System.Text;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void Main(string[] args)
        {
            string mensagem = "Produtividade em C#";
```

```
            Console.WriteLine("String Original:");
            Console.WriteLine(mensagem);
            byte[] conteudo = System.Text.Encoding.Unicode.GetByt
es(mensagem);

            Console.WriteLine("Concatenando todos os Bytes em uma
 string de bytes:");
            StringBuilder builder = new StringBuilder();
            for (int i = 0; i < conteudo.Length; i++)
            {
                builder.Append(conteudo[i].ToString("x2"));
            }
            Console.WriteLine(builder.ToString());

            Console.WriteLine("Usando a classe BitConverter para
colocar todos os bytes em uma string:");
            string bitString = BitConverter.ToString(conteudo);
            Console.WriteLine(bitString);

            Console.WriteLine("Obtendo a string em Unicode a part
ir dos Bytes:");
            string unicodeString = Encoding.Unicode.GetString(con
teudo, 0, conteudo.Length);
            Console.WriteLine(unicodeString);
        }
    }

}
```

Observe que esse exemplo ainda apresenta uma forma de listar os bytes de um array em uma string usando a classe `BitConverter` . Na próxima imagem, temos a saída produzida pelo código dessa listagem:

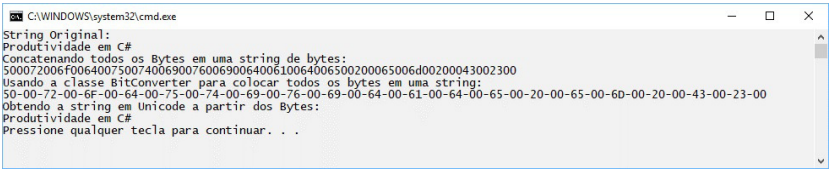


Figura 1.2: Convertendo um array de bytes em uma string

Particionando uma string em um array de strings

O método `Split` da classe `String` do C# é usado para particionar uma string em um array de strings baseado na lista de delimitadores informada. É possível informar como *split delimiters* tanto um único caractere quanto um array de caracteres ou array de strings.

Neste tópico, veremos exemplos de códigos que exploram a utilização deste método com diferentes delimitadores para particionar strings com o mínimo de esforço.

A forma mais simples de uso do método `Split` pode ser vista neste primeiro exemplo:

```
using System;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Livros recentes:");
            Console.WriteLine();
            //Livros separados apenas por vírgulas
            string livrosRecentes = "Produtividade em C#, Guia de
 Validação de Dados em C#, Guia de Validação de Dados em Visual B
asic, Guia de Validação de Dados em Python";
            string[] lista = livrosRecentes.Split(", ");
            foreach (string livro in lista)
                Console.WriteLine(livro);
        }
    }

}
```

Aqui, os nomes dos livros estão separados por uma vírgula

seguida de um espaço. Confira a saída obtida ao executar o código:

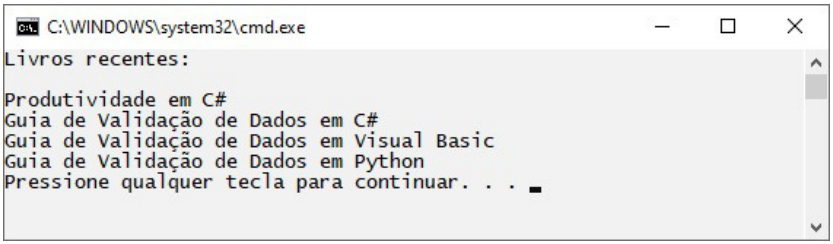


Figura 1.3: Particionando uma string em um array de strings

Em boa parte dos casos, esta forma básica de particionar uma string já será suficiente, variando apenas o caractere usado como delimitador. O próximo exemplo ilustra como tratar um cenário mais complexo:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            string pensamento = @"Nosso cérebro é o melhor brinqu
edo já criado:
Nele se encontram todos os segredos, inclusive o da felicidade.
A vida é maravilhosa se você não tem medo dela.";
            Console.WriteLine("Pensamento de Charles Chaplin:");
            Console.WriteLine();
            Console.WriteLine(pensamento);
            Console.WriteLine();
            Console.WriteLine("Palavras:");
            Console.WriteLine();
            string[] lista = pensamento.Split(new Char[] { ' ', '
,', '.', ':', '-', '\n', '\t' });
            foreach (string palavra in lista)
            {
                if (palavra.Trim() != "")
                    Console.WriteLine(palavra);
```

```
            }
        }
    }

}
```

Conforme você pode observar ao examinar o código, fornecemos um array de delimitadores que nos permitiram quebrar com pouco código uma string complexa em uma lista de palavras.

Testando se uma string é nula ou vazia

Escrever um código para testar se uma variável do tipo string contém *nulo* ou *vazio* é uma das tarefas que fazemos com maior frequência no dia a dia. Ao codificar esse tipo de teste, muitos desenvolvedores criarão códigos como o mostrado a seguir:

```
string nome = string.Empty;
if (nome == null || nome == string.Empty)
{
    Console.WriteLine("variável nome está vazia ou contém null");
}
else
{
    Console.WriteLine("nome: {nome}");
}
```

Esse trecho de código realmente efetua os dois testes mencionados, mas podemos realizar a mesma validação com menos código utilizando o método `IsNullOrEmpty` da classe `String`, conforme ilustrado no próximo fragmento de código:

```
string nome = string.Empty;
if (string.IsNullOrEmpty(nome))
{
    Console.WriteLine("variável nome está vazia ou contém null");
}
else
```

```
{
    Console.WriteLine("nome: {nome}");
}
```

1.3 FORMATANDO STRINGS

A linguagem C# oferece outras formas de formatar strings bastante úteis para o desenvolvedor, além da *interpolação de strings* vista previamente. Quando o nosso objetivo é formatar um valor específico, podemos usar tanto o método estático `Format` da classe `String`, quanto o método `ToString`, presente na classe base `Object` e herdado por todas as demais.

Usando String.Format para formatar strings

O exemplo a seguir demonstra como separar data e hora utilizando o método `Format` da classe `String`:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Formatação de data e hora
            DateTime agora = DateTime.Now;
            string dtStr = String.Format("{0:d} às {0:t}", agora)
;
            Console.WriteLine(dtStr);
        }
    }

}
```

Confira o resultado produzido por esse exemplo na próxima janela:

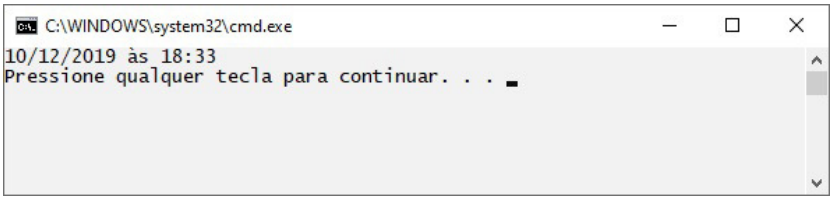


Figura 1.4: Formatando strings usando o método Format

Neste segundo exemplo, mostraremos como formatar o preço da gasolina em reais com duas casas decimais, utilizando novamente o método `String.Format` :

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Formatação de valor monetário
            decimal combustivel = 4.49m;
            string precoCombustivel = String.Format("Combustível:
 {0, 0:C2}", combustivel);
            Console.WriteLine(precoCombustivel);
        }
    }

}
```

Ao ser executado, esse código produzirá o seguinte resultado em uma máquina rodando Windows em português do Brasil:

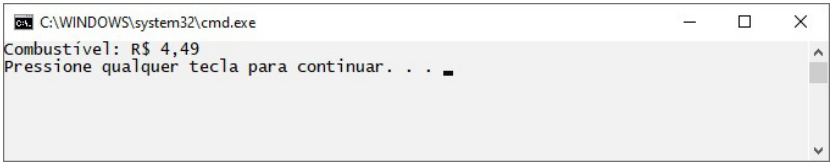


Figura 1.5: Formatando strings usando o método Format

O terceiro e último exemplo desta seção faz uso do método `ToString` e de uma máscara para transformar um valor numérico em uma string representando um CEP:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Formatação usando uma máscara
            string mascara = "00000-000";
            int cep = 22620172;
            Console.WriteLine(cep.ToString(mascara));
        }
    }

}
```

Confira na próxima imagem a saída produzida pelo código anterior:

Figura 1.6: Formatando strings usando o método ToString

Ao trabalhar com formatação de valores na plataforma .NET, tenha em mente que este é um recurso bastante poderoso e flexível que oferece suporte a múltiplas culturas e que requer algumas horas de estudo e testes para ser entendido, mas, uma vez dominado, trará benefícios significativos para o seu trabalho como desenvolvedor. Trata-se de um daqueles assuntos tão desafiantes e

apaixonantes quanto as expressões regulares, pois oferece uma infinidade de possibilidades.

> Para saber mais sobre este tema, consulte a seguinte página da documentação oficial:
>
> https://docs.microsoft.com/pt-br/dotnet/standard/base-types/formatting-types?view=net5

Substituindo o método `ToString` de um tipo

Cada classe que definimos em C# inclui o método `ToString`, pois este método é definido na classe `System.Object`, que corresponde à *classe base* de todas as classes. Por padrão, o método `ToString` retorna uma string contendo o nome da própria classe e está disponível em todas as classes que criamos.

Neste tópico, aprenderemos a substituir o método `ToString` de modo que possamos retornar informações sobre os objetos que sejam mais significativas e legíveis para humanos.

Ainda que o nome da classe retornado por padrão pelo método `ToString` possa ser útil, ele não transmite nenhuma informação sobre o conteúdo das propriedades do objeto. Confira rodando o exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class WebSite
    {
```

```
        public string Name { get; set; }
        public string URL { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            WebSite web = new WebSite();
            web.Name = "Claudio Ralha";
            web.URL = "http://www.claudioralha.com/";
            Console.WriteLine(web);
        }
    }

}
```

Ao ser executado, ele vai produzir a seguinte saída:

Figura 1.7: Exibindo o retorno do método ToString de um tipo

Isso ocorre porque o método `WriteLine` internamente chama o método `ToString` do objeto `web`. Para fornecer detalhes mais apropriados aos clientes de suas turmas, será necessário substituir o método `ToString` usando a sintaxe padrão, conforme ilustrado no exemplo a seguir:

```
class WebSite
{
    public string Name { get; set; }
    public string URL { get; set; }

    public override string ToString()
    {
        return $"{Name} - {URL}";
```

```
    }
}
```

As informações que você retorna de sua versão substituída do `ToString` podem ser usadas para muitos propósitos nos quais o texto legível é desejável. Um dos seus usos mais populares é dentro do depurador do Visual Studio. Quando o depurador mostra detalhes de variáveis locais ou observadas, a primeira informação exibida é gerada chamando o método `ToString`.

Em nosso caso, se rodarmos uma vez mais o exemplo, veremos que agora são exibidas informações úteis sobre a classe. Confira:

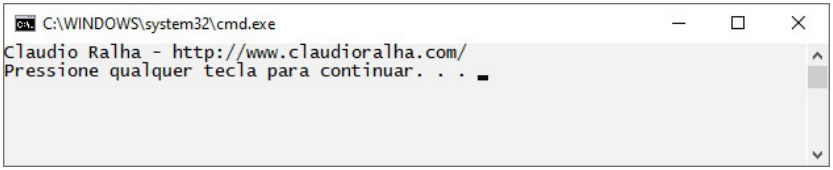


Figura 1.8: Substituindo o método ToString de um tipo

1.4 SORTEANDO UMA STRING DE UM ARRAY OU LISTA

Sortear valores é uma tarefa que ocorre com frequência no desenvolvimento de sistemas. Ao longo desta seção, demonstraremos como sortear um item de um array ou lista com o mínimo de código usando a classe `Random`.

O primeiro exemplo sorteia um funcionário a partir de uma lista mantida em um array. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            string[] funcionarios = { "Adriana", "Patrícia", "Mar
ia Vitória",
                                      "Carlos", "Vinícius", "Paul
o", "Leandro",
                                      "Karina", "Luana", "Juliana
 };

            Random rand = new Random();
            int indice = rand.Next(funcionarios.Length);
            Console.WriteLine($"Funcionário: {funcionarios[indice
]}");
        }
    }

}
```

A classe `Random` possui três métodos públicos: `Next`, `NextBytes` e `NextDouble`. O método `Next` retorna um número aleatório, `NextBytes` retorna uma matriz de bytes preenchidos com números aleatórios e `NextDouble` retorna um número aleatório entre 0.0 e 1.0.

O exemplo utiliza o método `Next` para sortear um valor que corresponde ao índice do array a ser retornado. O tamanho do array é passado como parâmetro para que não seja gerado nenhum índice inexistente.

Na próxima listagem, temos o mesmo código reescrito usando uma lista genérica:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            List<string> funcionarios = new List<string>()
            {
                "Adriana", "Patrícia", "Maria Vitória",
                "Carlos", "Vinícius", "Paulo", "Leandro",
                "Karina", "Luana", "Juliana"
            };


            Random rand = new Random();
            int indice = rand.Next(funcionarios.Count);
            Console.WriteLine($"Funcionário: {funcionarios[indice
]}");
        }
    }

}
```

Conforme você pode observar, esta é uma tarefa simples e rápida de implementar. A seguir, apresentamos um exemplo de uso do método `NextDouble` da classe `Random`.

1.5 GERANDO UMA STRING RANDÔMICA

Ao desenvolvermos sistemas, por vezes temos que criar uma rotina que gere uma senha temporária para os usuários. O exemplo a seguir ilustra como utilizar a classe `Random` para sortear essa senha. Veja:

```
using System;
using System.Text;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static string RandomString(int tamanho, bool caixa
Baixa)
```

```
        {
            StringBuilder sb = new StringBuilder();
            Random random = new Random();
            char ch;
            for (int i = 0; i < tamanho; i++)
            {
                ch = Convert.ToChar(Convert.ToInt32(Math.Floor(26
 * random.NextDouble() + 65)));
                sb.Append(ch);
            }
            if (caixaBaixa)
                return sb.ToString().ToLower();
            return sb.ToString();
        }

        static void Main(string[] args)
        {
            var senha = RandomString(8, true);
            Console.WriteLine($"Senha: {senha}");
        }
    }

}
```

A função `RandomString` deste exemplo recebe como parâmetros o tamanho da string a ser gerada e um valor booleano indicando se ela será em caixa baixa ou não.

Ao examinar esse código, é provável que você se pergunte se não há formas mais simples, elegantes ou concisas de gerar esse tipo de string. A seguir, vamos reproduzir algumas soluções disponíveis na seguinte thread do site StackOverflow:

https://stackoverflow.com/questions/1344221/how-can-i-generate-random-alphanumeric-strings

A primeira proposta fornecida por um desenvolvedor que se identifica apenas como *dtb* utiliza LINQ:

```
private static Random random = new Random();
```

```
public static string RandomString(int length)
{
    const string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    return new string(Enumerable.Repeat(chars, length)
      .Select(s => s[random.Next(s.Length)]).ToArray());
}
```

No caso de strings randômicas de até 22 caracteres, é possível utilizar um valor GUID como base para a string. Confira a seguir soluções propostas pelos usuários Douglas e Luis Perez para obter uma senha randômica de 8 caracteres:

```
Convert.ToBase64String(Guid.NewGuid().ToByteArray()).Substring(0,
8);
```

Ou com ainda menos código:

```
Convert.ToBase64String(Guid.NewGuid().ToByteArray()).Substring(0,
8);
```

1.6 UTILIZANDO COMENTÁRIOS ESPECIAIS PARA SINALIZAR O CÓDIGO

O Visual Studio reconhece um tipo especial de comentário que utiliza *tokens*, como `TODO`, `HACK` ou tokens criados pelo próprio desenvolvedor. Esse tipo de comentário é usado como atalho para sinalizar que um trecho de código não está finalizado, possibilitando ao desenvolvedor retornar rapidamente a esse ponto do código-fonte e dar seguimento ao trabalho.

Para sinalizar que algo precisa ser continuado depois, basta adicionar uma linha de comentário contendo o token `TODO` (em inglês, "A fazer"). Exemplo:

```
//TODO: adicionar um cabeçalho informativo
```

Quando quiser retornar a este local predefinido no código com mínimo de esforço, utilize a janela *Lista de Tarefas* (*Task List*) do Visual Studio. Clique no menu *View* e selecione a opção *Task List*. Veja que o comentário aparece na lista:

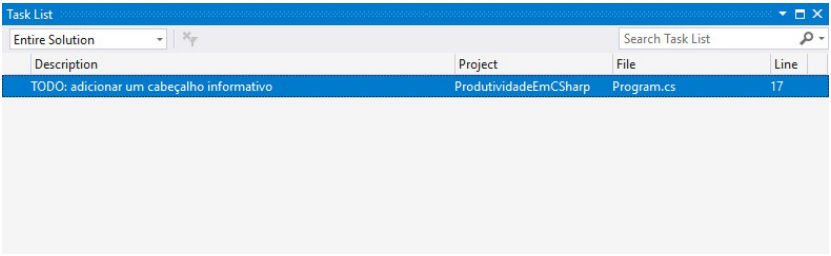


Figura 1.9: Inspecionando a janela Lista de Tarefas do Visual Studio

Efetue um duplo clique sobre o item desejado nesta janela e observe que o arquivo que contém o comentário é carregado no editor de código e o cursor de edição é movido para a linha que contém o token.

1.7 UTILIZANDO EXPRESSÕES REGULARES PARA VALIDAR DADOS

Ao longo dos últimos anos, escrevi vários livros em diversas linguagens para uma série chamada *Guia de Validação de Dados* baseada em *expressões regulares* e algoritmos de checagem de dígitos verificadores. Esta seção contém dois exemplos do *Guia de Validação de Dados em C#* incluídos para demonstrar o poder de síntese das expressões regulares.

As *expressões regulares* são usadas como ferramentas para executar com pouco código tarefas complexas, como localizar e

extrair múltiplas ocorrências de sequências de caracteres específicas dentro de um texto e validar se os dados fornecidos pelo usuário obedecem a um formato bem definido.

Veja a seguir um exemplo simples de como validar um CEP entrado pelo usuário baseado na máscara `99.999-999` usando expressões regulares em C#:

```
using System;
using System.Text.RegularExpressions;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            string padrao = @"^\d{2}\.\d{3}-\d{3}$";
            string resposta = string.Empty;
            Regex regex = new Regex(padrao);
            while (true)
            {
                Console.Write("CEP: ");
                resposta = Console.ReadLine().Trim();
                if (resposta.ToUpper() == "FIM") break;
                string resultado = regex.IsMatch(resposta) ? "vál
ido" : "inválido";
                Console.WriteLine($"{resposta} é um valor {result
ado}!\n");
            }
        }
    }
}
```

Ao analisar o exemplo, observe que utilizamos a classe `Regex` presente no namespace `System.Text.RegularExpressions`. Esta classe contém métodos para executar todo tipo de operações envolvendo expressões regulares. Como o nosso objetivo nesta seção é apenas validar entradas do usuário, vamos explorar apenas

o método `IsMatch`, que é usado para verificar se a string passada como parâmetro "casa" (combina) com a expressão regular armazenada. Note que utilizamos um verbatim (caractere `@`) antes da string da expressão regular armazenada na variável padrão para evitar a duplicação das barras invertidas.

As entradas do usuário são capturadas no loop `while` através do método estático `ReadLine` da classe `Console`. O método `Trim` é aplicado à variável `resposta` para remover eventuais espaços colocados por engano no início ou no fim do valor fornecido. Já o método `ToUpper` é usado para converter o valor entrado em letras maiúsculas para testar se este é igual à palavra `FIM`, o que significa que o programa será encerrado se o usuário digitar `FIM`, `Fim`, `fim` ou qualquer outra variação utilizando apenas essas três letras nesta ordem.

Devido ao seu poder de síntese, uma expressão regular (ER) nem sempre é algo simples de ser construído, uma vez que ela é capaz de condensar em uma única sequência de *metacaracteres* várias regras que compõem um padrão a ser testado. Para lidar com esses cenários mais complexos e trabalhosos, a melhor abordagem ao elaborar a expressão regular é a de "dividir para conquistar". Na prática, isso significa dividir a ER nas partes que a compõe e documentar cada uma delas no momento da sua criação, após efetuar uma bateria de testes.

Na tabela a seguir, temos um exemplo de documentação da expressão regular que utilizamos para validar o CEP no exemplo anterior:

Parte	Significado
^	O início de uma linha

`\d`	Um algarismo
`{2}`	Ocorrendo exatamente duas vezes
`\.`	Um ponto obrigatório
`\d`	Um algarismo
`{3}`	Ocorrendo exatamente três vezes
`\-`	Um hífen obrigatório
`\d`	Um algarismo
`{3}`	Ocorrendo exatamente três vezes
`$`	O fim da linha.

Confira na próxima imagem um exemplo de teste do programa de validação de CEPs:

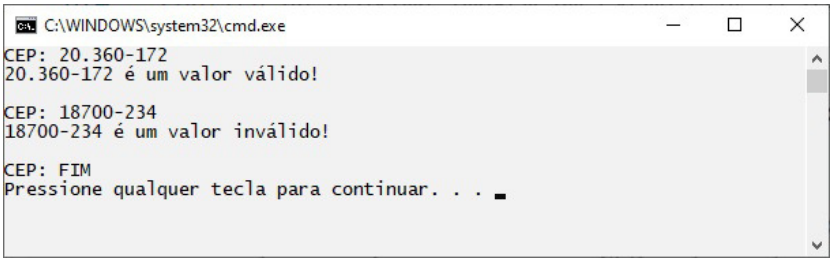


Figura 1.10: Validando entradas do usuário usando uma expressão regular

Observe que a expressão regular que utilizamos é capaz de reconhecer apenas o CEP informado segundo a máscara `99.999-999` . Este é o formato oficial para código postal e a expressão regular que criamos cumpriu o seu papel, mas o que fazer se quisermos aceitar também CEPs baseados nas máscaras `99999-999` e `99999999` ? Simples, bastaria alterar a expressão regular para aceitar os três formatos. Veja:

```
^(\d{8})|(\d{5}-\d{3})|(\d{2}\.\d{3}-\d{3})$
```

Infelizmente, o receio de ser obrigado a dar manutenção em expressões regulares complexas, por si só, é capaz de intimidar alguns desenvolvedores a ponto de não tirarem proveito deste recurso fantástico. O que muitas vezes eles não sabem ou se esquecem é que, primeiro, a maior parte das expressões regulares em uso em aplicações comerciais são relativamente simples e, segundo, que após uma expressão regular ter sido criada e bem testada, dificilmente precisará de manutenção.

Tenha em mente, contudo, que nem sempre as expressões regulares serão suficientes para garantir por si só que o dado que está sendo testado é válido. Em alguns casos, por exemplo, como CPF e CNPJ existem *dígitos verificadores* no final do número que também precisam ser validados juntamente com a máscara. Para estes cenários, precisamos combinar expressões regulares com rotinas de checagem de dígitos verificadores. O exemplo a seguir demonstra como validar CPFs:

```
using System;
using System.Text.RegularExpressions;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static bool ValidarCPF(string cpf)
        {
            int n1, n2;
            string[] cpfsInvalidos = new string[] { "00000000000"
, "11111111111", "22222222222", "33333333333", "44444444444", "55
555555555", "66666666666", "77777777777", "88888888888", "9999999
9999" };
            string parteNumerica = cpf.Replace(".", "").Replace("
-", "");
            foreach (string cpfInvalido in cpfsInvalidos)
```

```
            {
                if (parteNumerica == cpfInvalido) return false;
            }
            for (int x = 0; x <= 1; x++)
            {
                n1 = 0;

                for (int i = 0; i <= 8 + x; i++)
                {
                    n1 = n1 + Convert.ToInt32(parteNumerica.Subst
ring(i, 1)) * ((10 + x) - i);
                }

                n2 = 11 - (n1 - (Convert.ToInt32((n1 / 11) * 11))
);
                if ((n2 == 10) | (n2 == 11)) n2 = 0;
                if (n2 != Convert.ToInt32(parteNumerica.Substring
9 + x, 1))) return false;
            }
            return true;
        }

        static void Main(string[] args)
        {
            string padrao = @"^\d{3}\.\d{3}\.\d{3}-\d{2}$";
            string resposta = String.Empty;
            Regex regex = new Regex(padrao);
            while (true)
            {
                Console.Write("CPF: ");
                resposta = Console.ReadLine().Trim();
                if (resposta.ToUpper() == "FIM") break;
                string resultado = (regex.IsMatch(resposta) && Va
lidarCPF(resposta)) ? "válido" : "inválido";
                Console.WriteLine($"{resposta} é um valor {result
ado}!\n");
            }
        }
    }
}
```

Os valores fornecidos para validar um número de CPF devem conter 11 dígitos e incluir obrigatoriamente a presença de ponto e

hífen de acordo com a máscara `999.999.999-99`. É importante destacar que CPFs que apresentam o mesmo algarismo em todas as posições (`000.000.000-00`, `111.111.111-11` etc.) passam na validação dos dígitos verificadores, mas são considerados inválidos.

A expressão regular que utilizamos para validar o CPF verifica apenas se a entrada fornecida está em conformidade com a máscara `999.999.999-99`:

```
^\d{3}\.\d{3}\.\d{3}-\d{2}$
```

Veja a seguir um teste com o CPF fictício válido `637.658.268-04`:

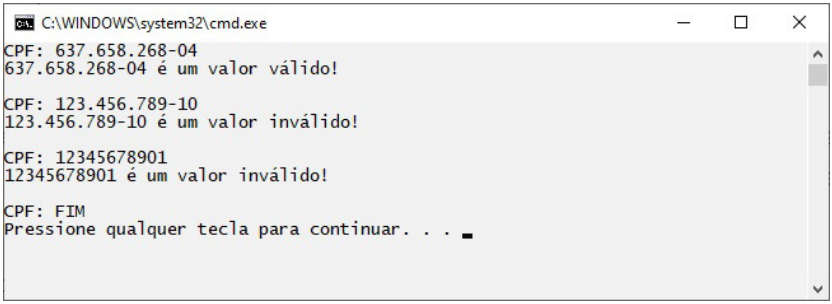


Figura 1.11: Validando entradas do usuário usando uma expressão regular e uma função de validação de dígitos verificadores

Caso deseje aceitar CPFs com ou sem máscara, utilize a seguinte expressão regular alternativa:

```
^(\d{11})|(\d{3}\.\d{3}\.\d{3}-\d{2})$
```

Para saber mais sobre o uso de expressões regulares em .NET e .NET Core, acesse a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/standard/base-types/regular-expressions

Ao longo deste capítulo inicial, exploramos diferentes formas de trabalhar com strings em C#, demonstrando como é possível chegar ao resultado desejado com pouco código quando se conhece bem a sua ferramenta de trabalho.

Assim como os idiomas estão em constante evolução e novas palavras vão surgindo ou ganhando novos significados com o passar dos anos, uma linguagem como o C# também ganha novas funcionalidades ao longo do tempo, com base nas ideias do time de desenvolvimento do compilador e em contribuições da comunidade técnica.

Para ter uma ideia do que o futuro nos reserva, visite a seguinte página da linguagem C# no GitHub:

https://github.com/dotnet/csharplang/projects/4

Capítulo 2

OPERADORES

O C# fornece um grande conjunto de operadores predefinidos, compatíveis com os tipos internos da linguagem. Operadores são símbolos que especificam quais operações serão executadas em uma expressão. A precedência e a associação dos operadores na expressão determinam a ordem de execução das operações, o que pode ser alterado pelo uso de parênteses.

A linguagem permite que alguns dos operadores existentes sejam sobrecarregados, alterando seu significado quando aplicados a um tipo definido pelo usuário.

Neste capítulo, veremos como empregar operadores que reduzem consideravelmente o tamanho do código-fonte gerado, sem reduzir a sua legibilidade. Nosso tour se iniciará com os *operadores de incremento e de decremento*, passando pelos *operadores especiais de atribuição*. Abordaremos a seguir os *operadores condicional nulo e de coalescência nula*, que podem simplificar a lógica e evitar erros. Em seguida, falaremos sobre o *operador ternário*, que nos ajuda a tornar o código mais conciso e que costuma confundir alguns desenvolvedores e desenvolvedoras. Na seção seguinte, falaremos um pouco sobre o uso dos operadores `is` e `is not` na criação de código legível (no próximo capítulo, forneceremos mais exemplos úteis quando

apresentarmos as melhorias incluídas no C# 9.0 relacionadas com a *correspondência de padrões*). A última parada será demonstrar como a *sobrecarga de operadores* pode tornar o nosso código mais simples.

Ao final, você terá constatado que o emprego de operadores é uma das melhores formas para se praticar a velha máxima do "menos é mais", tão em moda atualmente.

2.1 USANDO OS OPERADORES DE INCREMENTO ++ E DE DECREMENTO --

O operador de incremento (`++`) aumenta seu operando por `1` . Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int vitoria;
            vitoria = 1;
            Console.WriteLine(++vitoria);
            vitoria = 1;
            Console.WriteLine(vitoria++);
            Console.WriteLine(vitoria);
        }
    }

}
```

Confira o resultado na próxima imagem:

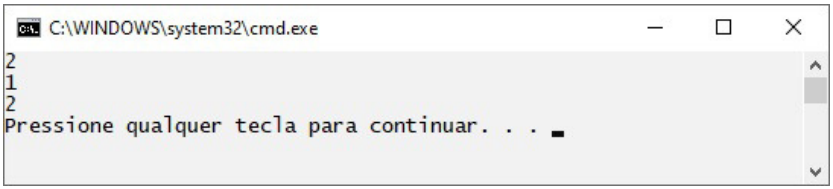


Figura 2.1: Usando o operador de incremento ++

Vale destacar que o operador pode aparecer antes ou depois de seu operando, o que naturalmente confunde alguns desenvolvedores, e é usado com frequência em testes de processos seletivos. No contexto deste livro, para ser mais produtivo, basta lembrar que escrever:

```
vitoria++
```

é mais rápido e menos cansativo que:

```
vitória = vitória + 1
```

O mesmo raciocínio vale para o operador de decremento `--`, usado para decrementar o operando em 1.

Os operadores de incremento `++` e decremento `--` são usados com frequência em loops `for`. O exemplo a seguir ilustra este cenário:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine(i);
```

```
            }
            for (int i = 5 - 1; i >= 0; i--)
            {
                Console.WriteLine(i);
            }
        }
    }

}
```

No próximo exemplo, utilizamos a variável `c` do tipo `char` como variável de controle do loop `for` e a incrementamos usando o operador de incremento. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            for (char c = 'a'; c <= 'e'; c++)
            {
                Console.WriteLine(c);
            }
        }
    }

}
```

2.2 USANDO OS OPERADORES ESPECIAIS DE ATRIBUIÇÃO

O C# dispõe de vários operadores de atribuição. Veja a seguir a lista extraída da documentação oficial:

- `x = y` para atribuição.
- `x += y` para incremento. Adicione o valor de y para o

valor de x, armazene o resultado em x e retorne o novo valor.

- `x -= y` para subtração. Subtraia o valor de y do valor de x, armazene o resultado em x e retorne o novo valor.
- `x \*= y` para atribuição de multiplicação. Multiplique o valor de y com o valor de x, armazene o resultado em x e retorne o novo valor.
- `x /= y` para atribuição de divisão. Divida o valor de x pelo valor de y, armazene o resultado em x e retorne o novo valor.
- `x %= y` para atribuição restante. Divida o valor de x pelo valor de y, armazene o resto em x e retorne o novo valor.
- `x &= y` para atribuição composta de um operador binário AND. Efetue um AND entre o valor de y e o valor de x, armazene o resultado em x e retorne o novo valor.
- `x |= y` para atribuição composta de um operador binário OR. Efetue um OR entre o valor de y e o valor de x, armazene o resultado em x e retorne o novo valor.
- `x ^= y` para atribuição composta de um operador binário XOR. Efetue um XOR entre o valor de y e o valor de x, armazene o resultado em x e retorne o novo valor.
- `x <<= y` para atribuição de deslocamento para a esquerda. Desloque o valor de x para a esquerda em y casas decimais, armazene o resultado em x e retorne o novo valor.
- `x >>= y` para atribuição de deslocamento para a direita. Desloque o valor de x para a direita em y casas decimais, armazene o resultado em x e retorne o novo valor.

Obviamente, a maior parte dos desenvolvedores não

empregará todos esses operadores no dia a dia. Se você utilizar os operadores de atribuição que envolvem as quatro operações básicas (soma, subtração, multiplicação e divisão), já sentirá a diferença, uma vez que estes são os que ocorrerão com maior frequência. Veja a seguir um exemplo simples no qual utilizamos esses quatro operadores de atribuição:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Valores iniciais das variáveis:");
            int x = 10;
            int y = 5;
            Console.WriteLine($"x: {x}   y: {y}");
            Console.WriteLine();

            Console.WriteLine("Empregando operador de atribuição
de soma:");
            x += y;
            Console.WriteLine($"x: {x}   y: {y}");
            Console.WriteLine();

            Console.WriteLine("Empregando operador de atribuição
de subtração:");
            x -= y;
            Console.WriteLine($"x: {x}   y: {y}");
            Console.WriteLine();

            Console.WriteLine("Empregando operador de atribuição
de multiplicação:");
            x *= y;
            Console.WriteLine($"x: {x}   y: {y}");
            Console.WriteLine();

            Console.WriteLine("Empregando operador de atribuição
de divisão:");
            x /= y;
```

```
            Console.WriteLine($"x: {x}   y: {y}");
        }
    }

}
```

Confira a saída produzida por esse exemplo na próxima imagem:

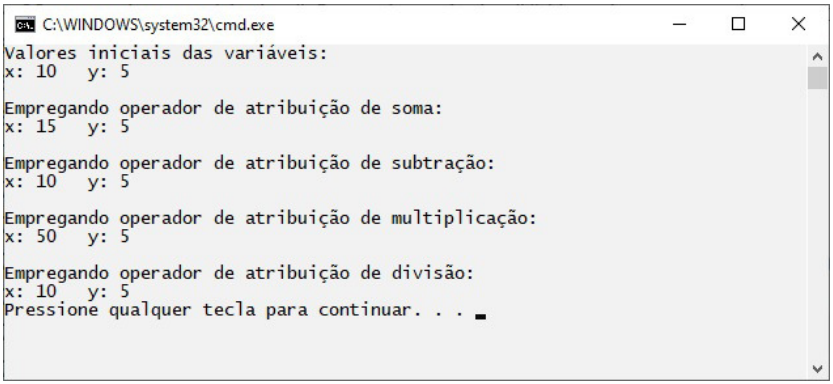


Figura 2.2: Aplicando os operadores de atribuição

No exemplo da próxima listagem, utilizamos o operador de atribuição de soma em um loop `for` :

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i += 2)
            {
                Console.WriteLine(i);
            }
        }
    }
```

```
}
```

2.3 USANDO O OPERADOR CONDICIONAL NULO ?

Uma das coisas mais frustrantes para quem desenvolve, ao utilizar as primeiras versões da linguagem C#, era a necessidade de escrever código para checar explicitamente a condição nula antes de poder usar um objeto ou suas propriedades. Veja a seguir um exemplo:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public Endereco Endereco { get; set; }
    }
    public class Endereco
    {
        public string Logradouro { get; set; }
        public string Bairro { get; set; }
        public string Cidade { get; set; }
        public string Estado { get; set; }
        public string Pais { get; set; }
        public string Cep { get; set; }

    }
    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa = new Pessoa() {
                Nome = "Cláudio Ralha",
                Endereco = new Endereco()
                {
                    Logradouro = "Rua dos Nerds 123",
```

```
                    Bairro = "Alto da Boa Vista",
                    Cidade = "Avaré",
                    Estado = "São Paulo",
                    Pais = "Brasil",
                    Cep = "18702-234"
                }
            };

            if (pessoa != null && pessoa.Endereco != null)
            {
                Console.WriteLine($"{pessoa.Nome}\n{pessoa.Endereco.Logradouro} - {pessoa.Endereco.Bairro}\n{pessoa.Endereco.Cidade} - {pessoa.Endereco.Estado} - {pessoa.Endereco.Pais}\nCep: {pessoa.Endereco.Cep}");
            }
        }
    }
}
```

A parte que nos interessa neste momento é justamente o trecho onde checamos se ambos os objetos não são nulos antes de tentar armazenar as informações neles contidas. Observe:

```
if (pessoa != null && pessoa.Endereco != null)
{
      Console.WriteLine($"{pessoa.Nome}\n{pessoa.Endereco.Logradouro} - {pessoa.Endereco.Bairro}\n{pessoa.Endereco.Cidade} - {pessoa.Endereco.Estado} - {pessoa.Endereco.Pais}\nCep: {pessoa.Endereco.Cep}");
}
```

Confira a saída produzida por este código na próxima imagem:

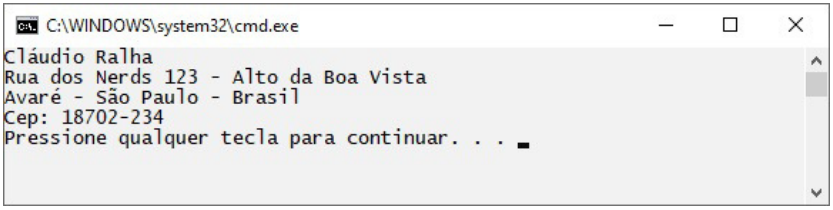


Figura 2.3: Testando se os objetos não são nulos antes de acessar seus membros

Essa necessidade de checar se um objeto continha nulo antes de acessarmos suas propriedades para evitar que uma exceção fosse disparada fazia com que o código de nossas aplicações ganhasse várias estruturas condicionais. Com a introdução do *operador condicional nulo* (*null-conditional operator*), tornou-se possível usar o `?` (ponto de interrogação) da mesma forma como utilizamos nos *tipos anuláveis* (*nullabe types*), ou seja, basta colocar o sinal `?` depois da instância e antes de chamar a propriedade. O trecho anterior poderia ser substituído pelo seguinte sem risco de ocorrer uma exceção:

```
Console.WriteLine($"{pessoa?.Nome}\n{pessoa?.Endereco?.Logradouro
} - {pessoa?.Endereco?.Bairro}\n{pessoa?.Endereco?.Cidade} - {pes
soa?.Endereco?.Estado} - {pessoa?.Endereco?.Pais}\nCep: {pessoa?.
Endereco?.Cep}");
```

Obviamente, nesse caso, não estamos considerando o fato de que ocorreria a impressão apenas dos espaços em branco no caso de a variável `pessoa` conter um valor nulo. Para testar essa condição, comente a declaração da variável `pessoa` atual e acrescente a seguinte linha de código no início do método `main`:

```
Pessoa pessoa = null;
```

Ao executar o projeto novamente, você verá que, mesmo no cenário em que a variável `pessoa` não contém uma instância do objeto `Pessoa`, o código usando operador condicional nulo não disparou uma exceção.

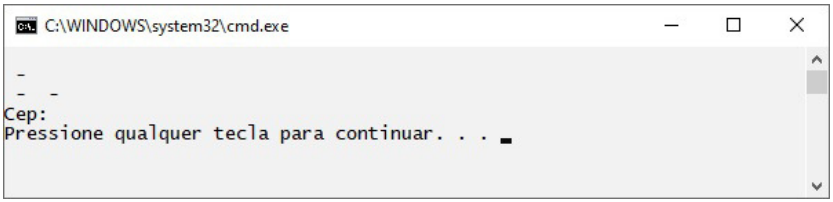


Figura 2.4: Usando o operador condicional nulo ?

> Para saber mais sobre o uso do operador condicional nulo, consulte:
>
> https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/null-conditional-operators

2.4 USANDO O OPERADOR DE COALESCÊNCIA NULA ??

O C# disponibiliza o *operador de coalescência nula* `??` , que nos permite verificar se há um valor nulo em uma variável, campo ou propriedade e atribuir um valor substituto em uma única linha de código. Para lidar com cenários como este, muitos desenvolvedores fariam algo assim:

```
static void Main(string[] args)
{
    Pessoa pessoa = null;
    string nome;
    if(pessoa!=null && pessoa.Nome!=null)
    {
        nome = pessoa.Nome;
    }
    else
```

```
    {
        nome = "Não informado";
    }
    Console.WriteLine(nome);
}
```

Para simplificar esse tipo de código, podemos recorrer ao operador `??` do C#, que é chamado *operador de coalescência nula*. Ele retornará o operando esquerdo se o operando não for nulo; caso contrário, ele retornará o operando direito. Veja como ficaria o exemplo reescrito:

```
static void Main(string[] args)
{
    Pessoa pessoa = null;
    var nome = pessoa?.Nome ?? "Não informado";
    Console.WriteLine(nome);
}
```

> Para obter informações complementares sobre o operador de *coalescência nula* e mais exemplos de aplicação do recurso, acesse a seguinte página da documentação oficial:
>
> https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/null-coalescing-operator

2.5 USANDO O OPERADOR TERNÁRIO ?:

O operador condicional ternário `?:` retorna um de dois valores dependendo do valor de uma expressão booleana, cuja sintaxe é mostrada a seguir:

```
condição ? primeira_expressão : segunda_expressão;
```

A condição deve ser avaliada como `true` ou `false`. Se for verdadeira, `primeira_expressão` será avaliada e se tornará o resultado. Caso a condição seja falsa, `segunda_expressão` será avaliada e se tornará o resultado. Veja a seguir um exemplo:

```
using System;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void Main(string[] args)
        {
            string classificacao;
            Console.Write("Entre a sua idade:");
            int idade = Convert.ToInt32(Console.ReadLine());
            classificacao = (idade > 18) ? "Maior de idade" : "Me
nor de idade";
            Console.WriteLine(classificacao);
        }
    }
}
```

Observe na próxima imagem a saída produzida por esse código:

Figura 2.5: Usando o operador ternário ?:

Como você pode notar, o uso do operador ternário torna o código muito mais conciso. Para obter mais informações sobre o operador ternário, acesse a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/conditional-operator

2.6 USANDO O OPERADOR LÓGICO IS NOT

No C# 9.0 passamos a dispor do operador *is not*, que nos permite tornar mais legíveis testes que antes eram feitos usando o operador `is` . Exemplos:

```
if (!(valor is null))
{
    //Código a executar
}

if (!(valor is string))
{
    //Código a executar
}
```

Com o suporte ao novo operador, agora podemos reescrever esses fragmentos de código conforme mostrado a seguir:

```
if (valor is not null)
{
    //Código a executar
}

if (valor is not string)
{
    //Código a executar
```

```
}
```

Para testar o uso do operador `is not`, execute o seguinte programa:

```
using System;

namespace ProdutividadeEmCSharp9
{
    class Program
    {
        static void Main(string[] args)
        {
            var valores = new object[] { "Produtividade em C#", t
rue, 9.0, 47, null, DateTime.Now, 'R' };

            //Antes do C# 9.0
            foreach (object valor in valores)
            {
                if (!(valor is null))
                    Console.WriteLine($"Tipo: {valor.GetType().Fu
llName} - Valor: {valor}");
                else
                {
                    Console.WriteLine("Valor nulo!");
                }
            }

            //A partir do C# 9.0
            foreach (object valor in valores)
            {
                if (valor is not null)
                    Console.WriteLine($"Tipo: {valor.GetType().Fu
llName} - Valor: {valor}");
                else
                    Console.WriteLine("Valor nulo!");
            }
        }
    }
}
```

2.7 EMPREGANDO A SOBRECARGA DE OPERADORES

Conforme mencionamos previamente, a linguagem C# permite que alguns dos operadores existentes sejam sobrecarregados, alterando seu significado quando aplicados a um tipo definido pelo usuário.

Para ilustrar como efetuar a sobrecarga de operadores, vamos utilizar o exemplo a seguir, no qual definimos o tipo `Distancia`. Observe:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Distancia
    {
        public int Metros { get; set; }

        public Distancia(int metros)
        {
            Metros = metros;
        }

        public static Distancia operator +(Distancia a, Distancia
 b)
        {
            Distancia resultado = new Distancia(0);
            resultado.Metros = a.Metros + b.Metros;
            return resultado;
        }

        public static Distancia operator -(Distancia a, Distancia
 b)
        {
            Distancia resultado = new Distancia(0);
            resultado.Metros = a.Metros - b.Metros;
            return resultado;
        }
```

```
        public static Distancia operator ++(Distancia a)
        {
            a.Metros++;
            return a;
        }

        public static Distancia operator --(Distancia a)
        {
            a.Metros--;
            return a;
        }

        public static bool operator ==(Distancia a, Distancia b)
            => a.Metros == b.Metros;

        public static bool operator !=(Distancia a, Distancia b)
            => a.Metros != b.Metros;

        public static bool operator <(Distancia a, Distancia b)
            => a.Metros < b.Metros;

        public static bool operator >(Distancia a, Distancia b)
            => a.Metros > b.Metros;

        public static bool operator <=(Distancia a, Distancia b)
            => a.Metros <= b.Metros;

        public static bool operator >=(Distancia a, Distancia b)
            => a.Metros >= b.Metros;

}

    class Program
    {
        static void Main(string[] args)
        {
            Distancia distancia1 = new Distancia(10);
            Distancia distancia2 = new Distancia(15);
            distancia1++;
            distancia2--;
            Distancia distancia3 = distancia1 + distancia2;
            Console.WriteLine($"distancia1: {distancia1.Metros}")
;
            Console.WriteLine($"distancia2: {distancia2.Metros}")
```

```
;
            Console.WriteLine($"distancia3: {distancia3.Metros}")
;
            if (distancia1 == distancia2)
                Console.WriteLine("distancia1 é igual a distancia
2!");
            else
                Console.WriteLine("distancia1 é diferente de dist
ancia12!");
            if (distancia1 >= distancia2)
                Console.WriteLine("distancia1 é maior ou igual a
distancia2!");
            else
                Console.WriteLine("distancia1 é menor que a dista
ncia12!");
        }
    }
}
```

Repare, no código da classe `Distancia`, em como é simples de se implementar a sobrecarga de operadores. Basta criar métodos estáticos e preceder o operador que está sendo sobrecarregado pela palavra-chave `operator`. Examine em seguida no método `Main` da classe `Program` como a sobrecarga de operadores facilita a escrita e a leitura do código.

Confira a saída produzida por esse exemplo na próxima imagem:

Figura 2.6: Empregando a sobrecarga de operadores

É importante destacar que nem todos os operadores podem ser sobrecarregados. Para saber quais operadores permitem a sobrecarga e quais não suportam, consulte a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/operator-overloading

Ao atingir este ponto, você já aprendeu os principais segredos relacionados a operadores em C# que costumam estar presentes no código dos desenvolvedores mais experientes. Não se preocupe em decorar todos os nomes exóticos dos operadores que mencionamos ou a sintaxe de uso de cada um deles. Procure apenas memorizar a quais tipos de cenários cada um dos operadores menos populares atendem. A sintaxe pode ser facilmente encontrada mantendo este livro por perto, anotando os exemplos de usos mais úteis em uma folha ou post-it na sua mesa de trabalho ou, em último caso, recorrendo ao Google.

CAPÍTULO 3

ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

A linguagem C# conta com várias estruturas de repetição e estruturas condicionais, em sua maioria derivadas da linguagem C++, que nos permitem ir muito além dos resultados obtidos com as instruções `for` e `if` dos nossos primeiros dias de programação.

Neste capítulo, veremos como e quando empregar laços `do...while`, `for`, `foreach` e `while`, como explorar estruturas condicionais `switch` e como utilizar *pattern matching* (em português, *correspondência de padrão*) para aumentar a legibilidade do código e torná-lo mais conciso. Você perceberá que as melhorias relacionadas à correspondência de padrões, introduzidas no C# da versão 7.0 à 9.0, tornaram a leitura do código tão fácil quanto pseudocódigo e deixaram desenvolvedores com background em outras linguagens como Visual Basic e Delphi em êxtase.

Iniciaremos o nosso estudo com uma revisão dos laços disponíveis em C#, explorando algumas possibilidades

desconhecidas para parte dos leitores e leitoras e, então, passaremos para os recursos introduzidos nas versões mais recentes da linguagem, que tornaram a instrução `switch` incrivelmente flexível e poderosa. Sinta-se à vontade para pular a revisão e avançar para a seção *Utilizando estruturas condicionais switch*, caso você esteja interessado somente nas novidades do C#.

3.1 CRIANDO ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO USANDO CODE SNIPPETS

O editor de código do Visual Studio dispõe de vários atalhos (em inglês, *code snippets*) para a inserção de blocos de código referentes a estruturas condicionais e de repetição, também conhecidas como estruturas de iteração. Confira os atalhos na tabela a seguir:

Atalho	Descrição	Locais válidos para inserir o snippet
`do`	Cria um loop `do while` .	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.
`else`	Cria um bloco `else` .	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.
`for`	Cria um loop `for` .	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.
`foreach`	Cria um loop `foreach` .	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.
`forr`	Cria um loop `for` que decrementa a variável de loop	Dentro de um método, um indexador, um acessador de

	depois de cada iteração.	propriedade ou um acessador de evento.
`if`	Cria um bloco `if`.	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.
`switch`	Cria um bloco `switch`.	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.
`while`	Cria um loop `while`.	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.

3.2 UTILIZANDO LAÇOS FOR

Saber quando utilizar cada uma das estruturas de iteração disponibilizadas é vital quando se deseja escrever um código o mais simples e legível possível. O C# suporta os seguintes laços: `do...while`, `for`, `foreach` e `while`.

Para descobrir qual deles empregar em cada caso, precisamos responder às seguintes questões:

1. Que tipo de conjunto será iterado?
2. O bloco de instruções do laço será executado um número predefinido de vezes ou enquanto uma condição for verdadeira?
3. O bloco de instruções do laço deverá ser executado pelo menos uma vez ou não?

Ao longo desta seção e das posteriores vamos apresentar quando utilizar cada uma das estruturas de repetições mencionadas. Note que, ainda que seja possível reescrever o código

de uma estrutura de iteração usando outra equivalente, sempre existirá a versão que será mais fácil de ler e que envolverá menos código. Por este motivo, reserve um tempo para refatorar partes do seu código que podem ser melhoradas.

Dentre as estruturas de repetição existentes, o *for* é o laço utilizado com maior frequência. Ele é usado quando sabemos de antemão quantas vezes o bloco será executado e é composto por três partes opcionais separadas pelo sinal de ponto e vírgula: *inicialização*, *condições* e *passos*.

```
for (inicialização de variáveis; condições; passos)
{
    //bloco de instruções a serem executadas até que a condição s
eja satisfeita
}
```

O exemplo a seguir ilustra as formas mais comuns de um laço `for` . Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Exemplo de incremento em 1
            for (int i = 1; i <= 5; i++)
            {
                Console.WriteLine(i);
            }

            //Exemplo de decremento em 1
            for (int i = 5; i >= 1; i--)
            {
                Console.WriteLine(i);
            }
```

```
            //Exemplo de incremento em 2
            for (int i = 1; i <= 5; i += 2)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Saiba que, apesar de usarmos variáveis do tipo `int` como *variáveis de controle do loop*, esta variável pode ser de qualquer tipo de dado numérico. Logo, podemos utilizar outros tipos como `double`, `decimal`, `long`, `char` etc. Observe:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Exemplo de incremento em 0.01
            for (double i = 1.01D; i < 1.10; i += 0.01D)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Confira a saída produzida por esse exemplo na imagem a seguir:

Figura 3.1: Utilizando uma variável de controle do loop for do tipo double

Em alguns cenários, precisaremos trabalhar com mais de uma estrutura de repetição ao mesmo tempo, criando fragmentos de código que são conhecidos como *laços aninhados* (em inglês, *nested loops*). É possível combinar diferentes estruturas de repetição em laços aninhados, sendo mais frequente o uso de laços `for` aninhados.

O próximo exemplo demonstra como calcular os números primos entre 2 e 100 usando dois laços `for` aninhados. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int i, j;
            Console.WriteLine("Números primeiros entre 2 e 100:\n
);
            for (i = 2; i <= 100; i++)
            {
                for (j = 2; j <= (i / j); j++)
                {
                    if ((i % j) == 0) break;
                }
                if (j > (i / j)) Console.WriteLine(i);
            }
```

```
        }
    }
}
```

Ao analisar esse código, observe que as variáveis de controle dos laços `i` e `j` foram declaradas antes dos loops para preservar o valor de `j` após o laço `for` interno. Observe também que usamos uma instrução `break` para abandonar o laço interno antes do final. É possível abandonar um loop com `break` e `return` a qualquer momento e avançar para a próxima iteração com `continue` sem executar código desnecessário.

Há naturalmente armadilhas que devem ser evitadas quando se busca produtividade e legibilidade. O próximo exemplo demonstra como trabalhar com duas variáveis de controle de loop em um único `for`:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0, j = 0; i < 5 && j < 10; i++, j++)
            {
                Console.WriteLine($"i: {i}, J: {j}");
            }
        }
    }
}
```

O fato de algo assim ser compilável e executável não significa que deva estar presente em nossos projetos. Para entender o porquê, basta pensar em quanto tempo um desenvolvedor gastaria para descobrir os intervalos de interações que este loop com duas variáveis de controle executará. Veja a seguir um segundo

exemplo, extraído da documentação oficial do C#, que reforça o que devemos evitar:

```
int i;
int j = 10;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++
, j--, Console.WriteLine($"Step: i={i}, j={j}"))
{
    // Body of the loop.
}
```

Exemplos como estes dois últimos que apresentamos só fazem sentido em provas de processos seletivos ou de certificação para avaliar o nível de conhecimento de candidatos em linguagem C#. Fuja da complexidade e mantenha o seu código o mais simples possível. Como já dizia a velha máxima: "O Diabo está nos detalhes"!

3.3 UTILIZANDO LAÇOS WHILE E DO WHILE

O laço `while` é a mais simples das estruturas de repetição. Ele executa um bloco de código desde que a condição dada seja verdadeira. Utilizamos os laços `while` e `do while` quando não sabemos previamente quantas iterações o laço terá.

Veja a seguir um exemplo:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i < 5)
            {
```

```
                Console.WriteLine(i);
                i++;
            }
        }
    }
}
```

Note que, enquanto a condição fornecida for verdadeira, o bloco de código do `while` será executado. Execute o programa e observe a saída gerada:

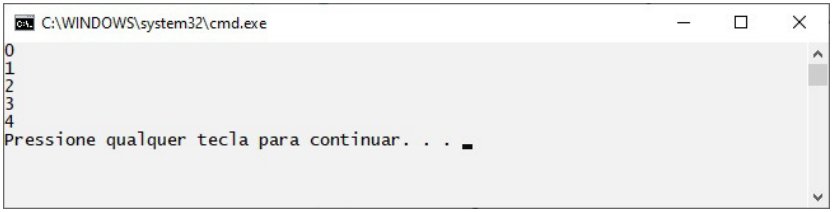


Figura 3.2: Utilizando um laço while

Agora troque o valor de inicialização de `i` na linha a seguir para `5`:

```
int i = 0;
```

e execute novamente o exemplo. Veja que agora o bloco de instruções do laço `while` não foi executado nenhuma vez. Isso ocorre porque a condição é avaliada antes do início do bloco de instruções.

Quando necessitamos que o bloco seja executado pelo menos uma vez, devemos usar a estrutura de repetição `do while`. O próximo exemplo ilustra o seu uso:

```
using System;

namespace ProdutividadeEmCSharp
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            do
            {
                Console.WriteLine(i);
                i++;
            }
            while (i < 5);
        }
    }
}
```

Neste caso, mesmo que troquemos o valor de inicialização de `i` para `5`, o bloco de instruções ainda será executado uma única vez.

Como os laços `while` e `do while` são muito mais simples que os laços `for`, não nos alongaremos fornecendo explicações mais detalhadas sobre o seu uso.

3.4 UTILIZANDO LAÇOS FOREACH

A linguagem C# suporta a estrutura de laço `foreach`, que nos permite percorrer todos os itens de uma coleção com o mínimo de esforço. Em um loop `foreach`, nós avaliamos cada elemento individualmente e deste modo um índice não é necessário. Sem índices, os loops são mais fáceis de escrever e os programas são mais simples.

O loop `foreach` retornará cada elemento de uma coleção na ordem em que foi definido. Isso é conhecido como enumeração e tem como benefício eliminar erros causados pelo tratamento incorreto de índices.

Observe no exemplo a seguir onde utilizamos `for` e `foreach` para exibir os itens do array `cores`:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] cores = { "azul", "vermelho", "amarelo", "ve
rde", "branco", "preto" };

            Console.WriteLine("Imprimindo as cores usando um loop
 For:");
            for (int i = 0; i < cores.Length; i++)
            {
                string valor = cores[i];
                Console.WriteLine(valor);
            }
            Console.WriteLine();

            Console.WriteLine("Imprimindo as cores usando um loop
 Foreach:");
            foreach (var valor in cores)
            {
                Console.WriteLine(valor);
            }
        }
    }

}
```

Confira na imagem a seguir a saída produzida por esse exemplo:

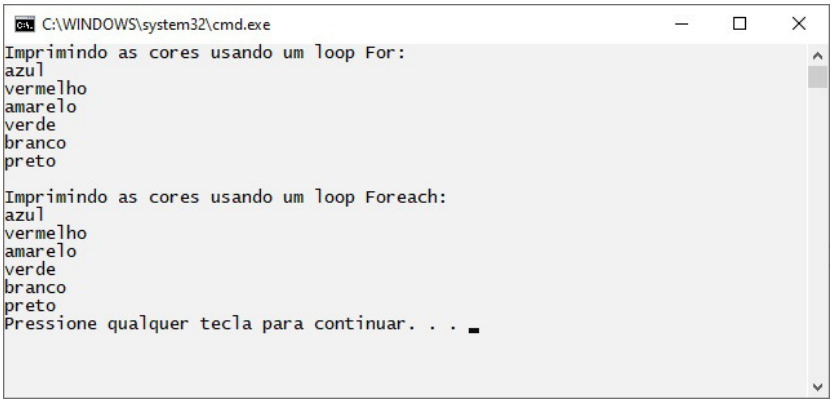


Figura 3.3: Empregando loops foreach

Ao examinar o código, repare que utilizamos a palavra reservada `var` na declaração da variável de controle do loop `foreach` `valor` para inferir o seu tipo e simplificar a sintaxe do loop. Por este motivo, o *code snippet* que é inserido pelo Visual Studio quando digitamos `foreach` seguido de *tab* é montado usando a palavra reservada `var` . Veja:

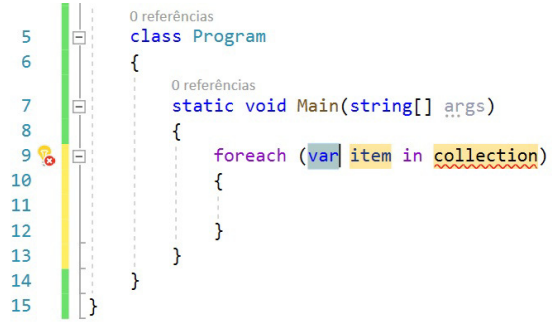


Figura 3.4: Comparando loops for com foreach

Apesar de `foreach` ser uma forma elegante e concisa de se construir loops, nem sempre será a solução para os nossos

problemas. Isso porque o uso de `foreach` impede que ocorra qualquer modificação na coleção durante o loop. Em outras palavras, não é possível remover um item da coleção, por exemplo.

3.5 UTILIZANDO ESTRUTURAS SWITCH

Até a versão 6 da linguagem C#, os desenvolvedores reclamavam de algumas limitações na estrutura condicional `switch`, que não era tão flexível como as estruturas equivalentes em linguagens como Visual Basic e Delphi. Com a chegada da versão 7, que passou a suportar *pattern matching* (abordado em detalhes a partir da próxima seção), estas limitações deixaram de existir e o código gerado com a estrutura condicional `switch` se tornou extremamente conciso e legível.

Ao longo desta seção, veremos usos do `switch` que são menos triviais e que não envolvem patterns e continuaremos com o tour pela seção seguinte. O primeiro exemplo utiliza um tipo enumerado retornado por `DayOfWeek`. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            switch (DateTime.Now.DayOfWeek)
            {
                case DayOfWeek.Saturday:
                case DayOfWeek.Sunday:
                    Console.WriteLine("Hoje é final de semana!");
                    break;
                default:
                    Console.WriteLine("Hoje é dia de trabalho!");
```

```
                    break;
                }
            }
        }
```

Observe que, nesse exemplo, temos o mesmo bloco de código sendo executado para os dias Sábado (em inglês, *Saturday*) e Domingo (*Sunday*). Para os dias de trabalho é executado o bloco `default`.

A saída gerada por este código obviamente vai depender do dia da semana em que ele for executado. Confira na imagem a seguir:

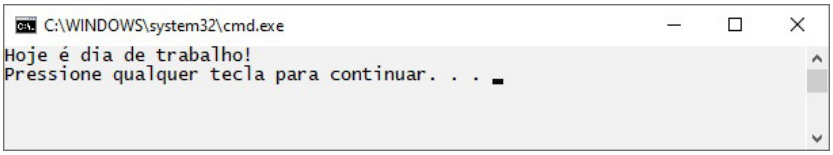


Figura 3.5: Empregando switch com uma variável de um tipo enumerado

Este segundo exemplo ilustra que é possível utilizar `switch` para avaliar uma variável do tipo `char` e usar `return` de dentro de um bloco do `switch`, cenário em que omitimos a instrução `break`. Além disso, ele demonstra que em C# a cláusula `default` do `switch` não é obrigatória. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        private static string ObterFruta()
        {
            Random rnd = new Random();
            int num = rnd.Next(0, 26); //0 a 25
            char letra = (char)('a' + num);
```

```
            switch (letra)
            {
                case 'a': return "abacate";
                case 'b': return "banana";
                case 'c': return "cereja";
                case 'd': return "damasco";
                case 'e': return "embaúba";
                case 'f': return "figo";
                case 'g': return "goiaba";
                case 'h': return "heisteria";
                case 'i': return "ingá";
                case 'j': return "jamelão";
                case 'k': return "kiwi";
                case 'l': return "laranja";
                case 'm': return "mamão";
                case 'n': return "nespera";
                case 'o': return "olho-de-boi";
                case 'p': return "pera";
                case 'q': return "quina";
                case 'r': return "romã";
                case 's': return "sapucaia";
                case 't': return "tamarindo";
                case 'u': return "uva";
                case 'v': return "veludo";
                case 'w': return "wampi";
                case 'x': return "xixá";
                case 'y': return "yamamomo";
                case 'z': return "zimbro";
            }
            return "Esta opção não é válida";
        }

        static void Main(string[] args)
        {
            for (int i = 1; i <= 10; i++)
            {
                Console.WriteLine($"{i}: {ObterFruta()}");
            }
        }
    }
}
```

A saída produzida por este código é randômica. Confira na próxima imagem um exemplo:

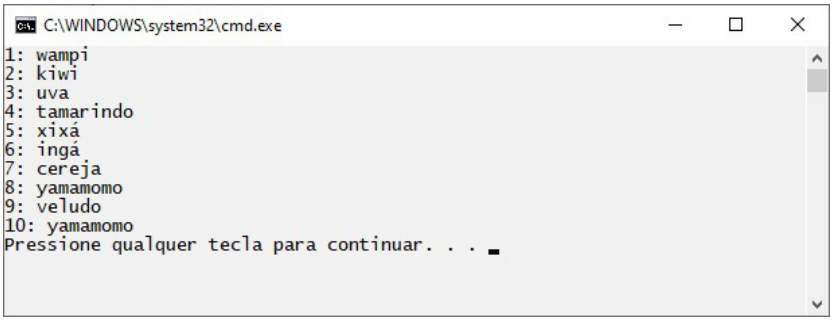


Figura 3.6: Empregando switch com uma variável do tipo char

Leitores que se divertiram na infância/adolescência com jogos como *Adedanha* e *Flor-Fruta* (também conhecido como *Stop*) olharão para esse código com saudosismo e adivinharão que o autor não conhecia todas essas frutas e teve que recorrer ao Google para achar algumas.

No próximo exemplo, demonstramos como é possível usar o `switch` com tipos anuláveis testando o caso nulo da variável. Neste caso, vamos tratá-lo no bloco `default`. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {

        static void Main(string[] args)
        {
            bool? passou = null;
            switch (passou)
            {
                case true:
                    Console.WriteLine("Parabéns!");
                    break;
                case false:
                    Console.WriteLine("Que pena!");
```

```
                    break;
                default:
                    Console.WriteLine("Precisamos saber o resulta
do!");
                    break;
            }
        }
    }
}
```

Obviamente, este código reescrito em C# 7.x ficaria mais legível:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {

        static void Main(string[] args)
        {
            bool? passou = null;
            switch (passou)
            {
                case true:
                    Console.WriteLine("Parabéns!");
                    break;
                case false:
                    Console.WriteLine("Que pena!");
                    break;
                case null:
                    Console.WriteLine("Precisamos saber o resulta
do!");
                    break;
            }
        }
    }
}
```

No final da próxima seção, focada em correspondência de padrões, apresentaremos mais exemplos de como ganhar produtividade com a instrução `switch` envolvendo as melhorias

introduzidas nas versões mais recentes do C#.

3.6 EMPREGANDO CORRESPONDÊNCIA DE PADRÕES

O conceito de *pattern matching* (em português, *correspondência de padrão*) foi introduzido no C# 7.0 com o objetivo de checar se um objeto reflete um *shape* especificado.

A correspondência de padrões é suportada por meio das expressões `is` e `switch`. Elas permitem inspecionar um objeto e suas propriedades para determinar se ele satisfaz o padrão procurado. Através da palavra-chave `when`, é possível especificar regras adicionais para o padrão.

Antes do C# 7.0, o operador `is` era usado para verificar o tipo de uma variável e, com base no tipo, retornava `true` ou `false`. A partir do C# 7.0, `is` fornece três tipos de correspondência de padrões:

- Padrão de constante
- Padrão de tipo
- Padrão var

Vamos ilustrar cada um deles através de exemplos. O *padrão de constante* permite verificar o objeto com qualquer valor. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
```

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

        static void Main(string[] args)
        {
            Pessoa pessoa = null;

            //checando valor null
            if (pessoa is null)
            {
                pessoa = new Pessoa
                {
                    Nome = "Cláudio Ralha",
                    Idade = 46
                };
            }

            //checando um valor string
            if (pessoa.Nome is "Cláudio Ralha")
                Console.WriteLine($"O nome é {pessoa.Nome}");

            //checando um valor Constante
            if (pessoa.Idade is 46)
                Console.WriteLine("A idade é 46 anos (mas não esp
alha, ok?).");
        }
    }
}
```

Ao ser executado, esse código produzirá a seguinte saída:

Figura 3.7: Verificando um objeto usando o padrão de constante

O padrão de tipo permite confirmar o tipo do objeto e atribuir

opcionalmente o valor a uma nova variável do tipo dado. O próximo exemplo demonstra como utilizá-lo:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
    }
    class Program
    {

        static void Main(string[] args)
        {
            object obj = 22;
            var pessoa = new Pessoa
            {
                Nome = "Cláudio Ralha",
                Idade = 45
            };

            if (obj is int)
                Console.WriteLine($"Variável obj possui um valor
do tipo inteiro.");

            if (obj is int i)
                Console.WriteLine($"Variável obj possui o seguint
e valor inteiro: {i}");

            if (pessoa is Pessoa)
            {
                Console.WriteLine("Variável pessoa é do tipo Pess
oa.");
            }

            if (pessoa is Pessoa p)
            {
                Console.WriteLine($"Variável pessoa é do tipo Pes
soa. Nome: {p.Nome}");
            }
        }
```

```
    }
}
```

Confira a saída produzida por esse exemplo na imagem a seguir:

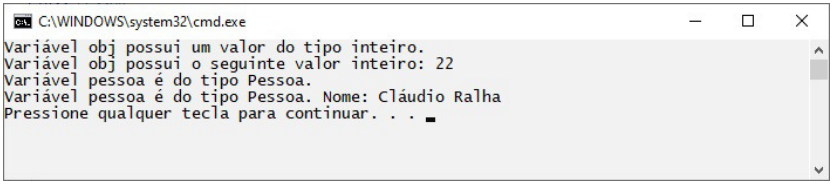


Figura 3.8: Verificando objetos usando o padrão de tipo

O padrão `var` é um caso especial do padrão de tipo, com a diferença de que o padrão corresponderá a qualquer valor, mesmo que seja nulo. Veja a seguir um exemplo:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Livro
    {
        public string Titulo { get; set; }
        public string Autor { get; set; }
    }


    class Program
    {

        static void Main(string[] args)
        {
            Livro livro = new Livro();
            if (livro is Livro l1)
            {
                Console.WriteLine($"Tipo da variável l1: {l1?.Get
Type()?.Name}");
            }
            livro = null;
```

```
            if (livro is Livro l2)
            {
                Console.WriteLine($"Tipo da variável l2: {l2?.Get
Type()?.Name}");
            }
            if (livro is var l3)
            {
                Console.WriteLine($"Tipo da variável l3: {l3?.Get
Type()?.Name}");
            }
        }
    }
}
```

Ao observar a saída gerada por esse código na imagem a seguir, note que quando testamos usando `is Pessoa l1` o valor será `true` apenas se a variável `l1` não contiver `null`. Por outro lado, quando usarmos `is var l3` o resultado será sempre `true`.

Figura 3.9: Verificando objetos usando o padrão var

Note também que foi necessário utilizar o *operador condicional null* junto do método `GetType` para evitar que fosse lançada uma *exception* do tipo `NullReferenceException` quando a variável é nula (em nosso exemplo, é o caso de `l3`).

A vantagem que temos ao usar este padrão é que a variável declarada com `var` é do tipo real do objeto.

Vale ressaltar que, apesar de a expressão padrão `is` funcionar muito bem para validação, haverá casos em que teremos muitas

instruções `if` e `else if` e poderá ser vantajoso migrar para a estrutura condicional `switch`, que finalmente atingiu o patamar tão sonhado por quem gosta de escrever código conciso e com boa legibilidade. Ela agora suporta os mesmos três padrões discutidos anteriormente.

O exemplo a seguir demonstra como fazer correspondência de padrão de tipo com a instrução `switch`:

```
using System;

namespace ProdutividadeEmCSharp
{

    class Animal
    {
        public string Nome { get; set; }
    }

    class Passaro : Animal
    {
        public string Alimentacao { get; set; }
    }

    class Cachorro: Animal
    {
        public int Idade { get; set; }
    }

    class Gato : Animal
    {
        public string Cor { get; set; }
    }

    class Program
    {
        private static void ReconhecerAnimal(Animal animal)
        {
            switch (animal)
            {
                case Passaro p:
                    Console.WriteLine($"{p.Nome} é um pássaro que
```

```
 adora comer {p.Alimentacao}.");
                    break;
                case Cachorro c:
                    Console.WriteLine($"{c.Nome} é um cachorro mu
ito dócil e brincalhão. Ele tem {c.Idade} anos.");
                    break;
                case Gato g:
                    Console.WriteLine($"{g.Nome} é um gato {g.Cor
} que adora miar no telhado dos vizinhos.");
                    break;
                default:
                    break;
            }
        }

        static void Main(string[] args)
        {
            var passaro = new Passaro() { Nome = "Namopelão", Ali
mentacao = "sementes de girassol"};
            var cachorro = new Cachorro() { Nome = "Neném", Idade
 = 3};
            var gato = new Gato() { Nome = "Sirius", Cor = "preto
 };
            ReconhecerAnimal(passaro);
            ReconhecerAnimal(cachorro);
            ReconhecerAnimal(gato);
        }
    }
}
```

Note que, ao definirmos variáveis em cada `case`, temos acesso às propriedades dos objetos. Na imagem a seguir, você pode observar a saída produzida pelo exemplo anterior:

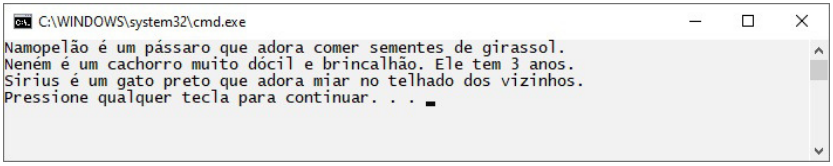


Figura 3.10: Empregando a correspondência de padrão de tipo usando switch

Um detalhe importante a ser notado é que não é possível especificar os blocos `case` do `switch` partindo de um tipo pai em direção ao tipo filho. O Visual Studio vai avisá-lo em tempo de codificação. Veja:

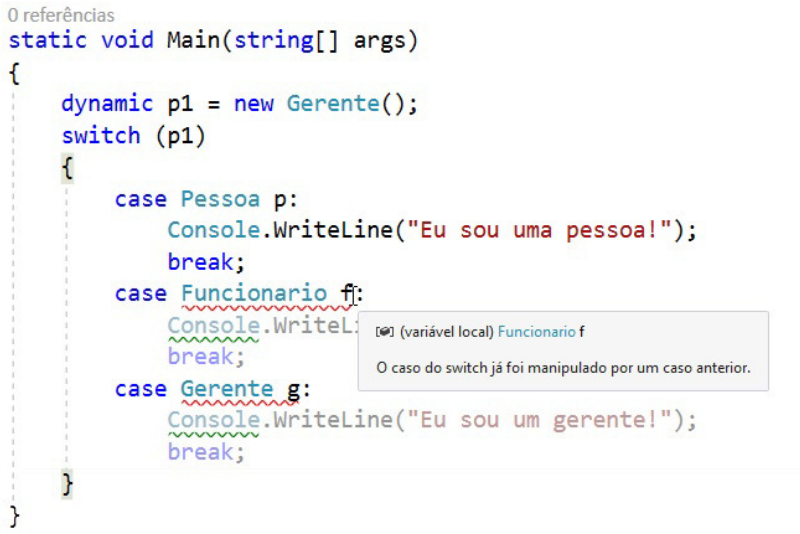


Figura 3.11: Sinalização de erro na ordem dos blocos case

O programa a seguir só será compilável se o *pattern matching* for empregado especificando primeiro os tipos derivados e se a variável `p1` for declarada como `dynamic` :

```
using System;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
    }
```

```
    class Funcionario : Pessoa
    {
        public string Cargo { get; set; }
    }

    class Gerente : Funcionario
    {
        public string Filial { get; set; }
    }

    class Program
    {

        static void Main(string[] args)
        {
            dynamic p1 = new Gerente();
            switch (p1)
            {
                case Gerente g:
                    Console.WriteLine("Eu sou um gerente!");
                    break;
                case Funcionario f:
                    Console.WriteLine("Eu sou um funcionário!");
                    break;
                case Pessoa p:
                    Console.WriteLine("Eu sou uma pessoa!");
                    break;
            }
        }
    }
}
```

Ao ser executado, ele produzirá a seguinte saída:

Figura 3.12: Empregando a correspondência de padrão com tipos derivados de um tipo pai

O próximo exemplo faz uso da cláusula `when` para especificar

filtros adicionais e mostra como a estrutura condicional se tornou poderosa:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        private static void ReconhecerCaractere(char c)
        {
            switch (c)
            {
                case char l when (c >= 'a' && c <= 'z') || (c >=
'A' && c <= 'Z'):
                    Console.WriteLine($"{c} é um caractere alfabé
tico.");
                    break;
                case char n when c >= '0' && c <= '9':
                    Console.WriteLine($"{c} é um caractere numéri
co.");
                    break;
                default:
                    break;
            }
        }

        static void Main(string[] args)
        {
            ReconhecerCaractere('a');
            ReconhecerCaractere('Z');
            ReconhecerCaractere('9');
        }
    }
}
```

Confira a saída produzida por este exemplo na imagem a seguir:

Figura 3.13: Usando a cláusula when para especificar filtros adicionais

Perceba que agora podemos resolver de forma elegante o que antes era feito através de múltiplas instruções `if`. Veja a seguir outro exemplo que faz uso da cláusula `when` em cada `case` do `switch`:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        private static void ReconhecerFaseDaVida(int idade)
        {
            string fase = String.Empty;
            switch (idade)
            {
                case int i when i <= 11:
                    fase = "Infância";
                    break;
                case int i when i >= 12 && i<=20:
                    fase = "Adolescência";
                    break;
                case int i when i >= 21 && i <= 74:
                    fase = "Idade Adulta";
                    break;
                case int i when i >= 75:
                    fase = "Velhice";
                    break;
            }
            Console.WriteLine($"{idade} anos = {fase}");
        }

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

```
        {
            ReconhecerFaseDaVida(1);
            ReconhecerFaseDaVida(18);
            ReconhecerFaseDaVida(46);
            ReconhecerFaseDaVida(84);
        }
    }
}
```

Ao examinar este código, alguns leitores vão estranhar a velhice começando em 75 anos. Esta mudança foi efetuada pela Organização Mundial de Saúde, uma vez que muitas pessoas com até 65 anos de idade continuam ativas no mercado de trabalho e apresentam expectativa de vida maior que anos atrás.

Confira na próxima imagem a saída produzida por esse código:

Figura 3.14: Usando novamente a cláusula when para especificar filtros adicionais

No exemplo da próxima listagem demonstramos como testar no `case` se a variável contém `null`. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Livro
    {
        public string Titulo { get; set; }
        public string Autor { get; set; }
    }
```

```
    class Program
    {

        static void Main(string[] args)
        {
            dynamic livro = new Livro();
            //livro = null;
            //livro = "Produtividade em C#";
            switch (livro)
            {
                case null:
                    Console.WriteLine("Variável livro contém null
!");
                    break;
                case Livro l:
                    Console.WriteLine("Variável livro contém obje
to do tipo Livro!");
                    break;
                default:
                    Console.WriteLine($"Variável livro contém obj
eto do tipo {livro.GetType().Name}!");
                    break;
            }
        }
    }
}
```

Note que há duas linhas comentadas neste código. Ao executar a primeira vez o exemplo com as linhas comentadas, você obterá a saída mostrada na próxima imagem:

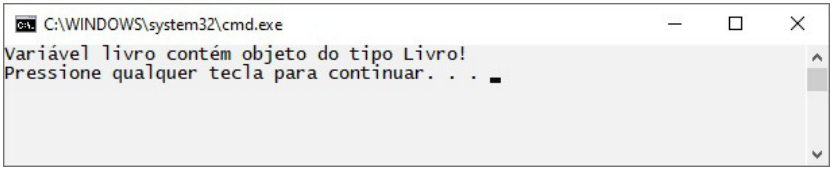


Figura 3.15: Testando o valor null em um bloco case de uma instrução switch

Descomente a primeira linha de comentário, execute novamente o código e observe o resultado. Em seguida,

descomente também a segunda linha e rode novamente o programa para testar os três cenários tratados no `switch`.

> Aviso de spoiler: para os leitores e leitoras que não ficaram contentes com a introdução da clausula `when` no C# 8.0 por considerarem que existem formas mais inteligentes de simplificar as estruturas `switch`, a boa notícia é que no C# 9.0 o `switch` se tornou ainda mais poderoso e simples de codificar com as melhorias nas correspondências de padrões, sobre as quais falaremos a seguir.

Ao começar a desenvolver em C# 9.0, você descobrirá que ao invés de escrever:

```
case int i when i >= 12 && i<=20:
    fase = "Adolescência";
    break;
```

Bastará escrever:

```
case i >= 12 and i<=20:
    fase = "Adolescência";
    break;
```

3.7 MELHORIAS NA CORRESPONDÊNCIA DE PADRÕES DO C# 9.0

A linguagem C# segue evoluindo e, enquanto você está lendo esta obra, o time de desenvolvimento está trabalhando nas funcionalidades a serem introduzidas nas próximas versões do compilador. O C# 9.0 nos trouxe várias melhorias nas

correspondências de padrões, conforme você pode observar na lista a seguir extraída da documentação oficial de lançamento da linguagem:

- *Padrões de tipo* testam se uma variável é de um tipo específico.
- *Padrões entre parênteses* impõem ou enfatizam a precedência de combinações de padrões.
- *Padrões de conjuntiva and* exigem os dois padrões para corresponder.
- *Padrões de disjuntiva or* exigem qualquer padrão para corresponder.
- *Padrões de negação not* exigem que um padrão não corresponda.
- Os padrões relacionais exigem que a entrada seja menor que, maior que, menor ou igual ou maior ou igual a uma determinada constante.

Foge ao escopo deste livro apresentar todas as possibilidades envolvendo cada uma dessas melhorias, mas saiba que esses padrões podem ser usados em qualquer contexto em que os padrões são permitidos: `is` expressões `switch` de padrão , expressões, padrões aninhados e o padrão do rótulo de uma instrução `case` de um `switch` .

> Nota: no capítulo 2, sobre Operadores, temos um exemplo de uso de *padrões de negação not* (consulte a seção *Usando o operador is not para tornar o código mais legível*).

Observe a seguir um exemplo onde aplicamos as melhorias na correspondência de padrões que só será compilável usando o C# 9.0 ou superior:

```
using System;

namespace ProdutividadeEmCSharp9
{

    static class Caractere
    {
        public static bool ELetra(this char c) =>
            c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';

        public static bool ELetraMinuscula(this char c) =>
            c is >= 'a' and <= 'z';

        public static bool ELetraMaiuscula(this char c) =>
            c is  >= 'A' and <= 'Z';

        public static bool ELetraOuNumero(this char c) =>
            c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or (>
= '0' and <= '9');

        public static bool ESinalDePontuacao(this char c) =>
            c is '!' or '?' or '.' or ',' or ':' or ';' or '-';
    }


    class Program
    {

        static void Main(string[] args)
        {
            var frase = "Bem-vindo ao C# 9.0!";
            if (frase is not null)
            {
                foreach (char c in frase)
                {
                    Console.WriteLine($"Caractere: {c} Letra ou N
úmero: {c.ELetraOuNumero()}");
                }
            }
```

```
        }
    }
}
```

Percebeu como o código agora está fácil de ler? Vamos nos concentrar no método `ELetra` a seguir:

```
public static bool ELetra(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

O `and` e o `or` finalmente chegaram ao C#! Este detalhe, junto da presença do `is`, tornou a leitura das condições muito mais simples. E ainda temos a possibilidade de usar parênteses opcionais para enfatizar a precedência, conforme ilustrado no método:

```
public static bool ELetraOuNumero(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or (>= '0' an
d <= '9');
```

O próximo exemplo mostra as melhorias na correspondência de padrões aplicadas a instruções `switch` e também só é compilável usando o C# 9.0 ou superior:

```
using System;

namespace ProdutividadeEmCSharp9
{
    class Program
    {
        static void Main(string[] args)
        {
            var valor = 3;
            switch (valor)
            {
                case < 0:
                    Console.WriteLine($"Valor {valor} é menor que
 0!");
                    break;
                case 0:
```

```
                    Console.WriteLine($"Valor {valor} é igual a 0
!");
                    break;
                case > 0 and <= 10:
                    Console.WriteLine($"Valor {valor} é maior que
 0 e menor ou igual a 10!");
                    break;
                default:
                    Console.WriteLine($"Valor {valor} é maior que
 10!");
                    break;
            }
        }
    }
}
```

Leitores e leitoras com background em linguagens como Visual Basic, Delphi, Python, Ruby e Lua devem estar se perguntando: "Mas afinal, por que não fizeram assim desde a versão 1.0?". Concordo, mas lembre-se de que às vezes ficamos presos tempo demais às nossas origens! No caso do C#, ao C++ e ao Java.

3.8 COMPACTANDO INSTRUÇÕES SWITCH USANDO EXPRESSÕES SWITCH

Em muitos casos, utilizamos uma instrução `switch` para retornar um valor em cada um de seus blocos `case`. Para lidar com estes cenários, o C# 8.0 introduz *expressões switch* que nos permitem aplicar uma sintaxe de expressão mais concisa, uma vez que elas descartam o uso de palavras-chave `case` e `break`.

Vejamos a seguir um exemplo de uso das *expressões switch*:

```
using System;

namespace ProdutividadeEmCSharp
{
    public enum Estacao
```

```
    {
        Primavera,
        Verao,
        Outono,
        Inverno
    }


    class Program
    {
        public static string Pedir(Estacao estacao) =>
            estacao switch
            {
                Estacao.Primavera => "Traz uma flor para mim!",
                Estacao.Verao => "Traz um sorvete para mim!",
                Estacao.Outono => "Traz um vinho para mim!",
                Estacao.Inverno => "Traz uma barra de chocolate p
ara mim!",
                _ => throw new ArgumentException(message: "Valor
do enum inválido!", paramName: nameof(estacao))
            };

    static void Main(string[] args)
        {
            Console.WriteLine(Pedir(Estacao.Primavera));
            Console.WriteLine(Pedir(Estacao.Verao));
            Console.WriteLine(Pedir(Estacao.Outono));
            Console.WriteLine(Pedir(Estacao.Inverno));
        }
    }
}
```

Perceba que há várias melhorias na sintaxe desta construção:

- O posicionamento da variável antes da palavra-chave `switch` ajuda a distinguir visualmente a expressão switch da instrução `switch` .
- Os corpos são *expressões*, não *instruções*.
- Os elementos `case` e `:` são substituídos por `=>` (mais conciso e intuitivo).
- O caso `default` é substituído por um descarte `_` .

Na próxima imagem podemos observar a saída gerada pelo exemplo anterior:

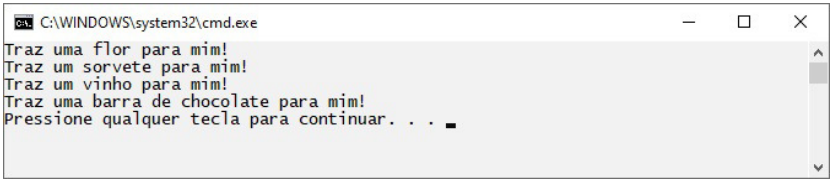


Figura 3.16: Empregando uma expressão switch

Fantástico, não é mesmo? Pois saiba que com as melhorias nas correspondências de padrão trazidas pelo C# 9.0 e o .NET 5.0 ainda podemos ir além. A partir desta versão do compilador podemos reescrever o último exemplo da seção anterior usando expressões `switch` como a mostrada a seguir:

```
using System;

namespace ProdutividadeEmCSharp9
{
    class Program
    {
        static void Main(string[] args)
        {
            var valor = 3;
            var mensagem = valor switch {
                < 0 => $"Valor {valor} é menor que 0!",
                0 => $"Valor {valor} é igual a 0!",
                > 0 and <= 10 => $"Valor {valor} é maior que 0 e
menor ou igual a 10!",
                _ => $"Valor {valor} é maior que 10!"
            };
            Console.WriteLine(mensagem);
        }
    }
}
```

Voilá! Simples, fácil e elegante como sempre sonhamos!

Para saber mais sobre *expressões switch*, consulte a seguinte seção da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-8#switch-expressions

Para saber mais sobre *correspondência de padrões* e obter outros exemplos, consulte:

https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-7#pattern-matching

https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-9#pattern-matching-enhancements

Conforme você deve ter percebido ao analisar os exemplos apresentados neste capítulo, quanto mais você conhecer os recursos fornecidos pela linguagem, mais será capaz de produzir códigos elegantes com menos esforço.

Lembre-se de que o C# não para de evoluir e de nos brindar com recursos que reduzem o trabalho braçal. Procure descobrir as novidades incluídas em cada versão do compilador liberada pela Microsoft e exemplos práticos de empregabilidade de cada nova funcionalidade. Aos poucos, você encontrará verdadeiras joias escondidas na linguagem e no Visual Studio e se tornará um profissional cada dia mais completo.

Capítulo 4

TIPOS E MEMBROS

Em linguagens orientadas a objetos, como C#, todo o nosso código está contido em tipos que são compostos por classes e estruturas. Os tipos possuem membros públicos e privados que representam seus dados e comportamentos, construtores e finalizadores.

Ao longo deste capítulo, veremos como tirar proveito de várias simplificações e melhorias incluídas na linguagem C# para torná-la mais poderosa e flexível. Nas próximas seções, você aprenderá a:

- empregar *tipos anuláveis* usando sintaxe simplificada;
- usar *literais binárias* e *separadores de dígitos*;
- criar múltiplos *construtores de instância* para uma classe;
- declarar uma *propriedade autoimplementada* como somente leitura ou somente escrita;
- atribuir um valor inicial a uma *propriedade autoimplementada* na sua declaração;
- simplificar o código de *propriedades calculadas* com *expressões lambda*;
- utilizar *propriedades* somente de inicialização em classes, structs e registros;
- iterar sobre um *tipo enumerado*;
- utilizar *inicializadores de objetos* e *inicializadores de*

coleções;

- utilizar *parâmetros opcionais* e *parâmetros nomeados*;
- utilizar *métodos de extensão* para estender uma classe;
- usar `dynamic` para retornar objetos diferentes em tempo de execução;
- utilizar `yield` em vez de criar *coleções* temporárias;
- utilizar `using` para declarar objetos descartáveis;
- utilizar *métodos de interface* padrão para fornecer implementação para membros de uma interface;
- utilizar *new expressions* para inicializar objetos;
- utilizar *registros* para criar tipos de referência imutáveis.

Conforme você pode observar, há muitas funcionalidades avançadas suportadas pela linguagem C# que acabam ficando de fora do código gerado pela maioria dos desenvolvedores. Conhecer e empregar estes recursos vai ajudar você a criar sistemas mais legíveis e robustos com menos código e a completar as suas tarefas em menos tempo.

4.1 CRIANDO TIPOS E MEMBROS USANDO CODE SNIPPETS

O editor de código do Visual Studio dispõe de vários atalhos (em inglês, *code snippets*) para a inserção de tipos e membros. Confira os atalhos na tabela a seguir:

Atalho	Descrição	Locais válidos para inserir o snippet
`class`	Cria uma declaração de classe.	Dentro de um namespace (incluindo o namespace global), uma classe ou um struct.

`ctor`	Cria um construtor para a classe que o contém.	Dentro de uma classe.
`enum`	Cria uma declaração enum.	Dentro de um namespace (incluindo o namespace global), de uma classe ou de um struct.
`exception`	Cria uma declaração para uma classe que deriva de uma exceção (Exception, por padrão).	Dentro de um namespace (incluindo o namespace global), de uma classe ou de um struct.
`indexer`	Cria uma declaração do indexador.	Dentro de uma classe ou de um struct.
`interface`	Cria uma declaração interface.	Dentro de um namespace (incluindo o namespace global), de uma classe ou de um struct.
`iterator`	Cria um iterador.	Dentro de uma classe ou de um struct.
`iterindex`	Cria um par de iterador e indexador "nomeado" usando uma classe aninhada.	Dentro de uma classe ou de um struct.
`namespace`	Cria uma declaração de namespace.	Dentro de um namespace (incluindo o namespace global).
`prop`	Cria uma declaração de propriedade autoimplementada.	Dentro de uma classe ou de um struct.
`propfull`	Cria uma declaração de propriedade com os acessadores `get` e `set`.	Dentro de uma classe ou de um struct.
`propg`	Cria uma propriedade implementada automaticamente do tipo somente leitura, com um acessador `set` particular.	Dentro de uma classe ou de um struct.
`sim`	Cria uma declaração de método principal `static int`.	Dentro de uma classe ou de um struct.
`struct`	Cria uma declaração estrutura.	Dentro de um namespace (incluindo o namespace global), uma classe ou um struct.

`svm`	Cria uma declaração de método principal `static void` .	Dentro de uma classe ou de um struct.

4.2 UTILIZANDO A SINTAXE SIMPLIFICADA EM TIPOS ANULÁVEIS

Os *tipos anuláveis* (em inglês, *nullable types*) foram introduzidos no C# 2.0 para permitir o suporte a valores nulos em tipos de valor. São instâncias da estrutura `System.Nullable<T>` , particularmente úteis quando estamos lidando com dados mantidos em banco de dados e precisamos ler valores numéricos e booleanos que podem estar ou não presentes nos registros. A declaração formal de um tipo anulável pode ser vista a seguir:

```
Nullable<T> variável = null;
```

Apesar de útil, recomendamos para fins de produtividade que seja usada a forma curta alternativa suportada pela linguagem, mais legível e simples:

```
T? variável = null;
```

O exemplo a seguir ilustra como declarar um tipo anulável inteiro usando a forma curta e como testar se ele contém valor ou não:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {

        static void Main(string[] args)
        {
            int? classificacao = null;
```

```
            int posicao = 0;
            //classificacao = 2;
            if (classificacao.HasValue) {
                posicao = classificação.Value;
            }
            Console.WriteLine($"Classificação: {classificacao}");
            Console.WriteLine($"Posição: {posicao}");
        }
    }
}
```

Quando a variável `classificacao` contém um valor diferente de `null`, o seu valor é convertido em inteiro e atribuído à variável `posicao`. Para verificar se a variável contém valor, é testada a propriedade `HasValue` e em seguida é atribuído o valor da propriedade `Value` à variável `posicao`.

Confira a seguir a saída gerada por esse exemplo:

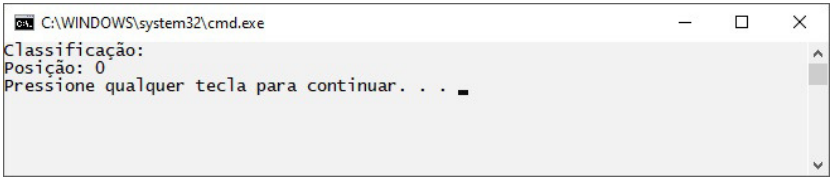


Figura 4.1: Testando se um tipo anulável contém valor

Após descomentar a linha comentada e repetir o teste, veremos a saída a seguir:

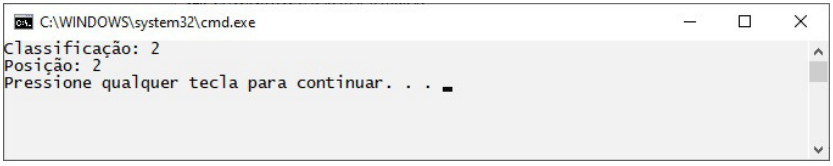


Figura 4.2: Repetindo o teste com a linha de código descomentada

Vale destacar que a evolução da linguagem fez com que a forma de testar se a variável `classificacao` possui ou não valor também fosse simplificada. Atualmente, podemos realizar o mesmo teste desta maneira:

```
if (classificacao!= null) {

    posicao = classificacao.Value;

}
```

Ou usando *pattern matching* introduzido no C# 7:

```
if (classificacao is int) {

    posicao = classificacao.Value;

}
```

Bem mais produtivo, não é verdade?

4.3 UTILIZANDO LITERAIS BINÁRIAS E SEPARADORES DE DÍGITOS

A partir do C# 7, é possível escrever uma literal numérica em binário além de hexadecimal. Observe:

```
int x = 0b10101010;
```

Note que esse recurso é bastante conveniente na definição de *máscaras de bits*, por exemplo.

Outra melhoria introduzida para tornar números grandes mais legíveis é a capacidade de agrupar os dígitos introduzindo separadores. Os separadores são inseridos por meio do caractere *underscore* (_) e podem ser usados em literais decimais,

hexadecimais e binárias. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int valorEmDecimal = 1_000_000_000;
            int valorEmHexadecimal = 0x7FFF_2345;
            int valorEmBinario = 0b1001_0110_1010_0101;
            Console.WriteLine($"valorEmDecimal: {valorEmDecimal}"
);
            Console.WriteLine($"valorEmHexadecimal: {valorEmHexad
ecimal}");
            Console.WriteLine($"valorEmBinario: {valorEmBinario}"
);
        }
    }

}
```

Obviamente, os separadores de dígitos só servem para facilitar a vida do desenvolvedor em tempo de codificação. Eles não afetam a saída gerada pelo código:

Figura 4.3: Utilizando literais binárias e separadores de dígitos

4.4 CRIANDO MÚLTIPLOS CONSTRUTORES DE INSTÂNCIA PARA UMA CLASSE

Construtores são tipos especiais de métodos usados para criar e

inicializar objetos. Eles permitem ao desenvolvedor criar objetos da classe, definir valores padrão e limitar a instanciação, dentre outras possibilidades. Ao trabalharmos com construtores, é preciso ter em mente os seguintes detalhes:

- Ao criarmos uma classe, um construtor de instância é automaticamente criado caso você não o crie de forma explícita em seu código. Este construtor é chamado de *construtor padrão* e tem como única função criar uma instância da classe.

- Em C#, os construtores usam o mesmo identificador que o nome da classe e não retornam um valor. Em construtores não se utiliza a palavra-chave `void` .

- Normalmente utilizamos o modificador de acesso `public` associado ao construtor para que outras classes possam instanciar objetos do seu tipo. É possível, entretanto, definir um construtor `private` ou `protected` para realizar tarefas específicas ou impedir que a classe seja instanciada.

- Quando você explicitamente escreve um construtor para uma classe, seja ele parametrizado ou não, você perde o construtor padrão que é criado automaticamente pelo compilador na ausência de um criado de forma explícita.

- Existem *construtores de instâncias* e o *construtor estático*, usado para inicializar campos de uma classe que possui *métodos estáticos*. Ao longo desta seção, focaremos apenas nos construtores de instância.

- Toda vez que uma classe é instanciada usando a palavra

`new` , o seu construtor é chamado.

O construtor padrão de uma classe não possui nenhum parâmetro e existe mesmo que você não o veja no seu código:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public string Sexo { get; set; }
        public DateTime DataNascimento { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa1 = new Pessoa()
            {
                Nome = "Beltrano de Tal",
                Sexo = "M",
                DataNascimento = new DateTime(1982, 8, 10)
            };

        }
    }
}
```

No exemplo a seguir, vamos sobrescrevê-lo com um construtor definido pela pessoa desenvolvedora:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public string Sexo { get; set; }
        public DateTime DataNascimento { get; set; }
```

```
        public Pessoa()
        {

        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa1 = new Pessoa()
            {
                Nome = "Beltrano de Tal",
                Sexo = "M",
                DataNascimento = new DateTime(1982, 8, 10)
            };

        }
    }
}
```

O construtor desse exemplo não possui código executável e existe somente para permitir a instância da classe. Ele pode ser inserido digitando o atalho `ctor` seguido de *tab*.

Assim como ocorre com outros métodos, os construtores também podem ser *sobrecarregados*, ou seja, podemos criar mais de um construtor para uma mesma classe. O próximo exemplo ilustra este cenário:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public string Sexo { get; set; }
        public DateTime DataNascimento { get; set; }

        public Pessoa()
```

```
        {

        }

        public Pessoa(string nome)
        {
            Nome = nome;
        }

        public Pessoa(string nome, string sexo, DateTime dataNasc
imento)
        {
            Nome = nome;
            Sexo = sexo;
            DataNascimento = dataNascimento;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa1 = new Pessoa();
            pessoa1.Nome = "Fulano de Tal";
            pessoa1.Sexo = "M";
            pessoa1.DataNascimento = new DateTime(1973, 4, 19);

            Pessoa pessoa2 = new Pessoa()
            {
                Nome = "Beltrano de Tal",
                Sexo = "M",
                DataNascimento = new DateTime(1982, 8, 10)
            };

            Pessoa pessoa3 = new Pessoa("Luana de Tal");
            pessoa3.Sexo = "F";
            pessoa1.DataNascimento = new DateTime(1990, 2, 22);

            Pessoa pessoa4 = new Pessoa("Mariana de Tal", "F", ne
 DateTime(1994, 7, 17));
        }
    }
}
```

Ao observar o código desta listagem, veja que criamos três construtores para a classe `Pessoa` que diferem no número de argumentos usados para inicializar o objeto quando a classe for instanciada. Compare as formas de inicializar os objetos `Pessoa1`, `Pessoa2`, `Pessoa3` e `Pessoa4`:

- `Pessoa1` utiliza o construtor sem argumentos. É necessário atribuir às linhas posteriores cada uma das propriedades de forma individual. Esta é a forma original suportada pela linguagem C# desde a sua criação.

- `Pessoa2` utiliza o construtor sem argumentos e um *inicializador de objeto* para atribuir cada uma das propriedades desejadas com menos esforço. Esta forma é uma alternativa à sobrecarga de construtores.

- `Pessoa3` utiliza um construtor que recebe como parâmetro apenas o atributo `Nome`. As linhas posteriores são usadas para atribuir outras propriedades do objeto.

- `Pessoa4` utiliza um construtor que aceita como parâmetro `Nome`, `Sexo` e `DataNascimento`. Todas as atribuições necessárias foram feitas usando o próprio construtor. Essa forma é mais concisa que a que utiliza inicializador de objeto, ainda que menos legível.

Vale destacar que podemos reduzir o tamanho do código sem perder legibilidade, conforme sinalizado na inicialização do objeto `Pessoa4`. Como o C# passou a suportar parâmetros nomeados, é possível reescrever a linha de declaração e inicialização do objeto `pessoa4` a seguir:

```
Pessoa pessoa4 = new Pessoa("Mariana de Tal", "F", new DateTime(1
```

```
994, 7, 17));
```

Da seguinte forma:

```
Pessoa pessoa4 = new Pessoa(nome:"Mariana de Tal", sexo: "F", dat
aNascimento: new DateTime(1994, 7, 17));
```

Muito mais legível, não concorda?

Outra melhoria que pode ser aplicada aos construtores é a *definição de corpo de expressão* (*expression body definition*) que pode ser usada quando o construtor puder ser implementado como uma única instrução. Isso quer dizer que este código:

```
public Pessoa(string nome)
{
    Nome = nome;
}
```

Pode ser implementado desta forma:

```
public Pessoa(string nome) => Nome = nome;
```

Em resumo, utilizar construtores parametrizados de instância juntamente com inicializadores de objetos nos permite simplificar a criação de objetos das nossas classes, reduzindo o número de linhas de código e o tempo de codificação.

4.5 DECLARANDO UMA PROPRIEDADE AUTOIMPLEMENTADA COMO SOMENTE LEITURA OU SOMENTE ESCRITA

A partir da versão 3.0, a linguagem C# passou a suportar *propriedades autoimplementadas* ou *automáticas*, o que tornou a declaração de uma propriedade muito mais concisa e fácil de ler. Em vez de declararmos o código desta forma:

```
private string _nome;

public string Nome
{
    get { return _nome; }
    set { _nome = value; }
}
```

Passamos a usar esta segunda maneira mais concisa:

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

Simples e rápida de codificar, especialmente se você souber que digitar "prop" seguido de *tab* no editor de código do Visual Studio fará com que o esqueleto da declaração seja automaticamente criado. O nosso trabalho será então substituir o tipo da propriedade pelo tipo desejado em cada caso e o nome da propriedade.

Até aqui a maioria dos desenvolvedores já sabe e aplica em sua jornada de codificação. O que muitos não sabem é que essa sintaxe de propriedades autoimplementadas também suporta a declaração de propriedades *somente de leitura* ou *somente de escrita*. Para declarar uma propriedade como somente leitura, basta configurar o escopo do assessor `set` como `private`:

```
public int Id { get; private set; }
```

Ou simplesmente omitir o assessor `set` da declaração, deste modo:

```
public int Id { get; }
```

Já para declarar uma propriedade como somente escrita, altere a visibilidade do assessor `get` como `private`. Exemplo:

```
public string Log { private get; set; }
```

Nesse caso não é possível remover o assessor `get` da declaração, pois propriedades autoimplementadas requerem a presença deste assessor.

Atribuindo um valor inicial a uma propriedade autoimplementada na sua declaração

As propriedades autoimplementadas também suportam a atribuição de um valor inicial no momento da sua declaração. Este recurso conhecido como *auto-property initializers* foi introduzido na versão 6.0 da linguagem. Observe a seguir dois exemplos:

```
public string Titulo { get; private set; } = "Produtividade em C#
;

public DateTime DataCadastramento { get; private set; } = DateTim
e.Now;
```

Note que, se o seu objetivo é criar uma propriedade somente de leitura, é possível ir além e utilizar uma expressão lambda conforme mostrado na próxima linha de código:

```
public string Nome => "Cláudio Ralha";
```

Utilizando propriedades somente de inicialização

O C# 9.0 introduziu um recurso conhecido como *propriedades somente de inicialização* (em inglês, *init-only properties* ou *init-only setters*), que oferece uma sintaxe consistente para inicializar membros de um tipo escrito pelo desenvolvedor. Acessadores `init` são usados em propriedades e indexadores no lugar de acessadores `set` para indicar que, após a etapa de inicialização, as propriedades se tornam apenas de leitura.

Isso nos permite fornecer valores iniciais para estas

propriedades através de inicializadores de objetos e de construtores parametrizados durante a criação do objeto e, em seguida, convertê-las em propriedades `readonly`.

Para conferir esse comportamento na prática, vamos utilizar o código a seguir, que só compilará no compilador do C# 9.0 que acompanha o .NET 5.0 ou superior:

```
using System;

namespace ProdutividadeEmCSharp1
{
    public class Livro
    {
        public string Titulo { get; init; }
        public string Autor { get; init; }
        public string Editora { get; init; }
        public int Ano { get; init; }

        public Livro() { }

        public Livro(string titulo, string autor, string editora,
int ano)
        {
            Titulo = titulo;
            Autor = autor;
            Editora = editora;
            Ano = ano;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var livro1 = new Livro()
            {
                Titulo = "Segredos do Visual Studio",
                Autor = "Cláudio Ralha",
                Editora = "Digerati",
                Ano = 2004
            };
            Console.WriteLine($"{livro1.Titulo} - Autor: {livro1.
```

```
Autor} - Editora:  {livro1.Editora} - Ano: {livro1.Ano}");
            var livroN = new Livro("Produtividade em C#", "Cláudi
o Ralha", "Casa do Código", 2020);
            Console.WriteLine($"{livroN.Titulo} - Autor: {livroN.
Autor} - Editora:  {livroN.Editora} - Ano: {livroN.Ano}");
            //Isto não é permitido
            //livroN.Ano = 2021;

        }
    }
}
```

Ao examinar esse exemplo, observe na classe `Livro` o uso de `init` nas propriedades definidas. Compile o código com a última linha comentada e, a seguir, repita a compilação com ela descomentada. Observe que será gerada uma mensagem de erro informando que a propriedade de inicialização ou indexador `Livro.Ano` só pode ser atribuída em um inicializador de objeto, em `this` ou `base` em um construtor de instância ou em um acessador `init`.

Empregando expressões lambda em propriedades calculadas

Ao escrevermos classes, muitas vezes precisamos criar propriedades calculadas como a propriedade `Total` do exemplo a seguir:

```
public class Item
{
    public int Quantidade { get; set; }
    public double Preco { get; set; }
    public double Total
    {
        get
        {
            return Quantidade * Preco;
        }
```

```
    }
}
```

Note que foi necessário escrever um método de apenas uma linha de código para calcular o valor `Total`. Não é algo prático, concorda? A partir do C# 6, passamos a utilizar uma forma mais concisa conhecida como membro *expression bodied*, que contém somente a expressão, ou seja, sem usar as chaves ou incluir um retorno explícito. Veja como fica o exemplo anterior reescrito usando essa funcionalidade:

```
public class Item
{
    public int Quantidade { get; set; }
    public double Preco { get; set; }
    public double Total => Quantidade * Preco;
}
```

Até a introdução desse recurso, só era possível utilizar expressões lambdas no corpo de métodos de uma classe. Agora, é possível implementar métodos e propriedades somente leitura a partir de expressões lambdas.

4.6 ITERANDO SOBRE UM ENUMERADO

Em alguns cenários, pode ser necessário percorrer toda a lista de constantes de um tipo enumerado, seja para exibir o conjunto de nomes, seja para retornar o conjunto de valores. Para ilustrar como executar estas ações com o mínimo esforço, vamos partir do enum `Estacoes`:

```
public enum Estacao
{
    Primavera = 0,
    Verao = 1,
    Outono = 2,
```

```
    Inverno = 3
}
```

A classe `Enum` fornece a classe base para enumerações e um conjunto de métodos, dentre os quais temos o método estático `Enum.GetNames` , usado para recuperar uma matriz dos nomes e retornar uma matriz de string dos nomes. Veja no exemplo a seguir como é simples utilizá-lo:

```
using System;

namespace ProdutividadeEmCSharp
{
    public enum Estacao
    {
        Primavera = 0,
        Verao = 1,
        Outono = 2,
        Inverno = 3
    }

    class Program
    {
        static void Main(string[] args)
        {
            foreach (var item in Enum.GetNames(typeof(Estacao)))
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

Ao executarmos esse código, obteremos a seguinte saída:

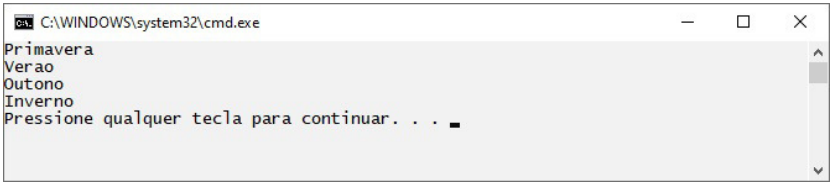


Figura 4.4: Listando as constantes de um tipo enumerado

Caso deseje recuperar a lista de valores do enum, utilize o método estático `GetValues` da classe `Enum` e converta os valores retornados para inteiros, como mostrado no próximo fragmento de código:

```
foreach (var item in Enum.GetValues(typeof(Estacao)))
{
    Console.WriteLine(Convert.ToInt32(item));
}
```

Confira o resultado na imagem a seguir:

Figura 4.5: Listando os valores de um tipo enumerado

4.7 UTILIZANDO INICIALIZADORES DE OBJETOS E DE COLEÇÕES

Ao inicializar valores de propriedade fora da classe, alguns desenvolvedores optam por inicializar as propriedades separadamente conforme ilustrado no exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class Pessoa
    {
        public string Nome { get; set; }

        public int Idade { get; set; }

        public char Sexo { get; set; }
```

```
    }

    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa = new Pessoa();
            pessoa.Nome = "Flavia";
            pessoa.Idade = 38;
            pessoa.Sexo = 'F';
        }
    }
}
```

Apesar de funcional, esta maneira é a menos produtiva. Desde a versão 3.0, o C# conta com um recurso chamado *inicializadores de objetos* (em inglês, *object initializers*), que nos permite reescrever o trecho de código a seguir:

```
Pessoa pessoa = new Pessoa();
pessoa.Nome = "Flavia";
pessoa.Idade = 38;
pessoa.Sexo = 'F';
```

da seguinte forma:

```
Pessoa pessoa = new Pessoa() { Nome = "Flavia" , Idade = 38, Sexo
 = 'F' };
```

Ao examinar a linha anterior, perceba que, além de poupar uma quantidade significativa de digitação, ainda ganhamos legibilidade do código ao optar por esta forma de inicializar as propriedades de um objeto.

Se você ainda não conhecia essa maneira de inicializar objetos e gostou da novidade, ficará mais feliz ainda em saber que o C# conta com o recurso de *inicializadores de coleção* (*collection initializers*) que nos permitem dar um passo além. Veja como é simples utilizá-los:

```
List<Pessoa> pessoas = new List<Pessoa>
{
    new Pessoa() { Nome = "Flavia" , Idade = 38, Sexo = 'F' },
    new Pessoa() { Nome = "Cláudio" , Idade = 46, Sexo = 'M' }
};
```

4.8 UTILIZANDO PARÂMETROS OPCIONAIS E PARÂMETROS NOMEADOS

O *polimorfismo* suportado pelas linguagens orientadas a objetos, como o C#, nos permite criar múltiplas versões de um método dentro de uma classe, variando apenas a sua *assinatura*. Ao longo da seção sobre construtores parametrizados no início deste capítulo, você já viu alguns exemplos de uso de polimorfismo e de parâmetros nomeados. A seguir, veremos mais um exemplo, só que dessa vez aplicado a métodos normais de instância:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static void Escrever(string mensagem)
        {
            Escrever(mensagem, false, ConsoleColor.White);
        }

        public static void Escrever(string mensagem, bool capital
izar)
        {
            Escrever(mensagem, capitalizar, ConsoleColor.White);
        }

        public static void Escrever(string mensagem, ConsoleColor
 cor)
        {
            Escrever(mensagem, false, cor);
        }
```

```
        public static void Escrever(string mensagem, bool capital
izar, ConsoleColor cor)
        {
            Console.ForegroundColor = cor;
            mensagem = capitalizar ? mensagem.ToUpper() : mensage
m;
            Console.WriteLine(mensagem);
        }

        static void Main(string[] args)
        {
            Console.BackgroundColor = ConsoleColor.Black;
            Console.Clear();
            Escrever("Produtividade em C#", ConsoleColor.Green);
            Escrever("Cláudio Ralha",false,ConsoleColor.Yellow);
            Escrever("Aprenda o jeito moderno de programar e extr
aia o máximo da linguagem em pouco tempo!");
        }
    }
}
```

No código dessa listagem, temos várias implementações do método `Escrever` . Conforme você pode observar, todas chamam internamente a implementação que possui o maior número de parâmetros. Ao ser executado, obteremos a seguinte saída:

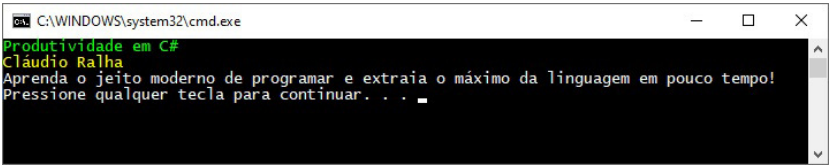


Figura 4.6: Utilizando parâmetros opcionais e parâmetros nomeados

Apesar de enxuto, o código desse exemplo pode ser simplificado utilizando o recurso de *parâmetros opcionais* introduzido na versão 3.0 da linguagem C#. Esses parâmetros são declarados com um valor default que será utilizado caso a

desenvolvedora não forneça um valor explícito para ele na chamada do método. Observe como ficaria o exemplo anterior reescrito de modo a tirar proveito desta funcionalidade:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static void Escrever(string mensagem, bool capital
izar = false, ConsoleColor cor = ConsoleColor.White)
        {
            Console.ForegroundColor = cor;
            mensagem = capitalizar ? mensagem.ToUpper() : mensage
m;
            Console.WriteLine(mensagem);
        }

        static void Main(string[] args)
        {
            Console.BackgroundColor = ConsoleColor.Black;
            Console.Clear();
            Escrever("Produtividade em C#", false, ConsoleColor.G
reen);
            Escrever("Cláudio Ralha",false,ConsoleColor.Yellow);
            Escrever("Aprenda o jeito moderno de programar e extr
aia o máximo da linguagem em pouco tempo!");
        }
    }
}
```

Ao examinar essa segunda listagem, você observará que o uso de parâmetros opcionais simplifica bastante o código-fonte da classe, ainda que não resolva todos os cenários. Note que foi necessário passar o segundo parâmetro para que pudéssemos especificar de forma explícita o terceiro na linha a seguir:

```
Escrever("Produtividade em C#", false, ConsoleColor.Green);
```

Se tentarmos utilizar qualquer uma das duas variações a seguir,

obteremos um erro de compilação:

```
Escrever("Produtividade em C#", , ConsoleColor.Green);
Escrever("Produtividade em C#", ConsoleColor.Green);
```

Para lidar com este tipo de limitação e permitir especificar apenas os parâmetros desejados e em ordem variável, foi introduzido no C# 4.0 o conceito de *parâmetros nomeados*. A próxima versão do nosso código de teste utiliza este recurso:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static void Escrever(string mensagem, bool capital
izar = false, ConsoleColor cor = ConsoleColor.White)
        {
            Console.ForegroundColor = cor;
            mensagem = capitalizar ? mensagem.ToUpper() : mensage
m;
            Console.WriteLine(mensagem);
        }

        static void Main(string[] args)
        {
            Console.BackgroundColor = ConsoleColor.Black;
            Console.Clear();
            Escrever("Produtividade em C#", cor: ConsoleColor.Gre
en, capitalizar: false);
            Escrever("Cláudio Ralha", cor: ConsoleColor.Yellow);
            Escrever("Aprenda o jeito moderno de programar e extr
aia o máximo da linguagem em pouco tempo!");
        }
    }
}
```

Observe que a ordem dos parâmetros foi invertida na linha de código a seguir e que é permitido misturar parâmetros nomeados com parâmetros não nomeados:

```
Escrever("Produtividade em C#", cor: ConsoleColor.Green, capitali
zar: false);
```

E nesta segunda linha de código, a chamada foi feita informando apenas a cor (terceiro parâmetro) sem passar o segundo parâmetro (capitalizar), ou seja, especificamos apenas os parâmetros que precisávamos utilizar:

```
Escrever("Cláudio Ralha", cor: ConsoleColor.Yellow);
```

Em resumo, o uso de parâmetros nomeados combinados com os parâmetros opcionais tornam o código em desenvolvimento mais conciso e legível, aumentando a produtividade e facilitando a manutenção.

4.9 UTILIZANDO MÉTODOS DE EXTENSÃO PARA ESTENDER UMA CLASSE

Métodos de extensão ou *extension methods* são um recurso introduzido na linguagem C# 3.0 para dar suporte a consultas LINQ que está disponível para classes e estruturas. Seu uso provê a capacidade de estender tipos existentes adicionando novos métodos sem a necessidade de modificar o tipo. Note que isso significa que podemos acrescentar métodos às classes cujo código não podemos manipular diretamente por não o termos disponível, e incluir novos métodos em estruturas e classes que não podiam ser estendidas pelo mecanismo de herança.

Para ilustrar como os métodos de extensão funcionam, vamos criar métodos extras para a classe `string` . Veja:

```
using System;
using System.Linq;
```

```
namespace ProdutividadeEmCSharp
{
    public static class MetodosExtensao
    {
        public static string ObterPrimeiraPalavra(this string val
or)
        {
            return valor.Split(null).FirstOrDefault();
        }

        public static string ObterUltimaPalavra(this string valor

        {
            return valor.Split(null).LastOrDefault();
        }

        public static string OrdenarPalavras(this string valor)
        {
            return OrdenarPalavras(valor,true);
        }

        public static string OrdenarPalavras(this string valor, b
ool ascendente)
        {
            var palavras = valor.Split(null);
            Array.Sort(palavras);
            if (ascendente == false) Array.Reverse(palavras);
            return String.Join(" ", palavras);
        }

        public static int ContarPalavras(this String valor)
        {
            return valor.Split(new char[] { ' ', '.', '?' }, Stri
ngSplitOptions.RemoveEmptyEntries).Length;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var frase = "Sou brasileiro e não desisto nunca";
            Console.WriteLine($"Frase original: {frase}");
            Console.WriteLine($"Primeira palavra: {frase.ObterPri
meiraPalavra()}");
```

```
            Console.WriteLine($"Última palavra: {frase.ObterUltim
aPalavra()}");
            Console.WriteLine($"Ordenando palavras (Asc): {frase.
OrdenarPalavras()}");
            Console.WriteLine($"Ordenando palavras (Desc):  {fras
e.OrdenarPalavras(false)}");
            Console.WriteLine($"Total de palavras: {frase.ContarP
alavras()}");
        }
    }
}
```

Ao examinar esse exemplo, considere os seguintes detalhes relacionados ao uso de métodos de extensão em C#:

- Um método de extensão é sempre estático e deve ser definido em uma classe estática.
- Um método de extensão pode ser apenas uma função que retorna ou não um valor. Não é possível definir uma propriedade, campo ou evento de extensão.
- O primeiro parâmetro na definição de um método de extensão especifica qual tipo de dado o método estende e ele é precedido pelo modificador `this` .
- Quando o método é executado, o primeiro parâmetro é vinculado à instância do tipo de dados invocada pelo método.

Perceba que a diferença dessa abordagem em relação à colocação desses métodos estáticos em uma classe de utilitários é que a chamada pode ser feita de forma semelhante à dos métodos de instância, apesar de esses métodos não serem de fato membros de instância do tipo. Em termos práticos, isso poupa o trabalho de lembrarmos onde os métodos estão realmente definidos, além de simplificar a sua sintaxe de uso.

Confira na imagem a seguir a saída produzida por esse exemplo:

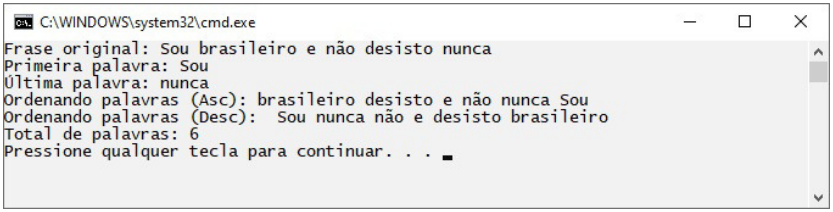


Figura 4.7: Estendendo uma classe com métodos de extensão

Ao utilizar métodos de extensão em seus projetos, atente-se aos seguintes pontos:

- Os métodos de extensão só estarão no escopo quando você importar explicitamente o namespace para seu código-fonte com uma diretiva `using` .
- Um método de extensão com o mesmo nome e assinatura de um método de instância não será chamado. Isso ocorre porque os métodos de extensão não podem ser usados para substituir os métodos existentes.

4.10 UTILIZANDO O TIPO DYNAMIC PARA RETORNAR OBJETOS DIFERENTES

Um dos recursos mais interessantes inseridos na linguagem C# é a palavra-chave `dynamic` , que nos permite, entre outras coisas, retornar objetos diferentes em tempo de execução baseado, por exemplo, nos parâmetros passados para um método. Em termos práticos, isso significa que um mesmo método pode retornar vários tipos de objetos, preenchendo uma lacuna deixada até então pelo polimorfismo no C#.

Ao contrário da palavra-chave `var` que usamos para inferir o tipo da variável baseado no seu valor inicial, `dynamic` não obtém as propriedades e métodos do objeto em tempo de compilação. Ela o faz em tempo de execução, o que a torna mais flexível em alguns cenários.

O exemplo a seguir ilustra como usar `dynamic` no retorno de um método:

```
using System;

namespace ProdutividadeEmCSharp
{

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

    class Autor : Pessoa
    {
        public string Email { get; set; }
    }

    class Livro
    {
        public string Titulo { get; set; }
        public int Edicao { get; set; }
        public int Ano { get; set; }
    }

    class Program
    {
        private static dynamic ObterObjeto(string objeto)
        {
            switch (objeto.ToLower())
            {
                case "pessoa":
                    Pessoa pessoa = new Pessoa() { Nome="Flavia",
 Sexo = 'F', Idade= 38};
```

```
                    return pessoa;
                case "autor":
                    Autor autor = new Autor() { Nome = "Cláudio",
 Sexo = 'M', Idade = 46, Email = "claudioralha@hotmail.com" };
                    return autor;
                case "livro":
                    Livro livro = new Livro() { Titulo = "Produti
vidade em C#", Edicao = 1, Ano = 2020 };
                    return livro;
                default:
                    return "Esta opção não é válida";
            }
        }

        static void Main(string[] args)
        {
            Pessoa pessoa = ObterObjeto("pessoa");
            Console.WriteLine($"Nome: {pessoa.Nome} - Sexo: {pess
oa.Sexo} - Idade: {pessoa.Idade}");
            Autor autor = ObterObjeto("autor");
            Console.WriteLine($"Nome: {autor.Nome} - Sexo: {autor
.Sexo} - Idade: {autor.Idade} - Email: {autor.Email}");
            Livro livro = ObterObjeto("livro");
            Console.WriteLine($"Título: {livro.Titulo} - Edição:
{livro.Edicao} - Ano: {livro.Ano}");
        }
    }
}
```

Ao examinar o código desta listagem, observe que o método `ObterObjeto` é usado para retornar três tipos diferentes de objetos (`Pessoa` , `Autor` e `Livro`) com o mínimo de código e de forma legível. Confira na imagem a seguir a saída gerada por esse exemplo:

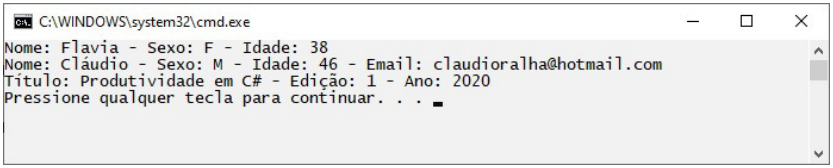


Figura 4.8: Utilizando dynamic para retornar objetos de tipos diferentes

Para saber mais sobre o uso de `dynamic` em C#, consulte a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/dynamic

4.11 UTILIZANDO YIELD EM VEZ DE CRIAR COLEÇÕES TEMPORÁRIAS

Com frequência, escrevemos códigos semelhantes ao do exemplo a seguir, nos quais criamos coleções temporárias para selecionar itens de uma outra coleção. Veja:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static List<int> ObterNumerosPares(List<int> lista)
        {
            List<int> listaTemporaria = new List<int>();

            foreach (int numero in lista)
            {
                if(numero % 2 == 0)
                {
                    listaTemporaria.Add(numero);
                }
            }
            return listaTemporaria;
        }

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

```
            List<int> numeros = new List<int>() {0,1,2,3,4,5,6,7,
,9,10};
            List<int> numerosPares = ObterNumerosPares(numeros);
            foreach (var numero in numerosPares)
            {
                Console.WriteLine(numero);
            };
        }
    }
}
```

Ao examinar esse código, observe que criamos a lista `listaTemporaria` do tipo `List<int>` no método `ObterNumerosPares`. De fato, não há nada de errado com esse exemplo, mas existem formas de obter o mesmo resultado com menos código: usando `yield` ou LINQ para objetos - sobre este falaremos no capítulo 7.

A linguagem C# desde a sua versão 2.0 conta com a *palavra-chave contextual* `yield` para lidar com cenários como esse. Veja na listagem a seguir como fica o exemplo anterior reescrito usando `yield`:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static IEnumerable<int> ObterNumerosPares(List<int> lista

        {
            foreach (int numero in lista)
            {
                if(numero % 2 == 0)
                {
                    yield return numero;
                }
            }
```

```
        }

        static void Main(string[] args)
        {
            List<int> numeros = new List<int>() {0,1,2,3,4,5,6,7,
,9,10};
            IEnumerable<int> numerosPares = ObterNumerosPares(num
eros);
            foreach (var numero in numerosPares)
            {
                Console.WriteLine(numero);
            };
        }
    }
}
```

Confira a seguir a explicação sobre o uso de `yield` extraída da documentação da linguagem C#:

> "Ao usar a palavra-chave contextual `yield` em uma instrução, você indica que o método, o operador ou o acessador `get` em que ela é exibida é um iterador. Usar `yield` para definir um iterador elimina a necessidade de uma classe adicional explícita (a classe que mantém o estado de uma enumeração, consulte `IEnumerator<T>` para obter um exemplo) ao implementar o padrão `IEnumerable` e `IEnumerator` para um tipo de coleção personalizado".
>
> **Importante**: uma palavra-chave contextual é usada para fornecer um significado específico no código em determinado contexto, mas não é uma palavra reservada no C#.

Ao serem executados, ambos os programas produzirão a seguinte saída:

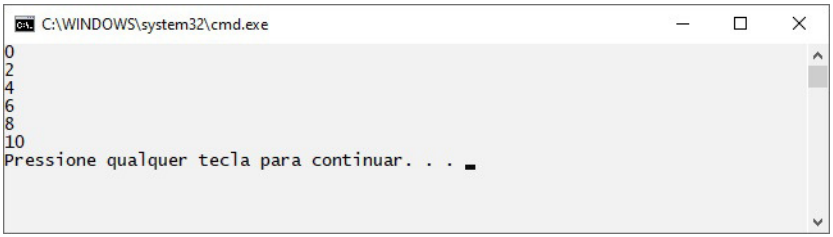


Figura 4.9: Utilizando yield em vez de criar uma coleção temporária

Vale destacar que existem algumas restrições de uso do `yield`. Não é possível incluir uma instrução `yield return` ou `yield break` em métodos anônimos e em métodos que contêm blocos inseguros e também há restrições de uso do `yield` em tratamento de exceções.

Para obter informações mais detalhadas, restrições de uso e outros exemplos de aplicação da palavra-chave `yield` em C#, consulte a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/yield

Para conferir a lista de palavras-chaves contextuais suportadas pelo C#, consulte a página a seguir:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/contextual-keywords

4.12 UTILIZANDO DECLARAÇÕES USING PARA SINALIZAR OBJETOS DESCARTÁVEIS

Até o C# 7.x, é necessário utilizar uma instrução `using` para definir o escopo dos objetos de tipo gerenciado que acessam recursos não gerenciados, pois ela garante o uso correto de objetos que implementam a interface `IDisposable`, evitando vazamentos de memória, além de manter presos recursos caros como conexões com bancos de dados. Veja a seguir um exemplo:

```
using System.IO;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void CriarArquivo()
        {
            using (var arquivo = new StreamWriter("exemplo.txt"))
            {
                arquivo.WriteLine("Produtividade em C#");
            }
        }

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

Ao analisar esse código, note que a instrução `using` mostra de forma explícita quando o recurso será limpo. Isso acontece quando a execução do código passa da chave de fechamento.

A partir do C# 8.0, podemos declarar um objeto descartável com a palavra-chave `using` precedendo a declaração da variável,

conforme mostrado na próxima listagem:

```
using System.IO;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void CriarArquivo()
        {
            using var arquivo = new StreamWriter("exemplo.txt");
            arquivo.WriteLine("Produtividade em C#");
        }

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

Declarações `using` são apenas declarações de variáveis locais. O objeto criado desta forma será descartado assim que sair do escopo, o que ocorre no final do bloco de instruções atual. Nesse exemplo, ao terminar a execução do método `CriarArquivo` .

Ao comparar as duas formas de se codificar, percebemos que a instrução `using` é mais explícita, oferecendo um controle mais preciso em relação à liberação dos recursos, enquanto a declaração `using` se limita a facilitar a leitura por não introduzir um nível extra de aninhamento. Pessoalmente, eu prefiro continuar usando instruções `using` .

Para saber mais sobre as pequenas diferenças de escopo entre a declaração `using` e a instrução `using`, recomendamos a leitura da documentação oficial e do seguinte artigo do mestre Macoratti:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/using-statement

http://www.macoratti.net/19/12/c8_using1.htm

4.13 UTILIZANDO MÉTODOS DE INTERFACE PADRÃO

O C# 8.0 introduz um novo recurso conhecido como *métodos de interface padrão* (em inglês, *default interface methods*), que vai bagunçar tudo o que você estudou no passado sobre interfaces e classes abstratas.

Os métodos de interface padrão nos permitem fornecer uma implementação para membros da interface. Deste modo, quando uma classe que implementa a interface não implementar o membro de interface especificado, será usado o método padrão da própria interface se este possuir um corpo.

Para entender os motivos que levaram o time de desenvolvimento do C# a incluir este recurso controverso na linguagem, transcrevemos a justificativa incluída na documentação oficial:

Esse recurso de linguagem permite que os autores de API adicionem métodos a uma interface em versões posteriores sem interromper a fonte ou a compatibilidade binária com implementações existentes dessa interface. As implementações existentes herdam a implementação padrão. Esse recurso também permite que o C# interopere com APIs que direcionam o Android ou o Swift, que dão suporte a recursos semelhantes. Os métodos de interface padrão também habilitam cenários semelhantes a um recurso de linguagem de "características".

Referência:
https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-8#default-interface-methods

Para ilustrar como utilizar métodos de interface padrão, vamos imaginar que tenhamos uma interface chamada `ILogger`, que originalmente foi definida como mostrada a seguir:

```
public interface ILogger
{
    void Info(string mensagem);
    void Error(string mensagem);
}
```

Após ser implementada em algumas classes do sistema, como `ConsoleLogger` e `DebugLogger`, surgiu a necessidade de adicionar um novo método chamado `Warn`. Sem usar o recurso de métodos de interface padrão introduzido no C# 8, teríamos que incluir o método `Warn` em todas as classes que implementam a interface `ILogger` conforme demonstrado no próximo exemplo:

```
using System;

namespace ProdutividadeEmCSharp
{
    public interface ILogger
    {
        void Info(string mensagem);
        void Error(string mensagem);
        void Warn(string mensagem);

    }

    public class ConsoleLogger : ILogger
    {
        public void Error(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método Er
ror da classe ConsoleLogger]");
        }

        public void Info(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método In
fo da classe ConsoleLogger]");
        }

        public void Warn(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método da
 classe ConsoleLogger]");
        }
    }


    public class DebugLogger : ILogger
    {
        public void Error(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método Er
ror da classe DebugLogger]");
        }

        public void Info(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método In
```

```
fo da classe DebugLogger]");
        }

        public void Warn(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método Wa
rn da classe DebugLogger]");
        }
    }


    class Program
    {

        static void Main(string[] args)
        {
            ILogger console = new ConsoleLogger();
            console.Error("Testando método Error da classe Consol
eLogger!");
            console.Info("Testando método Info  da classe Console
Logger!");
            console.Warn("Testando método Warn  da classe Console
Logger!");
            ILogger debug = new DebugLogger();
            debug.Error("Testando método Error da classe DebugLog
ger!  ");
            debug.Info("Testando método Info  da classe DebugLogg
er!  ");
            debug.Warn("Testando método Warn  da classe DebugLogg
er!  ");
        }
    }
}
```

Com o suporte à nova funcionalidade, basta implementar o método `Warn` na interface `ILogger` e sobrescrevê-lo onde de fato precisarmos. Para as demais classes, é possível utilizar a implementação do método `Warn` existente na interface. Veja:

```
using System;

namespace ProdutividadeEmCSharp
{
```

```
    public interface ILogger
    {
        void Info(string mensagem);
        void Error(string mensagem);

        void Warn(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método Wa
rn da interface ILogger]");
        }
    }

    public class ConsoleLogger : ILogger
    {
        public void Error(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método Er
ror da classe ConsoleLogger]");
        }

        public void Info(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método In
fo da classe ConsoleLogger]");
        }
    }


    public class DebugLogger : ILogger
    {
        public void Error(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método Er
ror da classe DebugLogger]");
        }

        public void Info(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método In
fo da classe DebugLogger]");
        }

        public void Warn(string mensagem)
        {
            Console.WriteLine($"{mensagem}  [Executando método Wa
```

```
rn da classe DebugLogger]");
        }
    }


    class Program
    {

        static void Main(string[] args)
        {
            ILogger console = new ConsoleLogger();
            console.Error("Testando método Error da classe Consol
eLogger!");
            console.Info("Testando método Info  da classe Console
Logger!");
            console.Warn("Testando método Warn  da classe Console
Logger!");
            ILogger debug = new DebugLogger();
            debug.Error("Testando método Error da classe DebugLog
ger!  ");
            debug.Info("Testando método Info  da classe DebugLogg
er!  ");
            debug.Warn("Testando método Warn  da classe DebugLogg
er!  ");
        }
    }
}
```

Confira na imagem a seguir a execução do método `Warn` da interface `ILogger` (terceira linha):

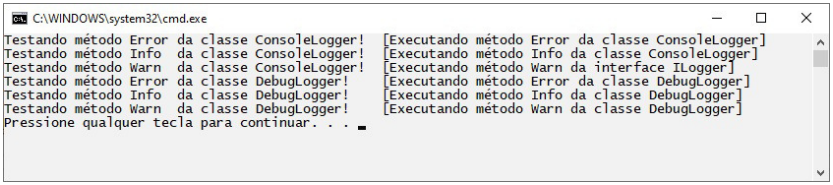


Figura 4.10: Executando o método warn definido na interface ILogger

Como curiosidade, saiba que essa funcionalidade já estava disponível em nível de linguagem MSIL há muitos anos, mas não

era possível tirar proveito dela em C# diretamente, apenas via *reflection*.

Com a chegada do C# 8.0, a implementação de membros de interface passou a ser suportada pelo compilador.

Este recurso será visto por alguns como uma benção, já que será conveniente e realmente útil em muitos cenários, e por outros como uma maldição, pois distorce a funcionalidade planejada para as classes de implementação. Isso porque ele permite alterar a assinatura da interface, fazendo com que as classes de implementação ganhem novas funcionalidades injetadas, que podem não fazer sentido ou, em casos mais extremos, gerar conflitos com algo presente na classe de implementação.

Como já dizia o velho ditado, "a diferença entre o remédio e o veneno está na dose". Portanto, procure analisar cada caso e utilizar este recurso com moderação, lembrando que as classes abstratas continuam disponíveis na linguagem.

Para um estudo mais avançado sobre *métodos de interface padrão*, recomendamos a leitura dos seguintes tutoriais:

https://docs.microsoft.com/pt-br/dotnet/csharp/tutorials/default-interface-methods-versions

https://docs.microsoft.com/pt-br/dotnet/csharp/tutorials/mixins-with-default-interface-methods

4.14 INICIALIZANDO OBJETOS USANDO NEW EXPRESSIONS

Desde o C# 1.0, nós nos acostumamos a instanciar objetos usando a sintaxe a seguir:

```
Pessoa pessoa1 = new Pessoa();
Pessoa pessoa2 = new Pessoa("Cláudio","Ralha");
```

O C# 3.0 nos trouxe a *inferência de tipo* e evoluímos para esta forma mais elegante:

```
var pessoa3 = new Pessoa("Iara","Ralha");
```

Com a chegada do C# 9.0, passamos a contar com uma nova forma de instanciar objetos conhecida como *new expressions*, um recurso que permite simplificar a escrita de instruções de criação de objetos. Em termos práticos, a funcionalidade torna possível eliminar o uso do nome das classes após a palavra-chave `new` desde que tenhamos especificado o nome do tipo no início da linha. Isso nos permite reescrever o exemplo inicial desta forma:

```
Pessoa pessoa1 = new();
Pessoa pessoa2 = new("Cláudio","Ralha");
```

Obviamente, a nova funcionalidade não é permitida com a palavra `var` . A linha a seguir impedirá o programa de compilar:

```
var pessoa3 = new("Iara","Ralha");
```

Perceba que nesse caso não há como o compilador inferir que classe o desenvolvedor deseja utilizar. Para conferir o uso do novo recurso na prática, execute o exemplo a seguir. Note que os nomes dos personagens usados neste exemplo pertencem à tragédia *Romeu e Julieta* de William Shakespeare, escrita entre 1591 e 1595:

```
namespace ProdutividadeEmCSharp9
{
    public class Pessoa
    {
        public string PrimeiroNome { get; set; }
        public string UltimoNome { get; set; }

        public int Idade { get; set; }

        public Pessoa() { }

        public Pessoa(string primeiroNome, string ultimoNome)
        {
            PrimeiroNome = primeiroNome;
            UltimoNome = ultimoNome;
        }

        public Pessoa(string primeiroNome, string ultimoNome, int
 idade) => (PrimeiroNome, UltimoNome, Idade) = (primeiroNome, ult
imoNome, idade);
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Antes do C# 9.0:
            Pessoa pessoa1 = new Pessoa();
            pessoa1.PrimeiroNome = "Romeu";
            pessoa1.UltimoNome = "Montéquio";
            pessoa1.Idade = 17;
            Pessoa pessoa2 = new Pessoa("Teobaldo", "Capuleto");
            Pessoa pessoa3 = new Pessoa("Julieta", "Capuleto",13)
;
            Pessoa pessoa4 = new Pessoa() { PrimeiroNome = "Julie
ta", UltimoNome = "Capuleto", Idade = 13 };
            //A partir do C# 9.0
            Pessoa pessoa5 = new();
            pessoa5.PrimeiroNome = "Romeu";
            pessoa5.UltimoNome = "Montéquio";
            pessoa5.Idade = 17;
            Pessoa pessoa6 = new ("Teobaldo", "Capuleto");
            Pessoa pessoa7 = new ("Julieta", "Capuleto", 13);
            Pessoa pessoa8 = new() { PrimeiroNome = "Julieta", Ul
timoNome = "Capuleto", Idade = 13 };
```

```
        }
    }
}
```

Apesar de não ser algo que implique em uma mudança significativa na forma como desenvolvemos e em um ganho de produtividade real como ocorreu com o uso do `var`, essa funcionalidade envolve uma nova forma de instanciar objetos que será certamente explorada em provas de certificação e de processos seletivos para o mercado de trabalho. Particularmente, prefiro continuar utilizando o nome da classe após a palavra `new`, mas é uma questão de gosto pessoal e não de obrigatoriedade.

4.15 UTILIZANDO REGISTROS PARA CRIAR TIPOS DE REFERÊNCIA IMUTÁVEIS

O principal recurso introduzido no C# 9.0 são os *registros* (em inglês, *records*) usados para permitir que objetos inteiros sejam imutáveis e se comportem como tipos de valor. Eles fazem uso das *propriedades somente de inicialização*, abordadas anteriormente, que permitem tornar as propriedades individuais imutáveis.

Tipos de registro são tipos de referência, semelhantes a classes, mas definidos usando a palavra-chave `record`. Veja um exemplo:

```
using System;

namespace ProdutividadeEmCSharp9
{
    public record Pessoa
    {
        public string PrimeiroNome { get; init; }
        public string UltimoNome { get; init; }

        public Pessoa() { }
```

```
        public Pessoa(string primeiroNome, string ultimoNome) =>
(PrimeiroNome, UltimoNome) = (primeiroNome, ultimoNome);
    }

    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa1 = new Pessoa("Cláudio", "Ralha");
            Console.WriteLine($"Olá, {pessoa1.PrimeiroNome} {pess
oa1.UltimoNome}!");
            //Só para lembrar, isto gera erro
            //pessoa1.PrimeiroNome = "Iara";
        }
    }
}
```

Ao analisar rapidamente esse código, a única diferença aparente que encontraremos é a substituição da palavra-chave `class` por `record`, certo? Pois saiba que apenas com registros podemos reduzir a declaração de um tipo a uma única linha. Veja:

```
namespace ProdutividadeEmCSharp9
{
    public record Pessoa (string PrimeiroNome, string UltimoNome)
;


    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa1 = new Pessoa("Cláudio", "Ralha");
        }
    }
}
```

Essa declaração de registro que torna as propriedades declaradas automaticamente somente de inicialização, por si só, já será suficiente para fazer com que muitos desenvolvedores queiram migrar imediatamente para o C# 9 e o .NET 5. Mas para

entender por que registros nos serão verdadeiramente úteis no dia a dia, vamos ver como eles se comportam em relação à cópia, à herança e à comparação de objetos.

Em muitos casos, desejamos criar um novo objeto a partir de outro, uma vez que alguns valores de propriedade são idênticos. Para estes cenários, o C# 9.0 disponibiliza a palavra-chave `with`. Ela nos permite criar um objeto a partir de outro, especificando quais alterações de propriedade desejamos efetuar. Observe:

```
static void Main(string[] args)
{
    Pessoa pessoa1 = new Pessoa("Cláudio", "Ralha");
    Pessoa pessoa2 = pessoa1 with { PrimeiroNome = "Iara" };
}
```

Para entender como isso se tornou tão fácil, segue a explicação do time de desenvolvimento da Microsoft:

> "Os registros possuem um método virtual oculto que é encarregado de "clonar" todo o objeto. Todo tipo de registro derivado substitui esse método para chamar seu construtor de cópia. Esse construtor de cópia se encadeia ao construtor de cópia do registro base. Uma expressão `with` simplesmente chama o método `clone` oculto e aplica o inicializador de objeto ao resultado.". (Em tradução livre de https://devblogs.microsoft.com/dotnet/welcome-to-c-9-0/)

Os registros, assim como as classes, oferecem suporte à herança simples de implementação. Ao criar registros em C#, tenha em mente as seguintes regras:

- Os registros não podem herdar de classes (com exceção da classe `object`) e as classes não podem herdar de registros.
- Os registros podem ser herdados de outros registros.

O próximo exemplo ilustra o uso de herança com registros:

```
using System;

namespace ProdutividadeEmCSharp9
{
    public record Pessoa
    {
        public string PrimeiroNome { get; }
        public string UltimoNome { get; }

        public Pessoa() { }

        public Pessoa(string primeiroNome, string ultimoNome) =>
(PrimeiroNome, UltimoNome) = (primeiroNome, ultimoNome);
    }


    public record Autor : Pessoa
    {
        public string Pseudonimo { get; }

        public Autor(string primeiroNome, string ultimoNome, stri
ng pseudonimo)
            : base(primeiroNome, ultimoNome) => Pseudonimo = pseu
donimo;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Autor autora = new Autor("Agatha", "Christie", "Mary
Westmacott");
            Console.WriteLine($"{autora.PrimeiroNome} {autora.Ult
imoNome} possui mais de 80 obras publicadas e é a mais prolífica
escritora de romances policiais da literatura. Além das obras pol
iciais,{autora.PrimeiroNome} também é autora de doze peças de tea
tro e de seis romances de época que foram assinados com o pseudôn
```

```
imo de {autora.Pseudonimo}, o primeiro publicado em 1930 e o sext
o em 1956.");
        }
    }
}
```

No construtor da classe `Autor`, veja a chamada ao construtor da classe base `Pessoa` para popular as propriedades `PrimeiroNome` e `UltimoNome` e a atribuição de valor da propriedade `Pseudonimo` declarada na própria classe `Autor`.

Em termos de comparação de objetos, registros funcionam como structs, pois substituem o método virtual `Equals` para permitir a comparação baseada em valor. Isso significa que cada propriedade será comparada com uma abordagem baseada em valor. Para comprovar, execute o próximo exemplo:

```
using System;

namespace ProdutividadeEmCSharp9
{
    public record Pessoa (string PrimeiroNome, string UltimoNome)
;


    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa1 = new Pessoa("Cláudio", "Ralha");
            Pessoa pessoa2 = new Pessoa("Cláudio", "Ralha");
            Pessoa pessoa3 = new Pessoa("Iara", "Ralha");
            Console.WriteLine("Usando Equals:");
            Console.WriteLine($"pessoa1 e pessoa2: {pessoa1.Equal
s(pessoa2)}");
            Console.WriteLine($"pessoa1 e pessoa3: {pessoa1.Equal
s(pessoa3)}");
            Console.WriteLine("Usando ==:");
            Console.WriteLine($"pessoa1 e pessoa2: {pessoa1 == pe
ssoa2}");
            Console.WriteLine($"pessoa1 e pessoa3: {pessoa1 == pe
```

```
ssoa3}");
        }
    }
}
```

Conforme você deve ter percebido em nossa rápida abordagem sobre o tema, o uso de registro pode tornar a tarefa de codificação muito mais simples e é algo que merece um estudo aprofundado. A documentação disponível ainda é escassa, mas isso deve mudar em pouco tempo.

> Importante: ao pesquisar sobre essa funcionalidade, você encontrará postagens e artigos mais antigos nos quais aparece a palavra-chave `data` no lugar de `record`. Para saber mais sobre registros, acesse:
>
> https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-9#record-types

Ao longo deste extenso capítulo, foram apresentados a você alguns recursos da linguagem C# que são desconhecidos para muitos desenvolvedores e desenvolvedoras, incluindo pessoas que possuem anos de estrada. As funcionalidades que abordamos vão ajudar você a resolver problemas complexos do dia a dia de forma inteligente e rápida, pois elas surgiram como respostas para dificuldades levantadas pelo time de desenvolvimento do C# e por outros desenvolvedores da comunidade técnica.

Capítulo 5

TUPLAS

Uma *tupla* (em inglês, *tuple*) é uma estrutura de dados que contém uma sequência de elementos de diferentes tipos de dados. Ela pode ser usada onde você deseja ter uma estrutura de dados para manter um objeto com propriedades, e se você não deseja criar um tipo separado para ela.

O suporte a tuplas foi introduzido no Framework .NET 4.0 através da classe `Tuple<T>` . Contudo, até o lançamento da versão 7.0 do C#, as tuplas eram ineficientes e não tinham nenhum suporte de linguagem, o que acabava desencorajando o seu uso. Dentre as desvantagens, a que mais incomodava era que os elementos de tupla do framework .NET 4.0 só podiam ser referenciados como `Item1` , `Item2` e assim por diante.

Com o suporte nativo da linguagem para tuplas incluído no C# 7.0, o desenvolvedor passou a dispor de nomes semânticos para os campos de uma tupla, usando tipos de tuplas novos e mais eficientes.

As principais vantagens oferecidas pelo emprego de tuplas em nosso código são as seguintes:

- Representar um conjunto único de dados com acesso fácil e manipulação do conjunto.

- Retornar múltiplos valores de um método sem usar parâmetros `out` .
- Passar múltiplos valores para um método usando um único parâmetro.

Ao longo deste capítulo, aprenderemos como tirar proveito deste poderoso recurso suportado pela linguagem C#. Iniciaremos estudando como retornar múltiplos valores de um método usando uma *tupla*. A seguir, aprenderemos como passar múltiplos valores para um método usando uma *tupla* ou *lista de tuplas*. Em seguida, demonstraremos como *desconstruir* os elementos de uma *tupla* e de uma *classe*. O capítulo termina apresentando como descartar *retornos de métodos* e *parâmetros out*.

Habilitando as tuplas do C# 7

Para usar a segunda geração de tuplas em seus programas, é necessário incluir o pacote `System.ValueTuple` através do gerenciador de pacotes NuGet. Isso é feito executando os seguintes passos:

1. Clique no menu *Ferramentas* do Visual Studio, selecione *Gerenciador de Pacotes do NuGet* e, a seguir, a opção *Gerenciar Pacotes do NuGet para a Solução*.

2. Clique na guia *Procurar* do gerenciador de pacotes e pesquise por `System.ValueTuple` . Clique no item encontrado na lista da esquerda e, a seguir, no botão *Instalar* no painel da direita.

3. A caixa de diálogos *Visualizar Alterações* será exibida. Clique no botão *Ok* para avançar.

4. Ao final da instalação, feche o gerenciador de pacotes clicando no botão de fechar existente na guia.

> **Importante**: a classe `Tuple` do .NET 4.0, mencionada anteriormente, continua existindo no framework, mas não será utilizada nos exemplos deste capítulo. A diferença é que `System.ValueTuple` é um tipo de valor (struct) enquanto `System.Tuple` é um tipo de referência (classe). `System.ValueTuple` expõe seus itens por meio de campos no lugar de propriedades.

5.1 RETORNANDO MÚLTIPLOS VALORES DE UM MÉTODO USANDO UMA TUPLA

Iniciaremos o nosso estudo sobre tuplas, criando um exemplo simples no qual temos uma tupla sem nome e uma tupla nomeada. Compare:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var casal1 = ("Eduardo", "Mônica");
            var casal2 = (Esposo: "Cláudio", Esposa: "Flávia");
            Console.WriteLine($"Casal 1 -> Esposo: {casal1.Item1}
 Esposa: {casal1.Item2}");
            Console.WriteLine($"Casal 2 -> Esposo: {casal2.Esposo
} Esposa: {casal2.Esposa}");
        }
```

```
    }
}
```

A tupla `casal1` foi inicializada usando constantes literais e não terá nomes de elementos criados usando as projeções de nome de campo de tupla no C# 7.1. Logo, para acessar seus itens utilizamos `Item1` ou `Item2`. Já a tupla `casal2`, conhecida como tupla nomeada, fornece nomes melhores para cada campo.

Note que ainda podemos nos referir aos elementos de uma tupla nomeada como `Item1`, `Item2` etc., mas fornecer nomes a cada elemento que criamos torna o seu uso muito mais produtivo.

Veja a seguir a saída gerada pelo exemplo anterior:

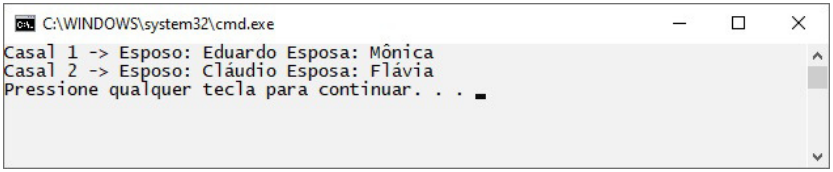


Figura 5.1: Acessando os itens das tuplas

Os nomes de campo para uma tupla a partir do C# 7.1 podem ser fornecidos por meio das variáveis usadas para inicializar a tupla. Isso é conhecido como *inicializadores de projeção de tupla*. O código da listagem a seguir ilustra este cenário:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var titulo = "A volta dos que não foram";
            var autor = "Jacinto Volta";
```

```
            var filme = (titulo, autor);
            Console.WriteLine($"Filme: {filme.titulo} Autor: {fil
me.autor}");
        }
    }
}
```

Confira a saída produzida por esse exemplo na janela a seguir:

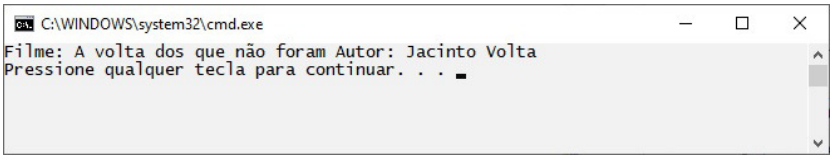


Figura 5.2: Referenciando os itens de uma tupla pelo nome

No próximo exemplo, utilizamos uma tupla nomeada para retornar mais de um valor de um método:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    static class Matematica
    {
        static internal(int Minimo, int Maximo) ObterMinimoMaximo
(List<int> lista)
        {
            int max = int.MinValue;
            int min = int.MaxValue;
            lista.ForEach(n =>{
                min = n < min ? n : min;
                max = n > max ? n : max;
            });
            return (min, max);
        }
    }

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

```
        {
            var numeros = new List<int>() { 22, 5, 78, 45, 89, 17
 };
            var resultado = Matematica.ObterMinimoMaximo(numeros)
;
            Console.WriteLine($"Mínimo: {resultado.Minimo} Máximo
: {resultado.Maximo}");
        }
    }
}
```

Observe na próxima imagem a saída produzida por este exemplo:

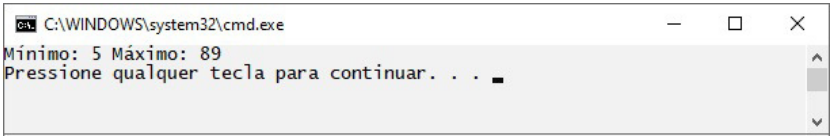


Figura 5.3: Retornando o menor e o maior valor de uma lista de inteiros usando uma tupla nomeada

5.2 PASSANDO MÚLTIPLOS VALORES PARA UM MÉTODO USANDO TUPLAS

Assim como podemos ter uma tupla como valor de retorno, também podemos ter como um dos parâmetros de um método uma tupla, uma lista ou um array de tuplas.

O primeiro exemplo desta seção ilustra como passar uma tupla como parâmetro para um método:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
```

```
        static void Listar((int, int, int, int, int, int,int) res
ultado)
        {
            var(concurso,d1,d2,d3,d4,d5,d6) = resultado;
            Console.WriteLine($"Resultado do concurso: {concurso}
);
            Console.WriteLine($"Dezenas sorteadas    : {d1} - {d2
} - {d3} - {d4} - {d5} - {d6}");
        }

        static void Main(string[] args)
        {
            var resultado = (1526, 4,9,16,34,38,45);
            Listar(resultado);
        }
    }
}
```

Ao analisar esse código, você verá que usamos a técnica de desconstrução na linha a seguir para tornar o código das linhas posteriores mais simples e legível:

```
var(concurso,d1,d2,d3,d4,d5,d6) = resultado;
```

Confira o resultado da execução desse exemplo na próxima imagem:

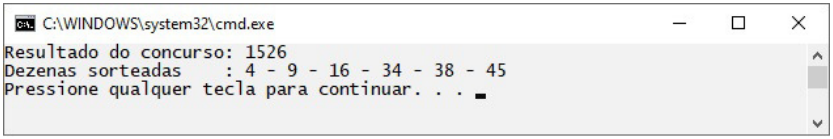


Figura 5.4: Passando múltiplos valores para um método usando uma tupla

Neste segundo exemplo, demonstramos como criar uma lista de tuplas, inicializá-la e em seguida passá-la como parâmetro para um método. Veja:

```
using System;
using System.Collections.Generic;
```

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Listar(List<(string, string, string)> palavra
s)
        {
            Console.WriteLine($"Português - Inglês - Espanhol");
            foreach (var (portugues, ingles, espanhol) in palavra
s)
            {
                Console.WriteLine($"{portugues} - {ingles} - {esp
anhol}");
            }
        }

        static void Main(string[] args)
        {
            var palavras = new List<(string, string, string)>
                                {
                                    ("garfo", "fork", "tenedor"),
                                    ("faca", "knife","cuchillo"),
                                    ("colher", "Spoon","cuchara")
                                };
            Listar(palavras);
        }
    }
}
```

Observe que também usamos a técnica de desconstrução no loop `foreach` do método `Listar` para simplificar o código. Ao ser executado, esse exemplo produz a saída mostrada a seguir:

Figura 5.5: Passando múltiplos valores para um método usando uma lista de tuplas

5.3 DESCONSTRUINDO OS ELEMENTOS DE UMA TUPLA

As tuplas do C# 7 contam com um interessante recurso conhecido como *desconstrução* que nos permite desmontar os elementos de uma tupla e acessá-los através de variáveis independentes. O exemplo a seguir demonstra o uso desta funcionalidade:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    static class Matematica
    {
        static internal (int Minimo, int Maximo) ObterMinimoMaxim
(List<int> lista)
        {
            int max = int.MinValue;
            int min = int.MaxValue;
            lista.ForEach(n => {
                min = n < min ? n : min;
                max = n > max ? n : max;
            });
            return (min, max);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var numeros = new List<int>() { 22, 5, 78, 45, 89, 17
 };
            var (minimo,maximo) = Matematica.ObterMinimoMaximo(nu
meros);
            Console.WriteLine($"Mínimo: {minimo} Máximo: {maximo}
);
        }
    }
```

```
}
```

A desconstrução ocorre nesta linha:

```
var (minimo,maximo) = Matematica.ObterMinimoMaximo(numeros);
```

Na linha seguinte é possível observar como ela simplifica o código:

```
Console.WriteLine($"Mínimo: {minimo} Máximo: {maximo}");
```

E o melhor de tudo é que não é necessário fornecer uma variável para cada um dos parâmetros retornados pelo método se você não pretende utilizar todos. Você pode informar o caractere `_` para ignorar cada campo que não utilizará. Veja:

```
static void Main(string[] args)
{
    var numeros = new List<int>() { 22, 5, 78, 45, 89, 17 };
    var (minimo, _) = Matematica.ObterMinimoMaximo(numeros);
    Console.WriteLine($"Mínimo: {minimo}");
}
```

Este recurso é conhecido como *descarte* e será tratado em mais detalhes após falarmos sobre o uso da desconstrução em classes.

5.4 DESCONSTRUINDO OS ELEMENTOS DE UMA CLASSE

Na seção anterior, vimos que a desconstrução nos permite desmembrar uma tupla em variáveis. A boa notícia para quem gostou dessa abordagem é que é possível utilizar a desconstrução com outros tipos. Para oferecer suporte à desconstrução, basta que o tipo implemente o método `Deconstruct` apenas com parâmetros `out`. Veja a seguir um exemplo:

```
using System;
```

```
namespace ProdutividadeEmCSharp
{

    class Pessoa
    {
        public string Nome { get; set; }

        public int Idade { get; set; }

        public void Deconstruct(out string nome, out int idade)
        {
            nome = Nome;
            idade = Idade;
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            Pessoa autor = new Pessoa() { Nome = "Cláudio Ralha",
 Idade = 47};
            //Aplicando a desconstrução
            var(nome,idade) = autor;
            Console.WriteLine($"O autor {nome} tem {idade} anos."
);
        }
    }
}
```

Um detalhe a ser notado é que podemos implementar essa funcionalidade para classes de terceiros empregando um método de extensão. A versão alterada do exemplo anterior mostrada a seguir simula este cenário:

```
using System;

namespace ProdutividadeEmCSharp
{

    public class Pessoa
    {
```

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

        public int Idade { get; set; }
    }

    public static class PessoaExtensoes
    {
        public static void Deconstruct(this Pessoa pessoa, out st
ring nome, out int idade)
        {
            nome = pessoa.Nome;
            idade = pessoa.Idade;
        }

    }


    class Program
    {
        static void Main(string[] args)
        {
            Pessoa autor = new Pessoa() { Nome = "Cláudio Ralha",
 Idade = 47};
            //Aplicando a desconstrução
            var(nome,idade) = autor;
            Console.WriteLine($"O autor {nome} tem {idade} anos."
);
        }
    }
}
```

Desse modo, é possível tornar mais simples o uso de uma classe por meio de desconstrução, mesmo quando não se tem acesso ao seu código-fonte.

5.5 DESCARTANDO RETORNOS DE MÉTODOS E PARÂMETROS OUT

Por vezes, o desenvolvedor pode desejar ignorar o resultado de um método. Isso é uma tarefa simples quando se trata do valor de

retorno do método, pois basta ignorá-lo como no exemplo:

```
namespace ProdutividadeEmCSharp
{

    class Program
    {
        private static int ObterValor() => 45;

        static void Main(string[] args)
        {
            var valor = ObterValor();

            ObterValor();
        }
    }
}
```

Nesse exemplo, temos duas chamadas ao método `ObterValor`. Na primeira, atribuímos o valor de retorno à variável `valor`. Na segunda chamada, o valor de retorno não é associado a nenhuma variável e é consequentemente descartado.

Enquanto ignorar o valor de retorno de um método sempre foi uma tarefa trivial em C#, fazer o mesmo com uma *variável out* antes do C# 7.0 não era algo simples, uma vez que a associação era obrigatória. Veja a seguir um exemplo:

```
namespace ProdutividadeEmCSharp
{

    class Program
    {
        private static void ObterValor(out int valor) => valor =
46;

        static void Main(string[] args)
        {
            int resultado;
            ObterValor(out resultado);
        }
```

```
    }
}
```

O C# 7.0 introduziu as variáveis `out`, que permitem que o código anterior seja reescrito da seguinte forma:

```
namespace ProdutividadeEmCSharp
{

    class Program
    {
        private static void ObterValor(out int valor) => valor =
46;

        static void Main(string[] args)
        {
            ObterValor(out var resultado);
        }
    }
}
```

Note que, mesmo neste caso, continuamos tendo uma variável sendo introduzida e associada, ainda que usando menos código. Felizmente, outro recurso incluído no C# 7 pode ser empregado para eliminar esse incômodo: *descartes* (em inglês, *discards*). O caractere underscore (_) é usado para indicar cada ocorrência do parâmetro ou variável que deve ser ignorado.

Você já o viu em funcionamento na seção anterior quando explicamos a desconstrução de tupla. Ele nos permite fazer algo como mostrado no próximo exemplo:

```
using System;

namespace ProdutividadeEmCSharp
{

    class Program
    {
```

```
        static void Main(string[] args)
        {
            var tupla = (1, 2, 4, 8, 16, 32);
            (_, var segundo, _, _, var quinto, _) = tupla;
            Console.WriteLine($"Segundo valor da Tupla: {segundo}
);
            Console.WriteLine($"Quinto valor da Tupla: {quinto}")
;
        }
    }
}
```

Ao ser executado, esse código produzirá a seguinte saída:

Figura 5.6: Descartando valores desnecessários durante a desconstrução de uma tupla

No contexto das variáveis `out`, utilizamos o descarte conforme mostrado no exemplo da próxima listagem:

```
namespace ProdutividadeEmCSharp
{

    class Program
    {
        private static void ObterValor(out int valor) => valor =
45;

        static void Main(string[] args)
        {
            ObterValor(out int _);
        }
    }
}
```

Note que é permitido inclusive descartar o tipo da variável, ou seja, a linha a seguir:

```
ObterValor(out int _);
```

Para ser reescrita da seguinte forma:

```
ObterValor(out _);
```

Como curiosidade, saiba que também podemos utilizar esse recurso para descartar uma variável. Veja:

```
namespace ProdutividadeEmCSharp
{

    class Program
    {

        static void Main(string[] args)
        {
            _ = 5 + 10;
        }
    }
}
```

Obviamente, não é possível ler o valor atribuído ao underscore. Se você tentar, obterá um erro de compilação. Experimente:

```
using System;

namespace ProdutividadeEmCSharp
{

    class Program
    {

        static void Main(string[] args)
        {
            _ = 5 + 10;
            Console.WriteLine($" Valor de _: {_}");
        }
    }
}
```

Ao tentar compilar o programa anterior, será gerada a seguinte

mensagem no Visual Studio em português: *O nome "_" não existe no contexto atual.*

Conforme você pôde observar ao longo das seções anteriores, o uso de tuplas simplifica muito o nosso trabalho. A grande questão para muitos é saber quando devemos criar uma classe e quando devemos trabalhar com uma tupla. Tenha em mente que, onde a criação de uma classe não faz muito sentido, a tupla provavelmente poderá ser usada. São cenários em que estamos lidando com valores não relacionados, nos quais o conteúdo agrupado é o que realmente importa.

Tuplas são um assunto vasto que facilmente ocuparia uma obra inteira. Neste capítulo, vimos um pouco do que é possível fazer com elas, do ponto de vista do aumento de produtividade do desenvolvedor. Para saber mais sobre o suporte a tuplas oferecido pelo C# 7.0 e posterior, acesse os seguintes artigos da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/tuples

https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-7#tuples

Para os casos em que é necessário escolher entre o uso de uma tupla e um tipo anônimo, recomendamos a leitura do seguinte artigo:

https://docs.microsoft.com/pt-br/dotnet/standard/base-types/choosing-between-anonymous-and-tuple

Capítulo 6

GENERICS

Generics é um recurso extremamente produtivo para a escrita de código, uma vez que evita retrabalho e maximiza o desempenho enquanto mantém a segurança de tipo. Incluído na versão 2.0 do framework .NET, generics nos permite postergar a especificação do tipo de dado dos elementos para o momento em que ele é usado no código, em vez de informá-lo na etapa de design do tipo, ou seja, é o código de chamada que decide o tipo ao instanciar uma classe genérica. Em termos práticos, isso significa que o código é montado de modo a não precisar se preocupar com o tipo, o que torna possível que a classe ou método que utiliza generics trabalhe com qualquer tipo de dado.

Apesar de passar despercebido para muitos desenvolvedores e desenvolvedoras e de assustar iniciantes, nós utilizamos generics com frequência em nosso dia a dia quando trabalhamos com listas, dicionários, LINQ e outros recursos mais avançados da linguagem. Por exemplo, ao criarmos uma lista de strings usando `List<T>` onde `T` é substituído por `string`, estamos usando generics para especificar o tipo. O parâmetro `T` é usado em generics para indicar que algo usa um parâmetro de tipo genérico. Esses parâmetros indicam os tipos que desejamos usar dentro de uma classe genérica.

Neste capítulo, veremos como explorar generics na construção de nossas próprias classes e métodos e como limitar os tipos que podem ser construídos a partir de um tipo genérico usando *constraints* tanto em tipos quanto em métodos. Em seguida, abordaremos como trabalhar com interfaces genéricas e não genéricas na definição de classes e coleções genéricas. O emprego de interfaces genéricas em nosso código ajuda a evitar ocorrências de operações de conversão *boxing* e *unboxing* em tipos de valor.

O capítulo termina abordando a disponibilidade de um grande número de classes e interfaces genéricas no próprio framework prontas para uso, para evitar que você perca tempo "reinventando a roda".

6.1 CRIANDO TIPOS GENÉRICOS

Para entender como o *generics* funciona na prática, vamos partir do exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class ClasseGenerica<T>
    {
        private T _value;

        public ClasseGenerica(T value)
        {
            _value = value;
        }
        public void IdentificarTipoDado()
        {
            Console.WriteLine($"Tipo fornecido -> T: {_value.GetT
ype()}");
        }
    }
```

```
    public class Livro
    {
        public string Titulo { get; set; }
        public string Autor { get; set; }
    }


    public class Program
    {
        static void Main(string[] args)
        {
            ClasseGenerica<int> varInt = new ClasseGenerica<int>(
18);
            varInt.IdentificarTipoDado();

            ClasseGenerica<string> varString = new ClasseGenerica
<string>("Produtividade em C#");
            varString.IdentificarTipoDado();

            Livro livro = new Livro() { Titulo="Produtividade em
C#", Autor="Claudio Ralha" };
            ClasseGenerica<Livro> varLivro = new ClasseGenerica<L
ivro>(livro);
            varLivro.IdentificarTipoDado();
        }
    }
}
```

Observe que nesse exemplo definimos uma classe genérica que recebe como parâmetro em seu construtor o tipo de dado a ser utilizado. Isso é informado com o *parâmetro de tipo genérico* `<T>`. Esse parâmetro é um *placeholder* (em inglês, *espaço reservado*) para o tipo específico que o desenvolvedor vai fornecer quando a classe for instanciada. É preciso obrigatoriamente fornecer esse argumento de tipo dentro dos *parênteses angulares* quando instanciamos a classe. Veja:

```
ClasseGenerica<int> varInt = new ClasseGenerica<int>(18);
ClasseGenerica<string> varString = new ClasseGenerica<string>("Pr
odutividade em C#");
```

```
ClasseGenerica<Livro> varLivro = new ClasseGenerica<Livro>(livro)
;
```

O segundo detalhe a ser notado é que utilizamos uma variável privada do tipo do parâmetro genérico `T` que vai armazenar o valor passado para a classe genérica:

```
private T _value;
```

Esta variável é populada no construtor da classe:

```
public ClasseGenerica(T value)
{
     _value = value;
}
```

No método `IdentificarTipoDado`, utilizamos o método `GetType` presente em todos os tipos criados em .NET e .NET Core para obter o tipo passado no parâmetro genérico `<T>`.

Confira a seguir, a saída gerada pelo exemplo anterior:

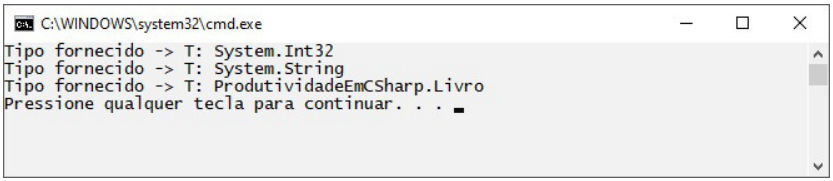


Figura 6.1: Definindo uma classe com um parâmetro genérico

Tenha em mente que o parâmetro genérico que vimos no exemplo anterior é nomeado por convenção como `<T>`, mas pode ter o nome que você desejar e é possível inclusive passar mais de um parâmetro para a classe. O código da próxima listagem ilustra esse cenário:

```
using System;
```

```
namespace ProdutividadeEmCSharp
{
    public class ClasseGenerica<T1,T2>
    {
        private T1 _value1;
        private T2 _value2;

        public ClasseGenerica(T1 value1, T2 value2)
        {
            _value1 = value1;
            _value2 = value2;
        }
        public void IdentificarTipoDado()
        {
            Console.WriteLine($"Tipos fornecidos -> T1: {_value1.
GetType()}  T2: {_value2.GetType()}");
        }
    }


    public class Program
    {
        static void Main(string[] args)
        {
            ClasseGenerica<int,string> teste = new ClasseGenerica
<int,string>(2020, "Feliz ano novo!");
            teste.IdentificarTipoDado();
        }
    }
}
```

Ao ser executado, esse segundo exemplo produzirá a seguinte saída:

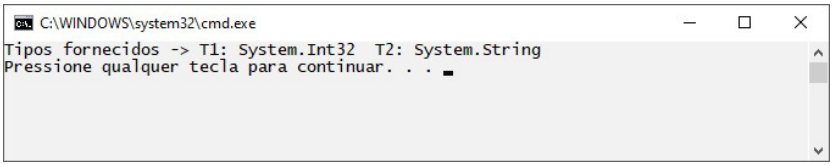


Figura 6.2: Definindo uma classe com dois parâmetros genéricos

6.2 UTILIZANDO CONSTRAINTS EM TIPOS GENÉRICOS

Apesar de toda a flexibilidade vista na seção anterior, haverá cenários em que não desejaremos que se possa criar qualquer tipo a partir de um tipo genérico que tenhamos codificado.

Restrições (em inglês, *constraints*) são especificadas usando a *palavra-chave contextual* `where` . Elas são necessárias, pois informam o compilador C# das funcionalidades que um argumento de tipo deve ter. Quando não temos nenhuma restrição, o argumento de tipo pode ser qualquer tipo. Apesar de soar como algo bom, isso obriga o compilador a assumir somente os membros de `Object` para o tipo, que é a classe base definitiva para qualquer tipo .NET.

Em outras palavras, *ao restringir o parâmetro de tipo, aumenta-se a quantidade de operações e chamadas de método permitidas àqueles com suporte do tipo de restrição e de todos os tipos de sua hierarquia de herança.*

A tabela a seguir, extraída da documentação oficial, lista os sete tipos de restrições atualmente suportados pelos generics:

Restrição	Descrição
`where T : struct`	O argumento de tipo deve ser um tipo de valor. Qualquer valor de tipo com exceção de `Nullable<T>` pode ser especificado.
`where T : class`	O argumento de tipo deve ser um tipo de referência. Essa restrição se aplica também a qualquer classe, interface, delegado ou tipo de matriz.
`where T : unmanaged`	O argumento de tipo não deve ser um tipo de referência e não deve conter nenhum membro de tipo de referência em nenhum nível de aninhamento.
	O argumento de tipo deve ter um construtor público sem parâmetros.

`where T : new()`	Quando usado em conjunto com outras restrições, a restrição `new()` deve ser a última a ser especificada.
`where T : <nome de classe base>`	O argumento de tipo deve ser ou derivar da classe base especificada.
`where T : <nome da interface>`	O argumento de tipo deve ser ou implementar a interface especificada. Várias restrições de interface podem ser especificadas. A interface de restrição também pode ser genérica.
`where T : U`	O argumento de tipo fornecido para `T` deve ser ou derivar do argumento fornecido para `U`.

O código do próximo exemplo ilustra como implementar uma restrição de tipo `T : <nome de classe base>`:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public DateTime DataNascimento { get; set; }
    }

    public class PessoaFisica: Pessoa
    {
        public char Sexo { get; set; }
        public string CPF { get; set; }
    }

    public class PessoaJuridica : Pessoa
    {
        public string CNPJ { get; set; }
    }

    public class ClasseGenerica<T> where T:Pessoa
    {
        private T _value;

        public ClasseGenerica(T value)
        {
```

```
            _value = value;
        }
        public void InformarResumo()
        {
            switch (_value)
            {
                case PessoaFisica pf:
                    string artigo = pf.Sexo == 'M' ? "O" : "A";
                    Console.WriteLine($"{artigo} cliente {pf.Nome
} possui o CPF {pf.CPF} e nasceu em {pf.DataNascimento.ToShortDat
eString()}.");
                    break;
                case PessoaJuridica pj:
                    Console.WriteLine($"A empresa {pj.Nome} possu
i o CNPJ {pj.CNPJ} e foi fundada em {pj.DataNascimento.ToShortDat
eString()}.");
                    break;
            }
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            PessoaFisica pessoa = new PessoaFisica() { Nome = "Cl
audio Ralha", Sexo = 'M', CPF = "123.456.789-10", DataNascimento
= new DateTime(1973,4,19)  };
            ClasseGenerica<PessoaFisica> varPF = new ClasseGeneri
ca<PessoaFisica>(pessoa);
            varPF.InformarResumo();
            PessoaJuridica empresa = new PessoaJuridica() { Nome
= "Ralha Teletransporte Express", CNPJ = "01.234.567/0001-89", Da
taNascimento = new DateTime(2003, 4, 23) };
            ClasseGenerica<PessoaJuridica> varPJ = new ClasseGene
rica<PessoaJuridica>(empresa);
            varPJ.InformarResumo();
        }
    }
}
```

Ao observar este código, note que implementamos a restrição de que só é possível aceitar argumento da classe `Pessoa` ou

classes derivadas desta, como `PessoaFisica` e `PessoaJuridica`. Veja também que empregamos no método `InformarResumo` o recurso de *correspondência de padrão de tipo* na instrução `switch` introduzido no C# 7.

O exemplo anterior produzirá a seguinte saída:

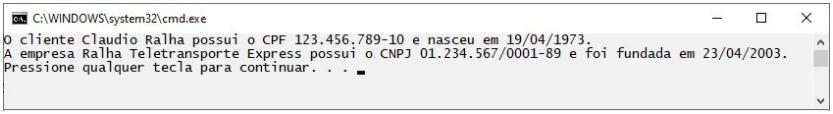


Figura 6.3: Implementando uma restrição de tipo de classe base

Caso o código de cliente tente criar uma instância da classe usando um tipo não permitido por uma restrição, o resultado será um erro em tempo de compilação. Para conferir isso na prática, altere o programa anterior incluindo a classe `Livro`:

```
public class Livro
{
    public string Titulo { get; set; }
    public string Autor { get; set; }
}
```

Em seguida, tente usá-la no método `Main` para criar uma instância da classe genérica:

```
Livro livro = new Livro() { Titulo = "Produtividade em C#", Autor
 = "Cláudio Ralha" };
ClasseGenerica<Livro> varLivro = new ClasseGenerica<Livro>(livro)
;
varLivro.InformarResumo();
```

Você obterá a seguinte sinalização de erro:

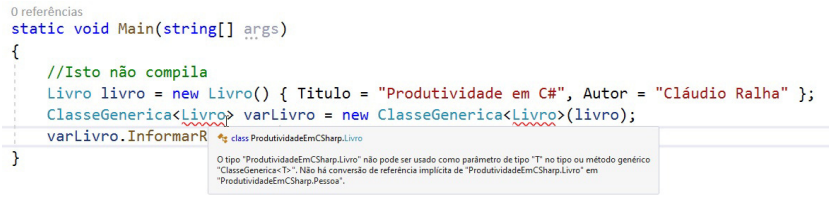


Figura 6.4: Gerando um erro decorrente da restrição de tipo de classe base

Ao utilizar constraints em generics saiba que você pode combinar múltiplas restrições na definição de uma única classe genérica. O fragmento de código a seguir, extraído da documentação oficial, ilustra este cenário:

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComp
arable<T>, new()
{
    // ...
}
```

Perceba como constraints em generics são um recurso poderoso que aumenta a segurança do código, evitando, dentre outras coisas, que novas pessoas de uma equipe de desenvolvimento utilizem uma determinada classe genérica para finalidades para as quais não foi planejada.

Apesar de os generics estarem presentes desde o C# 2.0, este é um modelo de programação que está em constante desenvolvimento. No C# 7.3, as restrições nos parâmetros de tipos passaram a suportar *restrições não gerenciadas* (voltada para tipos não gerenciados), *restrições de delegados* e *restrições de enumerados*.

6.3 UTILIZANDO CONSTRAINTS EM MÉTODOS GENÉRICOS

As constraints de parâmetros de tipos genéricos que utilizamos previamente em nível de tipo, podem ser aplicadas também em métodos genéricos. O exemplo a seguir ilustra este cenário:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    public class ClasseGenerica<T> where T: class
    {
        public string Concatenar<U>(List<U> items) where U : stru
ct
        {
            string resultado = string.Empty;
            bool primeiro = true;
            foreach (var item in items)
            {
                if (primeiro == false)
                {
                    resultado += ",";
                }
                else
                {
                    primeiro = false;
                }
                resultado += item.ToString();
            }
            return resultado;
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            string saida = String.Empty;
            List<int> inteiros = new List<int>(){0,1,2,3,4,5,6,7,
8,9};
```

```
            List<char> vogais = new List<char>() {'a','e','i','o'
,'u'};
            List<string> estacoes = new List<string>() {"Primaver
a","Verão","Outono","Inverno"};
            ClasseGenerica<string> classe = new ClasseGenerica<st
ring>();
            saida = classe.Concatenar<int>(inteiros);
            Console.WriteLine($"inteiros: {saida}");
            saida = classe.Concatenar<char>(vogais);
            Console.WriteLine($"vogais: {saida}");
            //Isso não compila
            //saida = classe.Concatenar<string>(estacoes);
            //Console.WriteLine($"estacoes: {saida}");
        }
    }
}
```

Note que aplicamos uma restrição de parâmetro de tipo `<T>` na classe genérica indicando que ela só pode receber como argumento um *tipo de referência* (ou seja, uma *classe* que neste exemplo não usamos para nada) e aplicamos uma restrição semelhante no método `Concatenar` indicando que ele só pode receber uma lista `List<U>` , na qual o parâmetro de tipo `U` obrigatoriamente precisa ser um *tipo de valor* (uma estrutura). Ao executarmos esse exemplo com o trecho de código final do método `Main` comentado, obteremos a saída a seguir:

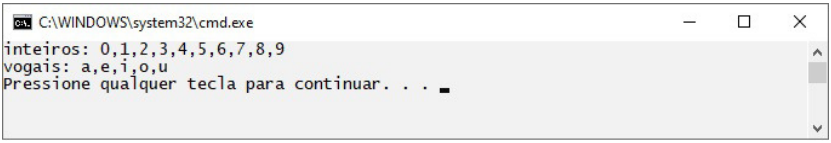


Figura 6.5: Implementando uma restrição de parâmetro de tipo em um método genérico

Ao descomentarmos o trecho final, veremos que o editor de código do Visual Studio sinalizará que não é possível utilizar o tipo `string` como parâmetro para o método `Concatenar` , por ele ser

um tipo de referência e não um tipo de valor. Veja:

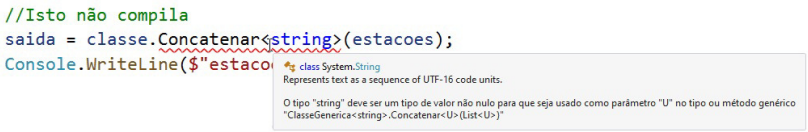


Figura 6.6: Gerando um erro decorrente da restrição de tipo no parâmetro de método

Isso nos permite limitar os tipos com os quais desejamos trabalhar no método com um mínimo de esforço.

Vale destacar que é possível definir um método genérico em uma classe não marcada como genérica. Na próxima seção, ilustraremos um exemplo que demonstra esse cenário.

6.4 DRIBLANDO LIMITAÇÕES DE CÁLCULOS EM MÉTODOS GENÉRICOS

Efetuar cálculos em tipos genéricos não é uma tarefa tão simples como pode parecer em um primeiro momento. Para entendermos por que isso ocorre, vamos partir do exemplo a seguir, que não é compilável:

```
using System;

namespace ProdutividadeEmCSharp
{
    public class Calculadora
    {
        public T Adicionar<T>(T a, T b)
        {
            return a + b;
        }
    }
```

```
    public class Program
    {
        static void Main(string[] args)
        {
            Calculadora calc = new Calculadora();
            int a = 5, b = 10;
            double c = 6.50, d = 8.12;
            string e = "Produtividade", f = " em C#";
            var resultado1 = calc.Adicionar(a, b);
            Console.WriteLine($"{a} + {b} = {resultado1} ({result
ado1.GetType()})");
            var resultado2 = calc.Adicionar(c, d);
            Console.WriteLine($"{c} + {d} = {resultado2} ({result
ado2.GetType()})");
            var resultado3 = calc.Adicionar(e, f);
            Console.WriteLine($"{e} + {f} = {resultado3} ({result
ado3.GetType()})");
        }
    }
}
```

Ao parar o mouse sobre a linha que contém a soma de `a` e `b` no método genérico `Adicionar` da classe `Calculadora`, você verá o aviso de erro informando que não é possível aplicar o operador `+` aos operandos do tipo `T`. Isso ocorre porque os generics são implementados em C# em tempo de execução e são verificados em tempo de compilação. Como não definimos qualquer restrição para o parâmetro de tipo, ele será do tipo `System.Object`.

Infelizmente, a classe base `Object` não define uma operação de adição, pois nem todos os objetos suportam esta operação. Como `T` pode ser qualquer tipo em tempo de execução, o compilador não pode determinar o significado de `a + b`, pois os tipos de `a` e de `b` ainda não foram determinados.

A primeira solução que nos vem à cabeça ao enfrentarmos este dilema é restringir os parâmetros de tipo definindo uma interface

que o tipo deverá implementar. Essa abordagem, todavia, não vai funcionar, pois as interfaces não podem conter métodos estáticos e os métodos de operador precisam ser estáticos.

> Conforme explicado no artigo de Rudiger Klaehn, publicado no site *Code Project*, com o sistema de restrições corrente, não é possível definir constraints para operadores:
>
> https://www.codeproject.com/Articles/8531/Using-generics-for-calculations

Para tornar esse código compilável, é preciso recorrer ao tipo estático `dynamic` introduzido no C# 4.0. Veja:

```
2 referências
public class Calculadora
{
    3 referências
    public T Adicionar<T>(T a, T b)
    {
        return (dynamic) a + b;
    }
}
```

Figura 6.7: Convertendo o resultado da soma para dynamic

O tipo `dynamic` criará uma *árvore de expressão* que será resolvida em tempo de execução. Confira a saída produzida pelo exemplo anterior ao ser executado:

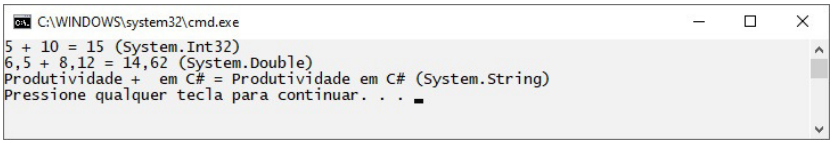


Figura 6.8: Efetuando somas com diferentes tipos de dados

Perceba que para contornar o problema foi necessário abrir mão da verificação de tipo em tempo de compilação usando `dynamic` . Esta abordagem tem um custo em termos de segurança e desempenho que precisa ser considerado.

Para saber mais sobre a criação de tipos genéricos em C# e sobre a definição de restrições nos parâmetros de tipos, acesse as seguintes páginas da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

6.5 CRIANDO INTERFACES GENÉRICAS

A criação de interfaces genéricas em C# é muito semelhante à criação de uma classe genérica. As interfaces genéricas são definidas para classes de coleção genéricas ou para as classes genéricas que representam itens na coleção.

Veja a seguir um exemplo no qual temos uma classe genérica

`Calculadora<T>` que implementa uma interface genérica `ICalculadora<T>`:

```
using System;

namespace ProdutividadeEmCSharp
{
    public interface ICalculadora<T>
    {
        T Adicionar(T a, T b);
        T Subtrair(T a, T b);
        T Multiplicar(T a, T b);
        T Dividir(T a, T b);
    }

    public class Calculadora <T> : ICalculadora<T>
    {
        public T Adicionar(T a, T b)
        {
            return (dynamic) a + b;
        }

        public T Subtrair(T a, T b)
        {
            return (dynamic) a - b;
        }

        public T Dividir(T a, T b)
        {
            return (dynamic) a / b;
        }

        public T Multiplicar(T a, T b)
        {
            return (dynamic) a * b;
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Calculadora<int> calc = new Calculadora<int>();
            int a = 10, b = 5;
```

```
            Console.WriteLine($"{a} + {b} = {calc.Adicionar(a, b)
}");
            Console.WriteLine($"{a} - {b} = {calc.Subtrair(a, b)}
");
            Console.WriteLine($"{a} * {b} = {calc.Multiplicar(a,
b)}");
            Console.WriteLine($"{a} / {b} = {calc.Dividir(a, b)}"
);
        }
    }
}
```

Ao ser executado, esse exemplo vai produzir a seguinte saída:

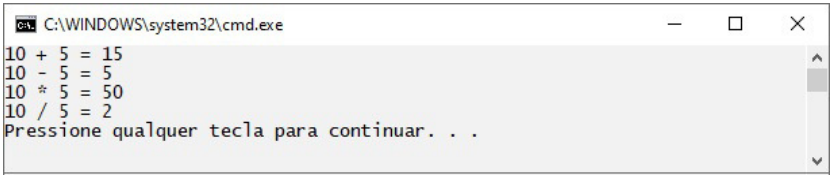


Figura 6.9: Criando uma classe que herda de uma interface genérica

Note que também é possível usar constraints na definição de uma interface. Poderíamos alterar o código da interface `ICalculadora`, do exemplo anterior, como a seguir:

```
public interface ICalculadora<T> where T : struct
{
    T Adicionar(T a, T b);
    T Subtrair(T a, T b);
    T Multiplicar(T a, T b);
    T Dividir(T a, T b);
}
```

Para que o código continue compilável, será necessário aplicar a restrição também à classe `Calculadora<T>`. Veja:

```
public class Calculadora<T> : ICalculadora<T> where T: struct
{
    public T Adicionar(T a, T b)
    {
```

```
        return (dynamic) a + b;
    }

    public T Subtrair(T a, T b)
    {
        return (dynamic) a - b;
    }

    public T Dividir(T a, T b)
    {
        return (dynamic) a / b;
    }

    public T Multiplicar(T a, T b)
    {
        return (dynamic) a * b;
    }
}
```

Cabe ressaltar que, se uma ou mais interfaces (genéricas ou não genéricas) forem especificadas como restrições em um parâmetro de tipo, somente os tipos que implementarem as interfaces enumeradas poderão ser usados. O exemplo a seguir ilustra este segundo cenário:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    public interface IObra
    {
        string Codigo { get; set; }
        string Titulo { get; set; }
        string Autores { get; set; }
    }

    public class Livro : IObra
    {
        public string Codigo { get; set; }
        public string Titulo { get; set; }
        public string Autores { get; set; }
```

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



    public class Quadro : IObra
    {
        public string Codigo { get; set; }
        public string Titulo { get; set; }
        public string Autores { get; set; }
    }


    public class Musica
    {
        public string Titulo { get; set; }
        public string Autores { get; set; }
        public string Letra { get; set; }
    }

    public class ListaObras<T> where T : class, IObra, new()
    {
        List<T> itens;

        public ListaObras()
        {
            itens = new List<T>();
        }

        public void Adicionar(T item)
        {
            itens.Add(item);
        }

        public void Listar()
        {
            foreach (T item in itens)
            {
                Console.WriteLine($"Código: {item.Codigo}   Títul
o: {item.Titulo}   Autores: {item.Autores}");
            };
        }

    }
```

```
    public class Program
    {
        static void Main(string[] args)
        {
            ListaObras<Livro> livros = new ListaObras<Livro>();
            livros.Adicionar(new Livro() { Codigo = "121", Titulo
 = "Segredos do Visual Studio.NET", Autores = "Cláudio Ralha" });
            livros.Adicionar(new Livro() { Codigo = "122", Titulo
 = "Produtividade em C#", Autores = "Cláudio Ralha" });
            Console.WriteLine("Listando livros:\n");
            livros.Listar();
            ListaObras<Quadro> quadros = new ListaObras<Quadro>()
;
            quadros.Adicionar(new Quadro() { Codigo = "201", Titu
lo = "O Girassol", Autores = "Iara Ralha" });
            quadros.Adicionar(new Quadro() { Codigo = "202", Titu
lo = "A menina no balanço", Autores = "Iara Ralha e Cláudio Ralha
" });
            Console.WriteLine("\nListando quadros:\n");
            quadros.Listar();
            //Isto não compila
            //ListaObras<Musica> musicas = new ListaObras<Musica>
();
        }
    }
}
```

Observe que a classe `ListaObras<T>` foi definida assim:

```
public class ListaObras<T> where T : class, IObra, new(){

}
```

Isso significa que só serão aceitos os tipos que forem classes, que implementem a interface `IObra` e que possuam um construtor default. Observe no final do método `main` que existe uma linha de código comentada. Esta linha contém um erro que demonstra a restrição imposta pela interface.

Ao executarmos este exemplo com a linha comentada, obteremos a seguinte saída:

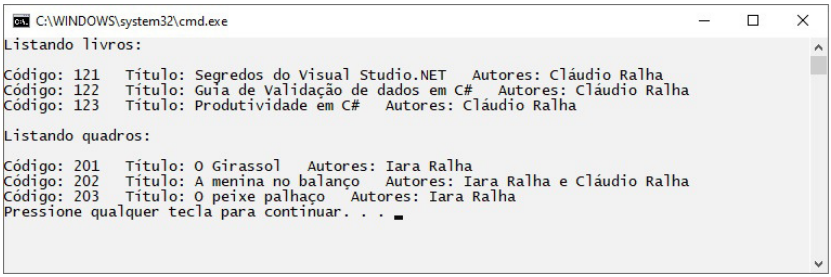


Figura 6.10: Utilizando uma interface em uma constraint de parâmetro de tipo

Descomente a linha comentada e veja que o código não é mais compilável. Isso ocorre porque a classe `Musica` não implementa a interface `IObra`.

6.6 UTILIZANDO TIPOS GENÉRICOS EXISTENTES NO FRAMEWORK

Nas seções anteriores, vimos um pouco da mecânica por trás da criação de tipos genéricos. Como se trata de um recurso de aplicação geral que evita retrabalho e maximiza o desempenho e a segurança de tipo, os próprios frameworks .NET e .NET Core já dispõem de um grande número de classes, estruturas e interfaces pré-construídas e testadas para tratar de classes e coleções genéricas nos namespaces `System.Collections.Generic` e `System.Collections.ObjectModel`. Estão disponíveis: `List<T>`, `Collection<T>`, `ReadOnlyCollection<T>`, `IEnumerable<T>`, `IEnumerator<T>`, `IList<T>`, `ICollection<T>`, `LinkedList<T>`, `Queue<T>`, `Stack<T>`, `SortedList<TKey,TValue>`, `Dictionary<TKey,TValue>`, `SortedDictionary<TKey,TValue>`, dentre outras.

Uma abordagem detalhada sobre cada um desses tipos foge ao escopo deste livro, mas você poderá obter exemplos de uso sobre os tipos desejados, usando como ponto de partida as seguintes páginas da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/standard/generics/collections

https://docs.microsoft.com/pt-br/dotnet/standard/collections/commonly-used-collection-types

https://docs.microsoft.com/pt-br/dotnet/standard/generics/interfaces

Reserve um tempo para aprofundar os seus conhecimentos em generics, pois esta é uma funcionalidade que vai lhe poupar horas de trabalho e muitas dores de cabeça. Para entender os tipos de problemas que os generics são capazes de resolver, recomendamos a leitura do seguinte artigo do mestre Macoratti, que também assina o prefácio deste livro:

https://imasters.com.br/back-end/c-3-motivos-importantes-para-usar-generics

Ainda que não seja possível condensar um assunto tão vasto em poucas páginas sem abrir mão de alguns detalhes, esperamos que você tenha percebido o quanto *generics* pode ser reutilizável e o quanto de trabalho ele é capaz de poupar. Lembre-se de que você poderá utilizá-lo para ir além de apenas criar classes, interfaces e

métodos genéricos, implementando coleções, estruturas e delegados genéricos. Trata-se de um recurso poderoso, elegante e simples de usar que precisa fazer parte do dia a dia do desenvolvedor e da desenvolvedora profissional.

Capítulo 7

LINQ

O LINQ (acrônimo de *Language Integrated Query*) é um modelo de manipulação de dados incluído originalmente no framework .NET 3.0 que foi projetado para fornecer aos programas a habilidade de selecionar dados da mesma forma como qualquer fonte de dados que implemente uma das seguintes interfaces: `IEnumerable`, `IEnumerable<T>` ou `IQueryable<T>`. Em termos práticos, isso significa que a partir de uma sintaxe única podemos criar expressões de consulta para retornar um conjunto de dados obtidos a partir de origens como arrays, listas, bancos de dados relacionais, documentos XML, planilhas do Excel, logs do Windows etc. Tudo isso sem perder a segurança de tipos!

Para suportar o LINQ, os frameworks .NET e .NET Core empregam vários recursos das linguagens C# e Visual Basic relacionados à produtividade, como inferência de tipos, tipos anônimos, métodos de extensão, expressões lambda, literais XML, tipos anuláveis, inicializadores de objetos e operador ternário `if`.

Os frameworks .NET e .NET Core oferecem provedores LINQ integrados para consulta nos seguintes tipos de dados:

- *LINQ para Objetos* - Usado para consultas sobre qualquer

tipo de objeto do C# na memória, como matrizes, listas e outros tipos de coleções.

- *LINQ para XML* - Usado para a criação e manipulação de documentos XML usando a sintaxe e mecanismo de consulta geral do LINQ.
- *LINQ para JSON* - Usado para a criação e manipulação de documentos JSON.
- *LINQ para Entidades* - Usado para consultas baseadas nos frameworks de mapeamento Objeto/Relacional Entity Framework e Entity Framework Core.
- *LINQ para SQL* - Usado para consultas a bancos de dados como alternativa ao LINQ para Entidades.
- *LINQ para Data Set* - Usado para consultas a bancos de dados baseadas no objeto `DataSet` , disponível desde a primeira versão do Framework .NET (apenas para sistemas legados).
- *LINQ Paralelo* - Estende o LINQ a Objetos com uma biblioteca de programação paralela que permite dividir uma consulta de modo que ela seja executada simultaneamente em múltiplos núcleos de um processador multicore.

Nas próximas páginas, apresentaremos como usar o LINQ para retornar um ou mais resultados (distintos ou não), demonstrando como efetuar filtragem, junções, uniões de duas fontes de dados e ordenação dos resultados. Focaremos apenas no LINQ para Objetos pela facilidade de construção de exemplos curtos que incluem as fontes de dados embutidas, mas tenha em mente que a sintaxe a ser empregada nos demais provedores LINQ é a mesma ou pelo menos similar.

Você verá que para realizar operações sobre as fontes de dados pode-se utilizar dois tipos de sintaxe: *consulta* (query) e *método* (também chamada de *high order functions* ou HOF). Ambas as alternativas são bastante produtivas, sendo a sintaxe de consulta mais legível e a sintaxe HOF, mais concisa.

Escolher entre uma sintaxe e outra é uma questão de preferência pessoal. Utilize a opção com a qual se sentir mais confortável ou então tire proveito de ambas, aplicando a opção mais adequada em cada caso. Ao longo da maior parte deste capítulo, trabalharemos com a sintaxe de consulta, por ser mais intuitiva e familiar a desenvolvedores com experiência prévia em linguagem SQL.

7.1 FILTRANDO DADOS USANDO LINQ

Para entender como as consultas LINQ funcionam, partiremos do exemplo a seguir onde utilizamos LINQ para efetuar uma consulta a uma fonte de dados representada por uma lista genérica de strings que retornará todos os nomes da lista que iniciam com a letra 'C':

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{

    public class Program
    {
        static void Main(string[] args)
        {
            List<string> nomes = new List<string>() { "Cláudio",
"Flávia", "Iara", "Cristiane", "Renan", "Vidal", "Lígia", "Ana",
```

```
"Branca", "Ceres", "Camila", "Elisa", "Karen", "Marco Aurélio", "
Francisco", "Carlos", "Lucas", "Ricardo", "Cláudia", "Michel", "G
ustavo", "Rafael" };
            var resultado = from n in nomes
                            where n.StartsWith("C")
                            select n;
            Console.WriteLine("Nomes que começam com a letra C:\n
);
            foreach (var item in resultado)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

Observe que a consulta que criamos usa uma sintaxe muito intuitiva e parecida com a adotada na linguagem SQL, voltada para bancos de dados relacionais. A mudança mais significativa é a ordem das cláusulas, que no LINQ difere iniciando pelo `from` em vez de `select`. Veja:

```
var resultado = from n in nomes
                where n.StartsWith("C")
                select n;
```

Confira na próxima imagem o resultado produzido por esse exemplo:

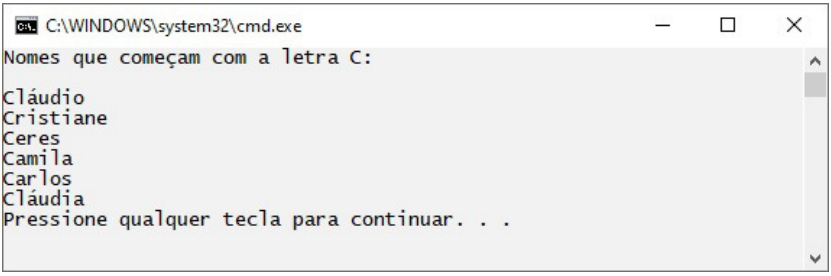


Figura 7.1: Filtrando a lista por nomes que iniciam com a letra C

Efetuar mudanças em uma consulta LINQ é uma tarefa simples, na maior parte dos casos. Para também incluir na consulta anterior nomes que começam com a letra 'K' (que possui em português som próximo ao da letra 'C'), basta alterá-la como mostrado a seguir:

```
var resultado = from n in nomes
                where n.StartsWith("C") || n.StartsWith("K")
                select n;
```

Para complicarmos um pouco, na próxima consulta desejamos obter a lista de nomes que iniciam com 'C' ou 'K' e possuem mais de 6 caracteres:

```
var resultado = from n in nomes
                where (n.StartsWith("C") || n.StartsWith("K")) &&
 n.Length > 6
                select n;
```

Observe que nesse caso usamos parênteses para garantir que os filtros sejam implementados como desejado e para aumentar a legibilidade.

O próximo exemplo ilustra como usar o operador `Contains` para executar uma busca *case-insensitive* em uma lista de nomes e retornar todos os elementos que contêm `ana`:

```
List<string> nomes = new List<string>() { "Flávia", "Ana Paula",
"Ana Cristina", "Ana Tereza", "Iara", "Cristiane", "Renata", "Júl
ia", "Luciana", "Juliana", "Giovanna" };
var resultado = from n in nomes
                where n.Contains("ana", StringComparison.OrdinalI
gnoreCase)
                select n;
```

O operador `Contains` é usado para checar se um elemento está contido em uma coleção ou não.

No exemplo a seguir, combinamos *LINQ para Objetos* com *expressões regulares* para obter a lista de nomes que possuem caracteres acentuados. Veja:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;

namespace ProdutividadeEmCSharp
{

    public class Program
    {
        static void Main(string[] args)
        {
            string padrao = @"[^a-zA-Z]";
            Regex regex = new Regex(padrao);

            List<string> nomes = new List<string>() { "Cláudio",
"Flávia", "Iara", "Cristiane", "Renan", "Vidal", "Lígia", "Ana",
"Branca", "Ceres", "Camila", "Elisa", "Karen", "Marco Aurélio", "
Francisco", "Carlos", "Lucas", "Ricardo", "Cláudia", "Michel", "G
ustavo", "Rafael" };
            var resultado = from n in nomes
                            where regex.IsMatch(n)
                            select n;

            Console.WriteLine("Nomes que possuem caracteres acent
uados:");
            foreach (var item in resultado)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

Na próxima imagem, temos a saída gerada por esse código:

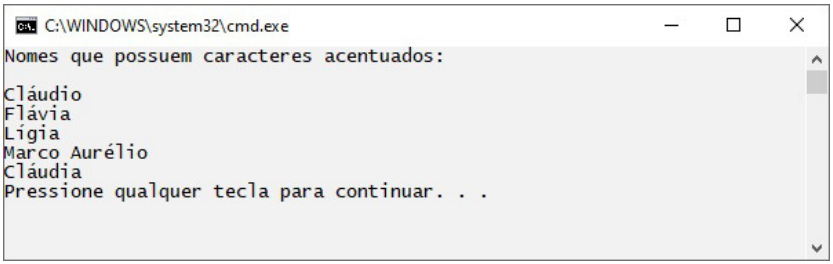


Figura 7.2: Filtrando a lista por nomes que contêm caracteres acentuados

O exemplo a seguir efetua uma consulta sobre uma lista de objetos do tipo `Pessoa`, retornando apenas as pessoas do sexo feminino:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Sexo { get; set; }
    }


    public class Program
    {
        static void Main(string[] args)
        {
            List<Pessoa> pessoas = new List<Pessoa>
                                        {
                                            new Pessoa() { Nome = "Fl
avia" , Idade = 38, Sexo = 'F' },
                                            new Pessoa() { Nome = "Cl
áudio" , Idade = 46, Sexo = 'M' },
                                            new Pessoa() { Nome = "Ia
ra" , Idade = 2, Sexo = 'F' },
                                            new Pessoa() { Nome = "Cr
```

```
istiane" , Idade = 43, Sexo = 'F' },
                                        new Pessoa() { Nome = "Re
nan" , Idade = 35, Sexo = 'M' }
                                    };

            var resultado = from p in pessoas
                            where p.Sexo == 'F'
                            select p;

            Console.WriteLine("Pessoas do sexo feminino:");
            foreach (var item in resultado)
            {
                Console.WriteLine(item.Nome);
            }
        }
    }
}
```

Observe que, na consulta anterior, obtemos como resultado uma lista de objetos do tipo `Pessoa`. Veja:

```
var resultado = from p in pessoas
                where p.Sexo == 'F'
                select p;

Console.WriteLine("Pessoas do sexo feminino:");
foreach (var item in resultado)
{
    Console.WriteLine(item.Nome);
}
```

Como só estamos usando a propriedade `Nome` no loop `foreach`, podemos modificar o trecho de código anterior conforme mostrado a seguir:

```
var resultado = from p in pessoas
                where p.Sexo == 'F'
                select p.Nome;

Console.WriteLine("Pessoas do sexo feminino:");
foreach (var item in resultado)
{
    Console.WriteLine(item);
```

```
}
```

Para obter apenas as pessoas do sexo feminino maiores de idade, bastaria incluir este critério na cláusula `where` conforme a consulta a seguir:

```
var resultado = from p in pessoas
                where p.Sexo == 'F' && p.Idade >=18
                select p.Nome;
```

7.2 EMPREGANDO OPERADORES DE ELEMENTOS COM RETORNO ÚNICO

O LINQ fornece operadores de elementos que retornam um único elemento ou um elemento específico de uma coleção. São eles: `Single` , `SingleOrDefault` , `First` , `FirstOrDefault` , `Last` e `LastOrDefault` .

Confira na lista a seguir a utilidade de cada um deles:

- `Single` - Retorna um único elemento específico de uma coleção de elementos se o elemento bate com os critérios especificados. Uma exceção é lançada se nenhuma ou mais de uma correspondência for encontrada para esse elemento na coleção.
- `SingleOrDefault` - Retorna um único elemento específico de uma coleção de elementos se o elemento bate com os critérios especificados. Uma exceção é lançada caso mais de uma correspondência seja encontrada para esse elemento na coleção. Um valor padrão é retornado se nenhuma correspondência for encontrada para esse elemento na coleção.
- `First` - Retorna o primeiro elemento de uma coleção de

elementos que atende aos critérios especificados, se um ou mais objetos forem encontrados. Uma exceção é lançada caso nenhuma correspondência seja encontrada para esse elemento na coleção.

- `FirstOrDefault` - Retorna o primeiro elemento de uma coleção de elementos que atende aos critérios especificados, se um ou mais objetos forem encontrados. Um valor padrão é retornado caso nenhuma correspondência seja encontrada para esse elemento na coleção.

No caso de dúvida sobre qual deles empregar, basta lembrar do seguinte:

- Se um conjunto de resultados contém mais de um elemento que atende aos critérios especificados e você quer que uma exceção seja lançada, use `Single` ou `SingleOrDefault`.

- Se um conjunto de resultados não contém nenhum registro que atenda aos critérios e você quer que um valor padrão seja devolvido, use `FirstOrDefault`.

Na maioria dos casos, a escolha mais adequada será `FirstOrDefault`, que executa mais rápido que `SingleOrDefault` e não lança exceção se não encontrar nenhum elemento que atenda aos critérios ou se encontrar mais de um elemento.

Para ilustrar o uso de `FirstOrDefault`, vamos alterar a última consulta apresentada na seção anterior. Veja:

```
var resultado = (from p in pessoas
                 where p.Sexo == 'F'
```

```
                select p).FirstOrDefault();

Console.WriteLine(resultado.Nome);
```

Veja que agora usamos parênteses para isolar a consulta original, antes de aplicarmos `FirstOrDefault` .

7.3 EMPREGANDO DISTINCT PARA OBTER RESULTADOS ÚNICOS

O método `Distinct` do LINQ é usado para retornar elementos distintos de uma fonte de dados. Empregá-lo sobre uma lista de tipos de valor ou sobre uma lista de strings é algo extremamente simples, conforme você pode observar nos exemplos a seguir:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{
    public class Program
    {
        static void Main(string[] args)
        {
            List<int> numeros = new List<int>()
            {
                1,2,3,4,5,6,3,4,5,2,1,6
            };

            var resultado = (from num in numeros
                      select num).Distinct();

            foreach (var item in resultado)
            {
                Console.WriteLine(item);
            }
```

```
        }
    }
}
```

Para casos como este, recomendamos utilizar a sintaxe de método (leia-se sintaxe HFO) no lugar da sintaxe de consulta que tornará o código mais legível. Compare:

```
var resultado = numeros.Distinct();
```

No próximo exemplo, demonstramos como usar um segundo overload do método `Distinct`, que recebe `IEqualityComparer` como argumento para os cenários em que precisamos fazer uma comparação *case-insensitive*. Veja:

```
using System;
using System.Linq;

namespace ProdutividadeEmCSharp
{
    public class Program
    {
        static void Main(string[] args)
        {
            string[] habilidades = { "Flexibilidade", "Autoconfia
nça", "Autoconhecimento", "Iniciativa", "Competitividade", "Visão
 no cliente",
                "Compreensão interpessoal", "Empatia", "Capacidad
e de liderança", "Persuasão", "Trabalho em Equipe", "Visão do neg
ócio", "empatia", "persuasão", "INICIATIVA" };
            var resultado = habilidades.Distinct(StringComparer.O
rdinalIgnoreCase);
            Console.WriteLine("Habilidades de um bom profissional
:\n");
            foreach (var item in resultado)
            {
                Console.WriteLine($"* {item}");
            }
        }
    }
}
```

Até aqui tudo funciona como esperado e não há nenhum trabalho extra a ser feito. As coisas começam a complicar quando surge a necessidade de aplicarmos `Distinct` a uma coleção de tipos complexos. O exemplo a seguir parece correto, mas vai falhar ao tentar retornar uma lista de pessoas distintas que participarão de uma viagem:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            List<Pessoa> pessoas = new List<Pessoa>()
            {
                new Pessoa(){Nome = "Pedro", Idade = 16, Genero =
'M'},
                new Pessoa(){Nome = "Paulo", Idade = 18, Genero =
'M'},
                new Pessoa(){Nome = "Marcela", Idade = 16, Genero
 = 'F'},
                new Pessoa(){Nome = "Roberta", Idade = 21, Genero
 = 'F'},
                new Pessoa(){Nome = "Ricardo", Idade = 19, Genero
 = 'M'},
                new Pessoa(){Nome = "Sofia", Idade = 14, Genero =
'F'},
                new Pessoa(){Nome = "Vanessa", Idade = 22, Genero
 = 'F'},
                new Pessoa(){Nome = "Rodrigo", Idade = 20, Genero
 = 'M'},
```

```
                new Pessoa(){Nome = "Rebeca", Idade = 25, Genero
= 'F'},
                new Pessoa(){Nome = "Henrique", Idade = 13, Gener
o = 'M'},
                new Pessoa(){Nome = "Pâmela", Idade = 21, Genero
= 'F'},
                new Pessoa(){Nome = "Alessandra", Idade = 19, Gen
ero = 'F' },
                new Pessoa(){Nome = "Pedro", Idade = 16, Genero =
'M'},
                new Pessoa(){Nome = "Vanessa", Idade = 22, Genero
 = 'F'}
            };
            var resultado = pessoas.Distinct();

            Console.WriteLine("Pessoas selecionadas para a viagem
:\n");
            foreach (var item in resultado)
            {
                Console.WriteLine($"* {item.Nome}");
            }
        }
    }
}
```

Confira o resultado na imagem a seguir:

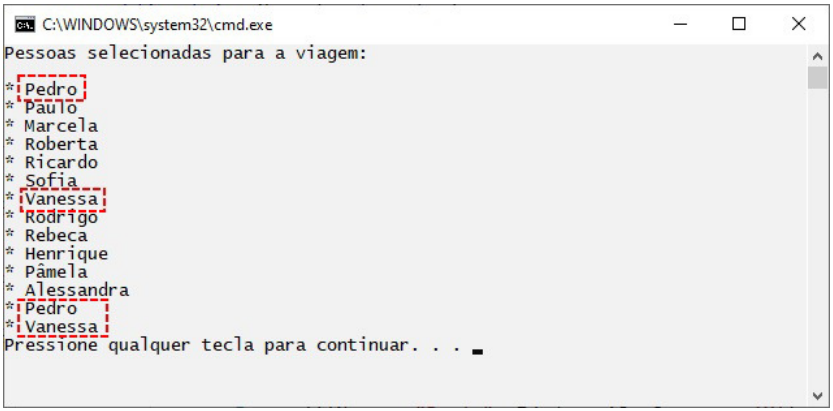


Figura 7.3: Método Distinct falhando ao retornar itens de uma coleção de um tipo complexo

Isso ocorre porque o comparador default checa apenas se as referências dos dois objetos são iguais, sem levar em consideração os valores das propriedades do tipo complexo. Há quatro soluções principais para este problema, a saber:

- Utilizar a versão sobrecarregada do método `Distinct` que usa a interface `IEqualityComparer` como argumento. Criar uma classe `PessoaComparer` que deverá implementar a interface `IEqualityComparer` (a interface contém os métodos `Equals` e `GetHashCode`). Feito isso, podemos criar uma instância da classe `PessoaComparer` e depois passar essa instância para o método `Distinct`.
- Substituir os métodos `Equals` e `GetHashCode` na classe `Pessoa`.
- Projetar as propriedades necessárias em um novo tipo anônimo que, por padrão, já substitui os métodos `Equals` e `GetHashCode`.
- Implementar a interface `IEquatable <T>`.

> Não apresentaremos aqui exemplos para as quatro abordagens, apenas a solução que envolve menos esforço do desenvolvedor, mas se desejar consultar uma comparação das quatro soluções, recomendamos a leitura do seguinte artigo:
>
> https://dotnettutorials.net/lesson/linq-distinct-method/

A abordagem que usaremos para resolver o problema, envolve a criação de um *tipo anônimo*. Para implementá-lo, basta substituir, no exemplo anterior, a linha a seguir:

```
var resultado = pessoas.Distinct();
```

Por esta:

```
var resultado = pessoas.Select(p => new { p.Nome, p.Idade, p.Gene
ro }).Distinct();
```

Execute o programa uma vez mais e veja que agora o resultado é o esperado. A abordagem que os desenvolvedores costumam usar com maior frequência é a que envolve substituir os métodos `Equals` e `GetHashCode` na classe `Pessoa`, o que resultaria no acréscimo de linhas de código. Perceba que a solução adotada não adicionou nenhuma linha extra ao código.

7.4 EMPREGANDO UNION PARA COMBINAR DUAS FONTES DE DADOS

O método `Union` do LINQ é usado para combinar as várias fontes de dados em uma fonte de dados única, removendo os elementos duplicados. Para ilustrar o seu uso, vamos partir de um exemplo que combina duas listas de tipos de valor:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{

    public class Program
    {
        static void Main(string[] args)
        {
            List<int> lista1 = new List<int>() { 1, 2, 3, 4, 5 };
            List<int> lista2 = new List<int>() { 3, 6, 2, 4, 7, 9
 };
            var resultado = lista1.Union(lista2).ToList();
```

```
            foreach (var item in resultado)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

Assim como ocorre com outros operadores do LINQ, como o `Distinct`, utilizar `Union` com tipos complexos requer a implementação de uma das soluções descritas na seção anterior.

O exemplo a seguir demonstra como usar tipos anônimos para combinar as duas listas (`pessoas1` e `pessoas2`) do tipo `Pessoa`:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            List<Pessoa> pessoas1 = new List<Pessoa>()
            {
                new Pessoa(){Nome = "Pedro", Idade = 16, Genero =
'M'},
                new Pessoa(){Nome = "Paulo", Idade = 18, Genero =
'M'},
                new Pessoa(){Nome = "Marcela", Idade = 16, Genero
 = 'F'},
                new Pessoa(){Nome = "Roberta", Idade = 21, Genero
```

```
 = 'F'},
                new Pessoa(){Nome = "Ricardo", Idade = 19, Genero
 = 'M'},
                new Pessoa(){Nome = "Sofia", Idade = 14, Genero =
'F'},
                new Pessoa(){Nome = "Vanessa", Idade = 22, Genero
 = 'F'},
                new Pessoa(){Nome = "Rodrigo", Idade = 20, Genero
 = 'M'},
                new Pessoa(){Nome = "Rebeca", Idade = 25, Genero
= 'F'},
            };

            List<Pessoa> pessoas2 = new List<Pessoa>()
            {
                new Pessoa(){Nome = "Henrique", Idade = 13, Gener
o = 'M'},
                new Pessoa(){Nome = "Pâmela", Idade = 21, Genero
= 'F'},
                new Pessoa(){Nome = "Alessandra", Idade = 19, Gen
ero = 'F' },
                new Pessoa(){Nome = "Pedro", Idade = 16, Genero =
'M'},
                new Pessoa(){Nome = "Vanessa", Idade = 22, Genero
 = 'F'}
            };
            var resultado = pessoas1.Select(p => new { p.Nome, p.
Idade, p.Genero }).Union(pessoas2.Select(p => new { p.Nome, p.Ida
de, p.Genero })).ToList();

            Console.WriteLine("Pessoas selecionadas para a viagem
:\n");
            foreach (var item in resultado)
            {
                Console.WriteLine($"* {item.Nome}");
            }
        }
    }
}
```

Para os cenários em que você precisa juntar duas fontes de dados em uma sem remover itens duplicados, utilize o método `Concat` em vez de `Union`.

7.5 EMPREGANDO DIFERENTES TIPOS DE JUNÇÕES

As associações suportadas pelo LINQ são semelhantes às *junções* (em inglês, *joins*) do SQL. Se você já está familiarizado com bancos de dados relacionais, não terá dificuldade em entender como construir as consultas equivalentes no LINQ. Para a grande maioria dos desenvolvedores e desenvolvedoras, o principal problema é recordar a sintaxe adotada, que não é muito intuitiva.

Uma junção é um tipo de consulta que mescla os dados de duas ou mais *fontes de dados* (objetos ou tabelas) em um único conjunto de resultados com base em alguma propriedade comum e permite também que sejam aplicadas algumas condições, se necessário.

O LINQ disponibiliza dois métodos para executar operações de junção. São eles:

- `Join` - Operador usado para unir duas fontes de dados com base na propriedade comum e retorna os dados como um único conjunto de resultados.
- `GroupJoin` - Operador usado para unir duas fontes de dados com base em uma chave ou propriedade comum, mas retorna o resultado como um grupo de sequências.

Existem quatro tipos de junções a serem consideradas quando trabalhamos com LINQ. Confira na imagem a seguir:

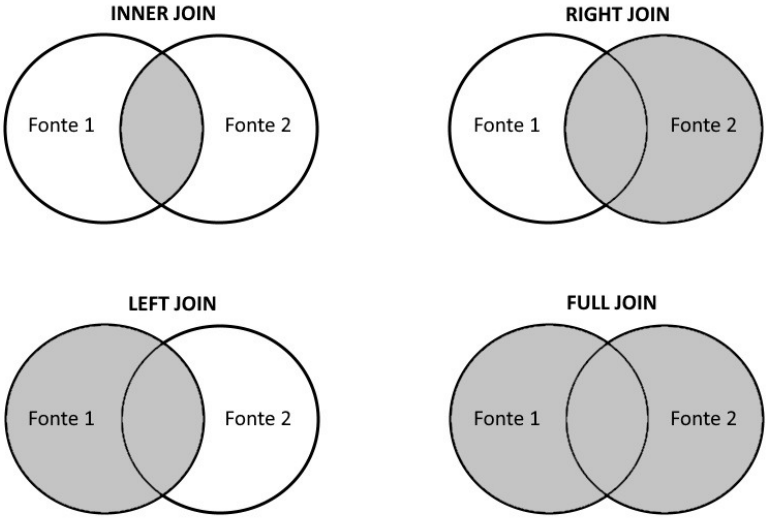


Figura 7.4: Tipos de junções suportadas pelo LINQ

Nas próximas subseções, veremos cada uma dessas junções suportadas, apresentando exemplos com a sintaxe de consulta que é muito mais intuitiva e fácil de ler que a sintaxe de método.

Junção interna (Inner Join)

A primeira junção que aprendemos ao estudar LINQ ou SQL é a *junção interna* ou *Inner Join*. Esse tipo de junção traz apenas a intersecção entre as duas fontes de dados, ou seja, retorna apenas os elementos correspondentes de ambas as fontes de dados.

No exemplo a seguir, a consulta trará os empregados que estão associados a departamentos existentes em uma empresa:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{

    public class Empregado
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Sexo { get; set; }
        public int DepartamentoId { get; set; }
    }

    public class Departamento
    {
        public int Id { get; set; }
        public string Nome { get; set; }
        public string Localizacao { get; set; }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            List<Empregado> empregados = new List<Empregado>
                                        {
                                            new Empregado() { Nome =
"Ramon" , Idade = 38, Sexo = 'M', DepartamentoId = 1 },
                                            new Empregado() { Nome =
"Cláudio" , Idade = 46, Sexo = 'M', DepartamentoId = 1 },
                                            new Empregado() { Nome =
"Leonardo" , Idade = 25, Sexo = 'M', DepartamentoId = 4 },
                                            new Empregado() { Nome =
"Luana" , Idade = 22, Sexo = 'F' , DepartamentoId = 3},
                                            new Empregado() { Nome =
"Fernanda" , Idade = 35, Sexo = 'F' , DepartamentoId = 0},
                                            new Empregado() { Nome =
"Natália" , Idade = 25, Sexo = 'F', DepartamentoId = 2 }
                                        };

            List<Departamento> departamentos = new List<Departame
nto>
                                        {
                                            new Departamento() { Id=1
```

```
, Nome = "TI" , Localizacao = "São Paulo" },
                                        new Departamento() { Id=2
, Nome = "RH" , Localizacao = "São Paulo" },
                                        new Departamento() { Id=3
, Nome = "Comercial" , Localizacao = "Alphaville" },
                                        new Departamento() { Id=4
, Nome = "Financeiro" , Localizacao = "Alphaville" },
                                        new Departamento() { Id=5
, Nome = "Marketing" , Localizacao = "Alphaville" }
                                    };

            var resultado = from empregado in empregados
                            join departamento in departamentos
                            on empregado.DepartamentoId equals de
partamento.Id
                            select new
                            {
                                Nome = empregado.Nome,
                                Departamento = departamento.Nome,
                                Localizacao = departamento.Locali
zacao
                            };

            Console.WriteLine("Exemplo de INNER JOIN:\n");

            foreach (var item in resultado)
            {
                Console.WriteLine($"{item.Nome} - {item.Departame
nto} - {item.Localizacao}");
            }
        }
    }
}
```

Ao examinar a consulta LINQ, note que a coluna ou propriedade que relaciona as duas fontes de dados é especificada na cláusula `on`. Neste exemplo, ela tem nomes diferentes nas duas entidades (`DepartamentoId` em `Empregado` e `Id` em `Departamento`):

```
var resultado = from empregado in empregados
                            join departamento in departamentos
```

```
                        on empregado.DepartamentoId equals de
partamento.Id
                        select new
                        {
                            Nome = empregado.Nome,
                            Departamento = departamento.Nome,
                            Localizacao = departamento.Locali
zacao
                        };
```

Confira a saída produzida por esta consulta na imagem a seguir:

Figura 7.5: Exemplo de junção interna

Observe que os elementos não correspondentes entre as duas fontes de dados foram removidos do conjunto de resultados (em nosso exemplo, apenas a Fernanda foi descartada). Note também que a consulta retorna uma coleção de um *tipo anônimo* que possui apenas os campos necessários (`Nome` , `Departamento` e `Localizacao`).

Junção externa esquerda (Left Outer Join)

A *junção externa esquerda* (em inglês, *left outer join*) ou simplesmente *junção esquerda* é uma junção na qual cada dado da primeira fonte de dados será retornado, independentemente de ter ou não dados correlatos presentes na segunda fonte de dados. Nos

casos em que não houver dados correspondentes na fonte de dados esquerda, serão necessários valores nulos para a segunda fonte de dados.

A implementação de uma junção externa esquerda é feita em duas etapas. Na primeira, é necessário executar uma junção interna usando uma junção de grupo com a palavra-chave `into`, deste modo:

```
var resultado = from empregado in empregados
                join departamento in departamentos
                on empregado.DepartamentoId equals departamento.I
d
                into temp
```

Na segunda etapa, precisamos incluir cada elemento da primeira fonte de dados (situada à esquerda) no conjunto de resultados, independentemente de esse elemento não ter correspondências na segunda fonte de dados (à direita). Para isso, precisamos chamar o método `DefaultIfEmpty` em cada sequência de elementos correspondentes da associação ao grupo. Veja:

```
var resultado = from empregado in empregados
                join departamento in departamentos
                on empregado.DepartamentoId equals departamento.I
d
                into temp
                from dx in temp.DefaultIfEmpty()
                select new
                {
                    Nome = empregado.Nome,
                    Departamento = (dx != null) ? dx.Nome : "Não
Informado",
                    Localizacao = (dx != null) ? dx.Localizacao :
"Não Informado"
                };

Console.WriteLine("Exemplo de LEFT OUTER JOIN:\n");
```

Ao analisar essa consulta, lembre-se de que o valor padrão para um tipo de referência é nulo. Consequentemente, é necessário verificar a referência nula antes de acessar cada elemento da coleção de departamentos. Aproveitamos a necessidade do teste para incluir a mensagem `Não Informado` em vez de simplesmente exibir `null`. Confira o resultado na imagem a seguir:

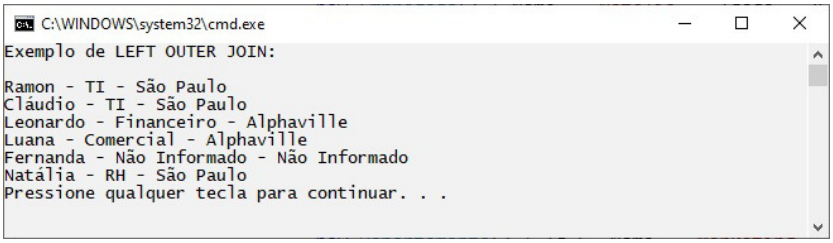


Figura 7.6: Exemplo de junção externa esquerda

Conforme você deve ter percebido, implementar uma *junção externa esquerda* em LINQ ou SQL não é algo tão trivial e intuitivo como gostaríamos. A boa notícia é que, para implementar a *junção externa direita* (em inglês, *right outer join*), basta trocar a ordem das fontes de dados.

Junção cruzada (Cross Join)

Ao combinarmos duas fontes de dados usando a *junção cruzada* (em inglês, *cross join*), cada elemento da primeira fonte de dados será mapeado com cada elemento da segunda fonte de dados. O resultado é um *produto cartesiano* das fontes de dados envolvidas na junção.

A junção cruzada é mais simples de montar do que a junção

interna e a junção externa esquerda, pois não utiliza a palavra-chave on para especificar uma propriedade comum que não é usada neste tipo de junção e também não há filtragem de dados. Veja a seguir um exemplo:

```
using System;
using System.Linq;

namespace ProdutividadeEmCSharp
{
    public class Program
    {
        static void Main(string[] args)
        {
            char[] vogais = "AEIOU".ToCharArray();
            char[] digitos = "123456789".ToCharArray();

            var posicoes =
                from v in vogais
                from d in digitos
                select v.ToString() + d;

            foreach (var posicao in posicoes)
            {
                Console.Write($"{posicao} ");
                if (posicao.EndsWith("9")) Console.WriteLine();
            }
        }
    }
}
```

Confira a saída produzida por este código na próxima imagem:

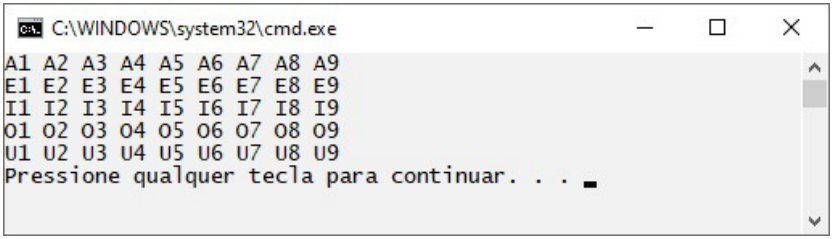


Figura 7.7: Exemplo de junção cruzada

Vale destacar que, neste tipo de junção, o número total de elementos na sequência resultante será o produto das duas fontes de dados envolvidas na junção (5 vogais x 10 dígitos = 50 elementos).

7.6 ORDENANDO O RESULTADO DAS CONSULTAS

Ao criarmos consultas LINQ, muitas vezes desejamos que os elementos sejam retornados em uma ordem ascendente ou descendente. Para ordenar elementos de um tipo complexo de forma ascendente, basta adicionar na consulta a cláusula `orderby` antes da cláusula `select`, especificando o nome da propriedade usada para fins de ordenação:

```
var resultado = from p in pessoas
                where p.Sexo == 'F'
                orderby p.Nome
                select p.Nome;
```

Confira:

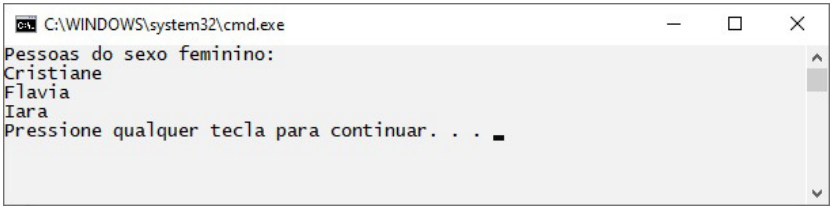


Figura 7.8: Ordenando os resultados de maneira ascendente

Caso deseje ordenar o resultado em ordem descendente, utilize a consulta a seguir:

```
var resultado = from p in pessoas
```

```
                where p.Sexo == 'F'
                orderby p.Nome descending
                select p.Nome;
```

Para aplicar a ordenação descendente no exemplo em que empregamos uma expressão regular sobre uma lista de strings, bastaria alterar a consulta como mostrado a seguir:

```
var resultado = from n in nomes
                where regex.IsMatch(n)
                orderby n descending
                select n;
```

Muito fácil, não é mesmo? E o melhor de tudo é que para ordenar mais de uma coluna usando a sintaxe de consulta é igualmente simples. No exemplo a seguir, utilizamos como primeiro critério de ordenação o sexo e como segundo, a idade:

```
var resultado = from p in pessoas
                orderby p.Sexo,p.Idade
                select p;
```

Note que nesta consulta não foi necessário especificar uma cláusula `where`. Para conferir o conjunto de resultados retornados por ela, altere o trecho final do exemplo anterior:

```
Console.WriteLine("Pessoas ordenadas por sexo e idade:\n");
foreach (var item in resultado)
{
    Console.WriteLine($"{item.Nome} - {item.Sexo} - {item.Idade}"
);
}
```

Executando uma vez mais o programa, você obterá a seguinte saída:

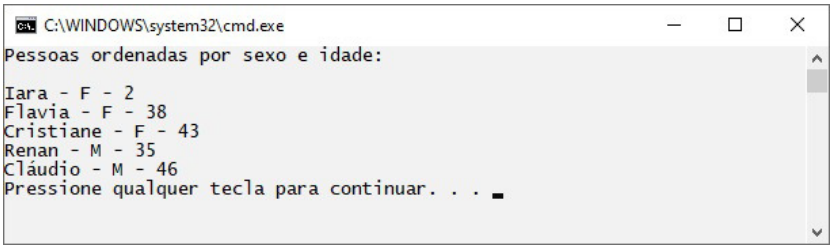


Figura 7.9: Ordenando de maneira ascendente os resultados por mais de um critério

Para reescrever a consulta anterior usando a sintaxe de métodos, teríamos que usar um operador `OrderBy` ou `OrderByDescending` para especificar a primeira propriedade de ordenação e um operador `ThenBy` para especificar a segunda propriedade de ordenação, sendo possível acrescentar mais chamadas ao método `ThenBy` para propriedades extras. Veja:

```
var resultado = pessoas.OrderBy(p => p.Sexo).ThenBy(p => p.Idade)
;
```

É importante destacar que LINQ é um assunto muito vasto que preencheria facilmente um livro inteiro mais extenso do que este que você está lendo agora. Ao longo deste capítulo, limitamo-nos a apresentar alguns recursos úteis disponíveis neste modelo de programação para despertar a curiosidade dos leitores e leitoras que ainda não vivenciaram a experiência de trabalhar com LINQ em seus projetos.

Para aqueles que utilizam LINQ de forma extensiva, uma ferramenta que não pode faltar é o LINQPad. O LINQPad 6 é compatível com as novidades introduzidas no C# 8 e no .NET Core 3.x e inclui vários templates de código prontos para fins de estudo. Você pode baixar a versão gratuita do programa, acessando:

https://www.linqpad.net/Download.aspx

Até o momento em que este livro estava sendo finalizado (outubro de 2020), ainda não havia nenhuma versão deste software compatível com o .NET 5.0.

Para saber mais sobre a criação de consultas LINQ, acesse a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries

Capítulo 8

EXCEÇÕES

As *exceções* ou *exceptions* são mecanismos primários fornecidos por linguagens, como C#, Visual Basic e Java para comunicar ao sistema ou script condições de erros ocorridos dentro do código, como a indisponibilidade de acesso à internet ou a uma API, incapacidade de conectar ao banco de dados com as credenciais passadas, erros de leitura ou escrita em um arquivo em uma pasta, efetuar a divisão de um número por zero etc.

Para tratar exceções, usamos estruturas `try-catch` e `try-catch-finally` , nas quais é frequente a presença de múltiplos blocos `catch` , voltados para diferentes tipos de exceções, e um bloco genérico disposto ao final, com o objetivo de capturar exceções que não foram capturadas pelos blocos anteriores mais especializados.

Para facilitar o nosso trabalho, o Visual Studio fornece a janela *Configurações de Exceção*, que vamos utilizar para interromper a execução quando um tipo específico de exceção for gerado. Você verá que é possível definir condições e que a interrupção da execução ocorre tanto com exceções tratadas quanto com as não tratadas. Com a execução pausada poderemos examinar a exceção lançada usando ferramentas como a *janela de Inspeção* ou *janela de Inspeção Rápida*, que serão detalhadas no capítulo sobre

depuração.

Uma exceção contém informações valiosas para o desenvolvedor, como a mensagem de erro, mantida na propriedade `Message`, e a pilha de chamadas, disponível na propriedade `StackTrace`. Além disso, pode conter ou não outra exceção, motivo pelo qual devemos sempre inspecionar a sua propriedade `InnerException`. Por meio dessa propriedade, temos acesso à exceção pai que causou a interrupção no fluxo normal de execução.

O C# nos permite lançar de forma explícita uma exceção usando a instrução `throw` ou relançar uma exceção capturada em um bloco `catch` após executar algum tratamento local usando a mesma instrução `throw`.

Para localizar a origem dos erros, veremos como rastrear o fluxo de execução inspecionando a janela *Pilha de Chamadas*, o que nos permitirá encontrar exceções não tratadas tanto no código do usuário quanto de terceiros ou do próprio framework.

A partir do C# 6.0, a linguagem passou a oferecer suporte a algumas melhorias no tratamento de exceções que facilitam a vida do desenvolvedor e que serão detalhadas neste capítulo. Nas próximas seções, você aprenderá como empregar *filtros de exceções* (inclusive na criação de log de erros) e como efetuar o tratamento de exceções em *métodos assíncronos*. Além disso, entenderá como relançar exceções sem perder o conteúdo da pilha de chamadas, um erro frequente entre os desenvolvedores, e descobrirá que podemos acessar informações sobre a última exceção lançada consultando a pseudovariável `$exception`.

Veremos que é possível estender a classe `Exception` do namespace `System` (a classe base de todas as exceções em C#), e criar exceções contendo propriedades e métodos extras adequados às suas necessidades. O capítulo termina mostrando como gerar, em poucos segundos, o esqueleto de uma classe de exceção personalizada usando o recurso *Gerar a partir do uso*.

8.1 CRIANDO BLOCOS DE TRATAMENTO DE EXCEÇÕES USANDO CODE SNIPPETS

O editor de código do Visual Studio dispõe de vários atalhos (em inglês, *code snippets*) para a inserção de blocos de tratamento de exceções e para a definição de uma classe que herde de `Exception`. Confira os atalhos na tabela a seguir:

Atalho	Descrição	Locais válidos para inserir o snippet
`exception`	Cria uma declaração para uma classe que deriva de uma exceção (`Exception` por padrão).	Dentro de um namespace (incluindo o namespace global), uma classe ou uma estrutura.
`try`	Cria um bloco `try-catch`.	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.
`tryf`	Cria um bloco `try-finally`.	Dentro de um método, um indexador, um acessador de propriedade ou um acessador de evento.

Descobrindo rapidamente quais tipos de exceções cada classe do framework pode disparar

Uma forma simples de descobrir que tipos de exceções cada

classe do framework .NET/.NET Core pode disparar é parar com o mouse sobre o nome da classe no editor de código.

Para testar, entre com o código de exemplo a seguir:

```
using System.IO;

namespace ProdutividadeEmCSharp
{
    class Program
    {

        static void Main(string[] args)
        {
            StreamReader sr = new StreamReader("produtividade.txt
);
        }
    }
}
```

Com o código já disponível, pare com o mouse sobre a classe `StreamReader` e inspecione as informações mostradas na tooltip. Confira na imagem a seguir:

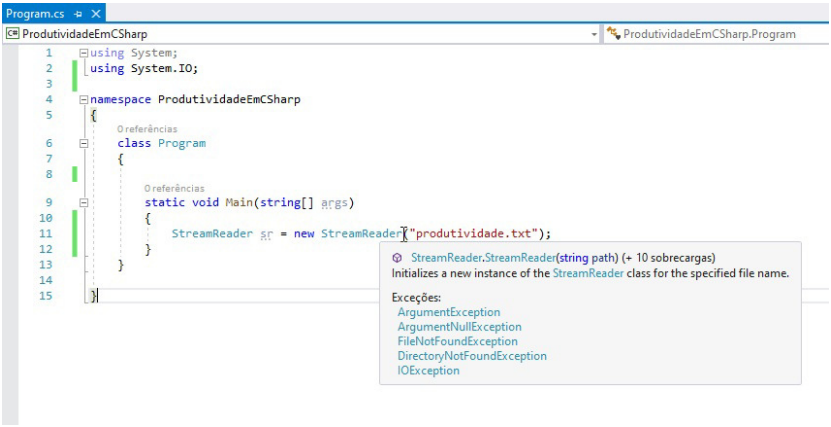


Figura 8.1: Descobrindo os tipos de exceção que podem ser disparados por um classe

E o melhor de tudo é que os nomes das classes de exceção listados são links para a documentação da classe gerada a partir de metadados, ou seja, basta um clique em `FileNotFoundException` para inspecionarmos os membros da classe. Veja:

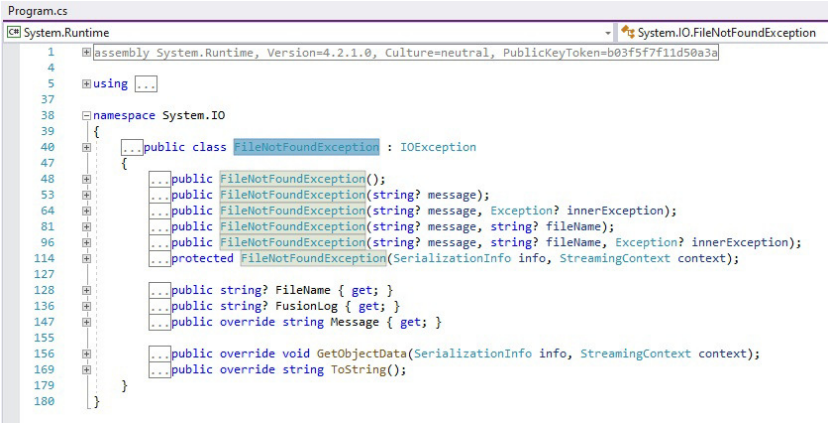


Figura 8.2: Inspecionando os membros da classe FileNotFoundException

8.2 TRATANDO EXCEÇÕES USANDO FILTROS DE EXCEÇÕES

Um *filtro de exceção* é um recurso introduzido no C# 6.0 que fornece muita flexibilidade para os desenvolvedores lidarem com as exceções. Durante a captura das exceções, eles fornecem instruções condicionais que retornam `true` ou `false` ao bloco `catch`. Veja a seguir um exemplo:

```
try
{
    //Código que dispara a exception
}
catch (Exception e) when (e.InnerException != null)
{
   //Código que trata a exception
```

```
}
finally
{

}
```

Conforme você pode observar, uma instrução `when` foi adicionada à direita do bloco de exceção. Isso significa que o bloco de código só será executado se a condição especificada for verdadeira.

É possível especificar vários níveis de bloco `catch` com condições diferentes e assim implementar uma abordagem de *fallback*. O próximo fragmento de código ilustra este tipo de construção mais complexa:

```
try
{
    //Código que dispara a exception
}
catch (InvalidOperationException ioe) when (e.InnerException != n
ull)
{
    //Código que trata a exception
}
catch (FieldAccessException fae) when (e.InnerException != null)
{
    //Código que trata a exception
}
catch(Exception exp)
{
    //Código que trata a exception
}
finally
{
}
```

Confira na próxima listagem um exemplo executável de uso da filtragem de exceção:

```
using System;
```

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Disparando a exceção...");
                throw new ArgumentNullException(paramName: "idade
);
            }
            catch (ArgumentNullException e) when (e.ParamName ==
"nome")
            {
                Console.WriteLine("Capturando a exceção dentro do
 bloco que contém o filtro nome...");
            }

        }
    }
}
```

Ao executar esse exemplo, veremos que a exceção continua sendo disparada, pois forjamos no bloco `Try` uma exceção do tipo `ArgumentNullException` que teria sido disparada por um parâmetro fictício `idade`, e o filtro que criamos só é ativado quando o parâmetro é `nome`. Veja:

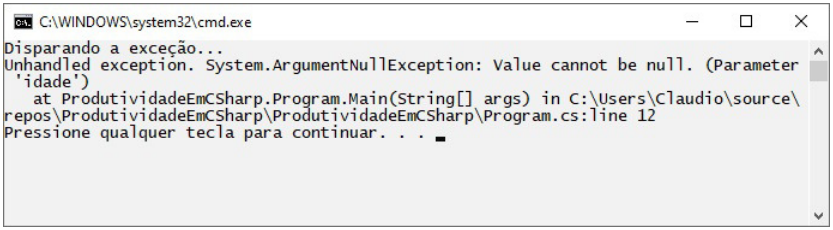


Figura 8.3: Falha no uso da filtragem de exceção

Para lidar com o problema que criamos, vamos fornecer um

mecanismo de *fallback*, incluindo um bloco `catch` extra que será disparado caso nenhum `case` anterior no fluxo seja ativado:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Disparando a exceção...");
                throw new ArgumentNullException(paramName: "idade
");
            }
            catch (ArgumentNullException e) when (e.ParamName ==
"nome")
            {
                Console.WriteLine("Capturando a exceção dentro do
 bloco que contém o filtro nome...");
            }
            catch (Exception e)
            {
                Console.WriteLine("Capturando a exceção dentro do
 bloco catch sem filtro.");
            }
        }
    }
}
```

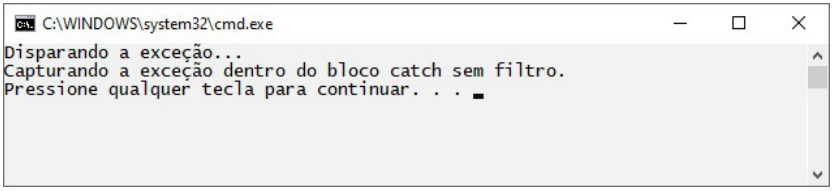


Figura 8.4: Utilizando um mecanismo de fallback em conjunto com a filtragem de exceção

Agora que já temos o programa de teste capturando todas as

exceções, vamos alterar o filtro para que possamos vê-lo em funcionamento. Altere o nome do parâmetro de `nome` para `idade`:

```
catch (ArgumentNullException e) when (e.ParamName == "idade")
{
    Console.WriteLine("Capturando a exceção dentro do bloco que c
ontém o filtro idade...");
}
```

Execute uma vez mais o exemplo e confira que o filtro funciona como esperado:

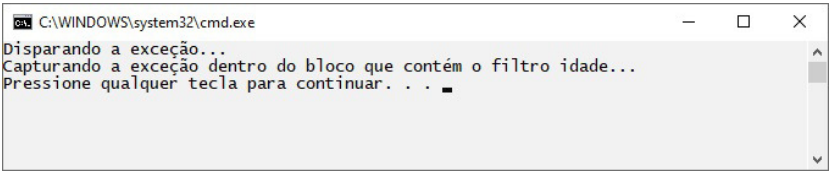


Figura 8.5: Capturando com sucesso uma exceção usando filtragem de exceção

8.3 EFETUANDO LOG DE ERROS USANDO FILTROS DE EXCEÇÕES

A funcionalidade *filtro de exceção* pode ser usada para implementar facilmente um mecanismo de log, uma vez que permite ter acesso à pilha de chamadas sem destruí-la. Para tanto, basta que o filtro criado retorne `false`, como no exemplo a seguir, no qual codificamos uma função log muito simples, apenas para exemplificar a ideia:

```
using System;
using System.Diagnostics;

namespace ProdutividadeEmCSharp
{
    class Program
```

```
    {
        static bool Log(Exception ex, string message, params obje
ct[] args)
        {
            Debug.Print(message, args);
            return false;
        }

        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Disparando a exceção...");
                throw new ArgumentNullException(paramName: "idade
");
            }
            catch (ArgumentNullException ex) when (Log(ex, "Parâm
etro não pode ser nulo!"))
            {
                Console.WriteLine("Capturando a exceção dentro do
 bloco ArgumentNullException...");
            }

            catch (Exception)
            {
                Console.WriteLine("Capturando a exceção dentro do
 bloco Exception...");
            }

        }
    }

}
```

Este exemplo utiliza o método `Print` da classe `Debug` para escrever uma mensagem na janela *Saída* (em inglês, *Output*) do Visual Studio quando o programa é executado em modo depuração, além de escrever na janela de console. Para exibir a janela *Saída*, clique no menu *Depurar*, no submenu *Janelas* e, a seguir, na opção *Saída*. Veja:

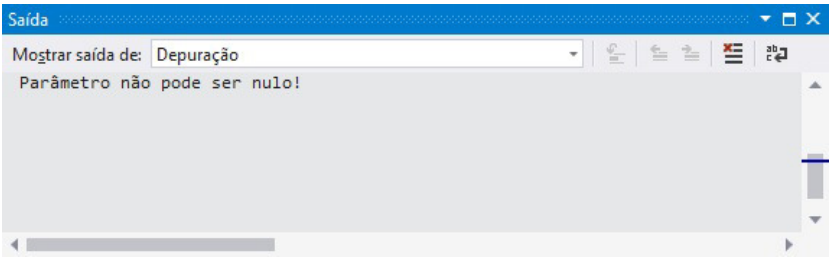


Figura 8.6: Inspecionando a mensagem de log na janela de Saída do Visual Studio

Para saber mais sobre o uso dos filtros de exceções em C#, acesse a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-9#exception-filters

8.4 INSPECIONANDO A PILHA DE CHAMADAS

A janela *Pilha de Chamadas* (em inglês, *Call Stack*) é usada para nos mostrar a pilha de chamadas de métodos e só está disponível durante a sessão de depuração.

Na prática, isso significa que devemos primeiro setar um ponto de interrupção, executar o programa em modo debug e, ao atingi-lo, carregar a janela *Pilha de Chamadas*. Para exibir a janela, clique no menu *Depurar*, selecione o submenu *Janelas* e, a seguir, a opção *Pilha de Chamadas* ou simplesmente tecle *Ctrl* + *Alt* + *c*.

Para ilustrar o uso desta ferramenta, vamos partir do código do

exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void Metodo2(string mensagem)
        {
            Console.WriteLine(Metodo1(mensagem));
        }

        static string Metodo1(string mensagem)
        {
            return $"Mensagem: {mensagem}\nTamanho: {mensagem.Len
gth}";
        }

        static void Main(string[] args)
        {
            Metodo2("Produtividade em C#");
            Metodo2(null);
        }
    }
}
```

Antes de executar o programa, marque um ponto de interrupção na primeira chamada ao `Metodo2` no método `Main`. Execute o programa em modo de depuração e exiba a janela conforme explicado anteriormente. Ao executar a segunda chamada ao `Metodo2`, uma exceção será lançada e será possível inspecionar a pilha. Veja na imagem a seguir:

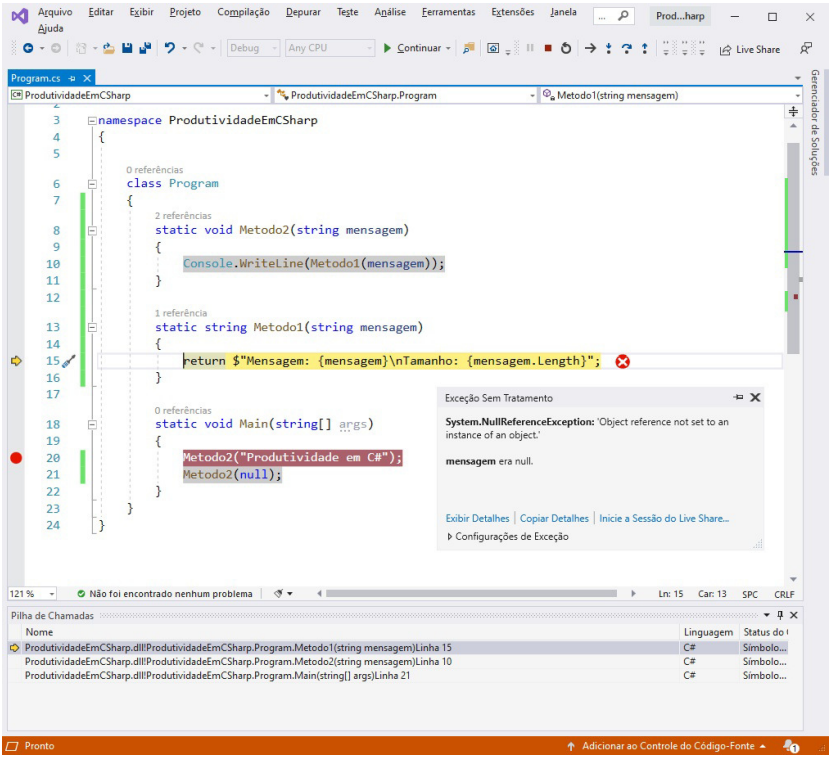


Figura 8.7: Exibindo a janela Pilha de Chamadas

Ao inspecionar a janela *Pilha de Chamadas*, considere que o rastreamento de pilha começa no método do nosso código. Nele a exceção é lançada (sinalizado pela seta amarela) e termina no método que contém o bloco `try...catch` que captura a exceção, caso exista, ou no método `main`, que é o ponto de entrada da nossa aplicação de console, no caso de uma exceção não tratada. Observe que a janela mostra em destaque a linha de execução atual e informa, sempre que possível, a linguagem usada para escrever cada método. Confira:

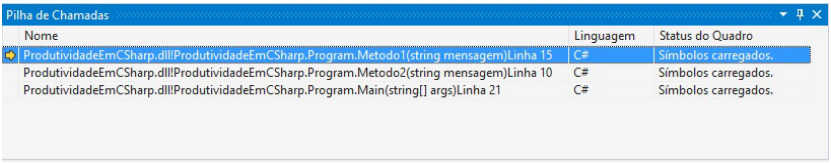


Figura 8.8: Inspecionando a ordem de chamada dos métodos na janela Pilha de Chamadas

Para visualizar os valores passados para os parâmetros de cada método, clique com o botão direito sobre um dos itens listados e selecione no menu de contexto a opção *Mostrar Valores de Parâmetro*. Veja:

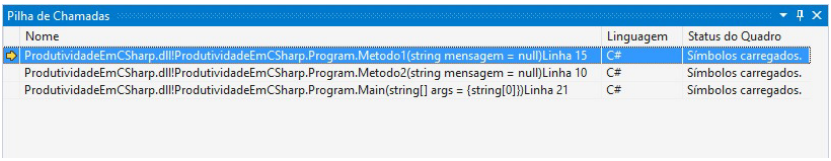


Figura 8.9: Inspecionando os valores passados para os métodos

Por meio dessa janela, podemos também adicionar, desabilitar e remover pontos de interrupção simplesmente clicando sobre o procedimento desejado e selecionando no menu de contexto as opções do submenu *Ponto de Interrupção*. Note que você pode efetuar um duplo clique sobre o nome de um método do seu projeto listado nesta janela para ver o código-fonte.

A janela *Pilha de Chamadas* é mais poderosa do que pode parecer em um primeiro momento, pois nos permite inspecionar não apenas a pilha de chamadas envolvendo o código que desenvolvemos, como também as chamadas às bibliotecas de terceiros e aos métodos do próprio framework, nos quais exceções também podem ocorrer. Mais adiante neste capítulo, explicaremos como habilitar a visualização desse tipo de rastreamento.

8.5 PRESERVANDO A PILHA DE CHAMADAS

Um detalhe importante a ser lembrado ao relançar uma exceção sem adicionar informações adicionais, é o de usar:

```
throw;
```

No lugar de:

```
throw ex;
```

Uma instrução `throw` vazia em um bloco `catch` emitirá um IL específico que lança novamente a exceção enquanto preserva o rastreio da pilha original (*stack trace*). Já uma instrução `throw ex` perderá o rastreamento de pilha para a fonte original da exceção e conterá apenas as informações que você especificar.

Para testar, execute o programa a seguir:

```
using System;
using System.Diagnostics;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void GerarException()
        {
            Console.WriteLine("Disparando a exceção...");
            throw new ArgumentNullException(paramName: "idade");
        }

        static void Main(string[] args)
        {
            try
            {
                GerarException();
            }
            catch (Exception ex)
            {
```

```
                throw;
                //throw ex;
            }

        }
    }

}
```

Utilizando a instrução `throw` vazia, obteremos a saída a seguir:

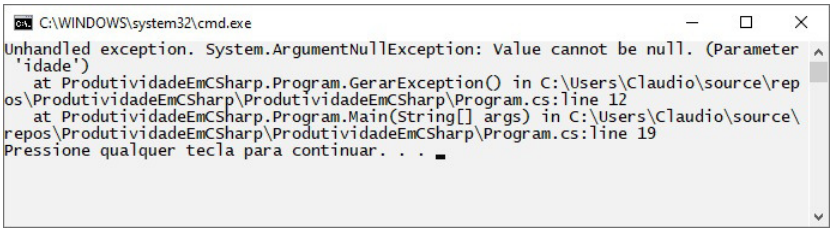


Figura 8.10: Utilizando a instrução throw

Repetindo o teste usando a instrução `throw ex` , obteremos a saída mostrada na próxima janela. Compare:

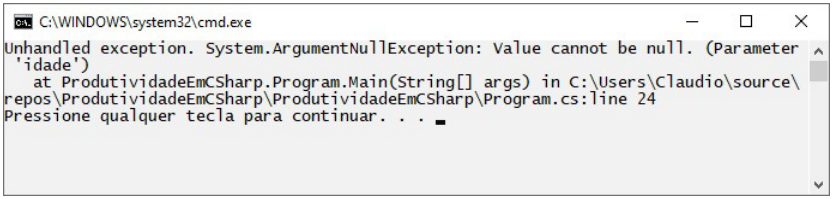


Figura 8.11: Utilizando a instrução throw ex

8.6 UTILIZANDO THROW EM CONTEXTOS QUE REQUEREM UMA EXPRESSÃO

Concebida originalmente para ser uma declaração, a palavra-

chave `throw` não podia ser usada nas primeiras versões da linguagem C# em contextos que exigiam uma expressão, como dentro de uma declaração ternária. Com o lançamento do C# 7.0, o compilador passou a permitir que o `throw` também seja usado como expressão.

Isso possibilita que utilizemos `throw` em uma expressão *null-coalescing* como a mostrada no exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        private string _nome;
        public string Nome
        {
            get => _nome;
            set => _nome = value ?? throw new ArgumentNullExcepti
on(null, $"O valor da propriedade {nameof(Nome)} não pode ser nul
o.");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Pessoa pessoa = new Pessoa();
                pessoa.Nome = null;
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Erro: {ex.Message}");
            }

        }
    }
}
```

Conforme vimos no capítulo sobre operadores, o operador de coalescência nula `??` (em inglês,*null-coalescing operator*) retornará o valor do operando esquerdo se não for null, caso contrário, ele avaliará o operando direito e retornará seu resultado.

Também podemos usar `throw` em um construtor *Expression–Bodied* (em português, *corpo de expressão*) para disparar a exceção quando o construtor recebe null como argumento. Veja:

```
public Pessoa(string nome) => _nome = nome ?? throw new ArgumentN
ullException();
```

As definições de corpo da expressão introduzidas no C# 6.0 e melhoradas no C# 7.0 possibilitam a implementação de um membro em uma forma bastante concisa e legível. Podemos utilizá-las sempre que a lógica para um método, construtor, destrutor, indexador ou propriedade, consistir em uma única expressão.

Para testar essa possibilidade, será necessário alterar a linha que cria a instância da classe `Pessoa` conforme mostrado a seguir:

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

> Para saber mais sobre `throw` em contextos que requerem expressões, consulte:
>
> https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/throw#the-throw-expression

8.7 UTILIZANDO O OPERADOR AWAIT EM BLOCOS CATCH E FINALLY

Antes do lançamento do C# 6.0, os desenvolvedores tinham que recorrer a todos os tipos de soluções estranhas para trabalhar com programação assíncrona e lidar com o tratamento de exceções, uma vez que não era possível utilizar a palavra-chave `await` nos blocos `catch` e `finally`. A partir desta versão da linguagem, já é possível utilizar o operador `await` sem restrições.

Embora não pareça uma mudança importante em um primeiro momento, é preciso lembrar que o consumo de APIs assíncronas é cada dia mais frequente. Além disso, a maioria dos aplicativos inclui *logging* ou recursos similares em cláusulas `catch`, o que pode se tornar mais complicado se o log for implementado como uma operação assíncrona em sistemas distribuídos. Como se não bastasse, existem ainda cenários em que desejamos executar algum trabalho de limpeza assíncrono em uma cláusula `finally`.

O exemplo a seguir, extraído da documentação oficial do C# 6.0, ilustra como utilizar o operador `await` nos blocos `catch` e `finally` de um método assíncrono:

```
public static async Task<string> MakeRequestAndLogFailures()
{
    await logMethodEntrance();
    var client = new System.Net.Http.HttpClient();
    var streamTask = client.GetStringAsync("https://localHost:100
00");
    try {
        var responseText = await streamTask;
        return responseText;
    } catch (System.Net.Http.HttpRequestException e) when (e.Mess
age.Contains("301"))
    {
        await logError("Recovered from redirect", e);
```

```
        return "Site Moved";
    }
    finally
    {
        await logMethodExit();
        client.Dispose();
    }
}
```

Para saber mais sobre o tratamento de exceções em métodos assíncronos, consulte a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/try-catch#exceptions-in-async-methods

8.8 INTERROMPENDO A EXECUÇÃO QUANDO UMA EXCEÇÃO FOR GERADA

O Visual Studio disponibiliza a janela *Configurações de Exceção* para gerenciamento de exceções. Por meio dela, podemos definir qual exceção quebrar, em que ponto interromper, adicionar novas exceções e adicionar condições para as exceções.

Para abrir essa janela, clique no menu *Depurar*, selecione o submenu *Janelas* e, a seguir, a opção *Configurações de Exceção*.

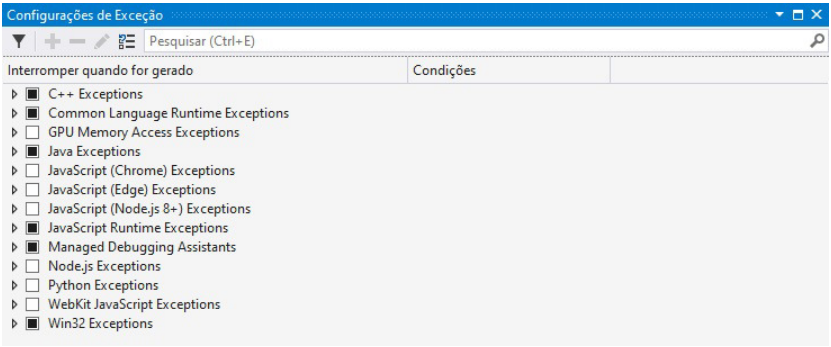


Figura 8.12: Janela de Configurações de Exceção

As exceções que nos interessam estão agrupadas na categoria *Common Language Runtime Exceptions*. Expanda este nó e inspecione a longa lista de exceções disponíveis:

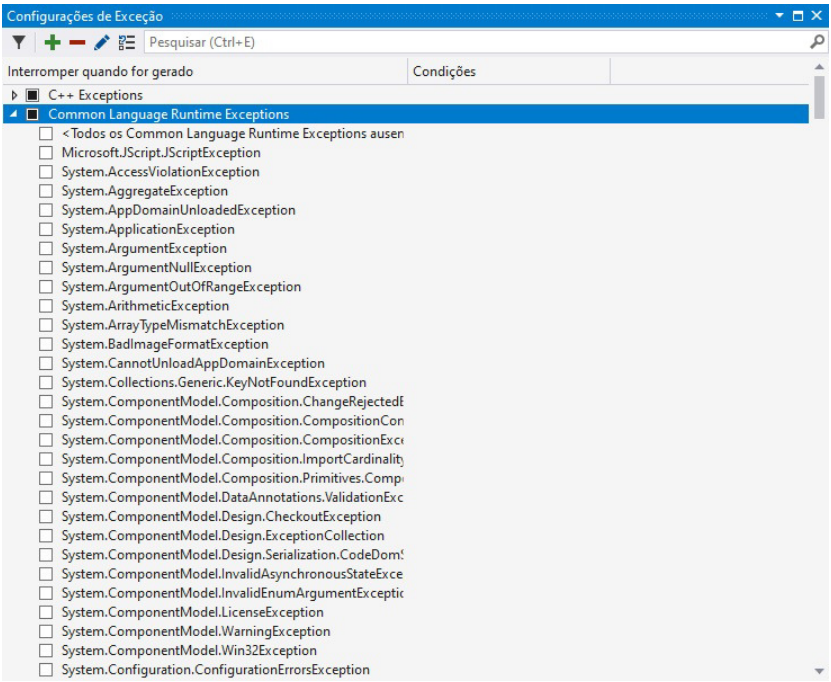


Figura 8.13: Lista de exceções da Common Language Runtime

Note que você pode selecionar todo o grupo de exceções da *Common Language Runtime* simplesmente clicando na caixa de verificação existente ou pode selecionar uma ou mais exceções clicando nas caixas de verificação dos nós que representam os tipos de exceção disponíveis. Observe que existe uma opção com o nome *Todos os Common Language Runtime Exceptions ausentes* para capturar as exceções não presentes nesta lista. Se preferir, você também pode clicar com o botão direito sobre a categoria de exceções e selecionar no menu de contexto a opção *Adicionar Exceção*. Neste caso, digite o nome da exceção seguido de enter para incluí-la na categoria.

A partir do Visual Studio 2017, podemos definir uma condição na exceção ou categoria de exceções. Com isso, somente quando houver uma condição correspondente, o IDE lançará uma exceção.

Para definir uma condição, clique com o botão direito do mouse sobre a exceção ou grupo de exceções desejado e selecione no menu de contexto a opção *Editar condições*. A caixa de diálogo *Editar condições* será exibida.

Especifique os nomes dos módulos para os quais você deseja gerar a exceção. Use o link *Adicionar condição* para incluir condições extras. Observe no aviso da janela que também é possível utilizar coringas na definição da condição e que o tipo de exceção afetada pela condição é mostrado no campo *Tipo*.

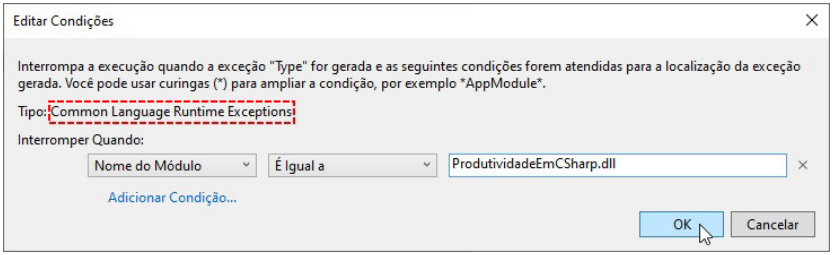


Figura 8.14: Definindo uma condição para o lançamento da exceção

Depois que a condição é aplicada, você verá que ela é exibida na coluna de condições da janela *Configurações de Exceção*.

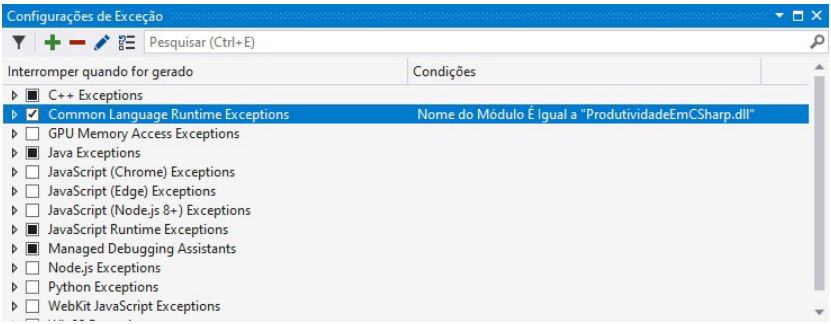


Figura 8.15: Condição de lançamento da exceção sendo listada na janela Configurações de Exceção

Essa funcionalidade é particularmente útil quando precisamos ignorar temporariamente algumas bibliotecas (que desenvolvemos ou foram criadas por terceiros) das quais já conhecemos os problemas existentes, enquanto investigamos novos bugs em outra parte do código.

Para experimentar na prática como essa janela afeta a nossa forma de trabalhar, vamos partir do exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static void DisparaETrataExcecao()
        {
            try
            {
                throw new AccessViolationException();
            }
            catch (Exception e)
            {
                Console.WriteLine($"Tratei a exceção! Mensagem: {
e.Message}");
            }
```

```
        }

        public static void DisparaExcecaoNaoTratada()
        {
            throw new AccessViolationException();
        }

        static void Main(string[] args)
        {
            DisparaETrataExcecao();
            DisparaExcecaoNaoTratada();
        }
    }

}
```

Ao analisar o código, note que temos um método `DisparaETrataExcecao` que gera uma exceção e a manipula, e um segundo método `DisparaExcecaoNaoTratada` que gera a mesma exceção sem tratá-la.

Na janela *Configurações de Exceção*, limpe as configurações anteriores clicando no botão *Restaurar a lista para as configurações padrões* presente na barra de botões e, em seguida, selecione o tipo de exceção `System.AccessViolationException` conforme mostrado na imagem a seguir:

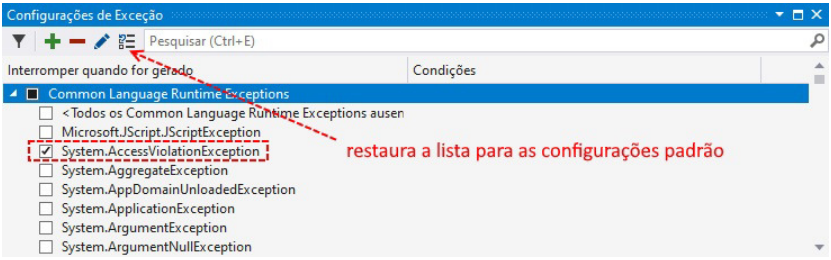


Figura 8.16: Filtrando apenas a exceção AccessViolationException

Uma vez feita a configuração, já podemos teclar *F5* para testar.

Você verá que a execução para no lançamento da exceção no método `DisparaETrataExcecao` , ou seja, antes de ser tratada no bloco `catch` da instrução `try...catch` . Clique no botão *Continue* ou tecle *F5* e observe que novamente a execução é interrompida com a sinalização de exceção lançada, desta vez no método `DisparaExcecaoNaoTratada` . Clicando em *Continue* uma vez mais, será exibido o aviso de exceção não tratada.

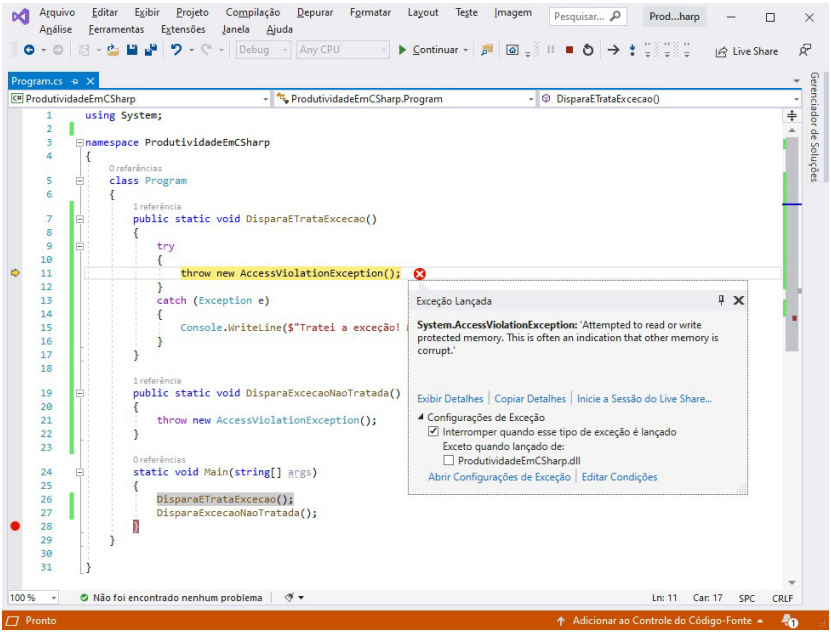


Figura 8.17: Execução interrompida com o lançamento da exceção do tipo AccessViolationException

Repare na imagem anterior que através da própria janela de lançamento da exceção é possível abrir a janela *Configurações de Exceção* e editar a condição atual que rastreou a exceção.

Para saber mais sobre a janela *Configurações de Exceção* e sobre filtragem por condição, consulte a seguinte página da documentação:

https://docs.microsoft.com/pt-br/visualstudio/debugger/managing-exceptions-with-the-debugger?view=vs-2019

8.9 CAPTURANDO EXCEÇÕES DO PRÓPRIO FRAMEWORK E DE BIBLIOTECAS DE TERCEIROS

O Visual Studio possui um recurso de depuração pouco conhecido chamado *Apenas meu código* (em inglês, *Just my Code*), que está ativo por padrão para melhorar a experiência de depuração. Ele esconde da lista pilha de chamadas, as chamadas aos métodos do framework e aos métodos das bibliotecas de terceiros colocando em seu lugar uma linha marcada como *[Código Externo]*. Veja:

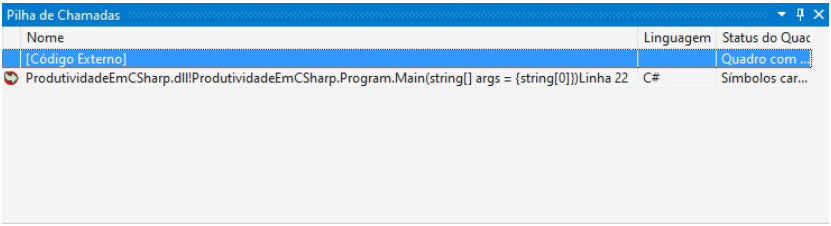


Figura 8.18: Janela Pilha de Chamadas exibindo apenas código de usuário

Esta abordagem simplifica o processo de depuração, permitindo que o desenvolvedor ou desenvolvedora se concentre em seu próprio código, e não no que foi gerado automaticamente ou desenvolvido por terceiros. Na documentação oficial, o código-fonte que geramos é chamado de *Código de usuário*. O restante é considerado como *código não-usuário* e inclui código gerado automaticamente pelos designers, como o código que inicializa um aplicativo ou coloca os controles nas janelas em aplicações Windows Forms e WPF, código de terceiros que incluímos na forma de pacotes via gerenciador de pacotes NuGet e o código do próprio framework. Até mesmo parte do seu código gerado, por exemplo, por uma ferramenta automática, pode entrar nessa classificação se for marcado com o atributo de *código não-usuário* `DebuggerNonUserCode`. Veja a seguir a janela Pilha de Chamadas mostrando *código não-usuário* para o mesmo programa:

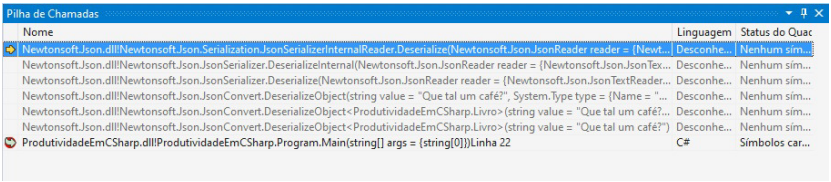


Figura 8.19: Janela Pilha de Chamadas exibindo métodos do código não-usuário (biblioteca de terceiros)

Ao desativar o recurso *Apenas meu código*, você verá a pilha de chamadas completa, o que pode fornecer dicas para solucionar o problema. Com ele ativo, ainda é possível inspecionar essas informações extras, mas é necessário clicar com o botão direito do mouse sobre a janela e selecionar no menu de contexto a opção *Exibir Código Externo*.

Quando este recurso está ligado, o Visual Studio interrompe a

execução apenas em exceções lançadas no seu próprio código. Sem que saibamos, muitas outras exceções são lançadas, sendo em sua maioria irrelevantes. Essas exceções ocorrem no próprio .NET framework e em bibliotecas de terceiros e geralmente não queremos ser notificados sobre elas.

Em cenários nos quais comportamentos anormais estão acontecendo, entretanto, desativar temporariamente este recurso pode ser útil para que possamos analisar com mais detalhes estas exceções. Por exemplo:

- Uma solicitação ao seu servidor retorna 500 sem motivo aparente.
- Uma chamada ao código da biblioteca de terceiros retorna resultados inesperados.
- Uma chamada ao código da biblioteca de terceiros gera uma exceção.

Para desativar o recurso *Apenas meu código*, execute os seguintes passos:

1. No menu *Depurar*, selecione *Opções*.
2. A caixa de diálogo *Opções* será exibida com a opção *Geral* da depuração pré-selecionada. Na lista de opções mostrada à direita, desmarque a caixa de verificação *Habilitar apenas Meu Código*. Em seguida, clique no botão *OK*.

Agora que você já sabe como ativar e desativar o recurso, vamos testá-lo com o código a seguir:

```
using System;
using Newtonsoft.Json;

namespace ProdutividadeEmCSharp
```

```
{
    class Livro
    {
        public string Titulo { get; set; }
        public string Autor { get; set; }

    }
    class Program
    {

        static void Main(string[] args)
        {
            string json = @"{
                      'Titulo': 'Produtividade em C#',
                      'Autor': 'Cláudio Ralha'
                    }";
            //json = "Que tal um café?";
            Livro livro = JsonConvert.DeserializeObject<Livro>(js
on);
            Console.WriteLine($"Título: {livro.Titulo}\nAutor: {l
ivro.Autor}");

        }
    }
}
```

Note que é necessário incluir o pacote `Newtonsoft.Json` usando o gerenciador de pacotes *NuGet* e incluir um ponto de interrupção na linha que efetua uma chamada ao método `DeserializeObject` da biblioteca.

Rode a primeira vez com a linha que produzirá o erro comentado e observe que o código funciona como esperado. Veja:

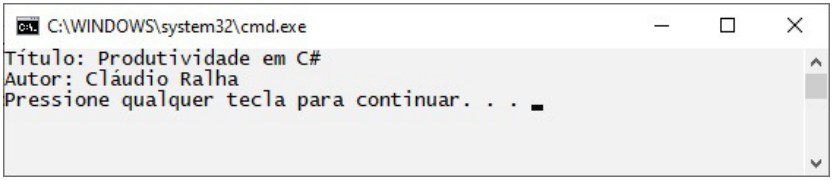


Figura 8.20: Funcionamento esperado do programa de teste

Em seguida, descomente a linha que criamos para forçar o lançamento da exceção e execute uma vez mais. Confira o resultado na próxima imagem:

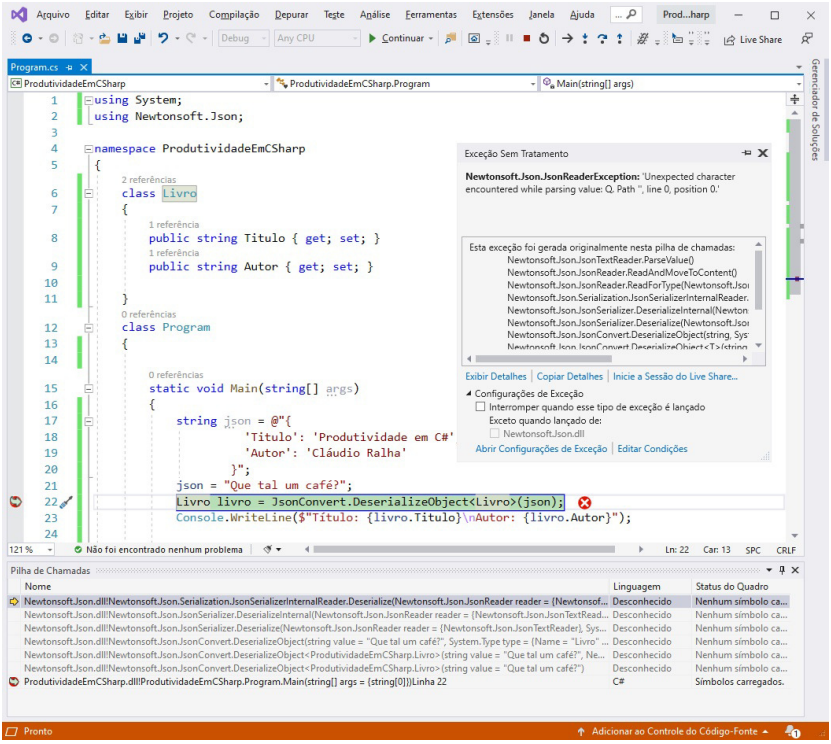


Figura 8.21: Exceção lançada em um método do código não-usuário

Ao compararmos as duas visualizações da pilha de chamada, ficamos com a sensação de que antes estávamos aprendendo a andar de bicicleta com segurança e que agora nos tiraram as rodinhas. Com o tempo, você descobrirá que esse recurso é particularmente útil para testar novas versões de bibliotecas de terceiros e do próprio framework, ainda não estáveis.

Para saber mais sobre como trabalhar com o recurso *Apenas meu código* desabilitado, consulte:

https://docs.microsoft.com/pt-br/visualstudio/debugger/just-my-code?view=vs-2019

https://docs.microsoft.com/pt-br/visualstudio/debugger/how-to-use-the-call-stack-window?view=vs-2019

8.10 CONSULTANDO A PSEUDOVARIÁVEL $EXCEPTION

Durante a depuração do código, por vezes nos deparamos com um bloco `catch` genérico em uma estrutura `try...catch` que sequer possui uma variável a ser inspecionada. Veja o exemplo:

```
0 referências
static void Main(string[] args)
{
    int a = 10, b = 0, c = 0;
    try
    {
        c = a / b;
    }
    catch (Exception)
    {
        Console.WriteLine("Ocorreu um erro!");
    }
}
```

Figura 8.22: Depurando o tratamento de uma exceção de um tipo desconhecido

Neste ponto, surge aquela dúvida: o que fazer para ter acesso

aos detalhes da exceção lançada sem precisar fazer alterações no código e reexecutá-lo. A boa notícia é que o Visual Studio dispõe de várias *pseudovariáveis* usadas para os mais diversos fins e uma delas é a `$exception` que armazena a última exceção lançada.

Você pode usar as janelas de *Inspeção*, *Inspeção Rápida (QuickWatch)* ou *Imediata*, que abordaremos em detalhes no capítulo sobre depuração, para inspecionar esta variável. Confira:

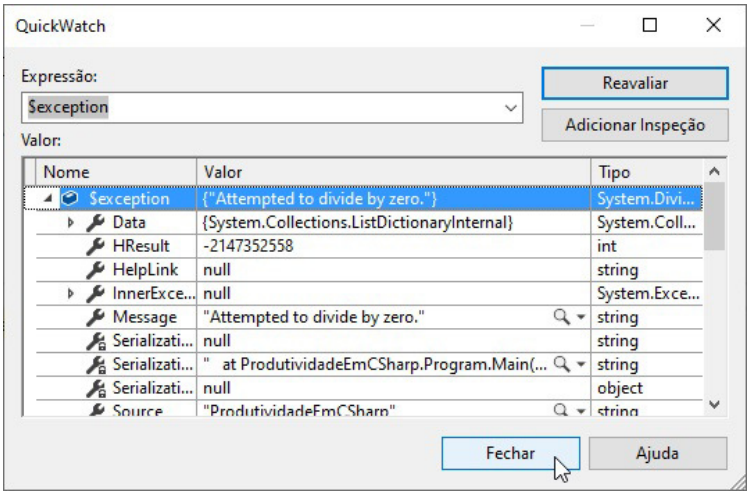


Figura 8.23: Consultando a pseudovariável $exception usando a janela de Inspeção Rápida

Para conhecer a lista completa de pseudovariáveis suportadas pelo Visual Studio, acesse:

https://docs.microsoft.com/pt-br/visualstudio/debugger/pseudovariables?view=vs-2019

8.11 CRIANDO CLASSES DE EXCEÇÃO PERSONALIZADAS

O Visual Studio conta com um recurso chamado *Geração a partir do Uso* que nos permite criar automaticamente classes, structs, interfaces, enumerações e métodos a partir de uma referência do seu uso. Abordaremos esta funcionalidade em detalhes no próximo capítulo, mas antes veremos como utilizá-la para criar o esqueleto de uma classe que herda da classe `Exception` a partir do seu uso. Considere o exemplo a seguir:

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Teste()
        {
            throw new MyCustomException();
        }



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

Para criar o código da classe `MyCustomException` usada no método `Teste`, basta parar com o mouse sobre a marca inteligente que é mostrada embaixo do nome e selecionar no menu a opção *Gerar classe `MyCustomException` em um novo arquivo*. Veja:

Figura 8.24: Gerando de forma automática o esqueleto de código da classe MyCustomException

Isso fará com que seja criado no projeto o arquivo `MyCustomException.cs` com o código mostrado a seguir:

```
using System;
using System.Runtime.Serialization;

namespace ProdutividadeEmCSharp
{
    [Serializable]
    internal class MyCustomException : Exception
    {
        public MyCustomException()
        {
        }

        public MyCustomException(string message) : base(message)
        {
        }

        public MyCustomException(string message, Exception innerE
xception) : base(message, innerException)
        {
```

```
        }

        protected MyCustomException(SerializationInfo info, Strea
mingContext context) : base(info, context)
        {
        }
    }
}
```

Percebeu quanto trabalho esta ferramenta é capaz de poupar? Vale destacar, entretanto, que só devemos criar classes de exceção personalizadas quando as já existentes não atenderem aos nossos propósitos. Na maioria dos cenários, elas atendem, então "Não perca tempo reinventando a roda"!

Neste capítulo, adiantamos alguns recursos relacionados à depuração que será tratada em detalhes mais adiante no capítulo 11. Isso foi necessário para agrupar de forma sequencial o máximo de informações relevantes para o desenvolvedor que já domina o básico do tratamento de exceções.

Antes de avançar para o próximo capítulo focado na geração de código, recomendamos a leitura da página a seguir que contém as melhores práticas para gerenciamento de exceções, reunidas pelo time de desenvolvimento do Visual Studio:

https://docs.microsoft.com/pt-br/dotnet/standard/exceptions/best-practices-for-exceptions

Capítulo 9

GERAÇÃO DE CÓDIGO

A *geração automática de código* é um assunto que atrai o interesse de todos os desenvolvedores que buscam produtividade e que faz parte do dia a dia inclusive daqueles que não percebem a sua presença.

Desde a seleção de um *template* para um novo projeto no Visual Studio até a criação de um banco de dados usando um *framework de mapeamento ORM*, como Entity Framework Core, passando pela simples criação de um membro de classe usando um *Code Snippet* ou de um formulário usando um *Designer*, estamos fazendo uso da geração de código para minimizar o trabalho "braçal" e tedioso. São facilidades que tornam o Visual Studio mais amado e completo a cada release e que ainda podem ser estendidas através de extensões gratuitas e pagas disponíveis no *Visual Studio Marketplace*.

Ao longo deste capítulo, veremos como explorar várias funcionalidades que nos pouparão muitas horas de trabalho e encantarão até os mais céticos. Iniciaremos o nosso estudo mostrando como usar fragmentos de dados nos formatos JSON e XML para criar de forma automática classes inteiras que possam armazená-los.

Abordaremos em sequência um recurso chamado *Geração a partir do uso*, que nos permite criar automaticamente classes, structs, interfaces, enumerações e métodos a partir de uma referência do seu uso.

Em seguida, demonstraremos as várias formas de usar *snippets de código*, como baixar snippets extras e como criar os seus próprios snippets de código. Se você tem dificuldade para memorizar certas sintaxes complexas da linguagem, vai amar esse recurso e usá-lo sem moderação.

O capítulo termina apresentando como baixar *templates de projeto* disponíveis on-line e como gerar os seus próprios templates de projeto. Você descobrirá nas próximas páginas que existe muita coisa boa e gratuita on-line esperando apenas o seu download.

9.1 GERANDO CLASSES A PARTIR DE JSON E XML

O Visual Studio dispõe de um gerador de classes a partir de dados fornecidos em JSON e XML. Para utilizá-lo, bastará copiar para o clipboard um trecho de código escrito em um desses dois formatos. No exemplo a seguir, temos um fragmento de código em JSON representando este livro:

```
{
  "Titulo": "Produtividade em C#",
  "Autor": "Cláudio Ralha",
  "Ano": 2021,
  "Editora": "Casa do Código"
}
```

Após efetuar a cópia para a área de transferência, o processo de geração de código é bem simples. Basta clicar no menu *Editar*,

selecionar o submenu *Colar especial* e escolher a opção *Colar JSON como classes* ou a opção *Colar XML como classes* dependendo do formato dos dados selecionados.

Veja o código gerado pelo Visual Studio para o JSON fornecido:

```
public class Rootobject
{
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public int Ano { get; set; }
    public string Editora { get; set; }
}
```

Observe que nesse caso o único ajuste que precisaremos fazer será renomear a classe de `Rootobject` para `Livro`. Pense em um JSON com cinquenta campos e você terá uma ideia imediata de quanto trabalho chato e cansativo esta ferramenta é capaz de nos poupar.

9.2 GERANDO CLASSES E STRUCTS A PARTIR DO SEU USO

A partir do Visual Studio 2015, a IDE passou a dispor de um recurso nativo de geração de código chamado *Geração a partir do Uso* (em inglês, *Generate from Usage*). Ele nos permite escrever um código que faça referência a classes, structs, interfaces, enumerações e membros que não existem e faz com que o ambiente de desenvolvimento os crie automaticamente. Isso é particularmente útil em alguns estilos de programação, como o TDD (*Test-driven Development - Desenvolvimento orientado a testes*).

Para ilustrar como criar o código de uma classe a partir do seu uso, vamos adicionar ao método `Main` de uma aplicação de console a linha de código a seguir:

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

Como a classe `Pessoa` ainda não existe no projeto, ela estará sublinhada em vermelho no editor de código do Visual Studio. Para gerar a classe `Pessoa` com um mínimo de esforço, execute os seguintes passos:

1. Clique com o botão direito do mouse sobre o nome da classe na linha anterior e selecione no menu de contexto a opção *Ações Rápidas e Refatorações*. Também é possível exibir esse menu parando o cursor de inserção sobre o nome `Pessoa` e teclando *Ctrl + ponto final*. Em ambos os casos, serão exibidas as opções mostradas na imagem a seguir:

Figura 9.1: Gerando uma classe a partir do seu uso

2. Selecione a opção *Gerar class "Pessoa" no novo arquivo*.

Pronto! O arquivo será gerado com o nome `Pessoa.cs`, mas

não será automaticamente carregado no editor de código para não atrapalhar a edição no arquivo em que estamos atualmente trabalhando. Abra-o e inspecione o código gerado.

Uma segunda maneira de criar a classe é usando a *smart tag* ou *marca inteligente* mostrada junto ao erro. Para exibi-la, basta parar o mouse sobre o nome da classe sublinhado em vermelho. Clique na seta existente junto ao símbolo para exibir o menu visto anteriormente e selecionar a opção *Gerar class "Pessoa" no novo arquivo*. Veja:

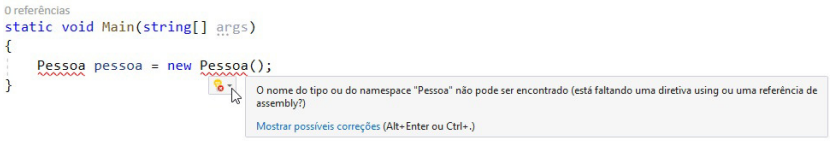


Figura 9.2: Utilizando a marca inteligente para acessar o menu de geração de código

Caso prefira usar o teclado, pressione *Ctrl+.* para abrir as opções de marca inteligente. Saiba que é possível utilizar essa combinação de teclas logo depois de digitar o nome da classe, evitando o mouse durante toda a operação.

É importante destacar que parte dos elementos que vamos criar usando esta funcionalidade serão apenas *stubs*, ou seja, membros sem código real criados apenas para permitir a compilação. Eles serão modificados posteriormente pelo desenvolvedor para adicionar a funcionalidade necessária. O recurso "Gerar a partir do uso" nos permite criar stubs para classes, estruturas, interfaces, métodos, propriedades, campos, enumerações e constantes de enumeração.

Ao examinar o código da classe `Pessoa` que acabamos de

criar, você notará que ela é gerada no namespace padrão do projeto e inclui as diretivas de uso que estão presentes para uma classe criada manualmente.

Caso deseje ter mais controle sobre a classe criada ou criar um struct, utilize o comando *Gerar novo tipo* no menu da marca inteligente. A caixa de diálogo *Gerar Tipo* será exibida:

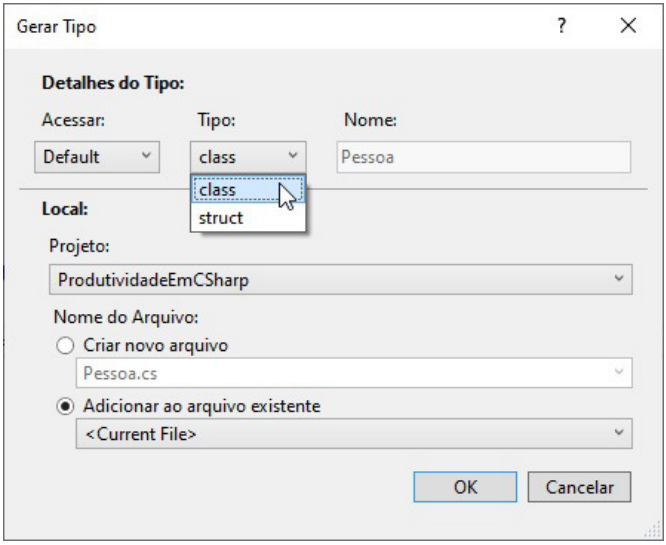


Figura 9.3: Construindo o tipo a partir da caixa de diálogo Gerar Tipo

Ao examinar esta janela, observe a lista suspensa *Tipo*. Ela especifica o tipo de código a ser criado (*classe* ou *struct*). Em *Acessar*, podemos escolher o modificador de acesso entre `internal` e `public`. Na lista de projetos, caso exista mais de um na solução, podemos escolher ao qual desejamos adicionar o código. Isso é útil para o TDD quando os testes são mantidos em um projeto separado do código da aplicação.

9.3 GERANDO PROPRIEDADES E CAMPOS A PARTIR DO SEU USO

As propriedades e campos podem ser gerados de maneira semelhante aos tipos. Se você adicionar o código a seguir, o membro `Nome` será sublinhado:

```
Pessoa pessoa = new Pessoa();
pessoa.Nome = "Cláudio Ralha";
```

Como esse membro não inclui parênteses, ele representa uma propriedade ou campo indefinido. Siga o mesmo procedimento descrito na seção anterior para ativar a marca inteligente e você verá o menu mostrado na imagem a seguir:

Figura 9.4: Gerando uma propriedade a partir do seu uso

Selecione a opção *Gerar propriedade 'Pessoa.Nome'*. O código a seguir será adicionado à classe `Pessoa` :

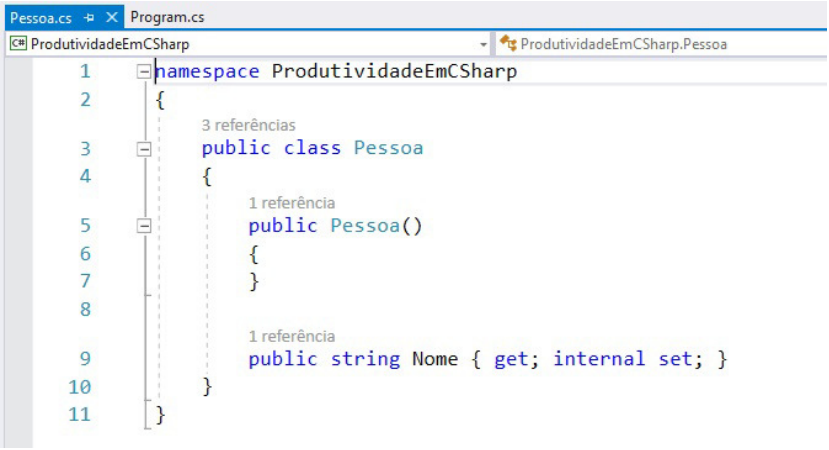


Figura 9.5: Inspecionando a propriedade Nome gerada

Observe que o tipo de dados é inferido do código. Em muitos casos, esse tipo estará correto. Em algumas poucas exceções, será necessário alterá-lo.

9.4 GERANDO MÉTODOS A PARTIR DO SEU USO

Para gerar um stub de um método, seja ele um *método de instância* ou um *método estático*, basta adicionar um membro indefinido com parênteses ou parâmetros. Adicione o seguinte código ao programa para testar esse recurso:

```
pessoa.Escrever("Produtividade em C\#");
```

Utilize a marca inteligente para gerar o método através da opção *Gerar método 'Pessoa.Escrever'* conforme explicado anteriormente. Confira na próxima listagem o código gerado para o método `Escrever`:

```
internal void Escrever(string v)
{
    throw new NotImplementedException();
}
```

Observe que o tipo de retorno do método foi especificado corretamente e que você provavelmente vai querer alterar a visibilidade do método alterando o modificador de acesso de `internal` para `public`. O mesmo pode ocorrer com o nome dos parâmetros.

Com relação ao nome dos parâmetros, se você utilizar variáveis como parâmetros no momento de criar o stub, os seus nomes serão utilizados para nomear os parâmetros. Para testar, adicione as seguintes linhas de código ao método `Main` da classe `Program.cs`:

```
string mensagem = "Produtividade em C#";
ConsoleColor cor = ConsoleColor.Green;
pessoa.Escrever(mensagem, cor);
```

Gere código para esta sobrecarga do método `Escrever`. Confira a seguir o código gerado pelo Visual Studio:

```
internal void Escrever(string mensagem, ConsoleColor cor)
{
    throw new NotImplementedException();
}
```

Outra forma de se obter o mesmo resultado é usando parâmetros nomeados. Veja um exemplo:

```
pessoa.Escrever(mensagem: "Produtividade em C#", cor: ConsoleColo
r.Green);
```

Ao criar as sobrecargas do método `Escrever`, observe que o Visual Studio oferece também a opção de adicionar o parâmetro `cor` ao método existente. Veja:

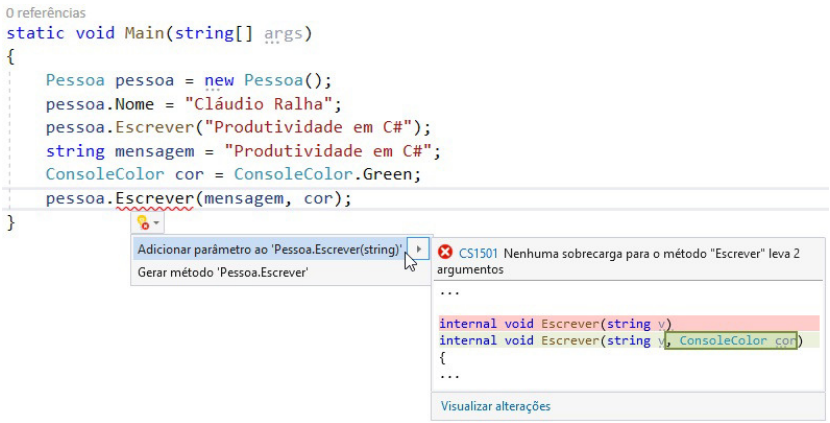


Figura 9.6: O gerador de código oferece a opção de adicionar parâmetro ao método previamente criado

Um detalhe importante a ser notado ao gerar stubs de método é o de que o uso da palavra-chave `var` impede a detecção do tipo correto de retorno para o método. Para ilustrar, gere um stub para o método `ECasado` da linha de código a seguir:

```
var casado = pessoa.ECasado();
```

Ao inspecionar o código gerado pelo Visual Studio, você verá que o uso da palavra-chave `var` fez com que o gerador colocasse como retorno `object` em vez de `bool`. Altere o tipo de retorno do método gerado `ECasado` para `bool` ou repita o procedimento partindo da linha de código a seguir:

```
bool casado = pessoa.ECasado();
```

9.5 GERANDO CONSTRUTORES A PARTIR DO SEU USO

A ferramenta de geração de código a partir do seu uso é um

aliado poderoso na busca por produtividade. Até agora, já vimos como criar classes, propriedades, campos e métodos. Vamos explorar mais um pouco esse recurso criando construtores.

Para criar um construtor parametrizado (o construtor padrão é gerado junto com a classe), basta partir de uma linha de código, como a mostrada a seguir, e utilizar a opção *Generate Constructor in 'Pessoa'*:

```
Pessoa autor = new Pessoa(nome:"Cláudio Ralha");
```

Confira o código gerado para o construtor:

```
public Pessoa(string nome)
{
    Nome = nome;
}
```

Voilà! O construtor gerado para a classe `Pessoa` atribui o parâmetro `nome` à propriedade `Nome` conforme esperado.

9.6 GERANDO ENUMERADOS A PARTIR DO SEU USO

Para finalizarmos o nosso tour sobre a geração de código a partir do seu uso no Visual Studio, vamos falar sobre o uso desta ferramenta com tipos enumerados. Ela só permite a adição de uma constante de enumeração, ou seja, é necessário que tenhamos criado previamente o tipo enumerado, ainda que vazio. Em outras palavras, se você já definiu um enum, você pode escrever código que contém referências a membros de enum indefinidos e usar o menu *Gerar* ou a marca inteligente para adicionar a constante à enumeração. A constante é adicionada como um novo item sem valor explícito.

Como exemplo, vamos criar um tipo enumerado chamado `eEstadoCivil`. Por questão de simplicidade, adicione-o no próprio arquivo `Pessoa.cs`. Defina-o como mostrado no fragmento de código a seguir:

```
public enum eEstadoCivil
{
    Solteiro,
    Casado
}
```

Obviamente, faltam estados nesse tipo enumerado. Na classe `Pessoa`, inclua a declaração da propriedade `EstadoCivil`:

```
public eEstadoCivil EstadoCivil { get; set; }
```

De volta ao arquivo `Program.cs`, adicione a seguinte linha de código:

```
autor.EstadoCivil = eEstadoCivil.Divorciado;
```

Utilize a marca inteligente para selecionar a opção *Gerar membro enum 'eEstadoCivil.Divorciado'*. Veja:

Figura 9.7: Adicionando o membro Divorciado ao tipo enumerado eEstadoCivil

Inspecione novamente o tipo enumerado e confira que a constante enumerada `Divorciado` foi adicionada.

```
public enum eEstadoCivil
{
    Solteiro,
    Casado,
    Divorciado
}
```

São recursos como este que aumentam a nossa capacidade de gerar uma grande quantidade de códigos com qualidade em um tempo reduzido, minimizando o trabalho braçal e fazendo com que sobre mais tempo para que a pessoa desenvolvedora pense no que está realmente fazendo.

> Para saber mais sobre a geração de código a partir do seu uso, visite a seguinte página da documentação oficial:
>
> https://docs.microsoft.com/pt-br/visualstudio/ide/walkthrough-test-first-support-with-the-generate-from-usage-feature?view=vs-2019

9.7 USANDO SNIPPETS DE CÓDIGO EM SEUS PROJETOS

Code snippets são pequenos fragmentos de código que podemos inserir em um arquivo de código no Visual Studio usando um atalho (em inglês, *shortcut*), como `ctor` seguido de *Tab* duas vezes ou usando o menu de contexto.

A lista de snippets disponíveis depende do tipo de arquivo em edição. Ao editarmos um arquivo `.cs`, teremos disponíveis os code snippets do C# que acompanham o Visual Studio e outros

instalados a partir do *Visual Studio Marketplace* ou desenvolvidos pelo próprio desenvolvedor.

Para obter snippets de código extras para o C#, execute os seguintes passos:

1. Acesse o endereço do *Visual Studio Marketplace*: https://marketplace.visualstudio.com/

2. Pesquise por *code snippets*. Uma lista de resultados será exibida incluindo snippets de código para outras linguagens e frameworks. Antes de avançar, perceba quantos snippets úteis estão disponíveis para poupar o nosso trabalho. Para usar os que estão marcados como FREE (gratuito), basta clicar no snippet desejado para exibir a página com uma descrição detalhada sobre ele. Experimente selecionar o item *C# Methods Code Snippets* criado por J. Sakamoto.

3. Clique no botão *Download*. Será efetuado o download de um arquivo de extensão do Visual Studio(`.vsix`). Abra-o para iniciar o instalador e clique no botão *Install* para efetuar a instalação.

4. Ao final você verá uma mensagem de sucesso e um aviso pedindo para fechar e reabrir o Visual Studio, caso ele esteja aberto, para que as mudanças tenham efeito.

5. Abra o Visual Studio e crie um projeto do tipo aplicação de console em C# para testar os novos snippets de código.

6. Acesse novamente a página do snippet de código no Visual Studio Marketplace e leia a seção *How to use?* na qual estão enumerados os atalhos de teclado disponíveis para os code

snippets que acabamos de instalar.

O uso de snippets de código acelera em muito a nossa produtividade, pois evita que tenhamos que memorizar sintaxes cada vez mais complexas para novas funcionalidades inseridas pelo time de desenvolvimento no C#.

Ao longo de quase todos os capítulos deste livro, você deve ter observado que apresentamos snippets de código que acompanham o Visual Studio para inserir os tipos de construções que estão sendo tratados. Nesta seção, veremos as maneiras disponíveis para inserir os snippets em nosso código.

Para ilustrar a primeira forma, baseada em um atalho de teclado, vamos utilizar o snippet de código de inserção de um método de instância sem parâmetros que faz parte do pacote que instalamos na seção anterior. Execute o seguinte roteiro:

1. Abra um arquivo de código em C# do projeto, como, por exemplo, `Program.cs` e digite `method` seguido de *Tab*. Veja que durante a digitação, o IntelliSense do Visual Studio lista os snippets de código que contêm a sequência de letras digitadas:

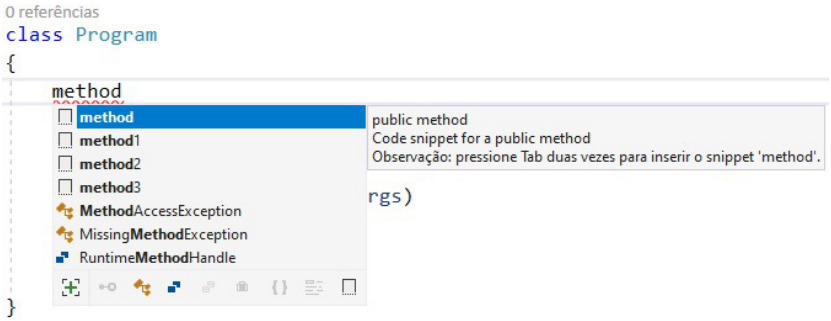


Figura 9.8: IntelliSense exibindo a lista de atalhos para os code snippets

2. Após a digitação do *Tab*, o código é inserido no lugar da palavra de atalho:

Figura 9.9: Código do método inserido via code snippet

3. Perceba que o modificador de acesso `public` está selecionado. Você pode digitar um modificador de acesso diferente, como `private` ou simplesmente teclar *Tab* para avançar para o retorno do método e trocá-lo de `void` para `string`, por exemplo, e, em seguida, teclar *Tab* novamente para selecionar e alterar o nome do método de `MyMethod` para algo significativo. Ao final, tecle *Esc* para tirar a seleção das partes da assinatura do método.

Note que, apesar de ser extremamente prática, esta forma de trabalhar exige que saibamos de antemão o atalho. O que fazer quando não conseguirmos lembrar o que digitar para inserir o snippet de código?

Para esses casos, temos algumas opções:

a. Procurar neste livro ou no Google.

b. Usar a inserção de snippets de código via menu de contexto que mostraremos a seguir.

c. Clicar no menu `Ferramentas` e selecionar a opção *Gerenciador de Snippets de Código*. Através desta ferramenta é possível inspecionar todos os snippets de código instalados.

Basta selecionar a linguagem C# em *Linguagens* e as categorias de snippets disponíveis serão exibidas no painel à esquerda. Expanda as categorias para inspecionar os snippets. Na imagem anterior, *methods* corresponde à categoria instalada pelo pacote que baixamos do Visual Studio Marketplace nesta seção. Ao navegar pelas listas de snippets de código, note que o caminho para o snippet selecionado é mostrado no campo *Local*:

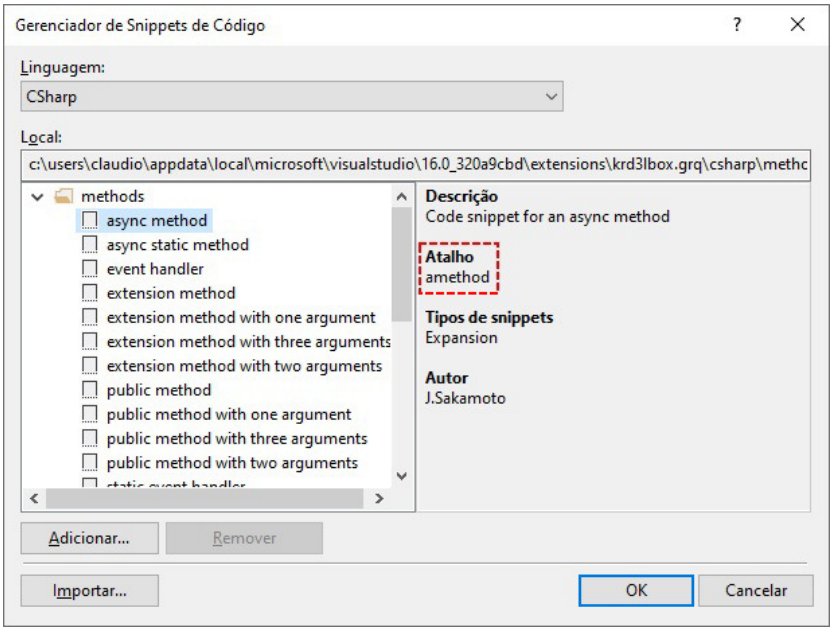


Figura 9.10: Descobrindo o atalho para o snippet desejado usando o Gerenciador de Snippets de Código

A segunda forma de inserir um snippet é clicando com o botão

direito do mouse sobre o editor de código em um arquivo de código C#, selecionar no menu de contexto o submenu *Snippets* e, a seguir, a opção *Inserir Snippet*. Veja:

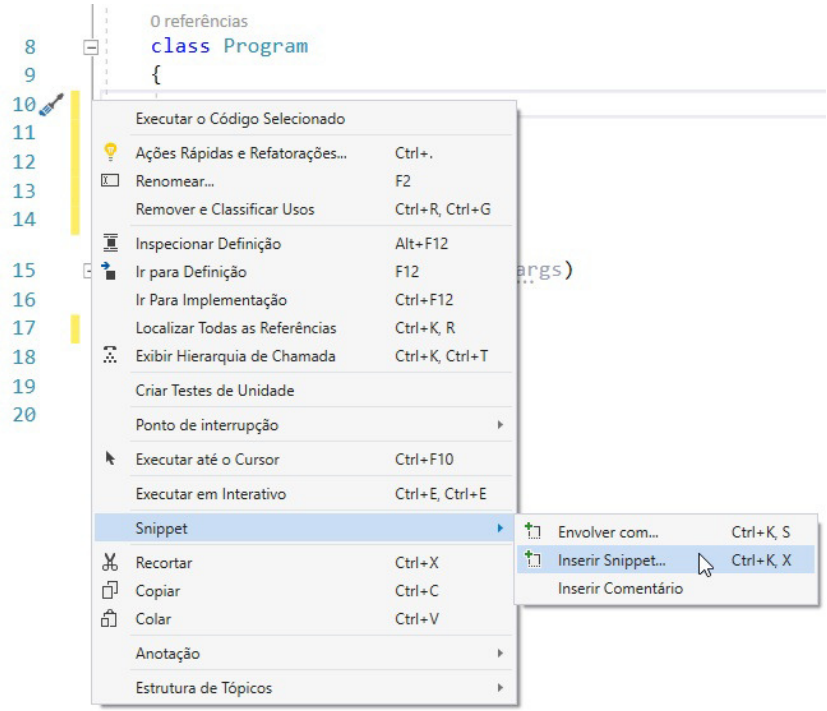


Figura 9.11: Acessando o menu de inserção de Snippets

Isso fará com que seja exibido o menu de inserção de snippets de código mostrado na próxima imagem. Você também pode exibi-lo diretamente teclando *Ctrl + K X*.

Para selecionar a categoria desejada, use o mouse ou as setas de teclado. A tecla *Tab* expande a categoria selecionada e insere o snippet.

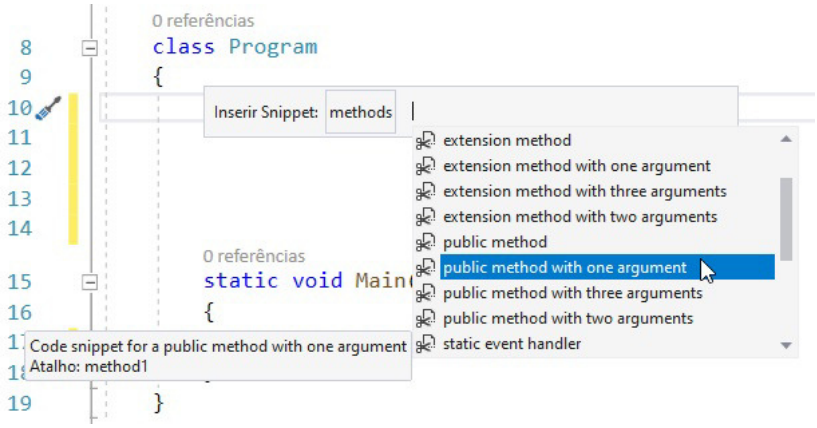


Figura 9.12: Menu de inserção de Snippets

Com o que vimos até aqui já é possível poupar muito trabalho, mas saiba que os snippets de código ainda nos permitem ir além. Em muitos cenários, temos um bloco de código que gostaríamos de mover para dentro de:

- um laço como `for` ou `while`;
- uma instrução condicional `if`;
- uma instrução `using`;
- uma instrução `try...catch` e outras possibilidades.

Para lidar com esses casos, basta selecionar o código a ser envolvido pelo snippet e teclar *Ctrl+K S* ou clicar com o botão direito do mouse sobre o código e selecionar no menu de contexto o submenu *Snippet* e a seguir a opção *Envolver com snippet*. Veja:

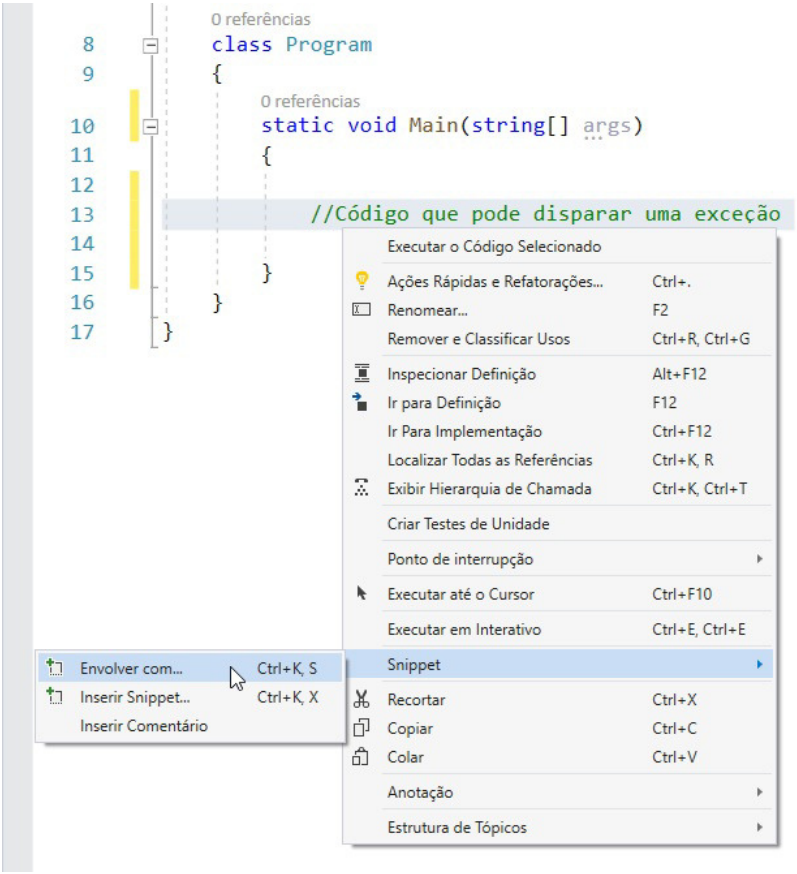


Figura 9.13: Acessando o menu de inserção de Snippets

No menu de snippets de código, selecione a opção desejada. Para esse teste, escolha `try`:

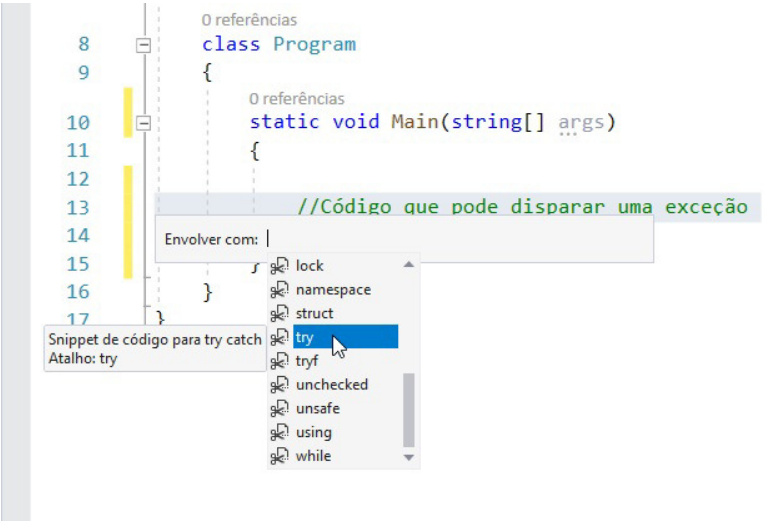


Figura 9.14: Usando a opção Envolver com para aplicar o snippet Try...catch

O código do snippet será adicionado. Repare que o bloco de código que envolvemos foi colocado dentro do `try` :

```
0 referências
class Program
{
    0 referências
    static void Main(string[] args)
    {

        try
        {
            //Código que pode disparar uma exceção

        }
        catch (Exception)
        {

            throw;
        }
    }
}
```

Figura 9.15: Snippet da estrutura try...catch adicionado

9.8 CRIANDO OS SEUS PRÓPRIOS SNIPPETS DE CÓDIGO

Podemos criar os nossos próprios snippets de código reutilizável para o Visual Studio.

Os snippets são arquivos com a extensão `.snippet` escritos em linguagem XML que podem ser editados no próprio Visual Studio. Veja a seguir o código do snippet criado para inserir `Console.Writeline()` usando o atalho `cw`:

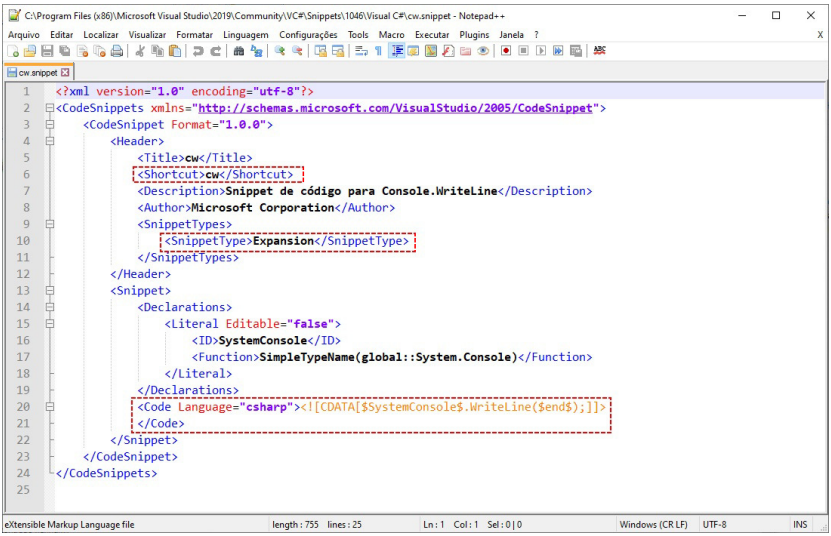


Figura 9.16: Inspecionando o código do snippet usado para inserir Console.Writeline()

É possível editar o XML diretamente no editor de código do Visual Studio ou em um programa similar, como o bom e velho Notepad++, mas isto exigiria um conhecimento maior da sua parte sobre a estrutura do XML que precisa ser gerada.

Como alternativa para simplificar o nosso trabalho e aumentar a nossa produtividade, é recomendável utilizar uma extensão, como o *Snippet Designer*, que oferece suporte visual para a construção dos snippets. Ela está disponível em:

https://marketplace.visualstudio.com/items?itemName=vs-publisher-2795.SnippetDesigner

Baixe-a e instale-a da mesma forma como fizemos com o pacote de snippets de métodos.

Com a extensão instalada, podemos criar um novo snippet do zero em C# executando os seguintes passos:

1. No menu *Arquivo*, selecione o submenu *Novo* e a seguir a opção *Arquivo...*. A caixa de diálogo *Novo Arquivo* será exibida.

2. Clique no item *Snippet Designer* do painel à esquerda. Em seguida, selecione o template *Code snippet* listado no painel central. Clique no botão *Abrir* para iniciar o designer de snippets.

3. O designer de code snippets será exibido. Veja:

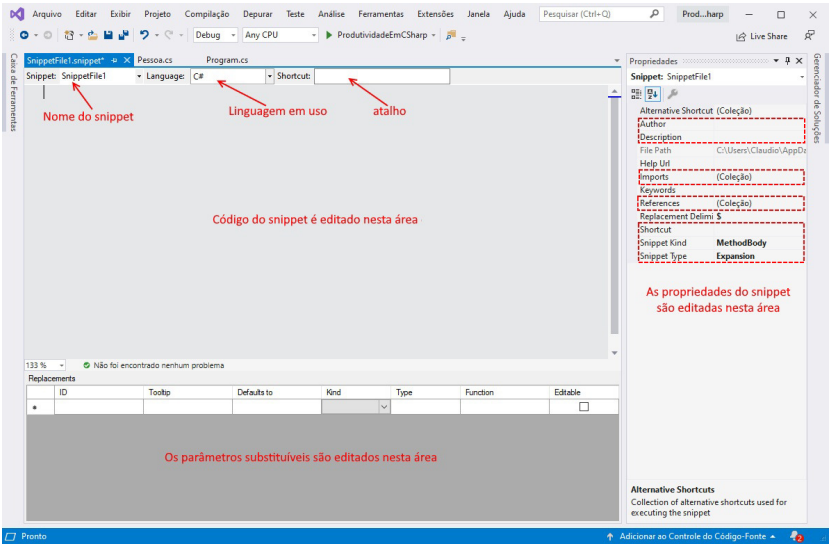


Figura 9.17: Tela principal do designer de code snippets

Para ilustrar como usar o designer, vamos criar um snippet que insira `Console.Writeline` com uma string interpolada. Para tanto, execute os seguintes passos:

1. No campo *Snippet*, digite `cwti` .

2. Em *Language*, selecione *C#*. O default é C++.

3. No campo *Shortcut*, digite novamente `cwti` .

4. No editor de propriedades, altere a propriedade *Author* para o seu nome e a propriedade *Description* para `Snippet para Console.WriteLine com texto interpolado` . Como o código que vamos inserir faz uso do caractere `$` , normalmente usado como delimitador de parâmetros substituíveis, altere a propriedade *Replacement Delimiter* de `$` para `#` . Veja:

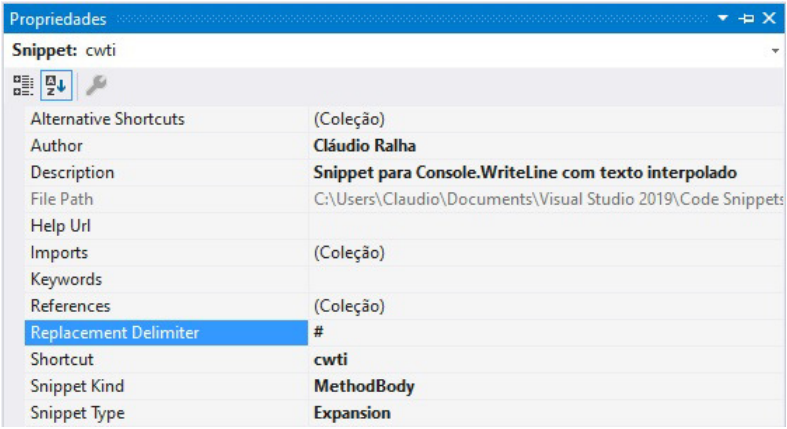


Figura 9.18: Alterando o delimitador padrão de parâmetros substituíveis

5. Na área reservada para a edição do snippet, entre com o seguinte código:

```
Console.WriteLine($"#texto#");
```

6. A linha anterior fará com que o parâmetro de substituição `texto` seja criado. No campo *Tooltip* da linha referente ao parâmetro, altere o conteúdo da coluna de `Texto` para `Texto a ser inserido`.

7. Tecle *Ctrl* + *s* para salvar o snippet. Ele será criado com o nome atribuído no campo *Snippet*. Os snippets que você criar em C# no visual Studio 2019 ficarão armazenados por padrão na seguinte pasta do seu computador: `%USERPROFILE%\Documents\Visual Studio 2019\Code Snippets\Visual C#\My Code Snippets`

Com o arquivo já salvo, alterne para um arquivo em C# no editor, como `Program.cs`, e digite `cwti` dentro do método `main`. Veja:

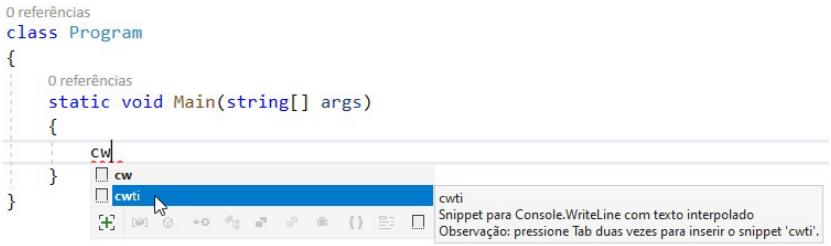


Figura 9.19: Digitando o atalho cwti para o snippet

Ao teclar *Tab*, o atalho será substituído pelo código a seguir. Note que a string `texto` está pré-selecionada e será substituída pelo texto digitado pelo desenvolvedor:

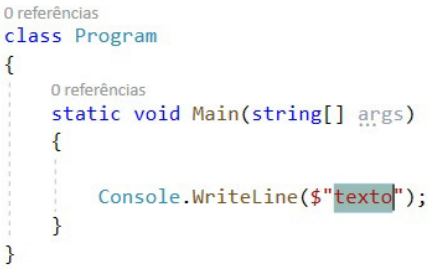


Figura 9.20: Código gerado via snippet

Ao construirmos nossos próprios snippets, temos a liberdade de criá-los de forma que sejam capazes de:

- Importar automaticamente os namespaces necessários.
- Suportar as operações de inserção do código do snippet e de envolver o código existente.
- Sinalizar em que parte do código eles podem ser inseridos.
- Permitir a substituição de parâmetros.

Um estudo completo sobre a criação de snippets foge ao

escopo deste livro, mas saiba que esta é uma excelente forma de organizar trechos reaproveitáveis do seu código e compartilhá-los com outros desenvolvedores. É possível inclusive iniciar a criação de snippet usando o *Snippet Designer* a partir de um trecho de código selecionado no editor. Basta clicar com o botão direito do mouse sobre o texto selecionado e escolher no menu de contexto a opção *Export as Snippet*. Veja:

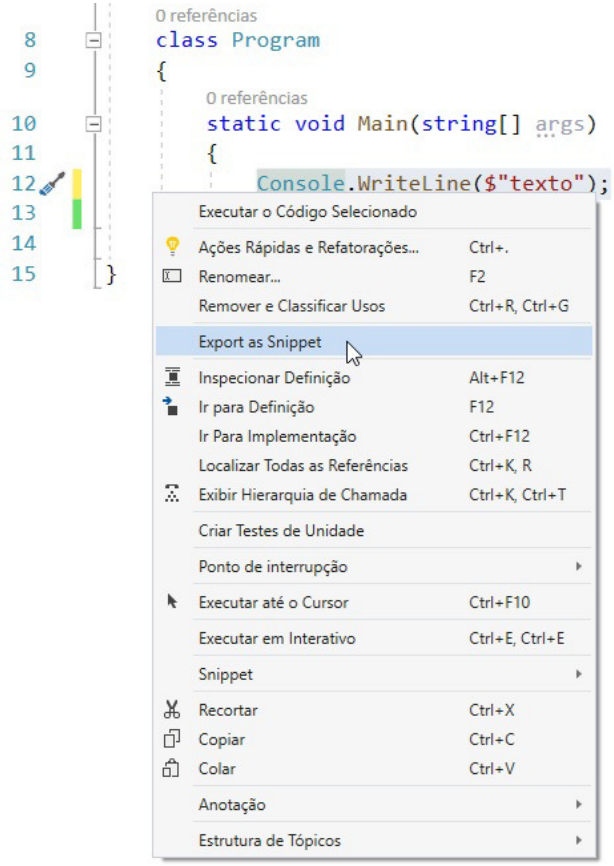


Figura 9.21: Exportando um trecho de código no editor como snippet

A forma mais rápida de aprender a criar snippets de código mais avançados é inspecionando o código de snippets existentes, alterando-os conforme a sua necessidade e salvando-os com um novo nome e atalho de teclado atribuído.

Para visualizar a lista de arquivos de snippets para a linguagem C# que acompanham o Visual Studio 2019, acesse a seguinte pasta:

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC#\Snippets\1046
```

> Para saber mais sobre a criação de snippets de código, incluindo como utilizar parâmetros de substituição e realizar importação de namespaces, consulte:
>
> https://docs.microsoft.com/pt-br/visualstudio/ide/walkthrough-creating-a-code-snippet?view=vs-2019

9.9 DOCUMENTANDO O CÓDIGO-FONTE COM COMENTÁRIOS XML

Ter à disposição documentação confiável para orientar o nosso trabalho é algo vital quando precisamos dar suporte a vários sistemas ao mesmo tempo. A presença de documentação do código facilita a evolução, manutenções corretivas e transferência de conhecimento para outros desenvolvedores.

Uma das formas disponíveis para efetuar a documentação do

código é usando *comentários e documentação XML*. Os comentários XML são um tipo especial de comentário que adicionamos acima da definição de qualquer tipo ou membro definido pelo usuário. Assim como ocorre com outros tipos de comentários, os comentários de documentação XML também são ignorados pelo compilador.

O compilador se limita a processar esses comentários em tempo de compilação, gerando ao final um arquivo de documentação XML que pode ser distribuído junto ao seu assembly .NET ou .NET Core. Esse arquivo permite que o Visual Studio use o IntelliSense para mostrar informações rápidas sobre tipos ou membros.

Arquivos de comentários XML também podem ser usados por ferramentas, como Swagger, DocFX, GhostDoc e Sandcastle para gerar sites de referência de API.

Para ativar a geração do arquivo de documentação XML, execute os seguintes passos:

1. Clique com o botão direito do mouse sobre o nome do projeto no *Gerenciador de Soluções* e selecione *Propriedades*.

2. A página de propriedades do projeto será exibida. Selecione a guia *Compilar* e ative a caixa de verificação *Arquivo de documentação XML*. Observe que você também pode alterar o local no qual o compilador grava o arquivo se desejar.

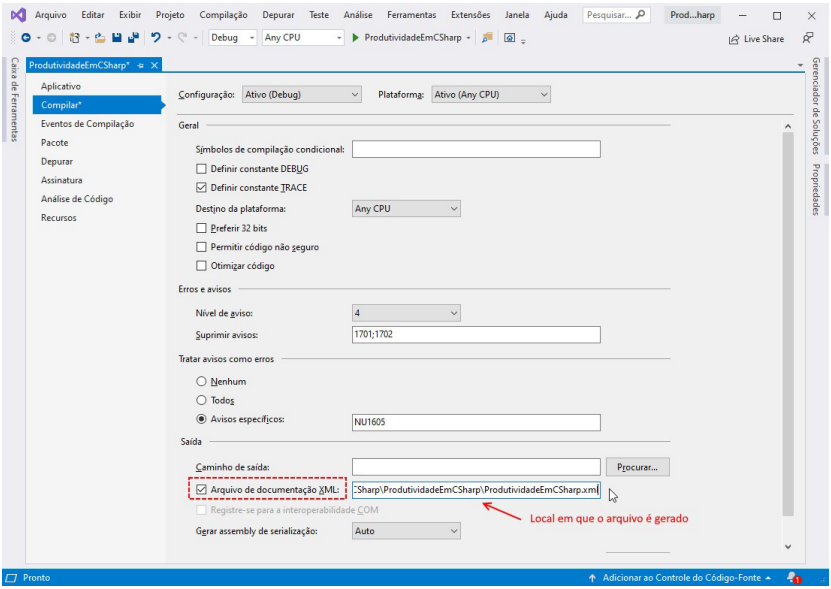


Figura 9.22: Ativando a geração do arquivo de documentação XML de um projeto no Visual Studio

3. Tecle *Ctrl* + *s* ou clique no botão *Salvar* da barra de botões do Visual Studio para efetivar a alteração. Feche, a seguir, a página de propriedades.

Para incluir comentários de documentação XML em seu código, basta parar com o cursor de inserção do editor de código na linha acima do tipo ou membro que deseja documentar e digitar barras triplas (`///`). Exemplo:

Figura 9.23: Comentário XML para um tipo preenchido pelo desenvolvedor

Conforme você pode observar, os comentários da documentação XML usam barras triplas e um corpo de comentário no formato XML. As tags que vão compor o XML gerado dependerão do tipo de elemento que está sendo documentado e se ele possui parâmetros. Veja na próxima imagem o esqueleto de documentação gerado pelo Visual Studio para um método:

```
/// <summary>
///
/// </summary>
/// <param name="mensagem"></param>
/// <param name="cor"></param>
1 referência
public static void Escrever(string mensagem, ConsoleColor cor)
{
    throw new NotImplementedException();
}
```

Figura 9.24: Esqueleto de comentário XML para um método

E, a seguir, a documentação XML já preenchida pelo desenvolvedor:

```
/// <summary>
/// Método estático para escrever uma mensagem colorida no Console.
/// </summary>
/// <param name="mensagem">Mensagem a ser exibida.</param>
/// <param name="cor">Cor a ser usada na mensagem. A cor é representada
/// por uma constante do tipo enumerado System.ConsoleColor.</param>
1 referência
public static void Escrever(string mensagem, ConsoleColor cor)
{
    throw new NotImplementedException();
}
```

Figura 9.25: Comentário XML para um método preenchido pelo desenvolvedor

Ao usarmos o método `Escrever` em nosso código, veremos a documentação sendo exibida como *tooltip*. Observe:

```
0 referências
static void Main(string[] args)
{
    Escrever("Produtividade em C#",);
}
```

void Program.Escrever(string mensagem, ConsoleColor cor)
Método estático para escrever uma mensagem colorida no Console.
cor: *Cor a ser usada na mensagem. A cor é representada por uma constante do tipo enumerado System.ConsoleColor.*

Figura 9.26: Documentação XML sendo exibida para o método Escrever

E o melhor de tudo é que se mudarmos a assinatura do

método, adicionando, por exemplo, um parâmetro extra, seremos lembrados pelo Visual Studio de que é preciso atualizar a documentação XML. Observe:

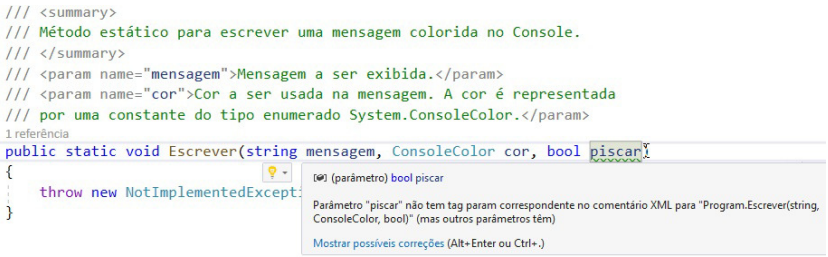


Figura 9.27: Sinalização de documentação XML desatualizada

Para solucionar o problema, clique em *Mostrar possíveis correções* e selecione no menu de contexto a opção *Adicionar nós de parâmetro ausentes*. Isso fará com que seja gerado uma tag para o parâmetro `piscar` que falta ser documentado. Veja:

```
/// <summary>
/// Método estático para escrever uma mensagem colorida no Console.
/// </summary>
/// <param name="mensagem">Mensagem a ser exibida.</param>
/// <param name="cor">Cor a ser usada na mensagem. A cor é representada
/// por uma constante do tipo enumerado System.ConsoleColor.</param>
/// <param name="piscar"></param>
1 referência
public static void Escrever(string mensagem, ConsoleColor cor, bool piscar)
{
    throw new NotImplementedException();
}
```

Figura 9.28: Incluindo o parâmetro ausente na documentação XML

Um efeito colateral da ativação dos comentários XML é que o Visual Studio passará a sinalizar todo o código C# do seu projeto que não contém comentários. Veja:

Figura 9.29: Sinalização de comentário XML ausente para um tipo

Isso obviamente é algo irritante e indesejável para a maioria dos desenvolvedores. Para desativar esse aviso, volte novamente à guia *Compilar* da página de propriedades do projeto, adicione o número `1591` no campo *Suprimir aviso* da seção *Erros e avisos* e salve o projeto. Caso já exista outro número neste campo, separe os dois números usando o caractere de ponto e vírgula.

Agora que você já tem o seu projeto devidamente configurado, já pode iniciar a documentação usando comentários XML. Estão disponíveis por default as seguintes tags: `<c>`, `<para>`, `<see>`, `<code>`, `<param>`, `<seealso>`, `<example>`, `<paramref>`, `<summary>`, `<exception>`, `<permission>`, `<typeparam>`,

`<include>`, `<remarks>`, `<typeparamref>`, `<list>`, `<returns>` e `<value>`.

Não nos aprofundaremos aqui no uso de cada uma dessas tags pois o nosso objetivo é apenas mostrar o caminho a ser trilhado. Você encontrará a documentação detalhada de cada tag a partir da lista disponível na seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/xmldoc/recommended-tags-for-documentation-comments

Para saber mais sobre documentação de código XML acesse também a seguinte página:

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/xmldoc/

As ferramentas externas de geração de documentação mencionadas nesta seção estão disponíveis em:

DocFx: https://dotnet.github.io/docfx/

GhostDoc: https://submain.com/products/ghostdoc.aspx

SandCastle: https://www.microsoft.com/download/details.aspx?id=10526

Swagger: https://swagger.io/

Vale destacar que algumas ferramentas, como o SandCastle oferecem suporte a tags extras como `<event>` e `<note>`. Nada impede que você também defina suas próprias tags, desde que elas sejam reconhecidas pelo programa que escolher para processar o XML gerado.

9.10 BAIXANDO TEMPLATES DE PROJETO DO VISUAL STUDIO MARKETPLACE

O Visual Studio é uma IDE extremamente poderosa e extensível. Graças à sua arquitetura modulável, é possível adicionar suporte a um grande número de linguagens e frameworks e baixar novos templates de projetos e pacotes para a linguagem desejada à medida que se tornarem necessários. Dentre outras vantagens, isso reduz o tempo e o espaço em disco necessário para a instalação do Visual Studio.

O conjunto inicial de templates de projeto de C# que acompanha o Visual Studio foi projetado para ser de uso geral, o que significa que parte deles oferece apenas o código inicial necessário para se começar a trabalhar. Não há como negar que esse esqueleto inicial já poupará muito trabalho, mas haverá cenários em que você desejará ir além e partir de um modelo de projeto mais elaborado que ofereça recursos avançados, como, por exemplo, suporte a log e autenticação. Nesses casos, é possível recorrer a templates criados por outros membros da comunidade e disponibilizados em sites, como o *Visual Studio Marketplace* ou o *GitHub*.

Nesta seção, veremos como baixar novos templates para o C# no Visual Studio Marketplace. Note que podemos fazer a busca

diretamente no site de forma semelhante à que fizemos com os snippets de código ou pesquisar os templates on-line por dentro do próprio Visual Studio.

Esta segunda alternativa é a mais produtiva, pois nos permite filtrar os resultados com maior facilidade e acessar a página do template quando for conveniente. Para comprovar, basta executar os seguintes passos:

1. No menu *Extensões* selecione a opção *Gerenciar extensões*.

2. A caixa de diálogo *Gerenciar extensões* será exibida. Utilize a caixa de pesquisa localizada na parte superior direita para procurar uma palavra-chave como, por exemplo, a palavra `bot` ou navegue pelo painel da esquerda até *Online* > *Visual Studio Marketplace* > *Modelos* > *Visual C#* e, a seguir, clique em uma das categorias de projetos disponíveis.

3. Selecione a extensão ou o template desejado. Clique no botão *Baixar* que será exibido. Após a instalação ser concluída, você verá um símbolo de check à direita do template instalado. Repare que no painel da direita é possível carregar a página web do template clicando em *Mais informações*.

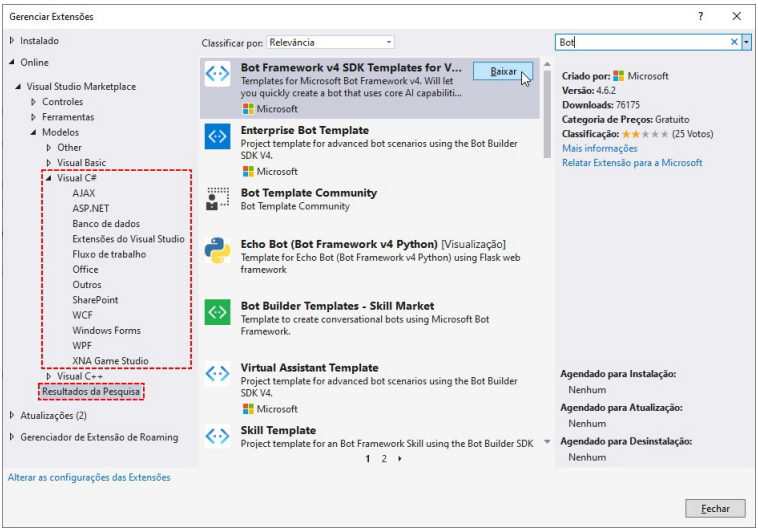


Figura 9.30: Pesquisando on-line novos templates para o Visual Studio

4. Clique no botão *Fechar* para encerrar a ferramenta de gerenciamento de extensões.

Uma vez instalado o novo template ou conjunto de templates, já poderemos usá-lo. Basta executar o seguinte roteiro:

1. Clique no menu *Arquivo* e selecione *Novo Projeto*.
2. A caixa de diálogo *Novo Projeto* será exibida. Para este exemplo, vamos pesquisar no campo de busca a palavra-chave `bot` para localizar todos os templates que possuam esta palavra em seus nomes.
3. Selecione o template desejado e clique em *Próximo*. A sequência de passos a partir deste ponto dependerá do tipo de projeto escolhido.

9.11 CRIANDO OS SEUS PRÓPRIOS TEMPLATES DE PROJETO

Ao trabalharmos no desenvolvimento de múltiplos sistemas, por vezes acabamos criando códigos que precisam estar presentes em vários projetos. Para lidar com esse tipo de problema, muitos desenvolvedores simplesmente copiam e colam o código de um projeto em outro, o que acaba gerando vários problemas ao longo do tempo.

Obviamente, o Visual Studio oferece maneiras mais inteligentes de encapsular e reutilizar o código previamente desenvolvido. Vejamos:

a) No caso de um projeto que represente o código inicial para novos desenvolvimentos, podemos gerar um novo template de projeto que ficará disponível posteriormente a partir da caixa de diálogo *Novo Projeto* do Visual Studio. Exemplo: um projeto em Angular 8 com código de back-end em C# (Web API) e referências para as principais bibliotecas necessárias, folhas de estilo em SASS, logo da empresa, componentes comuns (login, trocar senha, esqueci minha senha) etc.

b) No caso de uma biblioteca de classes, podemos gerar um *pacote NuGet* que poderá tanto ser tornado público quanto ser mantido privado para uso apenas na empresa.

A criação de um novo template de projeto a partir de um projeto existente é uma tarefa simples. Para simular esse cenário, execute os seguintes passos:

1. Crie um novo projeto do tipo aplicação de console em .NET

Core com o nome de `AppConsoleTemporizador`.

2. Altere o código do arquivo `Program.cs` conforme a listagem a seguir:

```
using System;
using System.Diagnostics;
using System.Threading;


namespace AppConsoleTemporizador
{
    class Program
    {

        static void Main(string[] args)
        {
            Stopwatch stopwatch = new Stopwatch();
            //Inicia cronômetro
            stopwatch.Start();

            //Código cujo tempo de execução desejamos medir
            for (int i = 0; i < 10; i++)
            {
                Thread.Sleep(1000);
            }

            //Para cronômetro
            stopwatch.Stop();

            //Tempo em horas, minutos e segundos
            Console.WriteLine("Tempo decorrido: {0:hh\\:mm\\:ss}"
, stopwatch.Elapsed);
        }
    }
}
```

Com esta mudança já temos o necessário para ilustrar a criação de um template. Se desejar, adicione um novo arquivo ao projeto, como um arquivo texto `leiame.txt`, para testar essa possibilidade.

Para criar o template, execute os seguintes passos:

1. No menu *Projeto*, selecione a opção *Exportar Modelo*.

2. O *Assistente de Exportação de Modelo* será exibido. A opção *Modelo de projeto* desejada já está selecionada por default. Clique no botão *Avançar*.

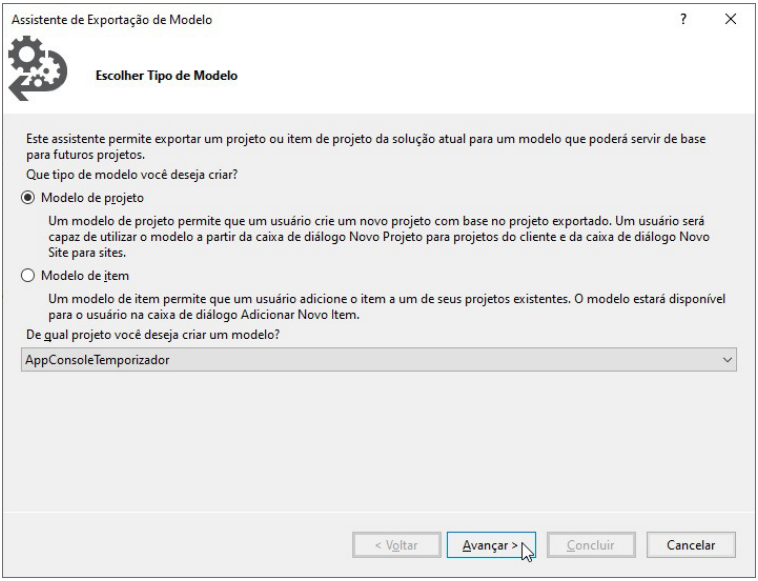


Figura 9.31: Assistente de Exportação de Modelo - Passo 1

3. No segundo passo, podemos fornecer um nome, descrição e ícone para o template. No campo *Nome do modelo*, digite `Aplicativo de Console .NET Core com Temporizador`. No campo *Descrição do modelo*, digite `Aplicação que utiliza a classe Stopwatch para medir o tempo de execução de um bloco de código`. Note que é possível fornecer um arquivo de imagem como ícone para o projeto.

4. Mantenha selecionada a opção *Importar automaticamente o modelo para o Visual Studio*. Observe que o campo *Local de saída* informa a pasta em que o template exportado será gravado como um arquivo compactado no formato `.zip`:

```
%USERPROFILE%\Documents\Visual Studio 2019\My Exported Templates\
```

Se você mantiver selecionada a opção *Exibir uma janela do gerenciador na pasta de arquivos de saída*, ao final da exportação o Explorador de Arquivos do Windows será carregado e exibirá a pasta que contém o template exportado, permitindo que você copie o template para outras máquinas.

Note que o Visual Studio utiliza internamente outra pasta para manter os templates criados pelo desenvolvedor, localizada em:

```
%USERPROFILE%\Documents\Documents\Visual Studio 2019\Templates\ProjectTemplates
```

Ao inspecionar esta pasta, você verá que existe uma estrutura de subpastas preparada para separar os templates por linguagens. Curiosamente, em nossos testes todos os templates foram criados diretamente na pasta `ProjectTemplates`.

Clique no botão *Concluir* para confirmar a criação do template.

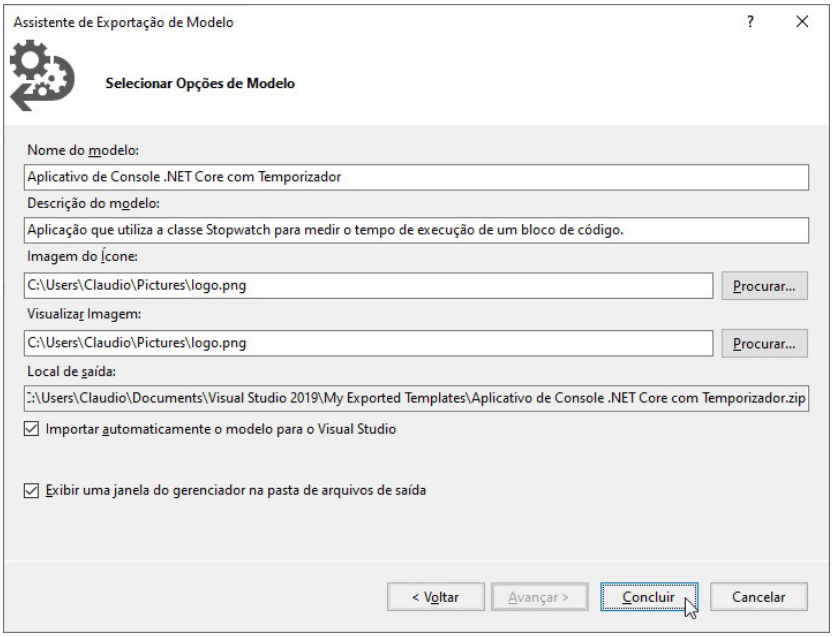


Figura 9.32: Assistente de Exportação de Modelo - Passo 2

Saiba que existem várias limitações no exportador de templates e na janela de criação de projetos que podem já ter sido resolvidas no momento em que você estiver lendo esta seção. No exportador de templates, faltam campos importantes que nos obrigam a descompactar o arquivo .zip, editar manualmente o arquivo de configuração `MyTemplete.vstemplate` para corrigir algumas configurações ausentes na interface gráfica, como o `<DefaultName>` e recompactar o template, se quisermos obter resultados mais profissionais.

O sistema de filtros e de tagueamento da caixa de diálogo *Novo Projeto* do Visual Studio 2019 ainda não funciona com os templates de projeto criados pelo desenvolvedor, o que significa

que teremos que procurar manualmente na lista pelo template que criamos com todos os filtros desligados. Confira o template que acabamos de criar na imagem a seguir:

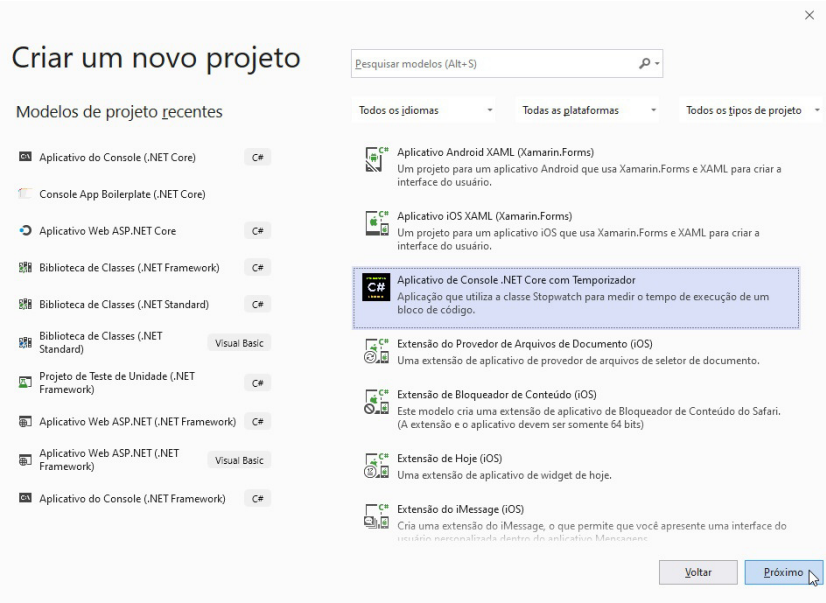


Figura 9.33: Caixa de diálogo Novo projeto exibindo o template de aplicação de console com temporizador

Para saber mais sobre a criação de templates de projeto para o Visual Studio 2019, acesse a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/visualstudio/ide/how-to-create-project-templates?view=vs-2019

A geração de código é um assunto extremamente vasto e em constante evolução. Neste capítulo, falamos sobre a criação automática de tipos e membros baseada em seu uso, a geração de classes a partir de dados em JSON e XML e a criação e uso de snippets de código e templates de projetos.

Nosso objetivo ao apresentar essas ferramentas foi o de despertar o seu interesse para o tema sem a pretensão de esgotá-lo. Reserve um tempo para pesquisar em fóruns, artigos e postagens de blogs quais são as melhores extensões, snippets de código, pacotes NuGet e templates de projeto disponíveis para simplificar o seu trabalho, pois quanto mais você procurar, mais opções interessantes pode descobrir.

Quem já trabalha com frameworks como o *Angular*, sabe que a ferramenta de linha de comando Angular CLI usada pelos desenvolvedores, gera cerca de metade de todo o código-fonte de um projeto de forma automática durante a sua construção. Conforme você já deve imaginar, isso poupa um esforço considerável e libera mais tempo para pesquisarmos no *Stack Overflow* quando algo não sai como esperávamos! :)

Capítulo 10

LIMPEZA DE CÓDIGO-FONTE

O Visual Studio 2019 simplifica o trabalho de reescrita do código legado fornecendo *analisadores de código*, um conjunto de *regras de estilo de código* para C# que podem ser versionadas e aplicadas a cada projeto em particular, *ações rápidas*, *ferramentas de refatoração*, *ferramentas de limpeza de código* e *ferramentas de relatório de métricas de código*.

Neste capítulo, veremos como configurar e exportar para arquivos `.editconfig` as regras de estilo de codificação que serão utilizadas pelos analisadores de código, pelas ações rápidas e pela ferramenta de limpeza de código. Abordaremos em seguida como configurar e efetuar a limpeza automática de código-fonte e como consultar as métricas de código geradas.

No final do capítulo, ainda explicamos como estender as funcionalidades dos analisadores de código e de ações rápidas e refatorações que abordaremos no próximo capítulo por meio de extensões gratuitas e pagas disponíveis no mercado. Para ilustrar o uso dessas extensões, demonstramos como instalar a extensão gratuita **Roslynator**, que acrescenta mais de 500 ações rápidas e refatorações ao Visual Studio.

Não se preocupe se tudo isso parecer novidade para você. Iniciaremos explicando todos os conceitos e ferramentas a serem utilizadas e em seguida passaremos à parte prática. Você verá que as funcionalidades que descreveremos serão empregadas de forma automática e imediata, no tempo de um estalar de dedos.

Tenha em mente que garantir a qualidade do código que está sendo produzido é um dever de todo mundo que desenvolve, pois o nosso código ainda é o nosso melhor cartão de visitas!

10.1 ENTENDENDO ANALISADORES, CORREÇÕES DE CÓDIGO E REFATORAÇÕES

Existem três conceitos básicos na análise de código que precisam ser entendidos para que possamos melhorar a qualidade do código de um projeto: *analisador de código*, *correção de código* e *refatoração*.

O *analisador* é executado no Visual Studio em segundo plano e ele analisa o código-fonte à medida que este vai sendo construído. Quando encontra um trecho de código que não está em conformidade com uma regra que precisa ser seguida (estilo de código), o analisador reporta um *diagnóstico*. O diagnóstico é exibido na IDE através de sublinhados no editor de código e na janela *Lista de erros* e pode ser corrigido de forma automática pelo desenvolvedor, se houver disponível uma *correção de código* (em inglês, *code fix*). As correções de código aplicáveis são mostradas no topo do menu que é exibido quando selecionamos no menu de contexto a opção *Ações Rápidas e Refatorações*, pois elas têm precedência sobre as refatorações. Os diagnósticos marcados como ocultos não são visíveis como rabiscos no editor de código.

Vale destacar que algumas correções de código podem fornecer a opção *Corrigir todas as ocorrências em*, que permite aplicar a mesma correção em múltiplos locais (documento atual, projeto e solução). Veja:

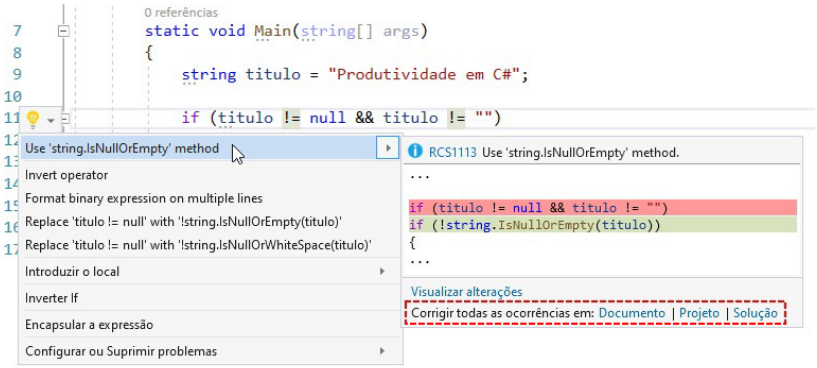


Figura 10.1: Opção para aplicar a mesma correção de código em múltiplos locais

A refatoração representa uma única operação fornecida sob demanda, como *extrair método*. Quando solicitada, o Visual Studio sugere uma lista de refatorações aplicáveis a um determinado intervalo do código.

Em termos práticos, você não precisa saber a diferença entre correções de código e refatorações para utilizá-las, especialmente porque as opções disponíveis são listadas no mesmo menu. A título de curiosidade, saiba que o Visual Studio as exibe de forma ligeiramente diferente. A correção de código é apresentada com um identificador e uma descrição no menu suspenso e também inclui o item *Suprimir* na parte inferior do menu de contexto. As refatorações são exibidas sem um identificador e descrição no menu suspenso.

10.2 IMPONDO REGRAS DE ESTILO DE CÓDIGO AOS PROJETOS

O Visual Studio é um ambiente de desenvolvimento altamente customizável e nos permite controlar de forma detalhada as convenções de estilo de código que desejamos empregar no código-fonte dos projetos. Essas regras são usadas para definir como os códigos das ações rápidas e da limpeza de código que veremos a seguir serão gerados.

Para acessar as configurações de estilo de código aplicáveis a projetos em C#, clique no menu *Ferramentas* e selecione *Opções*. A caixa de diálogo *Opções* será exibida. Navegue para as configurações de estilo de código em *Editor de texto* > *C#* > *Estilo de código*.

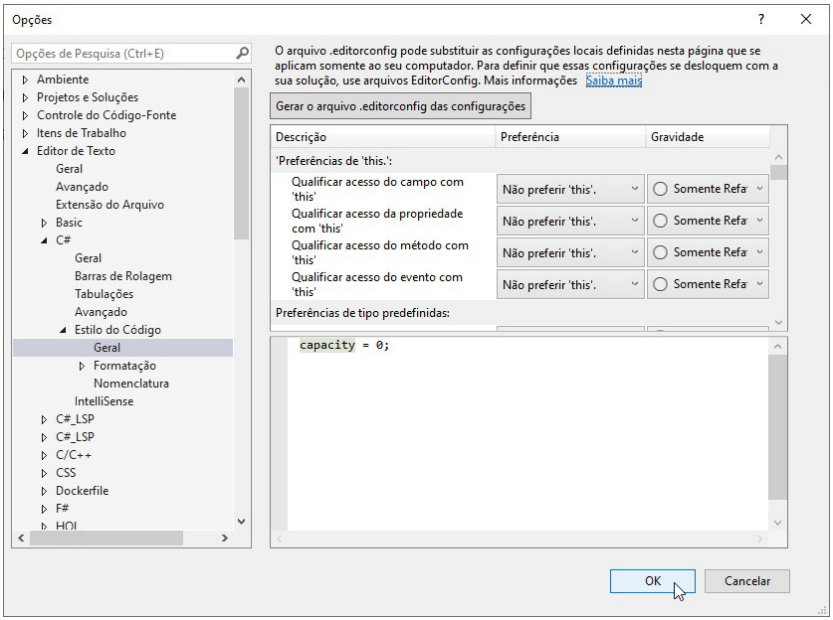


Figura 10.2: Opções de configurações de estilo de código para projetos em C#

Basta configurar as colunas *Preferência* e *Gravidade* para cada item do grid conforme as suas necessidades e clicar em seguida no botão *Ok* para efetivar a mudança. A *Gravidade* pode ser definida como *Somente Refatoração*, *Sugestão*, *Aviso* ou *Erro*. Para habilitar *Ações Rápidas* para um estilo de código, verifique se a configuração de Gravidade está definida como algo diferente de *Somente Refatoração*.

Note que dessa forma as configurações de estilo de código valerão apenas para a sua conta de personalização do Visual Studio, o que não é desejável quando estamos trabalhando em equipe.

Felizmente, o Visual Studio também suporta o uso de um arquivo `.editorconfig` contendo as convenções de codificação que serão aplicadas ao código-fonte de um projeto específico. Essa alternativa permite que as configurações de estilo de código sejam versionadas, partilhadas com outros desenvolvedores e reaproveitadas em projetos futuros.

Há várias formas de gerar o arquivo `.editorconfig` . A primeira é clicando com o botão direito do mouse sobre o nome do projeto no *Gerenciador de Soluções* e selecionando no menu de contexto o submenu *Adicionar* e, a seguir, a opção *Novo Item*. Na caixa de diálogo *Adicionar Novo Item* que será exibida, pesquise por `editorconfig` . Selecione qualquer um dos modelos de item de `Arquivo editorconfig` e clique no botão *Adicionar*.

A segunda maneira de gerar um arquivo `.editorconfig` é usando a caixa de diálogo *Opções* do Visual Studio. Navegue para as configurações de estilo de código em *Editor de texto* > *C#* > *Estilo de código*. Configure as opções de estilo de código conforme

desejado e a seguir clique no botão *Gerar o arquivo .editorconfig das configurações*. A caixa de diálogo *Salvar arquivo* será exibida. Clique no botão *Salvar* para gravar o arquivo. Em seguida, clique no botão *OK* para fechar a caixa de diálogo *Opções*. Esta é a forma mais rápida e produtiva de se gerar o arquivo do zero.

Uma terceira maneira de se adicionar o arquivo de configuração de estilos é importar um arquivo `.editorconfig` previamente criado para um projeto anterior. Experimente carregar o arquivo `.editorconfig` no editor de código do Visual Studio e verá que ele não contém nada que o vincule ao projeto em si, apenas configurações de estilo.

10.3 EXECUTANDO LIMPEZA DE CÓDIGO

As opções de configuração de estilo que configuramos na seção anterior podem ser aplicadas no código em C# usando os comandos *Limpeza de código* do Visual Studio 2019 e *Formatar documento* do Visual Studio 2017.

A limpeza de código formata seu arquivo ajustando detalhes como a indentação e a remoção de espaços extras no final de uma linha. Aplica também os estilos de código definidos no arquivo `EditorConfig` , caso tenha sido criado um para o projeto ou as configurações de estilo de código disponíveis na caixa de diálogo *Opções*.

Para realizar a limpeza de código no Visual Studio 2019 de um arquivo de código específico, clique no ícone de vassoura na parte inferior do editor ou pressione Ctrl+K, Ctrl+E. Veja:

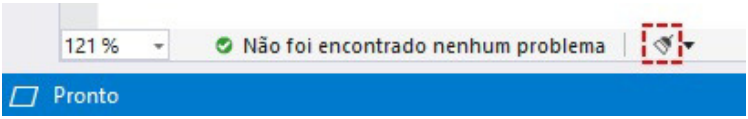


Figura 10.3: Botão de limpeza de código no Visual Studio 2019

Ao clicar na seta existente ao lado do botão de limpeza, você verá que é possível alternar entre dois perfis diferentes de limpeza e também acessar as opções de configuração:

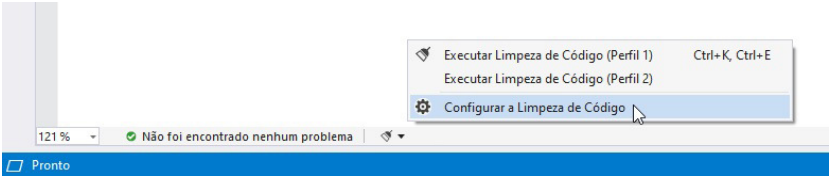


Figura 10.4: Acessando o menu de opções do botão de limpeza de código no Visual Studio 2019

À esquerda do botão de limpeza, temos uma outra novidade do Visual Studio 2019, o *indicador de saúde do documento*. Ele informa se não foi encontrado nenhum problema no documento atual, como no caso da imagem anterior, ou se foram encontrados erros e alertas, como no caso da próxima:

Figura 10.5: Aviso de erros e alertas no código

Nesse caso, é possível usar as setas para navegar entre os erros sinalizados ou clicar sobre o ícone de erros para carregar a janela *Lista de Erros*.

Na maioria dos casos, você vai desejar fazer a limpeza de código em todo o projeto ou em toda a solução. Para tanto, clique com o botão direito do mouse no nome do projeto ou no da solução no *Gerenciador de Soluções*, selecione o submenu *Análise e Limpeza de Código* e, a seguir, a opção *Executar Limpeza de Código (Perfil 1)*. Veja:

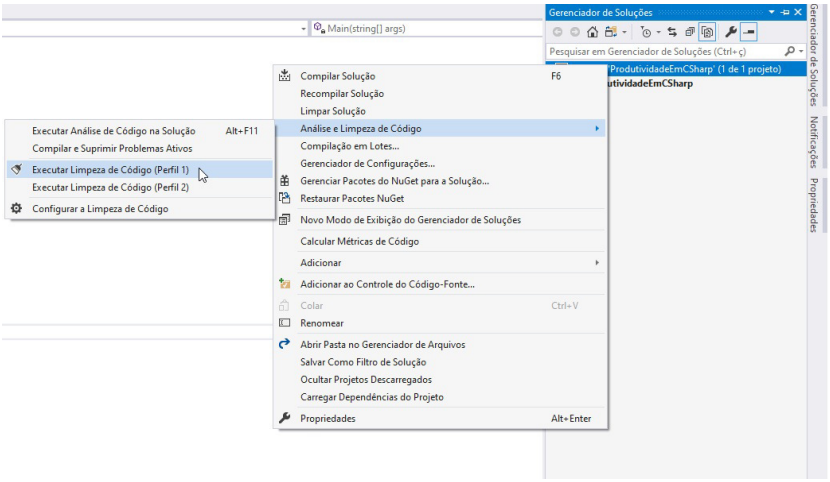


Figura 10.6: Executando limpeza de código em toda a solução

Para que a limpeza de código seja o mais eficiente possível, você deverá configurá-la de forma explícita com tudo que deseja que seja executado de forma automática usando a caixa de diálogo *Configurar a Limpeza de Código*. Selecione na lista *Reparados disponíveis* da janela todas as opções que deverão ser aplicadas e as mova para a lista *Reparadores incluídos* usando os botões de setas. Observe:

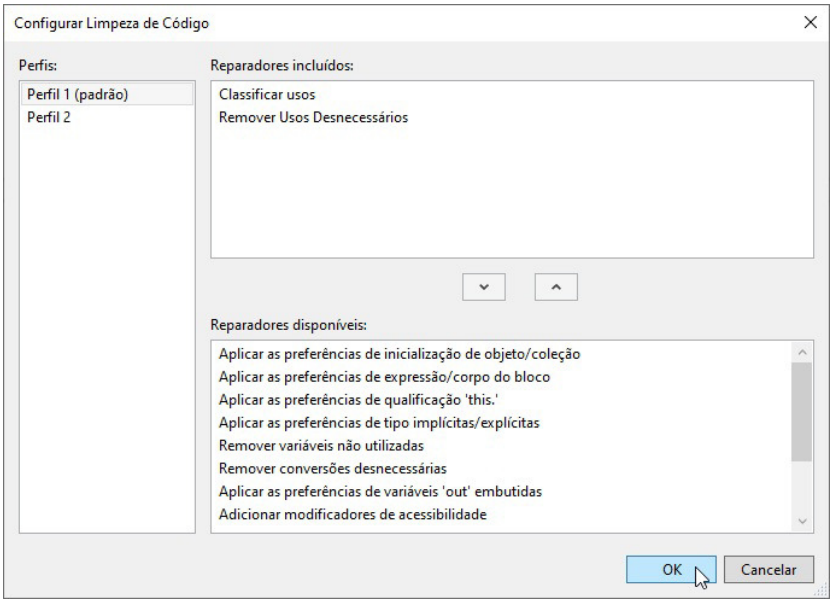


Figura 10.7: Configurando as opções a serem aplicadas na limpeza de código

Ao usar a limpeza de código, tenha em mente que ela não aplicará todas as *ações rápidas* e *refatorações* disponíveis nativamente no Visual Studio ou disponibilizadas por extensões de terceiros, devido às configurações de gravidade que mencionamos previamente e ao fato de que algumas ações envolvem fornecer um nome significativo para algo durante o procedimento.

As regras configuradas com uma gravidade de *Nenhum*, por exemplo, não participam da limpeza de código, mas ainda podem ser aplicadas de maneira individual através do menu *Ações rápidas* e *refatorações*.

De qualquer modo, a aplicação da limpeza de código corretamente configurada impactará de forma significativa no volume de trabalho que precisará ser realizado posteriormente de

forma manual pelo desenvolvedor.

10.4 VISUALIZANDO OS RESULTADOS DAS MÉTRICAS DE CÓDIGO

A janela *Resultados das Métricas de Código* exibe os dados gerados pela análise de métricas de código. Para exibir os resultados desses cálculos, execute os seguintes passos:

1. No menu *Analisar*, selecione o submenu *Calcular métricas de código* e, a seguir, a opção *Para a Solução*.
2. A janela será exibida com as métricas já calculadas. Expanda a árvore na coluna *hierarquia* para exibir detalhes de métricas de código.

Confira na imagem a seguir a janela de resultados de métrica para uma das aplicações de console usada como exemplo neste capítulo:

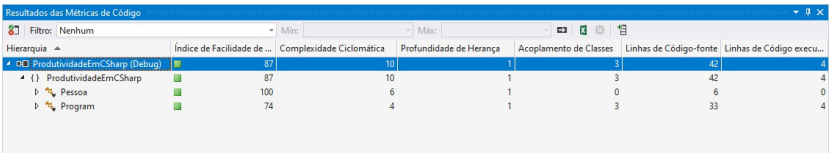


Figura 10.8: Exibindo os resultados das métricas de código

Vejamos o que cada uma das colunas de métricas de código significam:

- *Índice de Facilidade de Manutenção* (em inglês, *Maintainability Index*) - Calcula um valor de índice entre 0 e 100 que representa a relativa facilidade de manutenção do código. Um valor alto significa melhor facilidade de

manutenção. O valor ideal é 100.

- *Complexidade Ciclomática* (em inglês, *Cyclomatic Complexity*) - Mede a complexidade do código estrutural. Ele é criado pelo cálculo do número de diferentes caminhos de código no fluxo do programa (considere instruções ifs, switch/case, do, while). O valor ideal é 1.

- *Profundidade de Herança* (em inglês, *Depth of Inheritance*) - Indica o número de classes diferentes que herdam uma da outra, de volta para a classe base. Interfaces não são levadas em consideração. Quanto maior esse número, mais profunda a herança e maior o potencial para modificações na classe base que resultam em uma alteração significativa. Consequentemente, um valor baixo para esta métrica é bom e um valor alto é ruim. O valor ideal é 0.

- *Acoplamento de Classes* (em inglês, *Class Coupling*) - Mede o acoplamento de classes, ou seja, o quanto uma classe depende da outra. Devemos codificar de forma que tipos e métodos tenham alta coesão e baixo acoplamento. Um valor alto nesta métrica indica um design difícil de reutilizar e manter devido às suas muitas interdependências com outros tipos. O valor ideal é 0.

- *Linhas de Código-fonte* (em inglês, *Lines of Source code*) - Introduzida no Visual Studio 2019, esta métrica indica o número exato de linhas de código-fonte presentes no arquivo de origem, incluindo linhas em branco.

- *Linhas de Código Executável* (em inglês, *Lines of Executable code*) - Introduzida no Visual Studio 2019, esta métrica

indica o número aproximado de linhas ou operações de código executável. O valor calculado normalmente é próximo à métrica anterior, linhas de código, que é a métrica baseada em instrução MSIL usada no modo herdado.

Infelizmente, foge ao escopo deste livro uma abordagem mais profunda sobre esse assunto. Para saber mais sobre a janela resultados de métricas de código e o significado de cada métrica, use como ponto de partida as seguintes páginas da documentação oficial:

https://docs.microsoft.com/pt-br/visualstudio/code-quality/working-with-code-metrics-data?view=vs-2019

https://docs.microsoft.com/pt-br/visualstudio/code-quality/code-metrics-values?view=vs-2019

10.5 INSTALANDO EXTENSÕES DE INSPEÇÃO DE CÓDIGO E REFATORAÇÃO

Ao longo deste capítulo, vimos os principais recursos voltados para aplicação de regras de estilo e refatoração que estão presentes em todas as edições do Visual Studio. Estas ferramentas são suficientes para atender às necessidades da maioria dos desenvolvedores, mas existem várias extensões no mercado que podem suprir as necessidades mais específicas de alguns leitores.

Para aqueles que precisam de ferramentas mais completas de

inspeção de código e refatoração, a boa notícia é que existem excelentes produtos para o Visual Studio 2017 e Visual Studio 2019, tanto gratuitos quanto pagos. Dentre os gratuitos, podemos destacar:

- Roslynator https://marketplace.visualstudio.com/items?itemName=josefpihrt.Roslynator2019
- SonarLint para Visual Studio https://marketplace.visualstudio.com/items?itemName=SonarSource.SonarLintforVisualStudio2019
- StyleCopAnalyzers https://www.nuget.org/packages/stylecop.analyzers/
- CodeCracker https://www.nuget.org/packages/codecracker.CSharp/

Dentre as extensões pagas, destacam-se o *Resharper* da JetBrains e o *Visual Assist* da Whole Tomato que oferecem centenas de outros refactorings e um período de teste para que você possa avaliá-las:

- Resharper https://www.jetbrains.com/resharper/
- Visual Assist https://www.wholetomato.com/

A instalação dessas extensões pode ser feita diretamente pelo Visual Studio. Para testar essa alternativa, vamos instalar a extensão *Roslynator* executando os seguintes passos:

1. No Visual Studio 2019, as extensões ganharam um menu específico na IDE. Clique no menu *Extensões* e selecione a opção *Gerenciar Extensões*. No Visual Studio 2017, você encontrará esta opção no menu *Ferramentas*.

2. A caixa de diálogo *Gerenciar Extensões* será exibida. Note que a guia *Online* está ativa e que a ordenação default dos resultados é por popularidade. As extensões pagas que mencionamos já aparecem listadas nesta primeira página, pois elas estão há muito tempo no mercado e durante muitos anos foram as únicas opções disponíveis.

3. No campo *Pesquisar*, digite `Roslynator` para localizar a extensão gratuita mais popular (ela contém mais de 500 refactorings para C#).

4. O Visual Studio vai localizar a extensão. Clique no botão *Baixar*. A extensão será baixada e sua instalação será agendada para quando fecharmos o Visual Studio. Clique no botão *Fechar* para encerrar a caixa de diálogo *Gerenciar Extensões*.

5. Feche o Visual Studio para que a extensão *Roslynator* seja instalada e ativada. As funcionalidades desta extensão estarão disponíveis através do menu de ações rápidas. Veja a seguir um exemplo:

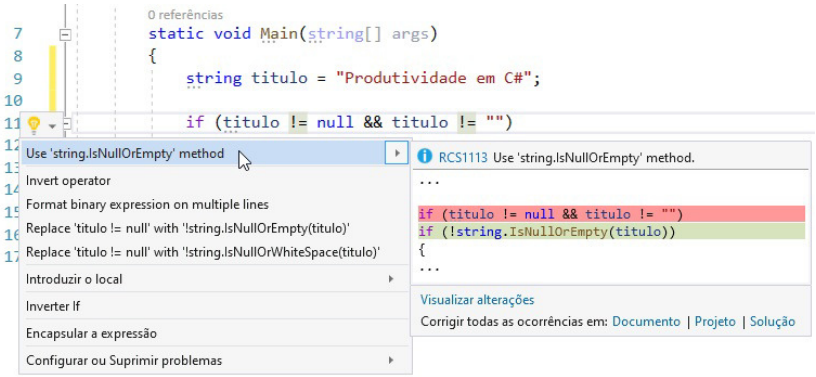


Figura 10.9: Aplicando uma alteração sugerida pelo Roslynator

Para configurar o *Roslynator* conforme as suas necessidades, execute os seguintes passos:

1. Clique no menu *Ferramentas* do Visual Studio e selecione *Opções*.
2. A caixa de diálogo *Opções* será aberta. Navegue até o submenu do *Roslynator* (neste caso específico, não utilize a caixa de pesquisa, pois se você buscar pelo termo `Roslynator` não verá a opção *Refactorings*).
3. Nos submenus *Refactorings* e *code fixes*, selecione as opções que deseja ativar. Observe que existe um campo de busca acima do grid que você pode utilizar para localizar mais facilmente uma regra específica que deseja ativar ou desativar.
4. Após efetuar todas as configurações desejadas, clique no botão *OK* para fechar a caixa de diálogo.

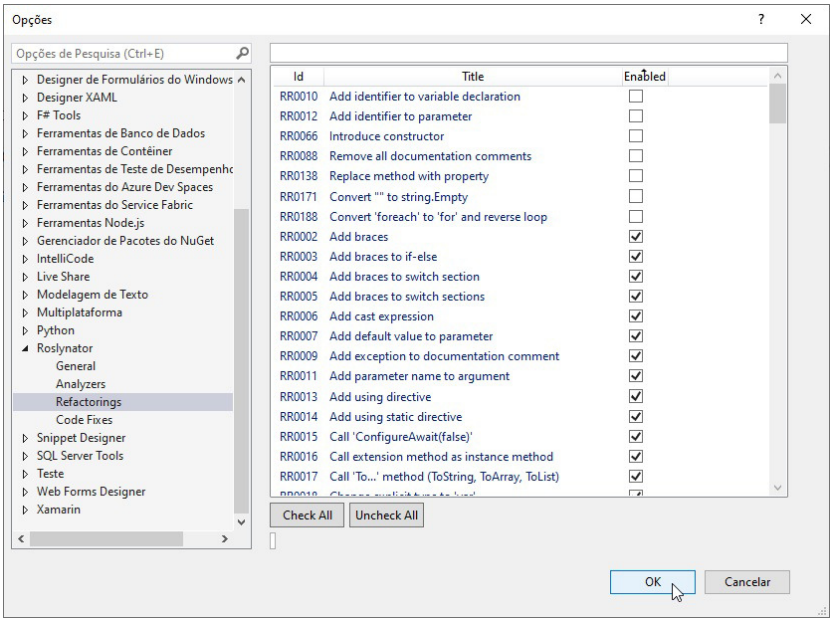


Figura 10.10: Configurando as refatorações ativas do Roslynator

Saiba que as ferramentas aqui sugeridas são apenas algumas dentre muitas gratuitas e pagas que foram construídas para otimizar e simplificar o nosso trabalho. Em fóruns como o Stack Overflow, em blogs técnicos sobre programação ou em tutoriais no YouTube, você encontrará sugestões de outras ferramentas fantásticas que merecem fazer parte do nosso dia a dia.

Capítulo 11

AÇÕES RÁPIDAS E REFATORAÇÃO

A reescrita de um código-fonte usando as melhores práticas para torná-lo mais fácil de manter, entender e estender, sem alterar seu comportamento, é conhecida como *refatoração* (em inglês, *refactoring*) e é uma tarefa cada vez mais presente na vida de um desenvolvedor ou desenvolvedora profissional.

Conforme vimos no capítulo anterior, o Visual Studio 2019 oferece a você, mesmo em sua versão gratuita, um grande leque de ferramentas que facilitam a reorganização de projetos já iniciados e a correção dos problemas encontrados. A lista inclui *analisadores de código*, *regras de estilo de código*, *ações rápidas*, *ferramentas de refatoração*, *ferramentas de limpeza de código* e *ferramentas de relatório de métricas de código*.

Neste capítulo, focaremos nas *ações rápidas* e *refatorações* para linguagem C#. Como existe um número muito grande de ações disponíveis nativamente na IDE, focaremos apenas nas mais importantes, ou seja, naquelas que você utilizará com maior frequência.

Explorando as ações rápidas

As *ações rápidas* (em inglês, *quick actions*) nos permitem corrigir, gerar mais código ou refatorar o código existente com um único comando. O recurso funciona da seguinte forma: o Visual Studio identifica locais em que o código está incorreto e locais em que o código pode ser simplificado e/ou melhorado e os sinaliza no editor de código com um rabisco ou sublinhado para o desenvolvedor. O desenvolvedor percebe que existe algo que pode ser modificado em um ponto específico do código e pode acessar o menu de ações rápidas de uma das seguintes formas:

a) Parando o mouse sobre o identificador ou palavra-chave sublinhada. Um ícone de marca inteligente de uma lâmpada ou chave de fenda será mostrado contendo uma pequena seta à direita que, ao ser clicada, mostra o menu de ações rápidas.

b) Clicando com o botão direito do mouse sobre o identificador ou palavra-chave. No menu de contexto que será exibido, selecione a opção *Ações Rápidas e Refatorações*. O menu de ações rápidas será exibido.

c) Parando com o cursor de inserção sobre o identificador ou palavra-chave e teclando *Ctrl* + . (ponto). O menu de ações rápidas será exibido.

É importante destacar que nem todos os locais onde uma ação rápida pode ser aplicada serão sinalizados de forma automática pelo Visual Studio. Além disso, em uma mesma linha de código podem existir diferentes ações rápidas que podem ser listadas dependendo de onde você parou o mouse ou o cursor de inserção antes de ativar o menu de ações rápidas.

Por meio de ações rápidas, é possível:

- Aplicar uma correção de código para uma violação de regra do analisador de código.
- Suprimir uma violação de regra do analisador de código ou configurar sua gravidade.
- Aplicar uma refatoração.
- Gerar um código a partir do seu uso.

Parte das ações rápidas suportadas pelo Visual Studio já foram explicadas no capítulo anterior. Por meio delas é possível implementar o recurso *Geração a partir do uso*, como vimos no capítulo 09. Nas próximas seções, focaremos em ações rápidas que aplicam correções no código existente e refatorações. Como o Visual Studio inclui um grande número de ações rápidas e algumas são muito específicas (e raramente utilizadas), abordaremos apenas as mais úteis.

Para conhecer as demais ações rápidas, acesse as seguintes páginas da documentação:

https://docs.microsoft.com/pt-br/visualstudio/ide/quick-actions?view=vs-2019

https://docs.microsoft.com/pt-br/visualstudio/ide/common-quick-actions?view=vs-2019

https://docs.microsoft.com/pt-br/visualstudio/ide/refactoring-in-visual-studio?view=vs-2019

Usando atalhos de teclado para refatorar

Parte das funcionalidades de refatoração de código suportadas pelo Visual Studio pode ser acessada via menu principal do Visual Studio ou através de atalhos de teclado (em inglês, *shortcuts*). No Visual Studio 2019, as funcionalidades ficam escondidas no submenu *Refatorar* do menu *Editar*.

Para facilitar a consulta e a memorização, enumeramos a seguir os atalhos de teclado usados por cada refactoring:

Refatoração	Atalho de teclado
Encapsular campo	Ctrl+R, Ctrl+E
Extrair interface	Ctrl+R, Ctrl+I
Extrair método	Ctrl+R, Ctrl+M
Remover parâmetros	Ctrl+R, Ctrl+V
Renomear	Ctrl+R, Ctrl+R
Reordenar parâmetros	Ctrl+R, Ctrl+O

Na maior parte dos casos, você acabará acessando esses recursos de refatoração por meio do menu de contexto ou pelo menu de ações rápidas. Memorizar os atalhos só será realmente útil se você tiver que realizar um trabalho extenso de refatoração de um sistema legado.

Agora que você já está familiarizado com os conceitos, configurações e ferramentas fornecidas pelo Visual Studio, é hora de nos divertimos um pouco testando várias funcionalidades disponibilizadas na forma de ações rápidas e refatorações que nos pouparão muito trabalho.

Classificando e removendo usings desnecessários

Um dos efeitos colaterais de utilizarmos templates de projetos e de arquivos é que acabamos muitas vezes com várias linhas de importação de namespaces que não são de fato usadas em nosso código. Estas importações são sinalizadas pelo Visual Studio em uma cor mais clara. Veja:

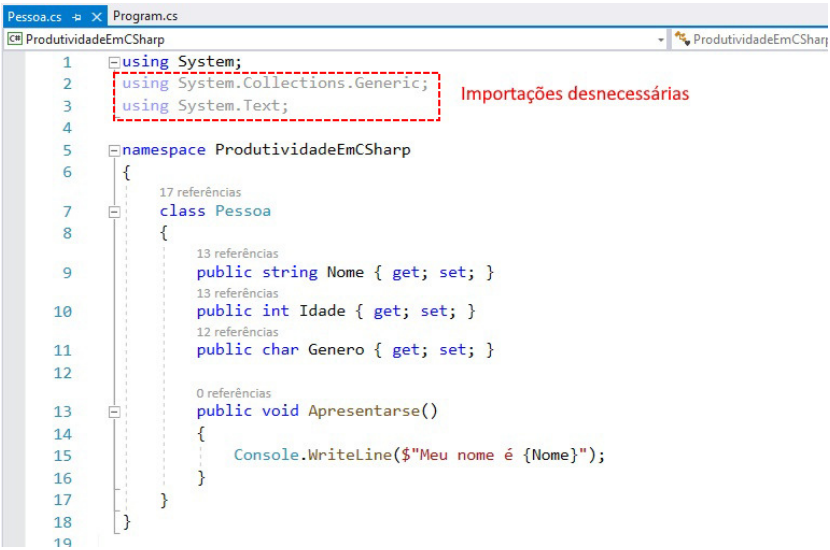


Figura 11.1: Sinalização de importações de namespaces não utilizados

Você pode remover manualmente cada linha desnecessária parando com o cursor de inserção sobre a linha e teclando *Ctrl + x* ou pode usar uma *Ação Rápida*. Basta clicar com o botão direito do mouse sobre qualquer linha referente às importações e selecionar no menu de contexto *Remover e Classificar Usos*. Esta ação removerá do arquivo atual qualquer importação de namespace não utilizado e, de quebra, ainda os ordenará.

Caso deseje apenas remover as importações desnecessárias, tecle *Ctrl* + . sobre uma das linhas de importação de namespace e selecione no menu de ações rápidas a opção *Remover Usos Desnecessários*.

Incluindo usings necessários

O Visual Studio é capaz de identificar quando tentamos utilizar em nosso código um tipo que está definido em assemblies do próprio framework .NET/.NET Core, pacotes NuGet ou outros namespaces do mesmo projeto ou de outros projetos da solução e cujo namespace não foi importado no arquivo atual.

Quando este cenário ocorre, a IDE nos oferece na ação rápida a inclusão da importação do namespace necessário. Veja:

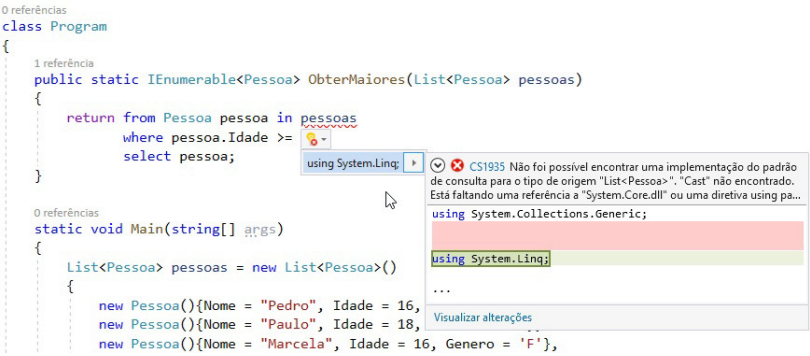


Figura 11.2: Incluindo de forma automática a importação de namespaces utilizados

Renomeando identificadores

Renomear identificadores de namespaces, tipos, membros e variáveis é uma tarefa extremamente simples e rápida no Visual Studio. Para ativar o recurso, basta clicar com o botão direito do

mouse sobre o identificador ou parar o cursor de inserção sobre ele e teclar *F2*. O identificador será selecionado e a janela da ferramenta *Renomear* será exibida no canto superior direito do editor de código. Basta digitar o novo nome para o identificador e teclar *Enter* ou clicar no botão *Aplicar*. Veja:

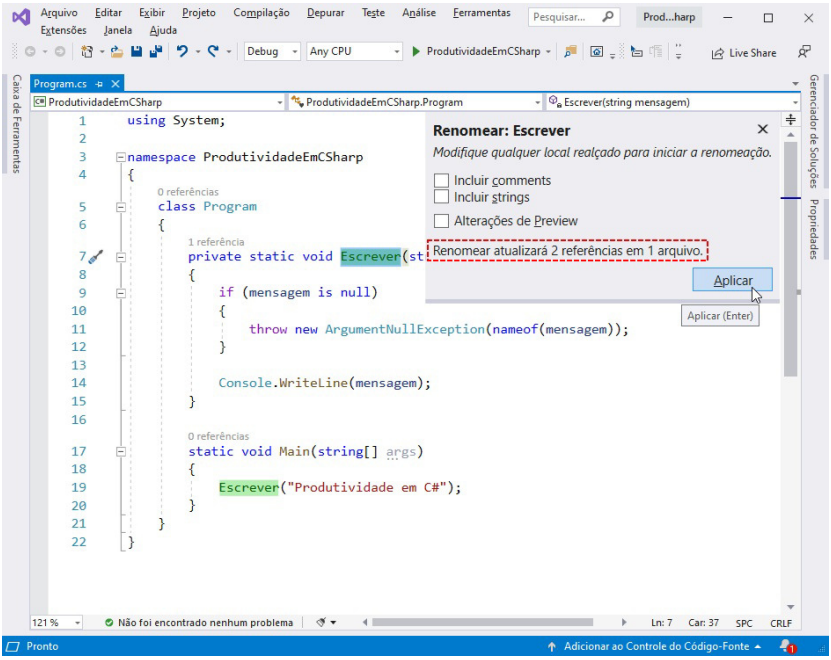


Figura 11.3: Utilizando a ferramenta Renomear para alterar o nome de um método

Conforme você observará nas próximas seções, esta ferramenta também é usada durante a aplicação de algumas ações rápidas para fornecer um nome significativo no lugar do nome original incluído pela ação durante a modificação do código existente. Ao usar a janela *Renomear*, você perceberá que ela informa em quantos locais ocorrerão substituição e informa também que existirão opções extras que serão incluídas, dependendo do tipo de

identificador que está sendo alterado. Por exemplo, ao usar a ferramenta para alterar o nome de uma classe, será exibida uma opção pré-selecionada que altera também o nome do arquivo que contém o código da classe.

11.1 DIVIDINDO O CÓDIGO DE UMA CLASSE EXTENSA EM MÚLTIPLOS ARQUIVOS

Criar pequenas classes e estruturas com poucas linhas de código facilita a leitura e manutenção do código. Essa abordagem é recomendada quando se adere aos princípios do SOLID, particularmente o *Princípio da Responsabilidade Única*. No entanto, em alguns casos, arquivos grandes são inevitáveis.

Apesar de ser possível aplicar *refatoração* do código em alguns cenários, pode ser conveniente em certos casos simplesmente quebrar a classe em múltiplos arquivos como uma solução intermediária para reduzir a complexidade. A linguagem C# torna isso possível ao adicionar a palavra-chave `partial` à definição da classe. Deste modo, teremos dois ou mais arquivos no quais a classe estará definida.

Para conferir o uso deste recurso na prática, vamos definir uma classe `Livro` e distribuí-la em dois arquivos, um contendo as propriedades, e o outro, os métodos. Nesta primeira listagem temos as propriedades que serão definidas no arquivo `Livro.Propriedades.cs`:

```
using System;

namespace ProdutividadeEmCSharp
{
    public partial class Livro
```

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

        public int Id { get; }

        public string Log { private get; set; }

        public string Titulo { get; set; } = "Produtividade em C#
;

        public string Autor { get; set; } = "Cláudio Ralha";

        public DateTime DataCadastramento { get; private set; } =
 DateTime.Now;

        public decimal Preco { get; private set; } = 50;
    }
}
```

No segundo arquivo, que chamaremos de `Livro.Metodos.cs`, temos o código dos métodos:

```
namespace ProdutividadeEmCSharp
{
    public partial class Livro
    {
        public string ObterResumo() {
            return "Produtividade em C# é um livro contendo um gr
ande número de dicas para facilitar o processo de codificação nas
 versões mais recentes da linguagem da Microsoft.";
        }

        public string ObterAutores()
        {
            return "Cláudio Ralha";
        }
    }
}
```

Confira a seguir os arquivos da classe `Livro` no *Gerenciador de Soluções*:

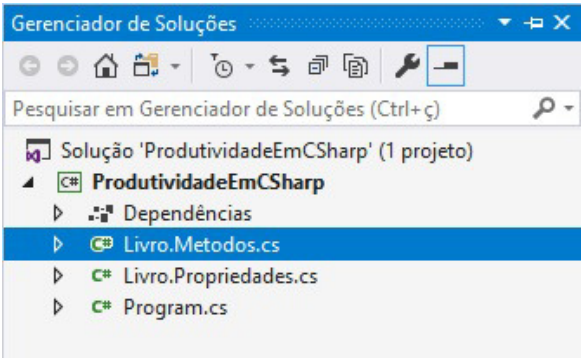


Figura 11.4: Particionando o código de uma classe em múltiplos arquivos

Obviamente, a forma de nomear os arquivos e de agrupar os membros da classe é apenas uma sugestão, você pode utilizar a convenção que for mais adequada para o seu caso.

Movendo um tipo para o arquivo correspondente

Ainda que não seja uma boa prática de programação, é possível definir em um único arquivo de código C# ou Visual Basic múltiplos tipos. Essa prática é comumente adotada quando se está criando algum código com o propósito de demonstração ou de prova de conceito e se deseja manter o máximo de simplicidade.

Para corrigir esse tipo de cenário, o Visual Studio disponibilizou a partir da versão 2017 um refactoring que move o tipo definido dentro do arquivo que contém múltiplas definições de tipos para um novo arquivo com o mesmo nome do tipo. Vamos ilustrar como usar este recurso, partindo do código do arquivo `Program.cs` mostrado a seguir:

```
using System;

namespace ProdutividadeEmCSharp
```

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


    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa = new Pessoa()
            {
                Nome = "Cláudio Ralha",
                Idade = 45
            };
            Console.WriteLine($"Nome: {pessoa.Nome}");
            Console.WriteLine($"Idade: {pessoa.Idade}");

        }
    }

}
```

Conforme você pode observar, a classe `Pessoa` foi definida no mesmo arquivo da classe `Program`. Para movê-la para um arquivo à parte, execute os seguintes passos:

1. Clique com o botão do mouse sobre o nome da classe e selecione no menu de contexto a opção *Ações Rápidas e Refatorações*.

2. No menu que será exibido, selecione a opção *Mover tipo para Pessoa.cs*.

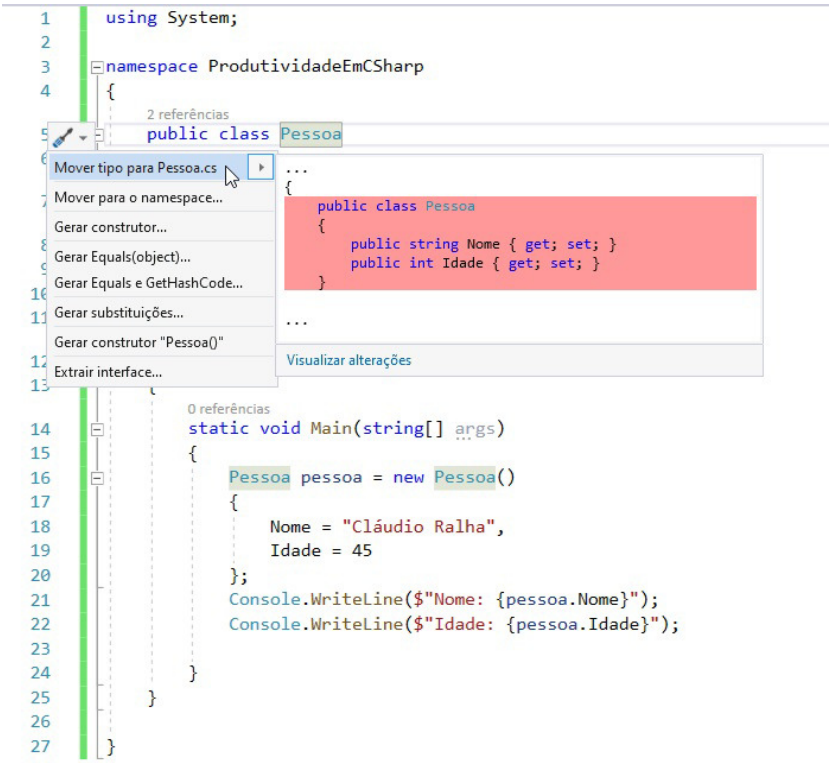


Figura 11.5: Movendo a classe para um arquivo exclusivo

Isso é tudo! Agora cada código está em seu devido lugar. Vale destacar que, se o arquivo `Pessoa.cs` já existisse, o arquivo previamente criado não seria usado. O Visual Studio iria criar um arquivo com o nome de `Pessoa1.cs` e salvar a definição do tipo nele.

Sincronizando o nome do tipo com o nome do arquivo

Há casos em que temos uma definição de tipo dentro de um

arquivo cujo nome não corresponde ao nome do tipo. A partir do Visual Studio 2017, é possível sincronizar o nome do tipo com o nome do arquivo através de novas refatorações suportadas pela IDE.

Vamos supor que tenhamos um arquivo nomeado como `Cliente.cs`, no qual está definida a classe `Pessoa`, cujo código está listado a seguir:

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

}
```

Para vermos essa refatoração em ação, execute os seguintes passos:

1. Clique com o botão direito do mouse sobre o nome de tipo. Selecione no menu de contexto a opção *Ações Rápidas e Refatorações*.

2. O menu da marca inteligente será exibido. Selecione a opção *Renomear tipo para Cliente*.

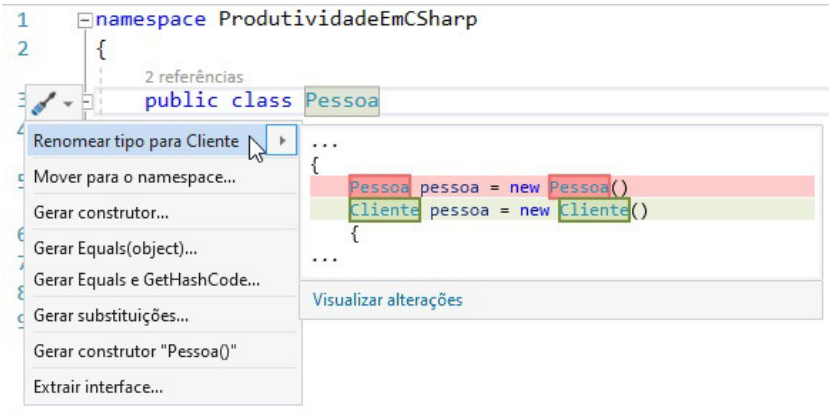


Figura 11.6: Renomeando o tipo para o nome do arquivo

Conforme esperado, a classe será renomeada para `Cliente`. Obviamente, para nomes curtos pode ser menos trabalhoso alterar manualmente o nome da classe para sincronizá-lo com o do arquivo, mas se o nome for complexo e longo, você vai agradecer ao time de desenvolvimento do Visual Studio por ter incluído essa funcionalidade.

É importante destacar que a Microsoft chegou a disponibilizar em versões de teste do Visual Studio 2017, uma opção que fazia o oposto, ou seja, renomeava o arquivo para o nome do tipo declarado (*Renomear arquivo para Pessoa.cs*, neste exemplo), mas essa refatoração não está mais disponível.

Sincronizando namespaces e nomes de pastas

Sincronizar os nomes de namespaces com os nomes e a estrutura de pastas em uso nos projetos nos ajuda a manter o código organizado de forma mais intuitiva. Para ilustrar este cenário, vamos criar uma classe `Pessoa` em um projeto de

aplicação de console:

```
namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }
}
```

O arquivo `Pessoa.cs` está localizado originalmente na pasta raiz. Confira na imagem a seguir:

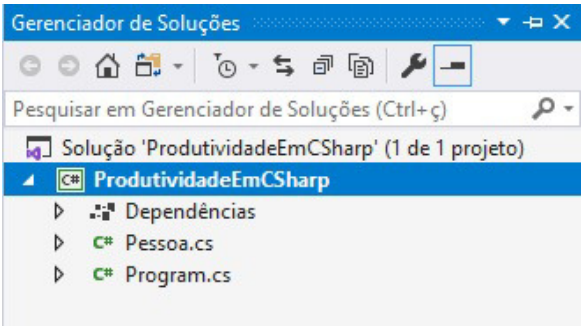


Figura 11.7: Estrutura de arquivos inicial do projeto

Em determinado momento do projeto, você decidiu que criaria uma pasta `Modelos` e, dentro dela, uma pasta `Entidades` para armazenar todas as classes de entidades do sistema. Para criar estas pastas, clique com o botão direito do mouse sobre o local desejado no *Gerenciador de Soluções* e selecione no menu de contexto o submenu *Adicionar* e a seguir a opção *Pasta*. Entre com o nome `Modelos`. Repita o processo fornecendo o nome `Entidades`. O resultado final pode ser visto a seguir:

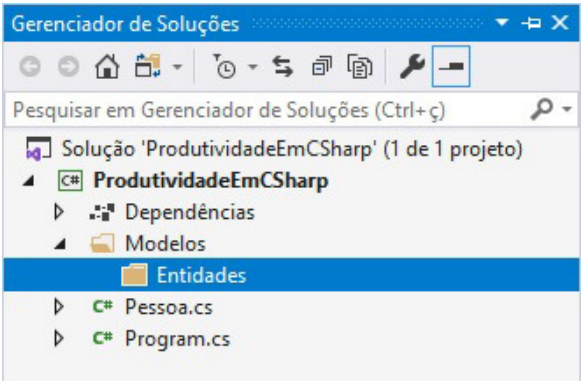


Figura 11.8: Estrutura de pastas criadas para organizar o código

Para mover o arquivo `Pessoa.cs` para a pasta `Entidades`, clique com o mouse sobre o arquivo `Pessoa.cs` no *Gerenciador de Soluções* e, enquanto mantém apertado o botão, arraste-o para o local desejado. Ao soltar o arquivo, uma caixa de confirmação será mostrada. Clique no botão *OK* para confirmar que deseja mover o arquivo.

Com o arquivo `Pessoa.cs` já no local adequado e aberto no editor de código, clique com o botão direito do mouse sobre o identificador do namespace e selecione no menu de ações rápidas a opção *Alterar o namespace para ProdutividadeEmCSharp.Modelos.Entidades*. Veja:

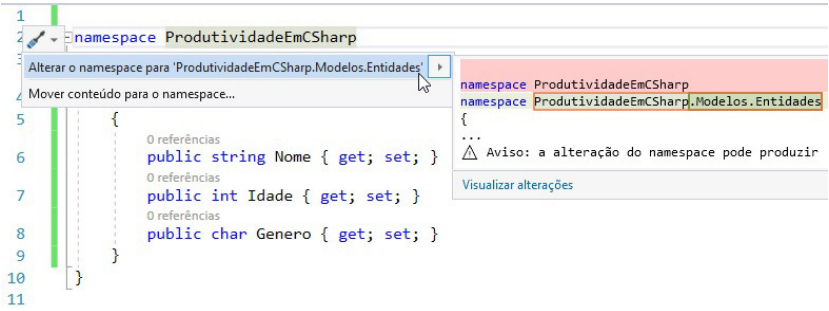


Figura 11.9: Aplicando a ação rápida de alteração de namespace

Pronto! O namespace foi ajustado sem risco de erro de digitação com o mínimo de esforço.

11.2 CONVERTENDO UM TIPO ANÔNIMO EM UMA CLASSE

O Visual Studio 2019 inclui uma ação rápida que nos permite converter um tipo anônimo em uma classe. Para testá-la, vamos utilizar o programa de exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var livro = new
            {
                Titulo = "Produtividade em C#",
                Autor = "Cláudio Ralha",
                Editora = "Casa do Código"
            };
            Console.WriteLine($"Título: {livro.Titulo}\nAutor: {l
ivro.Autor}\nEditora: {livro.Editora}");
```

```
        }
    }
}
```

Para testar esta ação rápida, clique com o botão direito do mouse sobre a palavra-chave `new` e selecione no menu de contexto *Ações Rápidas e Refatorações*. No menu seguinte, selecione *Converter em classe*. Veja:

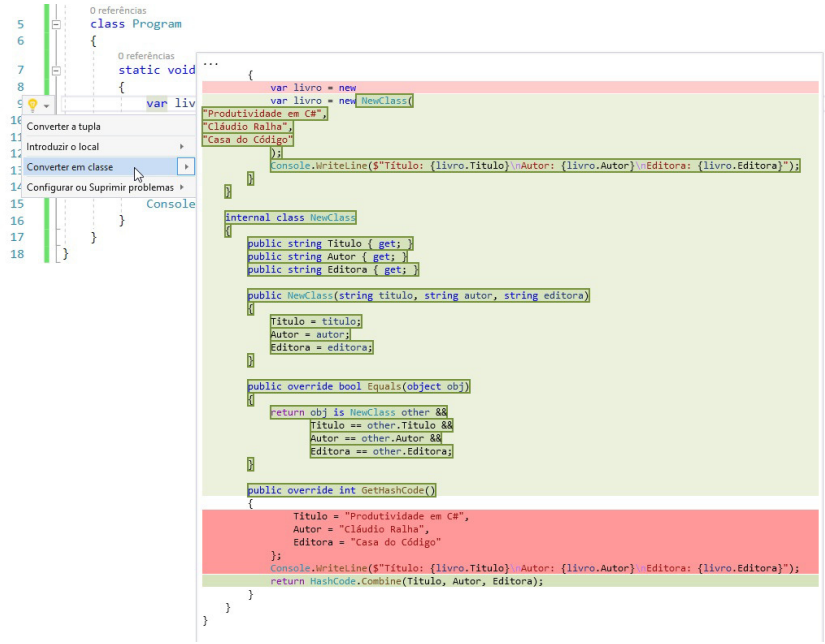


Figura 11.10: Convertendo um tipo anônimo em classe

A conversão será realizada e em seguida será ativada a ferramenta de Renomear para atribuir um nome mais intuitivo do que `NewClass` à classe. Para este exemplo, atribua `Livro`. O resultado final da conversão pode ser visto na listagem a seguir:

```
using System;
```

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var livro = new Livro(
                            "Produtividade em C#",
                            "Cláudio Ralha",
                            "Casa do Código"
            );
            Console.WriteLine($"Título: {livro.Titulo}\nAutor: {l
ivro.Autor}\nEditora: {livro.Editora}");
        }
    }

    internal class Livro
    {
        public string Titulo { get; }
        public string Autor { get; }
        public string Editora { get; }

        public Livro(string titulo, string autor, string editora)
        {
            Titulo = titulo;
            Autor = autor;
            Editora = editora;
        }

        public override bool Equals(object obj)
        {
            return obj is Livro other &&
                   Titulo == other.Titulo &&
                   Autor == other.Autor &&
                   Editora == other.Editora;
        }

        public override int GetHashCode()
        {
            return HashCode.Combine(Titulo, Autor, Editora);
        }
    }
}
```

Na maioria dos casos, será necessário algum ajuste manual para remover o código extra desnecessário.

11.3 PROMOVENDO UM MEMBRO DE UMA CLASSE A UMA INTERFACE OU TIPO BASE

O Visual Studio 2019 disponibiliza uma ação rápida que nos permite puxar um método definido em uma classe para uma interface que a classe implementa, ou para uma classe base da qual a classe que contém o método deriva.

Para simular um cenário em que este tipo de ação pode ser útil, vamos partir do código em desenvolvimento a seguir:

```
namespace ProdutividadeEmCSharp
{
    public interface ITributavel
    {

    }

    public abstract class Conta
    {
        public double Saldo { get; set; }
    }

    public class ContaPoupanca : Conta, ITributavel
    {
        public double CalcularTributo()
        {
            return Saldo * 0.05;
        }
    }

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

```
    }
}
```

Note que o programa que fornecemos nesta listagem não está completo. Vamos supor que, durante a codificação do método `CalcularTributo` na classe `ContaPoupanca`, você perceba que o método deve fazer parte da interface `ITributavel`, que deverá ser implementada em todos os tipos de contas que sofrerão tributação, incluindo `ContaInvestimento`, que ainda não foi codificada. O método em questão não pode fazer parte da classe abstrata `Conta`, pois há certos tipos de contas que não sofrem tributação.

Para promover o método `CalcularTributo` à interface `ITributavel`, basta clicar com o botão direito do mouse sobre o identificador e selecionar no menu de ações rápidas a opção. Veja:

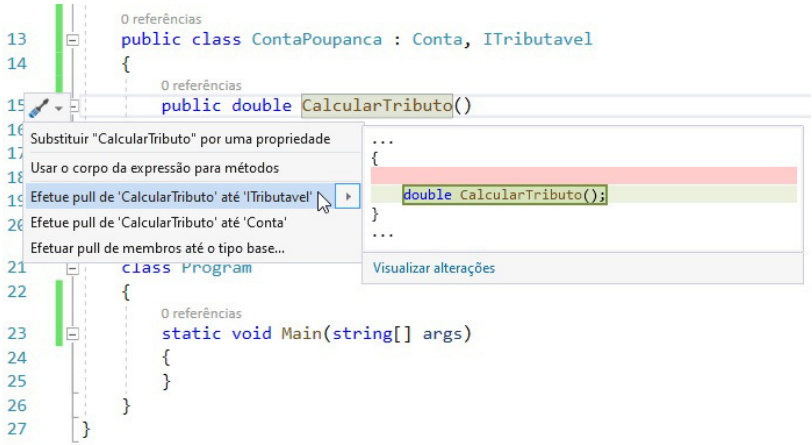


Figura 11.11: Promovendo o método CalcularTributo à interface ITributavel

Após a aplicação da ação, o código ficará como mostrado na próxima listagem:

```
namespace ProdutividadeEmCSharp
{
    public interface ITributavel
    {
        double CalcularTributo();
    }

    public abstract class Conta
    {
        public double Saldo { get; set; }
    }

    public class ContaPoupanca : Conta, ITributavel
    {
        public double CalcularTributo()
        {
            return Saldo * 0.05;
        }
    }

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

Note que agora o método `CalcularTributo` faz parte da interface `ITributavel` e sua implementação é feita na classe `ContaPoupanca`. Se tivéssemos promovido o método `CalcularTributo` à classe base `Conta` no lugar da interface `ITributavel`, o código do método seria movido para a classe base, ou seja, deixaria de existir em `ContaPoupanca`.

Nesta nossa simulação, a mudança pode parecer bem simples, pois todas as classes e interfaces foram codificadas no arquivo `Program.cs` para simplificar a explicação. Mas o real valor deste recurso é sentido quando aplicamos esta ação em projetos do mundo real, nos quais cada classe e interface está em um arquivo

diferente.

11.4 CONVERTENDO CAMPOS EM PROPRIEDADES

A ação rápida de encapsular um campo cria uma nova propriedade com o mesmo nome do campo. Para ilustrar o seu uso, vamos partir do exemplo da classe `Pessoa` a seguir:

```
namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome;
    }
}
```

Observe que a classe `Pessoa` utiliza um campo `Nome` no lugar de uma propriedade. Para converter este campo em uma propriedade, clique com o botão direito do mouse sobre o identificador da propriedade e selecione no menu de ações rápidas a opção *Encapsular campo "Nome" (e usar propriedade)*. Veja:

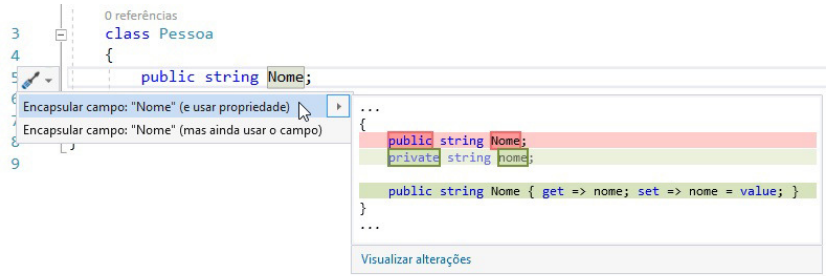


Figura 11.12: Convertendo um campo em uma propriedade

Confira o resultado na próxima listagem:

```
namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        private string nome;

        public string Nome { get => nome; set => nome = value; }
    }
}
```

Note que esta conversão provavelmente não agradará a maioria dos leitores e leitoras que esperavam ver uma conversão direta para uma *propriedade autoimplementada*. Para chegar a esse resultado, é necessário executar uma vez mais o menu de ações rápidas conforme descrito na próxima seção.

Convertendo propriedades em propriedades autoimplementadas

O C# suporta propriedades autoimplementadas como uma forma mais concisa de implementarmos propriedades. Na seção anterior, vimos que a conversão de um campo em uma propriedade usando uma ação rápida resultou na implementação de uma propriedade que ainda mantém a definição de uma variável interna declarada para armazenar o valor da propriedade.

Para converter a propriedade `Nome` em uma propriedade autoimplementada, clique com o botão direito do mouse sobre o identificador da propriedade e selecione no menu de ações rápidas, a opção *Usar a propriedade Auto*:

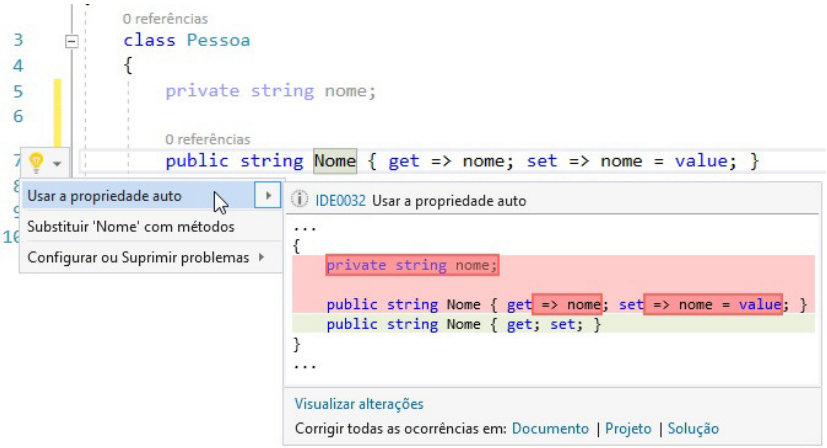


Figura 11.13: Convertendo uma propriedade em uma propriedade autoimplementada

Confira o resultado na próxima listagem:

```
namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
    }
}
```

Por enquanto, ainda não há como partir de um campo e chegar em uma propriedade autoimplementada usando uma única ação rápida, sem recorrer a um plugin externo. Todavia, esta é uma área do Visual Studio que vem recebendo melhorias constantes e é bem provável que essa restrição deixe de existir em breve.

11.5 PROMOVENDO FUNÇÕES LOCAIS A MÉTODOS

As *funções locais* foram introduzidas no C# 7.0 e permitem que

um método local seja definido e chamado dentro de outro método. Esse tipo de função pode ser declarada dentro de métodos, construtores, acessadores de propriedade, acessadores de eventos, métodos anônimos, expressões lambda, finalizadores e de outras funções locais.

Há desenvolvedores que apreciam a funcionalidade e outros que a consideram totalmente desnecessária. Confesso que me incluo no segundo grupo que nunca achou motivos reais suficientes para definir uma função local. A justificativa para as funções locais na documentação oficial é a seguinte:

"Funções locais tornam a intenção do seu código clara. Qualquer pessoa que ler seu código poderá ver que o método não pode ser chamado, exceto pelo método que o contém. Para projetos de equipe, elas também impossibilitam que outro desenvolvedor chame o método por engano diretamente de qualquer outro lugar na classe ou no struct". (Fonte: https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/local-functions.)

Para aqueles que não apreciam a funcionalidade, o Visual Studio 2019 disponibiliza uma ação rápida que permite converter uma função local em um método de uma classe.

Para testar este recurso, vamos utilizar o código de exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            int Somar(int a, int b)
            {
                return a + b;
            }

            Console.WriteLine(Somar(10, 20));
            Console.ReadKey();
        }
    }
}
```

Observe que dentro do método estático `Main`, ponto de entrada da aplicação, definimos a função local `Somar`. Para transformá-la em um método estático, basta clicar com o botão direito do mouse sobre o seu identificador e selecionar no menu de ações rápidas a opção *Converter em método*.

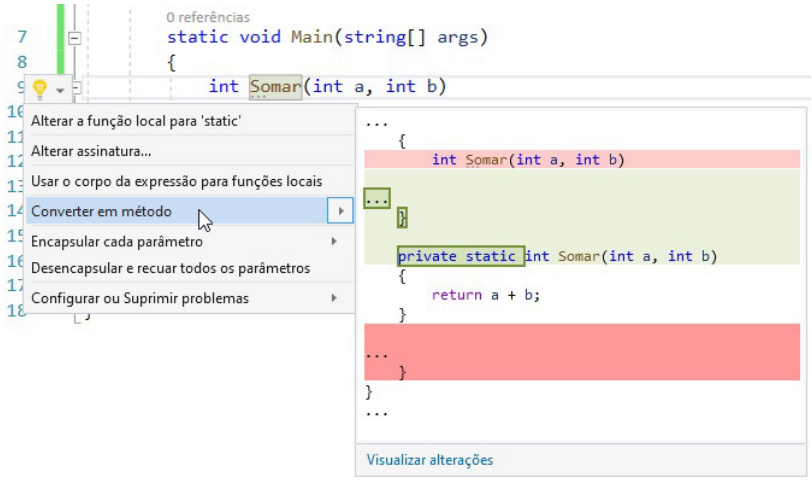


Figura 11.14: Convertendo uma função local em um método

Após a aplicação da ação, o código ficará como mostrado na próxima listagem:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Somar(10, 20));
            Console.ReadKey();
        }

        private static int Somar(int a, int b)
        {
            return a + b;
        }
    }
}
```

11.6 REORDENANDO OS PARÂMETROS DE UM MÉTODO

Ao criar bibliotecas de classes, você pode decidir *refatorar* seu código para fornecer uma ordem consistente de parâmetros. Quem já tentou realizar uma mudança desse tipo de forma manual em métodos que são largamente utilizados sabe o tempo que uma ação dessas consome e quão propensa a erros ela é. Felizmente, o Visual Studio dispõe de um excelente conjunto de ferramentas de refatoração.

Neste ponto, alguns leitores podem estar se perguntando: por que eu iria querer trocar a ordem de parâmetros de métodos que já estão bem testados? Existem alguns motivos principais, vejamos. Às vezes, ao ler o código existente, você encontrará métodos com parâmetros que são ordenados ilogicamente ou métodos em que os parâmetros aparecem em uma ordem diferente de outros métodos

semelhantes. Nesses cenários, pode ser desejável que os parâmetros apareçam em uma ordem consistente e lógica. Para atingir esse objetivo, será necessário atualizar a assinatura do método e o código de cada chamada para esse método.

A ferramenta *Reordenação de Parâmetros do Visual Studio* automatiza todo o processo. Em vez de atualizar manualmente o código, o comando fornece uma caixa de diálogo que permite alterar interativamente a ordem dos parâmetros de um método. Uma vez aceito, o Visual Studio modifica automaticamente o método e todas as referências a ele que aparecem na solução atual.

Para conferir como essa ferramenta pode ser usada para alterar a ordem dos parâmetros dos métodos existentes, partiremos do seguinte código de teste:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static void Escrever(string mensagem, bool capital
izar, ConsoleColor cor)
        {
            Console.ForegroundColor = cor;
            mensagem = capitalizar ? mensagem.ToUpper() : mensage
m;
            Console.WriteLine(mensagem);
        }

        static void Main(string[] args)
        {
            Console.BackgroundColor = ConsoleColor.Black;
            Console.Clear();
            Escrever("Produtividade em C#", true, ConsoleColor.Gr
een);
            Escrever("Cláudio Ralha", false, ConsoleColor.Yellow)
;
```

```
            Escrever("Aprenda o jeito moderno de programar e extr
aia o máximo da linguagem em pouco tempo!", false, ConsoleColor.W
hite);
        }

    }

}
```

Para utilizar a ferramenta de reordenação de parâmetros, execute os seguintes passos:

1. Clique com o botão direito do mouse sobre a declaração do método `Escrever` e selecione no menu de contexto a opção *Ações Rápidas e Refatorações*.

2. O menu da marca inteligente será então exibido. Clique na opção *Alterar Assinatura*.

3. A caixa de diálogo *Alterar Assinatura* será exibida. Observe que os parâmetros estão sendo exibidos em uma lista na mesma ordem da assinatura do método. Selecione o parâmetro desejado e use os botões de setas para movê-lo para a posição desejada. Repita esse procedimento para os demais parâmetros que desejar mover.

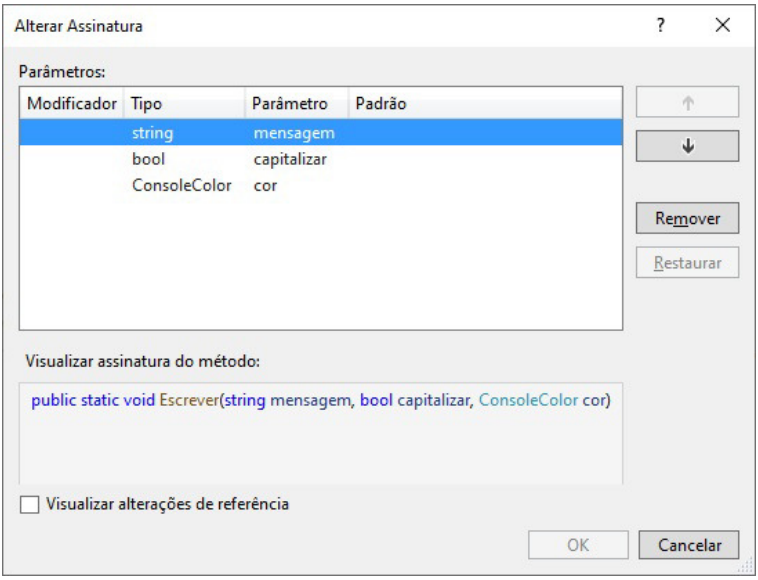


Figura 11.15: Acessando a ferramenta Alterar Assinatura

4. Para este exemplo, mudaremos apenas o parâmetro `cor` para a segunda posição. Quando estiver satisfeito com a nova assinatura, clique no botão *OK* para confirmar a mudança.

Isso é tudo! Simples, rápido e sem erros! Compile e execute o projeto para verificar que as alterações foram corretamente aplicadas a todos os projetos da solução atual.

11.7 ADICIONANDO NOMES DE ARGUMENTOS

No capítulo 4, apresentamos alguns exemplos de uso de parâmetros nomeados que ajudam a aumentar a legibilidade do código, especialmente quando um método recebe vários

argumentos do mesmo tipo.

O Visual Studio 2017 e superior inclui uma ação rápida que nos permite trabalhar com parâmetros nomeados. Para testar o recurso, vamos utilizar o programa a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {

        static void Main(string[] args)
        {
            var aniversario = new DateTime(1973, 4, 19);
            Console.WriteLine($"Eu nasci em uma {aniversario.ToLo
ngDateString()}!");
        }
    }
}
```

Observe que o construtor da classe `DateTime`, utilizado para instanciar o objeto, recebe três argumentos numéricos. Para identificarmos com facilidade qual deles representa o mês e qual representa o dia da data, clique com o botão direito do mouse sobre o primeiro argumento (neste exemplo, `1973`) e selecione no menu de ações rápidas a opção *Adicionar nome de argumento "year" (incluindo argumentos à direita)*. Veja:

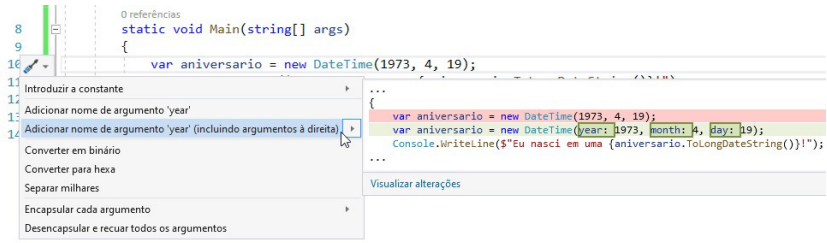


Figura 11.16: Incluindo nomes para os argumentos

Após aplicar a ação, a linha de código de declaração da variável `aniversario` ficará como mostrado a seguir:

```
var aniversario = new DateTime(year: 1973, month: 4, day: 19);
```

11.8 ADICIONANDO CHECAGEM DE NULOS PARA PARÂMETROS

O Visual Studio 2019 inclui ações rápidas que nos permitem verificar os parâmetros passados para um método ou construtor de classe. Para testar esse recurso, vamos utilizar o programa a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        private static void Escrever(string mensagem)
        {
            Console.WriteLine(mensagem);
        }

        static void Main(string[] args)
        {
            Escrever("Produtividade em C#");
        }
    }
}
```

Ao clicar com o botão direito do mouse sobre o parâmetro `mensagem` do método `Escrever` e acessar o menu de *Ações Rápidas*, veremos as seguintes opções:

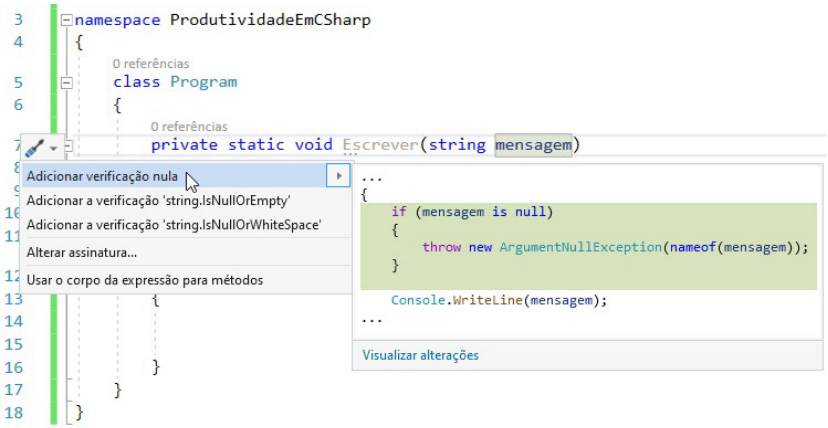


Figura 11.17: Incluindo checagem de nulo para o parâmetro mensagem

Note que as três opções relacionadas à adição de verificação nos permitem testar:

- Apenas por nulos.
- Por nulos ou string vazia.
- Por nulos, string vazia ou espaços em branco.

Confira a seguir o código do método `Escrever` após a refatoração usando a primeira opção, *Adicionar verificação nula*:

```
private static void Escrever(string mensagem)
{
    if (mensagem is null)
    {
        throw new ArgumentNullException(nameof(mensagem));
    }

    Console.WriteLine(mensagem);
}
```

Esse recurso se torna ainda mais útil e produtivo quando temos múltiplos parâmetros de tipos de referência no mesmo método.

Nesse caso, é mostrada uma opção extra que permite adicionar checagem de nulo para todos os parâmetros em uma única ação. Confira:

Figura 11.18: Incluindo checagem de nulo para todos os parâmetros

11.9 USANDO O COMANDO DE EXTRAÇÃO DE MÉTODO

Quando um método cresce além do esperado e sua manutenção se torna difícil, ou quando temos uma seção de um método que será duplicada em outras partes do nosso código, uma boa solução é considerar a extração de parte do código em seu próprio método. Para a nossa sorte, o Visual Studio pode automatizar esse processo de refatoração através do comando *Extrair Método*. Quando usado corretamente, o processo simplifica o código melhorando sua legibilidade e facilitando a sua manutenção, uma vez que incentiva a reutilização e reduz a duplicação de código.

O funcionamento do comando de extração de método é bem simples. Após o desenvolvedor selecionar um número de linhas dentro de um membro de uma classe, o Visual Studio vai extrair o

código em seu próprio método, substituindo as linhas originais por uma chamada para o novo membro.

Durante a geração do código, a ferramenta vai examinar o código que está sendo extraído para determinar se um ou mais valores de retorno serão necessários e para identificar quais variáveis devem se tornar parâmetros do novo método. Cada um desses itens é implementado automaticamente.

A forma mais simples de extração de método move uma seção de código reutilizável para seu próprio método. Este novo método não requer parâmetros e não retornará um valor. O método será privado e será um membro estático ou de instância para corresponder ao membro do qual o código foi extraído. Para ilustrar este cenário, vamos partir do código da próxima listagem:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.BackgroundColor = ConsoleColor.Black;
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Clear();
            Console.WriteLine("Produtividade em C#");
            Console.WriteLine("Cláudio Ralha");
            Console.WriteLine("Aprenda o jeito moderno de program
ar e extraia o máximo da linguagem em pouco tempo!");
        }

    }

}
```

Para extrair um conjunto de linhas de códigos e transformá-las

em um novo método, execute os seguintes passos:

1. Selecione as linhas desejadas. Elas precisam ser linhas consecutivas. Em nosso caso, vamos selecionar as seguintes linhas:

```
Console.BackgroundColor = ConsoleColor.Black;
Console.ForegroundColor = ConsoleColor.Green;
Console.Clear();
```

2. Clique com o botão direito do mouse sobre a seleção e selecione no menu de contexto a opção *Ações Rápidas e Refatorações.*

3. O menu da marca inteligente será exibido. Clique na opção *Extrair Método.*

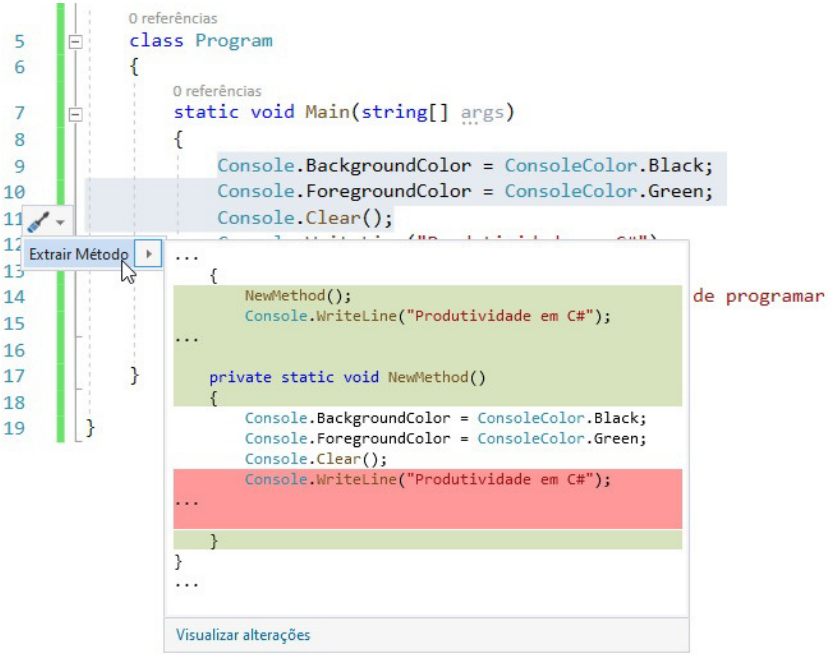


Figura 11.19: Aplicando a ação rápida de extração de método

4. O método recém-criado será nomeado como `NewMethod` e imediatamente a ferramenta de renomear método será ativada para que possamos trocar o nome do método.

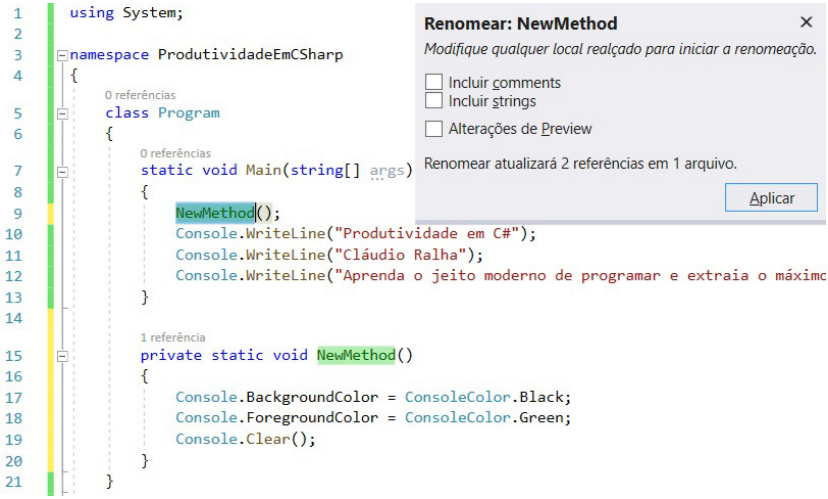


Figura 11.20: Acessando a ferramenta de renomear

5. Altere o nome do método deste exemplo para `ConfigurarTela` e a seguir clique no botão *Aplicar*.

Observe que o novo nome é assumido ao longo do código. Após a extração, o código selecionado foi movido para um novo método e substituído por uma chamada para esse método.

Em alguns casos, o novo método exigirá parâmetros e um valor de retorno. O Visual Studio analisará o código selecionado pelo desenvolvedor para determinar se esse é o caso e criá-los de acordo. Para testar esse cenário, vamos partir do código da listagem a seguir:

```
using System;
```

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Nome:");
            string nome = Console.ReadLine();
            Console.Write("Idade:");
            int idade = Convert.ToInt16(Console.ReadLine());
            string status = string.Empty;
            status = idade >= 18 ? "Maior de Idade" : "Menor de i
dade";
            Console.WriteLine(status);
        }

    }

}
```

Selecione as duas linhas a seguir e aplique a extração de método:

```
string status = string.Empty;
status = idade >= 18 ? "Maior de Idade" : "Menor de idade";
```

Observe que o novo método exigirá um parâmetro e retornará a string informando o status do usuário. Renomeie o método para `obterStatusMaioridade`:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Nome:");
            string nome = Console.ReadLine();
            Console.Write("Idade:");
            int idade = Convert.ToInt16(Console.ReadLine());
```

```
            string status = obterStatusMaioridade(idade);
            Console.WriteLine(status);
        }

        private static string obterStatusMaioridade(int idade)
        {
            string status = string.Empty;
            status = idade >= 18 ? "Maior de idade" : "Menor de i
dade";
            return status;
        }
    }

}
```

Em casos mais complexos, teremos códigos reutilizáveis que podem ser extraídos, mas somente se o novo método incluir mais de um valor de retorno. Quando esse for o caso, a operação de refatoração criará parâmetros de saída ou de referência para esses valores de retorno.

Conforme você pode observar, essa é uma daquelas ferramentas que poupa o desenvolvedor e desenvolvedora do trabalho braçal e de boa parte do trabalho intelectual de refatorar o código-fonte.

Removendo variáveis não utilizadas

Durante o desenvolvimento dos métodos de uma classe ou struct, por vezes declaramos variáveis que acabam não sendo usadas. O editor do Visual Studio nos avisa da presença de código desnecessário, sublinhando os identificadores das variáveis em verde, e oferece uma ação rápida que nos permite removê-las. Veja no exemplo a seguir a variável `resultado` declarada, porém não utilizada no método `Somar`:

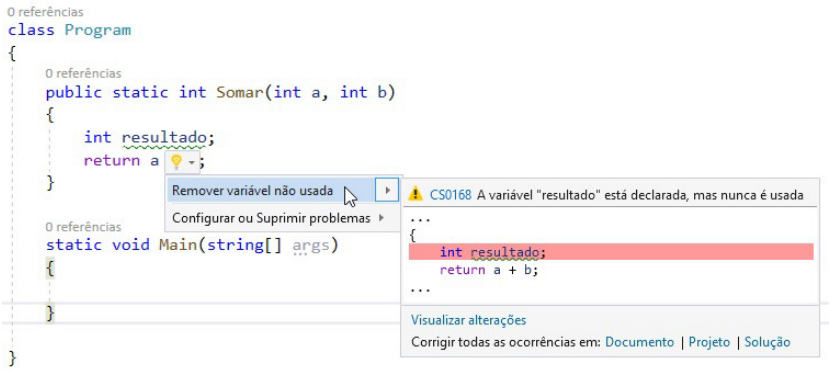


Figura 11.21: Removendo variáveis não utilizadas

Como alternativa à ação rápida, você pode parar com o cursor de inserção na linha que contém a variável não usada e teclar *Ctrl + x* para excluí-la.

11.10 CONVERTENDO STRING.FORMAT EM STRING INTERPOLADA

Ao efetuarmos manutenções em código legado, por vezes esbarramos com o uso do método `Format` da classe `string` para a montagem de cadeias de caracteres complexas. Essa era a forma recomendada para a montagem de strings antes da linguagem passar a suportar a interpolação de strings abordada no capítulo 1 deste livro.

O Visual Studio 2017 e versões posteriores oferecem uma ação rápida que permite converter o código montado com o método `Format` em uma string interpolada. Para testar o recurso, vamos partir do exemplo a seguir:

```
using System;
```

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            string titulo = "Produtividade em C#";
            string autor = "Cláudio Ralha";
            string editora = "Casa do Código";
            string mensagem = string.Format("Obrigado por adquiri
r o livro {0}, escrito por {1} e publicado pela editora {2}", tit
ulo, autor, editora);
            Console.WriteLine(mensagem);
        }
    }
}
```

Para testar a refatoração, clique com o botão direito do mouse sobre o método `Format` e selecione a ação rápida *Converter para cadeia de caracteres interpolada*.

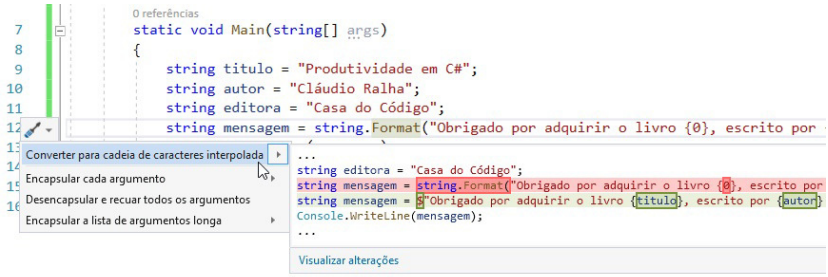


Figura 11.22: Substituindo string.Format por uma string interpolada

Como resultado, você obterá a seguinte string interpolada:

```
string mensagem = $"Obrigado por adquirir o livro {titulo}, escri
to por {autor} e publicado pela editora {editora}";
```

11.11 CONVERTENDO UM LOOP FOR EM UM LOOP FOREACH

Há casos em que desejamos simplificar o código, transformando um laço `for` em um laço `foreach`. Para cenários em que o loop `for` contém todas as três partes (inicializador, condição e iterador), o Visual Studio oferece a ação rápida *Converter em foreach*.

Para ilustrar como usar essa ação, vamos partir do exemplo a seguir:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var pessoas = new List<string>() {"Cláudio","Flávia",
"Iara"};
            for (int i = 0; i < pessoas.Count; i++)
            {

                Console.WriteLine(pessoas[i]);
            }
        }
    }
}
```

Ao examinar o código, note que o laço `for` contém todas as três partes e que a variável de controle do laço é usada apenas para iterar pelos itens, ou seja, este é um laço que pode ser simplificado. Para aplicar a ação, clique com o botão direito do mouse sobre a palavra-chave `for` e selecione no menu de ações rápidas a opção *Converter em foreach*. Veja:

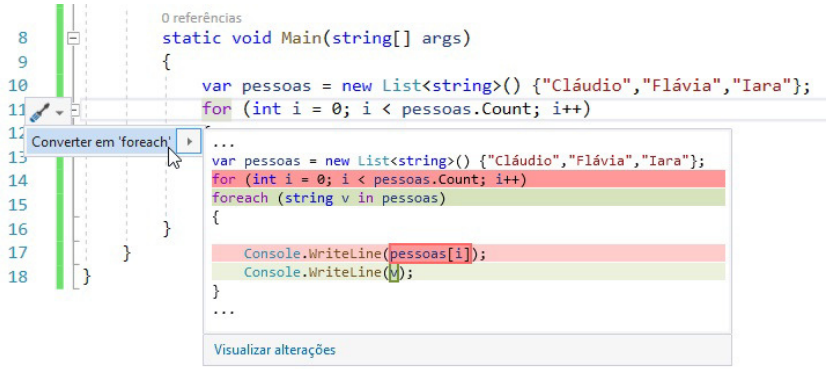


Figura 11.23: Convertendo um laço for em laço foreach

Após converter o laço `for` em laço `foreach`, a ferramenta *Renomear* será exibida para que possamos atribuir um nome significativo à variável de controle do loop. Neste exemplo, utilizamos `pessoa`. Confira o resultado final da conversão na próxima listagem:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var pessoas = new List<string>() {"Cláudio","Flávia",
"Iara"};
            foreach (string pessoa in pessoas)
            {

                Console.WriteLine(pessoa);
            }
        }
    }
}
```

11.12 CONVERTENDO UM LOOP FOREACH EM UM LOOP FOR

Em alguns cenários, pode ser necessário efetuar o caminho inverso que fizemos na seção anterior, ou seja, converter um loop `foreach` em um loop `for`. Para ilustrar essa possibilidade, podemos utilizar o próprio código gerado anteriormente.

Para aplicar a ação, clique com o botão direito do mouse sobre a palavra-chave `foreach` e selecione no menu de ações rápidas a opção *Converter em for*. Veja:

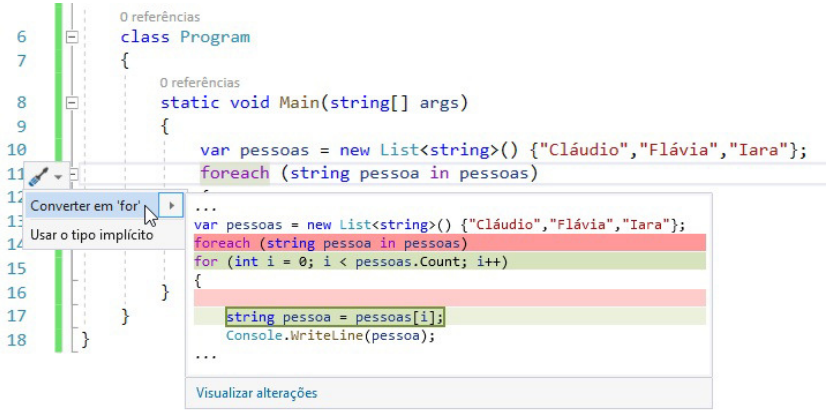


Figura 11.24: Convertendo um laço foreach em laço for

Confira na listagem a seguir o resultado da conversão:

```
using System;
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            var pessoas = new List<string>() {"Cláudio","Flávia",
"Iara"};
            for (int i = 0; i < pessoas.Count; i++)
            {
                string pessoa = pessoas[i];
                Console.WriteLine(pessoa);
            }
        }
    }
}
```

11.13 CONVERTENDO UM LOOP FOREACH EM LINQ OU EXPRESSÕES LAMBDA

Um dos refactorings rápidos introduzidos no Visual Studio 2019 permite converter um laço `foreach` em uma *consulta LINQ* ou *expressão lambda*. Para testar esse recurso, vamos utilizar o programa a seguir, que contém um loop `foreach` no método `ObterMaiores`. Veja:

```
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }

    class Program
    {
        public static IEnumerable<Pessoa> ObterMaiores(List<Pesso
a> pessoas)
        {
            foreach (Pessoa pessoa in pessoas)
            {
                if (pessoa.Idade >=18)
```

```
                    yield return pessoa;
                }
                yield break;
            }

            static void Main(string[] args)
            {
                List<Pessoa> pessoas = new List<Pessoa>()
                {
                    new Pessoa(){Nome = "Pedro", Idade = 16, Genero =
'M'},
                    new Pessoa(){Nome = "Paulo", Idade = 18, Genero =
'M'},
                    new Pessoa(){Nome = "Marcela", Idade = 16, Genero
 = 'F'},
                    new Pessoa(){Nome = "Roberta", Idade = 21, Genero
 = 'F'},
                    new Pessoa(){Nome = "Ricardo", Idade = 19, Genero
 = 'M'},
                    new Pessoa(){Nome = "Sofia", Idade = 14, Genero =
'F'},
                    new Pessoa(){Nome = "Vanessa", Idade = 22, Genero
 = 'F'},
                    new Pessoa(){Nome = "Rodrigo", Idade = 20, Genero
 = 'M'},
                    new Pessoa(){Nome = "Rebeca", Idade = 25, Genero
= 'F'},
                    new Pessoa(){Nome = "Henrique", Idade = 13, Gener
o = 'M'},
                    new Pessoa(){Nome = "Pâmela", Idade = 21, Genero
= 'F'},
                    new Pessoa(){Nome = "Alessandra", Idade = 19, Gen
ero = 'F' }
                };
                IEnumerable<Pessoa> maiores = ObterMaiores(pessoas);
            }
        }
}
```

Para converter o trecho do código do loop em uma expressão LINQ, execute os seguintes passos:

1. Clique com o botão direito do mouse sobre a palavra-chave

`foreach` e selecione no menu de contexto a opção *Ações Rápidas e Refatorações*.

2. O menu de ações rápidas será exibido. Selecione a opção *Converter para LINQ*.

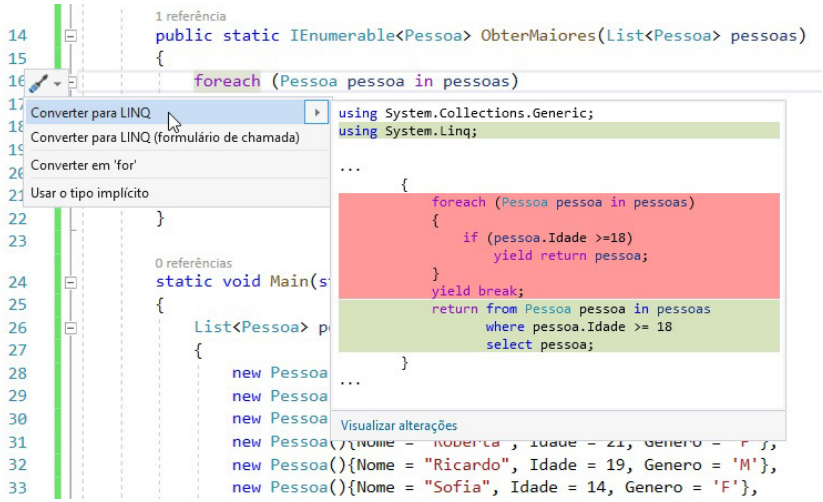


Figura 11.25: Convertendo um loop foreach em uma consulta LINQ

Após a conversão, o método `ObterMaiores` ficará como mostrado no trecho de código a seguir:

```
public static IEnumerable<Pessoa> ObterMaiores(List<Pessoa> pesso
as)
{
    return from Pessoa pessoa in pessoas
           where pessoa.Idade >= 18
           select pessoa;
}
```

Caso você prefira utilizar uma expressão lambda em vez do loop `foreach`, utilize a opção *Converter para LINQ (formulário de chamada)* do menu de ações rápidas. O resultado pode ser visto

no próximo fragmento de código:

```
public static IEnumerable<Pessoa> ObterMaiores(List<Pessoa> pesso
as)
{
    return pessoas.Where(pessoa => pessoa.Idade >= 18).Select(pes
soa => pessoa);
}
```

11.14 CONVERTENDO ESTRUTURAS CONDICIONAIS IF EM SWITCH

Estruturas condicionais `if` podem se tornar bastante complexas e difíceis de compreender quando são compostas de várias cláusulas `else if` . Com as melhorias introduzidas no C# 7.0, tornou-se simples aplicar instruções `switch` para substituí-las na maioria dos cenários.

Para ilustrar como efetuar essa transformação usando uma ação rápida, vamos partir do exemplo a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    abstract class Funcionario
    {

    }

    class Vendedor:Funcionario
    {

    }

    class Supervisor : Funcionario
    {

    }
```

```
    class Gerente : Funcionario
    {

    }

    class Program
    {
        static void Apresentarse(Funcionario funcionario)
        {
            if (funcionario is Vendedor)
            {
                Console.WriteLine("Eu sou um vendedor.");
            }
            else if (funcionario is Supervisor)
            {
                Console.WriteLine("Eu sou um supervisor.");
            }
            else if (funcionario is Gerente)
            {
                Console.WriteLine("Eu sou um gerente.");
            }
        }
        static void Main(string[] args)
        {
            Gerente gerente = new Gerente();
            Apresentarse(gerente);
        }
    }
}
```

Para aplicar a transformação, clique com o botão direito do mouse sobre a palavra-chave `if` e selecione no menu de ações rápidas a opção *Converter em Switch*:

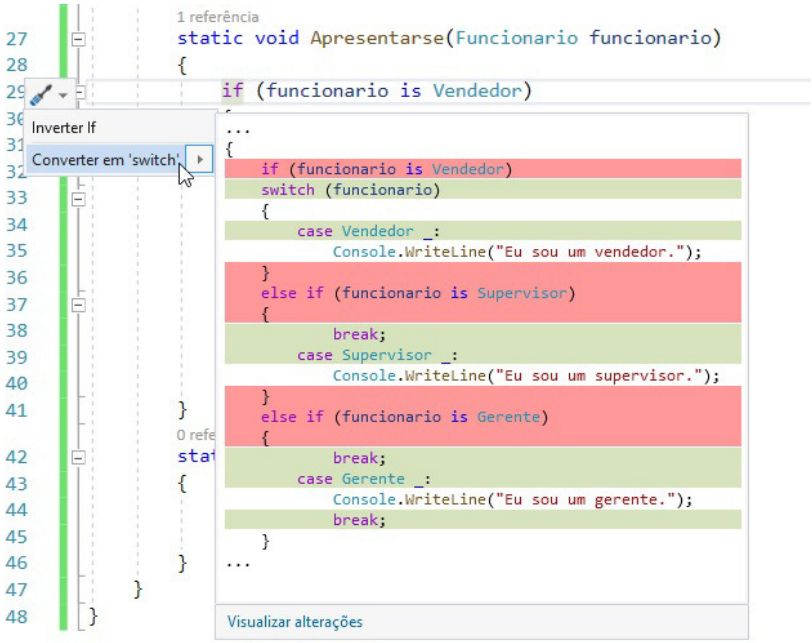


Figura 11.26: Convertendo uma instrução if em switch

Confira a seguir o código do método `Apresentarse` após a aplicação da ação:

```
static void Apresentarse(Funcionario funcionario)
{
    switch (funcionario)
    {
        case Vendedor _:
            Console.WriteLine("Eu sou um vendedor.");
            break;
        case Supervisor _:
            Console.WriteLine("Eu sou um supervisor.");
            break;
        case Gerente _:
            Console.WriteLine("Eu sou um gerente.");
            break;
    }
```

```
}
```

11.15 ADICIONANDO CLÁUSULAS CASE AUSENTES EM UM SWITCH

Ao criarmos uma estrutura condicional `switch` no Visual Studio, normalmente utilizamos o *code snippet*, que gera automaticamente e sempre que possível os blocos `case` adequados para o tipo de dado que está sendo avaliado. Veja a seguir um exemplo:

```
namespace ProdutividadeEmCSharp
{
    enum EstadoCivil
    {
        Solteiro,
        Casado
    }


    class Program
    {
        static void Main(string[] args)
        {
            var estadoCivil = EstadoCivil.Casado;

            switch (estadoCivil)
            {
                case EstadoCivil.Solteiro:
                    break;
                case EstadoCivil.Casado:
                    break;
                default:
                    break;
            }
        }
    }
}
```

No mundo real, nem sempre saberemos todos os estados

possíveis de um tipo enumerado em um primeiro momento, e pode ser necessário adicionar novas constantes para o tipo posteriormente. Exemplo:

```
enum EstadoCivil
{
    Solteiro,
    Casado,
    UniaoEstavel,
    Divorciado,
    Separado,
    Viuvo
}
```

Para casos como esse, o Visual Studio 2017 e posterior oferece uma ação rápida que nos permite incluir de forma automática cláusulas `case` para as constantes ausentes. Veja:

Figura 11.27: Adicionando cases ausentes em um switch

Após a aplicação da ação, a instrução `switch` ficará conforme

mostrado na listagem a seguir:

```
switch (estadoCivil)
{
    case EstadoCivil.Solteiro:
        break;
    case EstadoCivil.Casado:
        break;
    case EstadoCivil.UniaoEstavel:
        break;
    case EstadoCivil.Divorciado:
        break;
    case EstadoCivil.Separado:
        break;
    case EstadoCivil.Viuvo:
        break;
    default:
        break;
}
```

11.16 UTILIZANDO INICIALIZADORES DE OBJETOS

O Visual Studio 2017 e superior inclui uma ação rápida que permite usar *inicializadores de objetos* em vez de invocar o construtor e ter linhas adicionais de instruções de atribuição. Vamos ilustrar esse cenário através do exemplo a seguir:

```
namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
```

```
            var pessoa = new Pessoa();
            pessoa.Nome = "Fernando";
            pessoa.Idade = 37;
            pessoa.Genero = 'M';
        }
    }
}
```

Para executar esta ação, clique com o botão direito do mouse sobre a palavra-chave `new` e selecione no menu de ações rápidas a opção *A inicialização do objeto pode ser simplificada*. Veja:

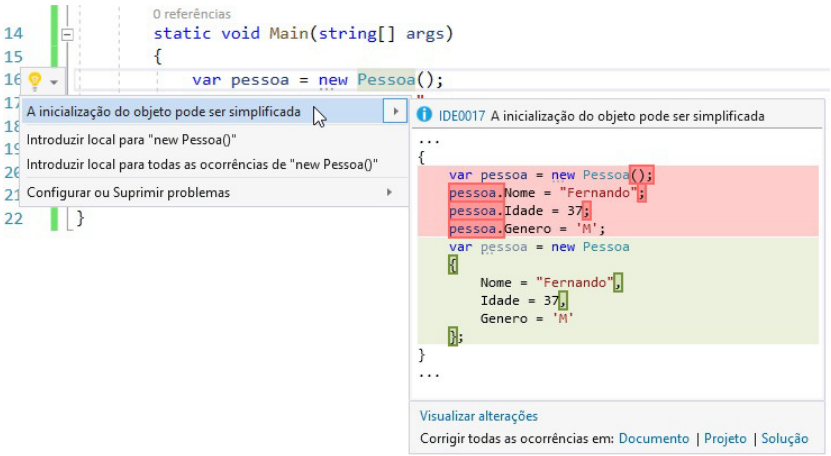


Figura 11.28: Simplificando a inicialização de um objeto

Após a aplicação da ação, obteremos o seguinte resultado:

```
var pessoa = new Pessoa
{
    Nome = "Fernando",
    Idade = 37,
    Genero = 'M'
};
```

11.17 UTILIZANDO INICIALIZADORES DE COLEÇÕES

O Visual Studio 2017 e superior também inclui uma ação rápida que permite usar *inicializadores de coleções* para simplificar o nosso código. Observe o exemplo a seguir:

```
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var primos = new List<int>();
            primos.Add(2);
            primos.Add(3);
            primos.Add(5);
            primos.Add(7);
            primos.Add(11);
            primos.Add(13);
            primos.Add(17);
            primos.Add(19);
        }
    }
}
```

Para executar esta ação, clique com o botão direito do mouse sobre a palavra-chave `new` e selecione no menu de ações rápidas a opção *A inicialização de coleção pode ser simplificada*. Veja:

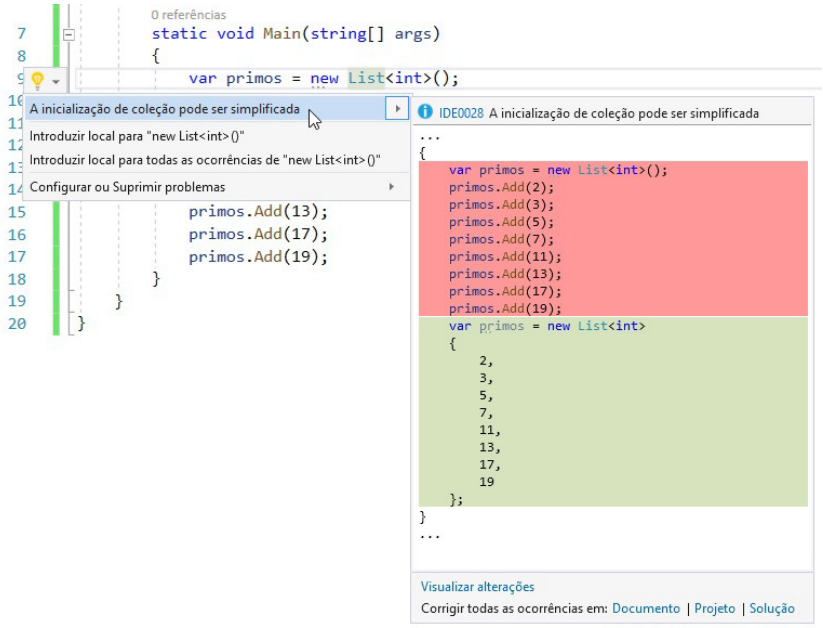


Figura 11.29: Simplificando a inicialização de uma coleção de inteiros

Após a aplicação da ação, o código-fonte do nosso programa de teste ficará como mostrado na próxima listagem:

```
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            var primos = new List<int>
            {
                2,
                3,
                5,
                7,
                11,
                13,
```

```
                17,
                19
            };
        }
    }
}
```

Veja agora um segundo exemplo no qual temos uma coleção `pessoas` composta de uma lista de objetos do tipo `Pessoa`:

```
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var pessoa1 = new Pessoa
            {
                Nome = "Renan",
                Idade = 35,
                Genero = 'M'
            };
            var pessoa2 = new Pessoa
            {
                Nome = "Cristiane",
                Idade = 43,
                Genero = 'F'
            };
            var pessoas = new List<Pessoa>();
            pessoas.Add(pessoa1);
            pessoas.Add(pessoa2);
        }
    }
}
```

Clique novamente com o botão direito do mouse sobre a

palavra-chave `new` e selecione no menu de ações rápidas a opção *A inicialização de coleção pode ser simplificada*. Veja:

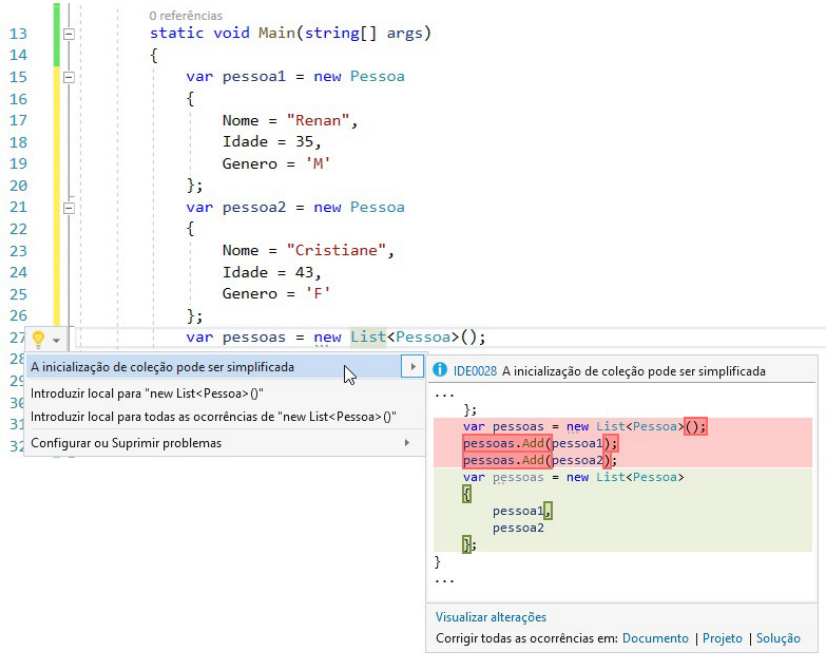


Figura 11.30: Simplificando a inicialização de uma lista de objetos do tipo Pessoa

Confira o resultado a seguir:

```
using System.Collections.Generic;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }
    class Program
    {
```

```
        static void Main(string[] args)
        {
            var pessoa1 = new Pessoa
            {
                Nome = "Renan",
                Idade = 35,
                Genero = 'M'
            };
            var pessoa2 = new Pessoa
            {
                Nome = "Cristiane",
                Idade = 43,
                Genero = 'F'
            };
            var pessoas = new List<Pessoa>
            {
                pessoa1,
                pessoa2
            };
        }
    }
}
```

Conforme você pode observar, o resultado é realmente melhor do que o que tínhamos antes da ação, mas acredito que você também gostaria que a ferramenta fornecesse como saída o trecho de código mostrado na próxima listagem:

```
var pessoas = new List<Pessoa>
{
    new Pessoa {
    Nome = "Renan",
    Idade = 35,
    Genero = 'M'
},
   new Pessoa {
    Nome = "Cristiane",
    Idade = 43,
    Genero = 'F'
}};
```

Infelizmente, essa ação ainda não oferece um resultado tão

avançado como o esperado. Quem sabe isso já tenha mudado quando você estiver lendo este livro.

Com o que você aprendeu neste capítulo, não há mais motivos para entregar ou aceitar código mal escrito, uma vez que o Visual Studio é capaz de sinalizar e corrigir rapidamente boa parte do código que pode ser melhorado e formatado segundo as regras de estilo que definirmos ou simplificarmos.

Parodiando aquela consagrada escola inglesa para bruxos, eu costumo brincar que o conhecimento reunido neste livro é como as aulas de "Defesa contra as Artes das Trevas" que ajudam a nos proteger de pessoas que acham que fazemos mágica em vez de software e que não entendem que consertar código malfeito ou remover funcionalidades desnecessárias também consome um bom tempo de um ou mais *sprints* do *Scrum*.

Capítulo 12

DEPURAÇÃO

A *depuração* é o processo que usamos para encontrar erros em um projeto. Para executá-la, recorremos normalmente à *depuração interativa*, que é feita anexando o depurador a um processo em execução e investigando a execução e o estado do programa com um vasto arsenal de ferramentas de depuração oferecidas pelo Visual Studio.

Não há mágica, nem bala de prata quando o assunto é depuração. Depurar é um processo cansativo, moroso e cheio de obstáculos, principalmente quando o código não foi otimizado para a depuração e não se conhece a fundo as ferramentas de depuração oferecidas pelo ambiente de desenvolvimento.

O domínio dos recursos do depurador do Visual Studio é vital para quem busca produtividade, pois reduz de forma significativa o tempo gasto com a depuração, além de transformar você em um desenvolvedor ou desenvolvedora mais eficaz.

Em muitos casos, alguns minutos percorrendo o código com o depurador serão suficientes para encontrar e solucionar o bug. Quando isso não for o bastante, será necessário recorrer a outras frentes como investigar logs, analisar despejos, efetuar testes de unidade, criação de perfil etc.

Preparamos este capítulo para todos que já sofreram e ainda sofrem depurando código alheio. Ao longo das próximas páginas, iniciaremos o nosso estudo abordando os vários tipos de *pontos de interrupção* suportados pelo Visual Studio, incluindo *pontos de interrupção condicionais*, *tracepoints*, *pontos de interrupção de função* e os novos *pontos de interrupção de dados* suportados apenas no .NET Core 3.0 ou superior. Veremos também como compartilhar pontos de interrupção com outros desenvolvedores usando os recursos de exportação e importação de pontos de interrupção.

Após dominarmos os pontos de interrupção, veremos como navegar pelo código em modo de depuração e como usar o recurso *Editar e Continuar* suportado pela linguagem C#.

O capítulo termina abordando como visualizar e alterar dados usando várias janelas de depuração incluídas no Visual Studio. Veremos como tirar proveito das janelas *Saída*, *Imediata*, *Locais*, *Automáticos*, *Inspeção*, *Inspeção Rápida* e como usar *DataTips* de forma eficiente. Abordaremos também como utilizar visualizadores de depuração nativos do Visual Studio para visualizar dados no formato JSON, XML, Datasets etc., como baixar visualizadores extras a partir do Visual Studio Marketplace e o que é necessário para criar seus próprios visualizadores de depuração.

12.1 FORÇANDO O MODO DE INTERRUPÇÃO DO DEPURADOR

Antes de iniciarmos a depuração, normalmente definimos pontos de interrupção em nosso código, o que faz com que o

programa seja pausado em uma linha de código específica quando a execução atinge o ponto sinalizado. Em termos práticos, isso significa que todos os threads em execução são pausados na linha de código em questão.

Esse é o caminho padrão a ser seguido, mas haverá casos em que desejaremos entrar nesse modo de interrupção para efetuar a depuração mesmo sem termos pontos de interrupção definidos previamente. Para tanto, basta clicar no menu *Depurar* e selecionar a opção *Pausar Execução* ou teclar *Ctrl + Alt + Break*. Você verá que a execução será pausada e uma seta verde será mostrada na linha em que a execução parou. Confira:

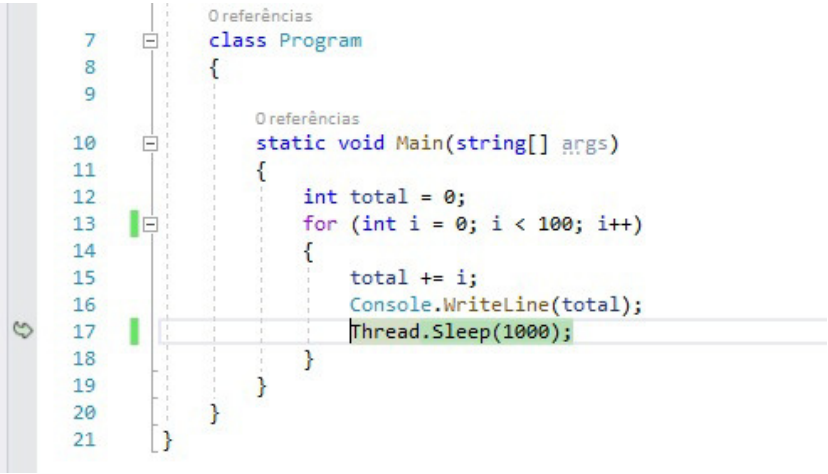


Figura 12.1: Forçando a entrada no modo de interrupção do Visual Studio

A partir deste ponto, você pode usar as opções de navegação pelo código descritas em detalhes na seção *Navegando pelo código em modo de interrupção* deste capítulo.

12.2 DEFININDO PONTOS DE INTERRUPÇÃO NO SEU CÓDIGO

O Visual Studio nos permite definir pontos de interrupção (em inglês, *breakpoints*) em qualquer linha de código executável. Isso nos possibilita pausar a execução do depurador onde desejarmos.

Para definir um ponto de interrupção no código-fonte com o mínimo de esforço, basta parar com o cursor de inserção na linha desejada e teclar *F9* ou clicar com o botão esquerdo do mouse na margem da extrema esquerda, ao lado de uma linha de código. O ponto de interrupção será exibido como um círculo vermelho na margem esquerda. Ao teclar *F9* novamente ou clicar uma segunda vez com o mouse, o ponto será desmarcado.

O Visual Studio suporta diferentes tipos de pontos de interrupção em C#:

- Pontos de interrupção simples.
- Pontos de interrupção condicionais.
- Pontos de interrupção de dados.
- Pontos de interrupção de funções.
- Tracepoints.

O gerenciamento de todos os pontos de interrupção é feito através da janela *Pontos de Interrupção*. Para exibi-la, clique no menu *Depurar*, depois no submenu *Janelas* e, por fim, na opção *Pontos de Interrupção* ou simplesmente tecle *Ctrl + D, B*. A janela será exibida com a lista de pontos de interrupção ativos e desabilitados. Veja:

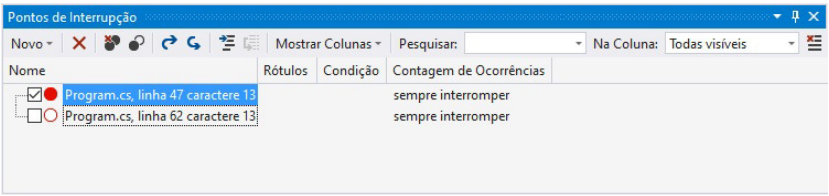


Figura 12.2: Janela de gerenciamento de pontos de interrupção do Visual Studio

Os pontos de interrupção ativos aparecem com círculos vermelhos cheios, enquanto os desabilitados são mostrados com círculos vazios tanto na margem esquerda do editor quanto na janela *Pontos de Interrupção*. Os tracepoints são sinalizados com losangos.

Para que os pontos de interrupção estejam disponíveis, o código do nosso projeto precisa ser compilado em modo debug. Isso é feito selecionando a opção Debug na barra de botões do Visual Studio.

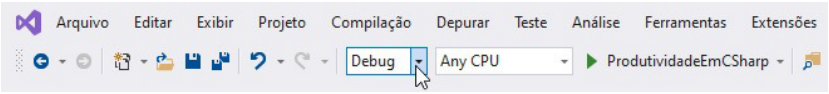


Figura 12.3: Selecionando a compilação em modo debug

Nos próximos tópicos, veremos como tirar proveito de cada um dos tipos de pontos de interrupção suportados.

Exportando e importando pontos de interrupção

O Visual Studio nos permite compartilhar o estado e o local dos pontos de interrupção com outros desenvolvedores. Para isso, basta exportar um ou mais pontos de interrupção para um arquivo XML e importá-los em outro computador que esteja rodando a

IDE.

Note que essas funcionalidades nos possibilitam partilhar um ponto de interrupção específico, um conjunto deles que atenda aos critérios de pesquisa informados na janela de gerenciamento ou, se preferir, a relação completa de pontos de interrupção criados pelo desenvolvedor.

Para exportar um ou mais pontos de interrupção a partir da janela de pontos de interrupção, selecione as caixas de verificação correspondentes e, a seguir, clique com o botão direito do mouse em um deles. Selecione *Exportar itens selecionados*, escolha um local de exportação e, em seguida, selecione *salvar*. Se quiser salvar todos, clique no ícone *Exportar todos os pontos de interrupção* na barra de botões.

Para importá-los, selecione o ícone *importar pontos de interrupção de um arquivo*, navegue até o local do arquivo XML e selecione *Abrir*. Veja:

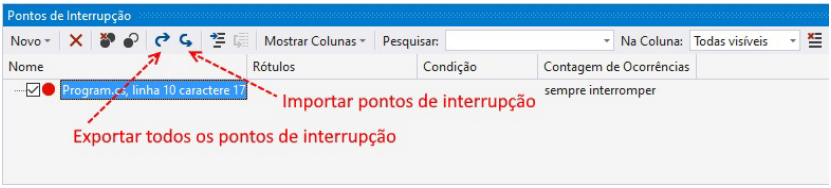


Figura 12.4: Recursos de exportação e importação de pontos de interrupção do Visual Studio

Desativando temporariamente pontos de interrupção

Apesar de ser fácil e rápido marcar e desmarcar pontos de interrupção simples, existirão momentos em que desejaremos desativar temporariamente um ou mais pontos em vez de

simplesmente excluí-los. Essa alternativa é particularmente útil quando desejamos preservar condições incluídas em um ponto de interrupção, rótulos associados e *tracepoints*.

Para desabilitar um ponto de interrupção sem excluí-lo a partir do próprio editor, clique com o botão direito do mouse sobre o ponto de interrupção e selecione no menu de contexto a opção *Desabilitar ponto de interrupção*. Para reabilitá-lo, clique novamente com o botão direito do mouse sobre ele e selecione *Habilitar ponto de interrupção*. Para obter o mesmo resultado usando a janela de gerenciamento de pontos de interrupção, basta marcar ou desmarcar a caixa de verificação existente junto a cada ponto listado.

Atribuindo rótulos a pontos de interrupção

Durante a depuração de programas com código complexo em que se tem vários pontos de interrupção definidos, pode ser produtivo atribuir rótulos a cada um deles.

A atribuição dos rótulos pode ser feita tanto através do editor quanto da janela de gerenciamento de pontos de interrupção. Em ambos os casos, basta executar os seguintes passos:

1. Clique com o botão direito do mouse sobre o ponto de interrupção. No menu de contexto que será exibido, selecione *Editar rótulos*.
2. A caixa de diálogo *Editar rótulos de ponto de interrupção* será exibida. Forneça um rótulo para o ponto de interrupção no campo *Digite um novo rótulo* e a seguir clique no botão *OK*.

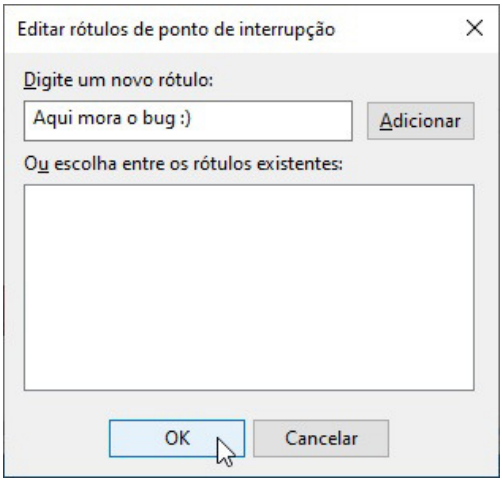


Figura 12.5: Atribuindo um rótulo a um ponto de interrupção

O rótulo atribuído será exibido na coluna *Rótulos*. Veja:

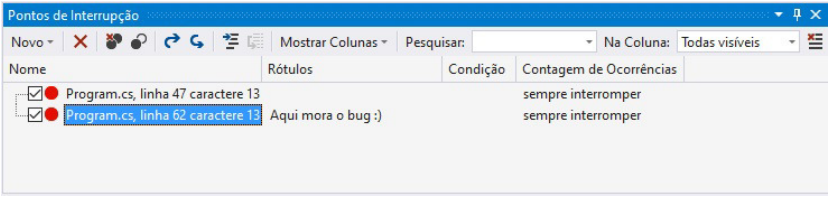


Figura 12.6: Rótulo do ponto de interrupção sendo exibido na janela de gerenciamento de pontos de interrupção

Qualquer pessoa que já tenha mudado de residência sabe a importância de colocar identificadores nas caixas para que seja possível encontrar algo rapidamente quando necessário.

Observe que existe um campo *Pesquisar* na janela *Pontos de Interrupção* que nos permite filtrar os breakpoints existentes. Se digitarmos `bug` seguido de *Enter* nesse campo de busca, veremos que será exibido apenas um ponto de interrupção que contém esta

palavra em seu rótulo. Confira:

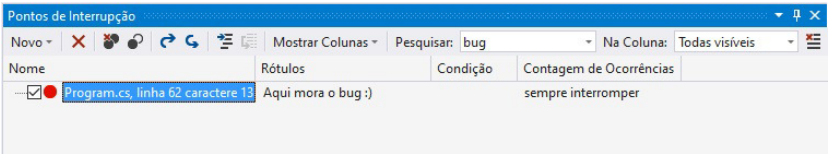


Figura 12.7: Filtrando os pontos de interrupção da janela de gerenciamento de pontos de interrupção

12.3 UTILIZANDO PONTOS DE INTERRUPÇÃO CONDICIONAIS

Pontos de interrupção condicionais foram criados para minimizar o tempo e o esforço mental que dedicamos à depuração. O depurador do Visual Studio suporta três opções de condições em breakpoints: expressão condicional, filtro e contagem de acesso.

Para ilustrar como utilizá-los em nossos projetos, vamos partir de um programa bem simples:

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int contador = 0;
            for (int i = 0; i <= 10; i++)
            {
                contador += i;
            }
        }
    }
}
```

Execute os seguintes passos para criar o ponto de interrupção:

1. Adicione um ponto de interrupção na linha a seguir:

```
contador += i;
```

2. Clique com o botão direito do mouse sobre o ponto de interrupção e selecione *Condições* no menu de contexto. Para esse teste, desejamos que o ponto só seja ativado quando a variável `contador` for maior que `10`. Para tanto, entre como `contador > 10` no campo mais à direita da definição da expressão condicional e, a seguir, clique no botão *Fechar*. Veja:

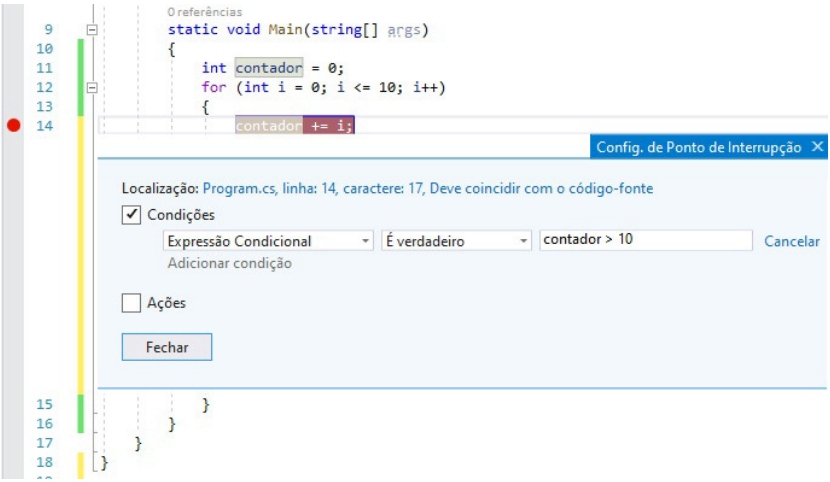


Figura 12.8: Definindo uma condição para a execução de um ponto de interrupção

3. Note na próxima imagem que o ponto de interrupção passou a ser sinalizado por um sinal de mais (+) dentro do círculo vermelho, indicando a presença de uma condição:

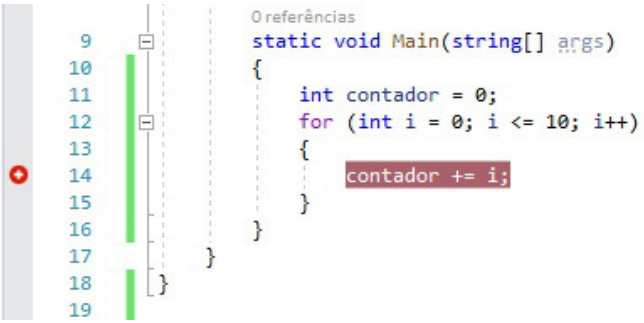


Figura 12.9: Sinalização de um ponto de interrupção condicional

Ao executar o programa, você verá que na primeira vez em que ocorre a parada no ponto de interrupção o contador estará com o valor 15. Confira:

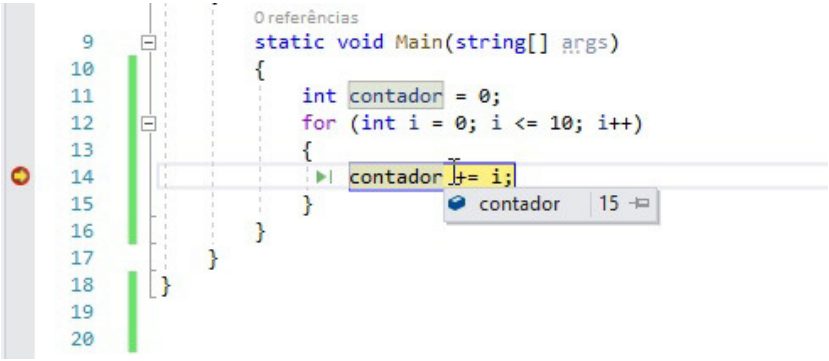


Figura 12.10: Inspecionando o valor da variável contador

Repare no tempo de depuração que o uso desses pontos de interrupção mais inteligentes consegue nos poupar. O depurador do Visual Studio suporta o uso de condições para definir pontos de interrupção que serão ativados nos seguintes cenários:

- Uma expressão condicional é avaliada como verdadeira.

- O valor de uma expressão condicional se altera.
- Atingimos uma contagem de acesso definida pelo desenvolvedor.
- Uma condição de filtro configurada para disparar somente em dispositivos especificados ou em processos e threads especificados.

Para saber mais sobre pontos de interrupção condicionais, consulte:

https://docs.microsoft.com/pt-br/visualstudio/debugger/using-breakpoints?view=vs-2019#breakpoint-conditions

12.4 UTILIZANDO PONTOS DE INTERRUPÇÃO DE DADOS

Em projetos .NET Core 3.0 ou superior, podemos definir *pontos de interrupção de dados*. Esse tipo especial de ponto de interrupção interrompe a execução quando uma propriedade de um objeto específico é alterada.

Para ilustrar como explorar esse novo recurso, vamos partir do exemplo a seguir:

```
using System;
using System.Threading;

namespace ProdutividadeEmCSharp
{
    class Pessoa
```

```
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Pessoa pessoa = new Pessoa() { Nome = "Pedro", Idade
= 18, Genero = 'M' };
            Console.WriteLine($"Nome: {pessoa.Nome}  Idade: {pess
oa.Idade}");
            Thread.Sleep(5000);
            pessoa.Idade = 19;
            Console.ReadLine();
        }
    }
}
```

Para definir um ponto de interrupção de dados, execute os seguintes passos:

1. Inicie a depuração e aguarde até que um ponto de interrupção seja atingido. Para esse exemplo, defina o ponto de interrupção na linha a seguir:

```
Pessoa pessoa = new Pessoa() { Nome = "Pedro", Idade = 18, Genero
 = 'M' };
```

2. Inspecione o objeto a ser monitorado, em nosso caso `pessoa`, expandindo as suas propriedades na janela *Inspeção*, *Automáticos* ou *Locais*.

3. Clique com o botão direito do mouse na propriedade desejada (`Idade`) e selecione no menu de contexto a opção *Interromper quando o valor é alterado*.

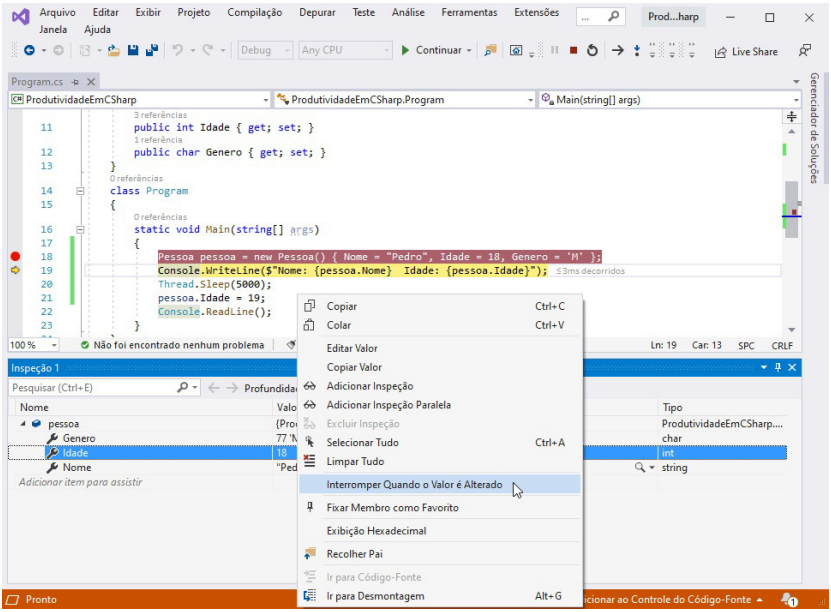


Figura 12.11: Definindo um ponto de interrupção de dados

A propriedade é sinalizada com um ponto de interrupção. Veja:

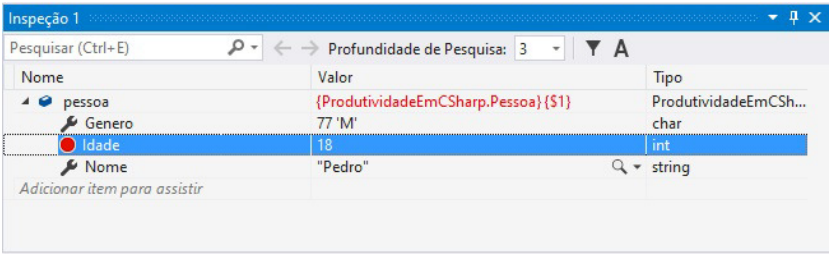


Figura 12.12: Sinalização de um ponto de interrupção de dados na janela de Inspeção

4. Tecle *F5* para continuar a execução e aguarde até que a execução pare na linha que altera o valor da propriedade `Idade` .

Observe que os valores anterior e atual são exibidos:

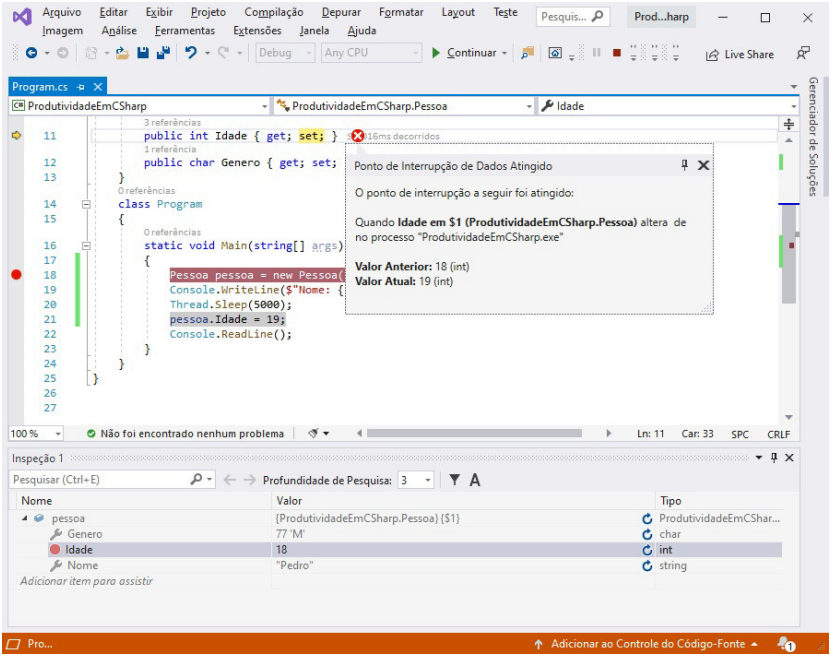


Figura 12.13: Execução parada em um ponto de interrupção de dados

Quando bem utilizado, esse recurso pode reduzir consideravelmente o tempo de depuração de um projeto. Vale destacar que esse tipo de ponto de interrupção tem algumas limitações, além de só estar disponível no Visual Studio 2019 para projetos .NET Core 3.0 ou superior. Segundo a documentação oficial, os pontos de interrupção de dados não funcionarão para:

- Propriedades que não são expansíveis na dica de ferramenta ou nas janelas Inspeção, Locais e Automáticos.
- Variáveis estáticas.
- Classes com o atributo `DebuggerTypeProxy` .

- Campos dentro de structs.

> Para saber como proceder nesses casos, consulte a seguinte página de resolução de problemas da documentação oficial:
>
> https://docs.microsoft.com/pt-br/visualstudio/debugger/troubleshoot-data-breakpoint-errors?view=vs-2019

12.5 UTILIZANDO PONTOS DE INTERRUPÇÃO DE FUNÇÃO

Pontos de interrupção de função são usados quando desejamos interromper a execução sempre que uma função é chamada. Imagine, por exemplo, que você sabe o nome da função, mas não a sua localização ou que existem várias funções com o mesmo nome (leia-se *métodos sobrecarregados*) e deseja parar a execução em todas as chamadas.

Para ilustrar como utilizar esse tipo de ponto de interrupção, vamos utilizar o seguinte programa de teste:

```
using System;
using System.Threading;

namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void Escrever()
        {
```

```
            Console.WriteLine("Executando método Escrever()");
        }

        static void Escrever(string mensagem)
        {
            Console.WriteLine("Executando método Escrever(string
mensagem)");
        }


        static void Main(string[] args)
        {
            Escrever();
            Thread.Sleep(5000);
            Escrever("Produtividade em C#");
            Console.ReadLine();
        }
    }
}
```

Para definir um ponto de interrupção de função, execute os seguintes passos:

1. No menu *Depurar*, selecione o submenu *Novo ponto de interrupção* e, a seguir, a opção *Ponto de interrupção de função*. Esta opção também está disponível na janela *Pontos de Interrupção*. Nesse caso, você deve clicar em *Novo* e selecionar a opção *Ponto de interrupção de função*.

2. A caixa de diálogo *Novo ponto de interrupção de função* será exibida. Digite o nome da função `Escrever` no campo *Nome da função*.

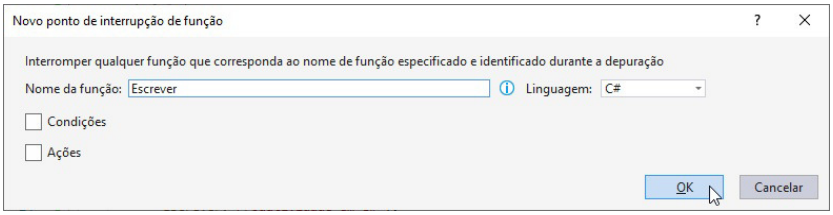


Figura 12.14: Definindo um ponto de interrupção de função

3. Note que é possível ainda definir condições e ações, se assim você desejar. Na janela *Pontos de Interrupção*, o ponto de interrupção que acabamos de criar será mostrado com o nome da função. Veja:

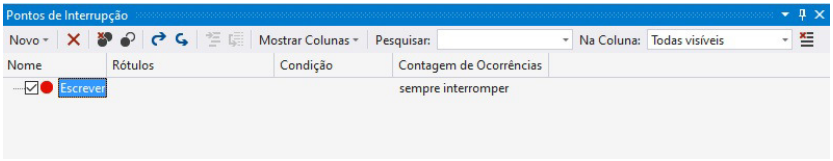


Figura 12.15: Ponto de interrupção de função listado na janela de pontos de exibição

4. Tecle *F5* para executar o programa. A execução vai parar a cada chamada ao método `Escrever`.

> Consulte a documentação oficial para conhecer as formas que podemos utilizar para restringir a especificação de função:
>
> https://docs.microsoft.com/pt-br/visualstudio/debugger/using-breakpoints?view=vs-2019#BKMK_Set_a_breakpoint_in_a_source_file

Ainda nesse exemplo, se desejássemos que o ponto de interrupção de função fosse ativado apenas para a segunda sobrecarga do método `Escrever`, precisaríamos especificar os tipos de parâmetro da função sobrecarregada. Neste caso: `Escrever(string)`.

12.6 ENVIANDO INFORMAÇÕES PARA A JANELA DE SAÍDA USANDO TRACEPOINTS

Os *tracepoints* permitem ao desenvolvedor enviar informações para a janela de saída do Visual Studio em condições configuráveis sem modificar o código-fonte ou parar a execução. Isso é feito ao atribuir uma cadeia de caracteres de saída à caixa de seleção *Ações* na janela *Configurações de ponto de interrupção*.

Vamos ilustrar como utilizar esse recurso com o programa de teste a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static int Fibonacci(int n)
        {
            int a = 0;
            int b = 1;
            for (int i = 0; i < n; i++)
            {
                int temp = a;
                a = b;
                b = temp + b;
            }
            return a;
        }
```

```
        static void Main(string[] args)
        {
            for (int i = 0; i < 15; i++)
            {
                Console.WriteLine(Fibonacci(i));
            }
            Console.ReadLine();
        }
    }
}
```

Para definir um tracepoint, é necessário criar inicialmente um ponto de interrupção simples. Após a sua criação, execute os seguintes passos:

1. Pare o mouse sobre o círculo vermelho do ponto de interrupção e clique no ícone de engrenagem. A janela *Configurações de ponto de interrupção* será exibida.

2. Marque a caixa de seleção *Ações*. Note que o círculo vermelho se transforma em um losango, o que indica que você mudou de um ponto de interrupção para o tracepoint.

3. As opções relacionadas às ações serão exibidas. Insira a mensagem que você deseja exibir na caixa de texto *Mostrar uma mensagem na Janela de Saída*. Para este exemplo, digite: `a: {a} b: {b}` . Note que é possível, se você desejar, forçar a parada da execução ao atingir o tracepoint desmarcando a opção *Continuar execução do código*.

4. Se necessário, adicione condições que determinam se a mensagem é exibida. Para isso, marque a caixa de seleção *Condições* e informe as condições.

5. Clique no botão *Fechar*.

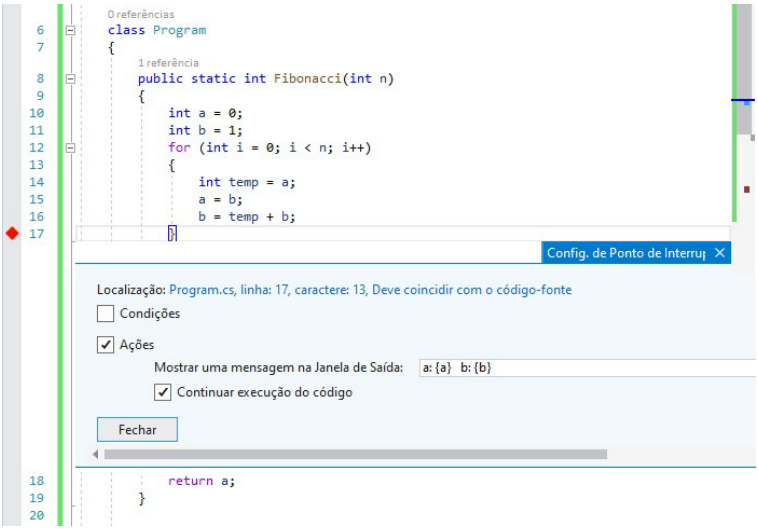


Figura 12.16: Configurando a saída que será gerada pelo tracepoint

6. Tecle *F5* para executar o programa em modo de depuração.

A janela *Saída* da sua instância do Visual Studio provavelmente não estará visível neste momento. Para exibi-la, clique no menu *Depurar*, selecione o submenu *Janelas* e, a seguir, a opção *Saída*. Confira na imagem a seguir a saída produzida pelo tracepoint:

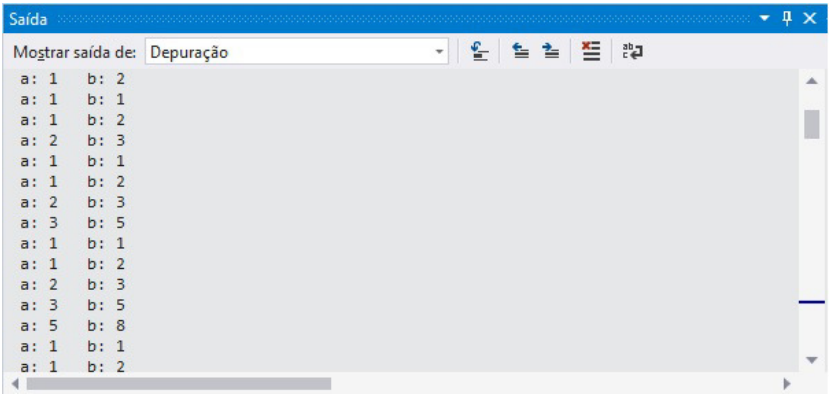


Figura 12.17: Inspecionando os valores impressos pelo tracepoint na janela de saída

Ao digitar o texto a ser exibido na caixa de mensagem *Mostrar uma mensagem na Janela de Saída* note que o IntelliSense está ativo. Experimente digitar um sinal de `$` e observe que existem variáveis internas do Visual Studio que podem ser inseridas nas mensagens, como `$FUNCTION` representando a função atual ou `$CALLER` representando a função chamadora da função atual. Veja na imagem a seguir a lista de variáveis mostrada pelo IntelliSense:

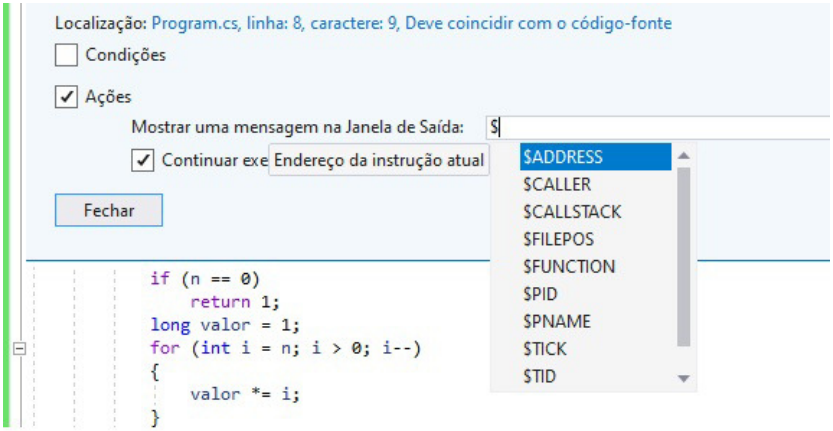


Figura 12.18: Usando variáveis internas nas mensagens

Perceba que este é um daqueles recursos que pode nos poupar um tempo considerável de depuração quando bem explorado, pois nos permite gerar um log da execução sem alterar o código-fonte. Para saber mais sobre *tracepoints*, consulte a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/visualstudio/debugger/using-tracepoints?view=vs-2019

12.7 NAVEGANDO PELO CÓDIGO EM MODO DE INTERRUPÇÃO

Quando estamos no modo de interrupção, podemos depurar de forma interativa, enquanto observamos como a execução do código progride. Temos à disposição as seguintes opções da

navegação de código:

- *Continue* (F5) - Esta opção abandonará o modo de interrupção e continuará a execução do programa até o próximo ponto de interrupção ser atingido, se ele existir. Nesse caso, o Visual Studio entrará novamente no modo de interrupção.

- *Pular método/Step Over* (F10) - Executará a linha atual e será interrompida na próxima linha de código.

- *Intervir/Step Into* (F11) - É usada quando a próxima linha de execução é um método ou uma propriedade. Quando clicado, o depurador entrará na primeira linha do código do método.

- *Executar para este lugar* - Este recurso, introduzido no Visual Studio 2017, permite continuar a execução e interromper em um local especificado sem um ponto de interrupção. Para isso, basta clicar na seta verde que aparece no início de cada linha quando o mouse está parado sobre ela. Outra forma de chegar ao mesmo resultado é parar com o cursor de inserção na linha desejada e teclar *Ctrl + F10* ou clicar com o botão direito do mouse na linha em questão e selecionar no menu de contexto a opção *Executar até o Cursor*. Veja:

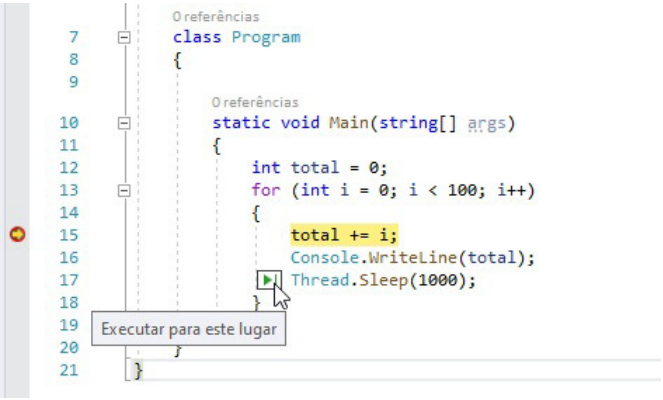


Figura 12.19: Usando a opção de navegação Executar para este lugar

- *Definir próxima declaração* (Ctrl + Shift + F10)_ - Esta opção do menu de contexto permite pular a execução de uma parte do código, definindo de forma arbitrária a próxima linha de código a ser executada. Na prática, isso significa que a linha atual marcada pela seta amarela não será executada. Como alternativa, é possível arrastar a seta amarela com o mouse para qualquer linha do nosso código-fonte.

A capacidade de alterar o fluxo de execução simplesmente arrastando a seta amarela nos permite testar caminhos de execução de código diferentes ou executar novamente o código sem reiniciar o depurador. Isso é algo poderoso e flexível, mas que precisa ser usado com certo cuidado, pois não é possível reverter o aplicativo para um estado anterior após mover o ponteiro de execução.

A opção *Executar até o cursor*, mencionada anteriormente, é mais flexível do que aparenta em um primeiro momento. Você

pode iniciar automaticamente a depuração do código-fonte simplesmente clicando com o botão direito do mouse sobre uma linha qualquer do código-fonte e selecionando esta opção no menu de contexto. Isso fará com que a execução avance até o ponto onde o cursor está localizado, caso não exista nenhum ponto de interrupção definido no caminho. Se existir, tecle *F5* para avançar e veja que a execução para no ponto onde o comando foi sinalizado.

Reiniciando o aplicativo rapidamente

Durante a depuração de um código, por vezes nos distraímos com algum evento externo, como uma ligação, uma mensagem de WhatsApp ou um colega de trabalho nos chamando e por vezes precisamos recomeçar. Essas breves interrupções são suficientes para arruinar minutos de depuração na maioria dos casos.

Para esses cenários, saiba que o Visual Studio oferece um recurso chamado *Reiniciar* que nos permite economizar um pouco de tempo. Para executá-lo, basta clicar no menu *Depurar* e selecionar a opção *Reiniciar* ou teclar *Ctrl* + *Shift* + *F5*. Isto fará com que o programa seja interrompido e reiniciado. O depurador será pausado no primeiro ponto de interrupção encontrado pela execução do código.

Utilizando o recurso Editar e Continuar

O recurso *Editar e Continuar* é uma daquelas funcionalidades oferecidas pelo Visual Studio para a linguagem C# que já existe há algum tempo, mas que é pouco explorada pelos desenvolvedores.

A primeira coisa que devemos saber sobre o recurso é que ele

só funciona em builds de depuração e que está ativo por padrão.

Essa funcionalidade permite que você faça alterações no código enquanto estiver no modo de interrupção. Em outras palavras, você não precisa parar de depurar e executar o programa novamente ao encontrar o bug. Basta alterar o código no modo de interrupção e o Visual Studio seguirá com a execução.

Na imagem a seguir, temos um exemplo muito simples, mas que é suficiente para ilustrar a funcionalidade. Nesta simulação, estamos depurando um programa e percebemos que na linha seguinte a que paramos, precisaremos modificar a saída produzida pelo método `WriteLine` da classe `Console`. Veja:

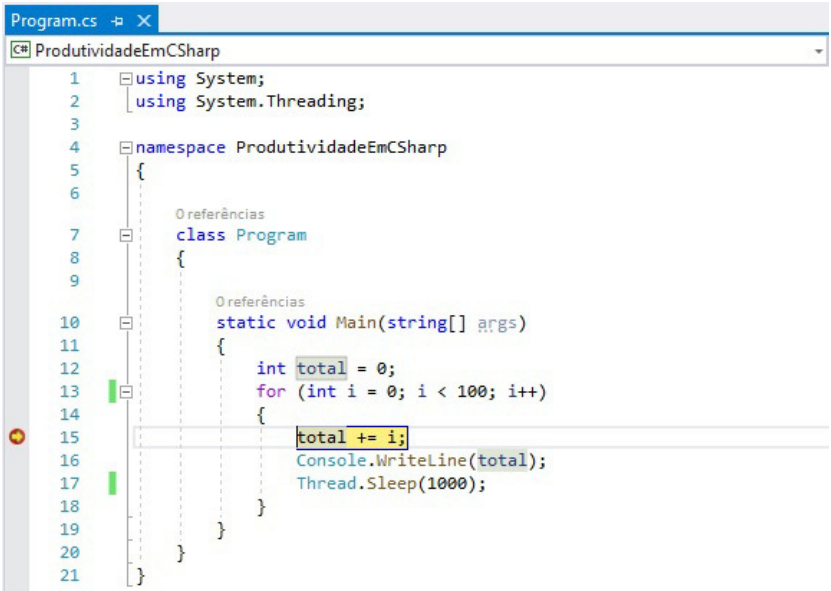


Figura 12.20: Ilustrando um local onde podemos utilizar Editar e Continuar

Para resolver o problema, nós simplesmente editamos o

argumento passado para o método `WriteLine` na linha 17 e seguimos com a depuração:

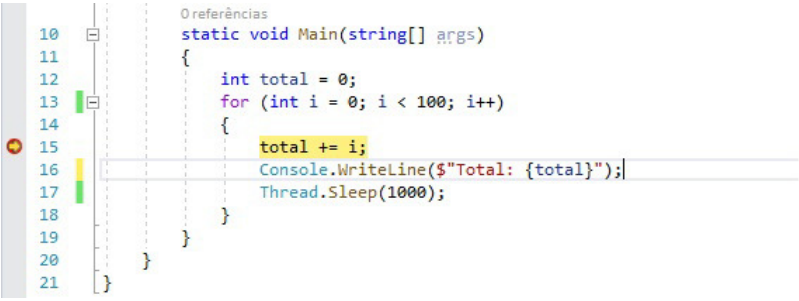


Figura 12.21: Empregando o recurso Editar e Continuar

Simples assim! Obviamente, nem todas as alterações são suportadas. Não é possível, por exemplo, fazer alterações dentro de uma função lambda.

Para conhecer a lista de alterações em C# que suportam esse recurso, consulte:

https://docs.microsoft.com/pt-br/visualstudio/debugger/supported-code-changes-csharp?view=vs-2019

https://github.com/dotnet/roslyn/wiki/EnC-Supported-Edits

Para mais detalhes sobre como usar o Editar e Continuar, acesse a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/visualstudio/debugger/how-to-use-edit-and-continue-csharp?view=vs-2019

12.8 VISUALIZANDO OS DADOS DURANTE A DEPURAÇÃO

O Visual Studio nos oferece um poderoso conjunto de ferramentas para a visualização dos dados durante a depuração. Temos disponíveis as seguintes opções:

- Janela Saída (Output)
- Janela Imediata (Immediate)
- Janela Locais (Locals)
- Janela Automáticos (Autos)
- Janelas de Inspeção (Watch)
- Janela de Inspeção Rápida (QuickWatch)
- Janelas de Inspeção Paralela (Parallel Watch)
- Visualizadores de Depuração (Debugger Viewers)
- Dicas de dados (DataTips)

Através de janelas, como *Locais*, *Automáticos*, *Inspeção* e *Inspeção Rápida*, podemos não só visualizar os valores de variáveis como também alterá-los durante a depuração do código. Para tanto, basta efetuar um duplo clique nos valores dessas variáveis e alterá-los. Também é possível alterar o valor de uma variável por meio das janelas *Imediata* e *Dicas de dados*.

Utilizando a janela Saída

A janela *Saída* (em inglês, *Output*) exibe mensagens geradas pelo Visual Studio durante a importação de pacotes, compilação, depuração, gerenciamento de código-fonte, publicação etc. É através dela que visualizamos mensagens de erro e de advertências emitidas pelo compilador, dentre outras. O que muitos não sabem

é que também é possível escrever nessa janela mensagens de debug criadas pelo próprio desenvolvedor para auxiliar no processo de depuração do código.

Para exibir a janela, clique no menu *Exibir* e selecione a opção *Saída* ou pressione *Ctrl+Alt+O*. Para escrever na janela Saída, devemos usar os métodos das classes `Debug` e `Trace` presentes no namespace `System.Diagnostics`. O exemplo a seguir ilustra o uso do método `WriteLine` da classe `Debug`:

```
using System;
using System.Diagnostics;

namespace ProdutividadeEmCSharp
{

    class Program
    {

        static void Main(string[] args)
        {
            Debug.WriteLine("Houston, temos um problema!");
            Console.ReadLine();
        }
    }
}
```

Confira o resultado na próxima imagem:

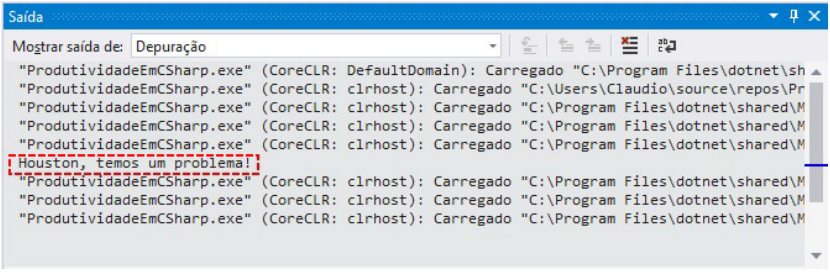


Figura 12.22: Escrevendo mensagens na janela Saída do Visual Studio

Ao examinar as mensagens geradas na janela Saída, você vai notar que ela contém "lixo", ou seja, várias mensagens que não nos interessam no momento. É possível filtrar a origem das mensagens que serão ou não exibidas nessa janela, clicando com o botão direito do mouse sobre a janela e selecionando as opções relacionadas à exibição no menu de contexto. Veja:

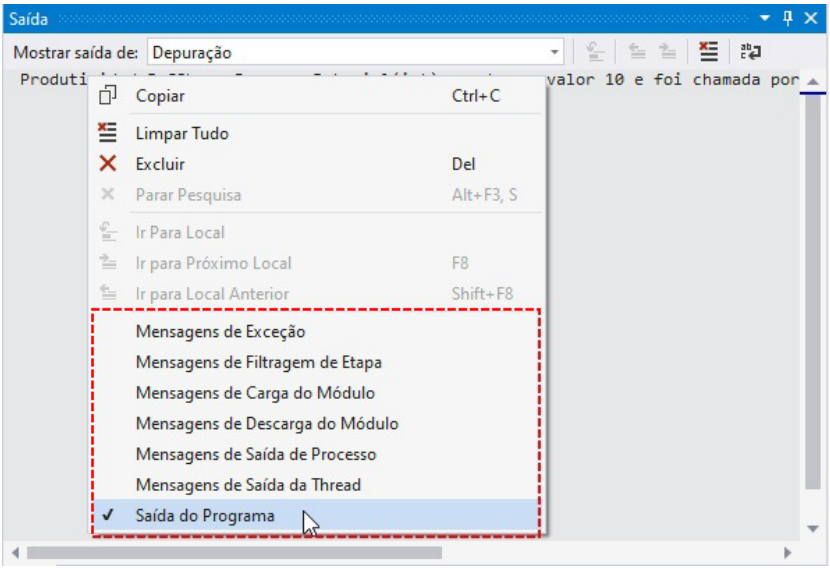


Figura 12.23: Especificando os filtros de mensagens ativos da janela Saída usando o menu de contexto

Também é possível realizar essa seleção através da caixa de diálogo *Opções* do Visual Studio, acessível através do menu *Ferramentas*. Para exibir os filtros de mensagem, selecione *Depuração* e, a seguir, *Janela de Saída* no menu da esquerda. Confira na imagem a seguir:

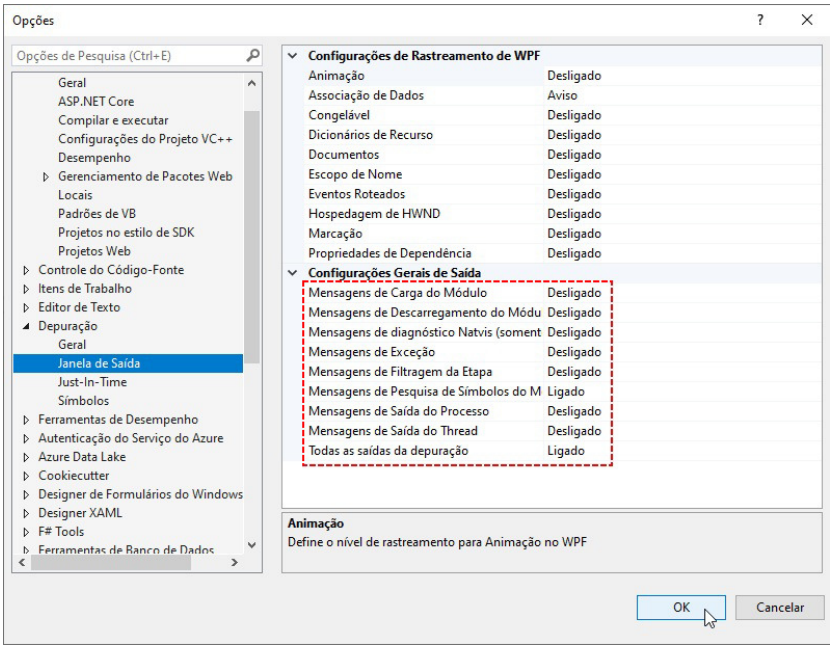


Figura 12.24: Especificando os filtros de mensagens ativos da janela Saída usando a caixa de diálogo Opções

Caso deseje manter a saída produzida pelo seu código para compartilhar com outros desenvolvedores, saiba que o Visual Studio permite salvar durante a depuração o conteúdo da janela Saída com um mínimo de esforço. Basta clicar no menu *Arquivo* e selecionar a opção *Salvar Saída como*. Por padrão, o arquivo será nomeado como `Saída-Depuração.txt` .

Saiba também que é possível, se você desejar, forçar a exibição da janela Saída toda vez que o build iniciar. Para tanto, execute os seguintes passos:

1. Clique no menu *Ferramentas* e selecione *Opções*.
2. A caixa de diálogos *Opções* será exibida. No painel da

esquerda, clique em *Projetos e Soluções*.

3. Na lista de opções mostrada na parte direita da janela, selecione a caixa de verificação *Mostrar janela de saída no início da compilação*.
4. Clique no botão *OK*.

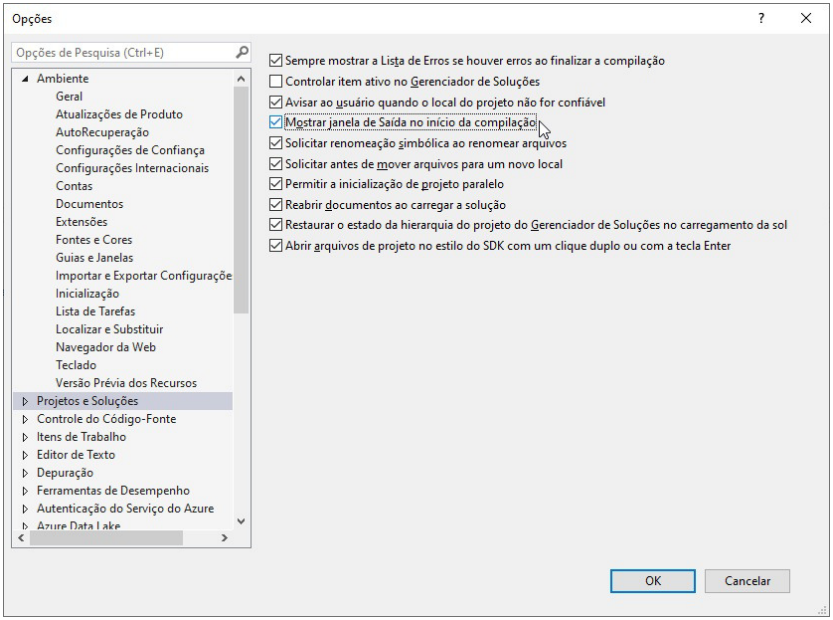


Figura 12.25: Ativando a exibição da janela Saída no início da Compilação

Utilizando a janela Imediata

A janela *Imediata* é usada para depurar e avaliar expressões, executar instruções e imprimir valores de variáveis. Ela funciona como uma janela de linha de comando no Windows ou Linux que permite repetir as últimas entradas digitadas através das setas do cursor e limpar a janela usando a opção *Limpar tudo* presente no menu de contexto.

Para ilustrar o seu uso, vamos utilizar o programa a seguir:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProdutividadeEmCSharp
{
    class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public char Genero { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<Pessoa> pessoas = new List<Pessoa>()
            {
                new Pessoa(){Nome = "Pedro", Idade = 16, Genero =
'M'},
                new Pessoa(){Nome = "Paulo", Idade = 18, Genero =
'M'},
                new Pessoa(){Nome = "Marcela", Idade = 16, Genero
 = 'F'},
                new Pessoa(){Nome = "Roberta", Idade = 21, Genero
 = 'F'},
                new Pessoa(){Nome = "Ricardo", Idade = 19, Genero
 = 'M'},
                new Pessoa(){Nome = "Sofia", Idade = 14, Genero =
'F'},
                new Pessoa(){Nome = "Vanessa", Idade = 22, Genero
 = 'F'},
                new Pessoa(){Nome = "Rodrigo", Idade = 20, Genero
 = 'M'},
                new Pessoa(){Nome = "Rebeca", Idade = 25, Genero
= 'F'},
                new Pessoa(){Nome = "Henrique", Idade = 13, Gener
o = 'M'},
                new Pessoa(){Nome = "Pâmela", Idade = 21, Genero
= 'F'},
                new Pessoa(){Nome = "Alessandra", Idade = 19, Gen
```

```
ero = 'F' }
            };
            Console.ReadLine();
        }
    }
}
```

Ao examinar esse código, note que incluímos uma referência a `System.Linq`. Apesar de não ser usado diretamente pelo programa, ele será necessário para efetuarmos consultas LINQ na lista `pessoas`.

Para exibir a janela Imediata, clique no menu *Depurar*, selecione o submenu *Janelas* e, a seguir, a opção *Imediata* ou pressione *Ctrl+Alt+I*.

Experimente digitar `pessoas` na janela Imediata durante a depuração seguido de *Enter*. Note que ela oferece suporte ao IntelliSense. Será exibido o total de itens na coleção juntamente com os elementos que a compõem. Veja:

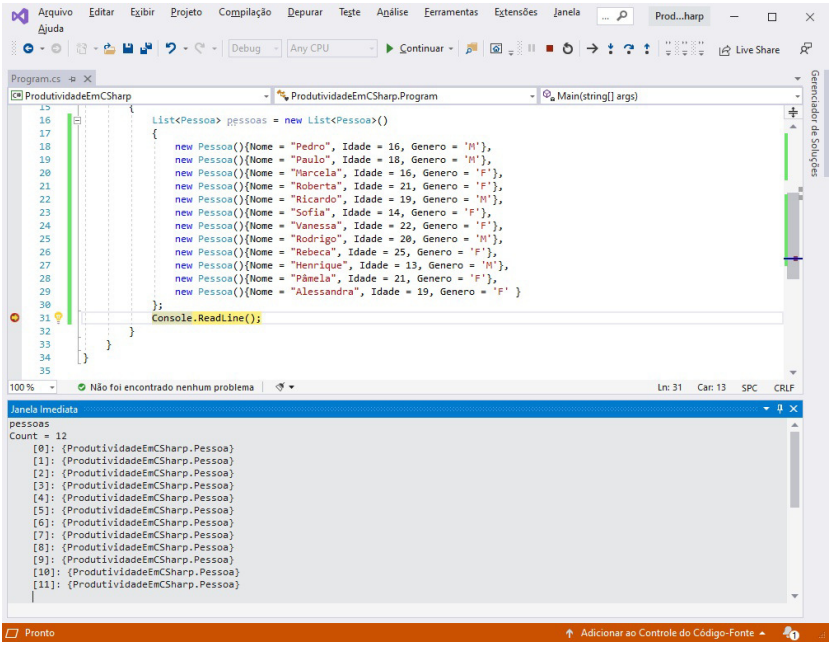


Figura 12.26: Inspecionando a lista pessoas na janela Imediata

Caso deseje saber apenas o total de itens na lista `pessoas`, basta digitar:

```
pessoas.Count
```

Ao analisarmos a lista retornada na janela Imediata vista na imagem anterior, podemos perceber que ela não retorna informações úteis para fins de depuração. Podemos alterar isso facilmente utilizando o atributo de depuração `DebuggerDisplay`, que será explicado em detalhes mais adiante. Para tanto, execute os seguintes passos:

1. Adicione a linha a seguir acima da declaração da classe `Pessoa`:

```
[DebuggerDisplay("Nome: {Nome} - Idade: {Idade} - Gênero: {Genero
.ToString()}")]
```

2. Inclua referência para o namespace `System.Diagnostics` no código do arquivo:

```
using System.Diagnostics;
```

Após efetuarmos as alterações e repetirmos o teste anterior, veremos a seguinte saída:

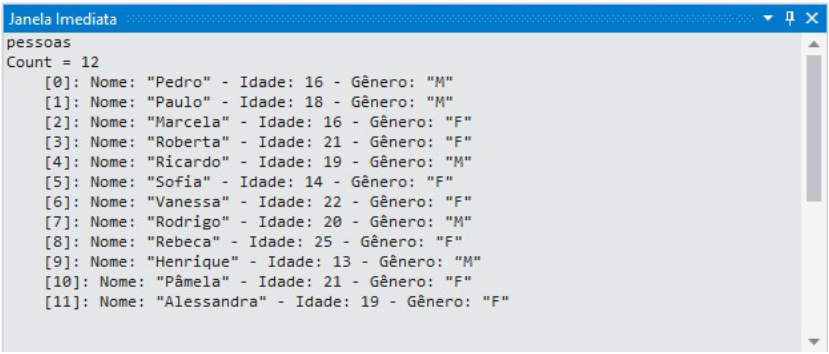


Figura 12.27: Utilizando o atributo DebuggerDisplay para alterar a visualização da lista pessoas

A janela Imediata avalia expressões compilando e usando o projeto atualmente selecionado. Em outras palavras, você pode usar expressões lambdas para descobrir na lista acima todas as mulheres maiores de idade. Basta executar:

```
pessoas.Where(p => p.Idade>=18 && p.Genero == 'F').Select(p => p.
Nome)
```

Utilizando as janelas Autos e Locals

As janelas *Automáticos* e *Locais* (em inglês, *Autos* e *Locals*) exibem valores de variáveis durante uma sessão de depuração de

código-fonte em C#. Essas janelas são realmente úteis quando desejamos monitorar o escopo de objetos ao longo da depuração.

A janela Automáticos mostra as variáveis usadas em torno do ponto de interrupção atual. Em C# e Visual Basic, esta janela exibirá qualquer variável usada na linha atual ou anterior. Para exibi-la durante a depuração, clique no menu *Depurar*, selecione o submenu *Janelas* e, a seguir, a opção *Autos*.

A janela Locais mostra as variáveis definidas no escopo local, que geralmente é o método ou a função atual. Para exibi-la durante a depuração, clique no menu *Depurar*, selecione o submenu *Janelas* e, a seguir, a opção *locais*.

Para entender a diferença entre elas, vamos partir do código do exemplo a seguir:

```
namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int a, b, c, d, e, f, g;
            a = 1;
            b = 2;
            c = 3;
            d = 4;
            e = 5;
            f = 6;
            g = 7;
        }
    }
}
```

Inclua um ponto de interrupção na linha `d = 4` e execute o programa. Exiba as duas janelas e compare as variáveis listadas em

cada uma delas. Confira nas imagens a seguir:

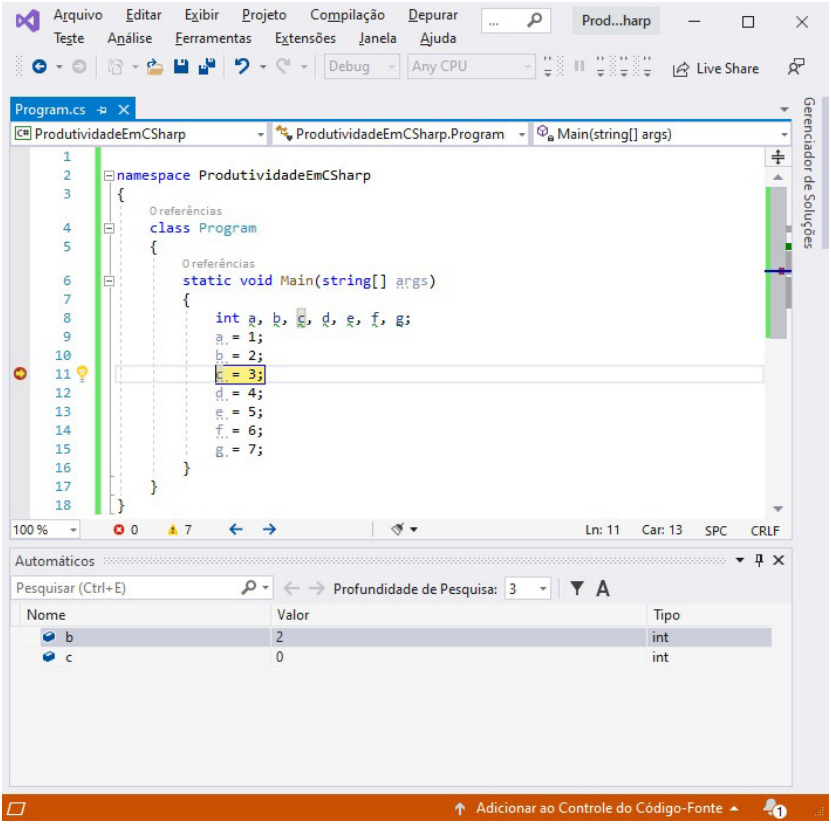


Figura 12.28: Inspecionando variáveis na janela Automáticos

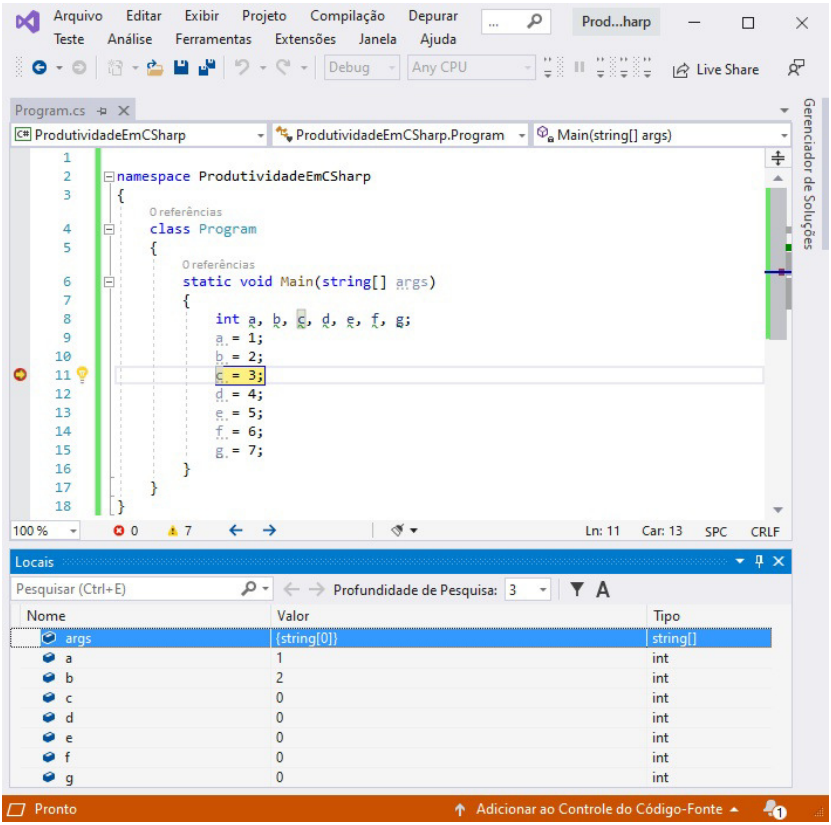


Figura 12.29: Inspecionando variáveis na janela Locais

Um detalhe importante a ser considerado é o de que ambas as janelas podem ser usadas para examinar os valores de retorno de método. Isso é particularmente útil quando os valores não são armazenados em variáveis locais. Um método pode ser usado como um parâmetro ou como o valor retornado de outro método. Confira na imagem a seguir os valores retornados pelos métodos `soma` e `subtracao`:

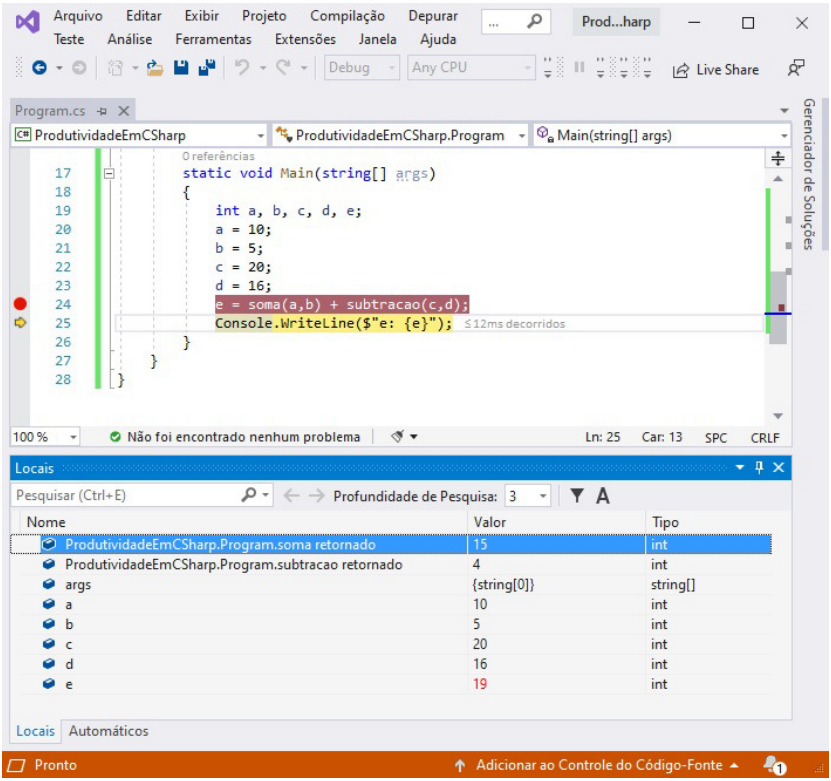


Figura 12.30: Inspecionando os valores retornados por métodos na janela Locais

Matrizes e objetos são exibidos nas janelas de depuração como controles de árvore. Para navegar pela árvore, basta selecionar a seta à esquerda de um nome de variável para expandir a exibição e mostrar campos e propriedades. Exemplo:

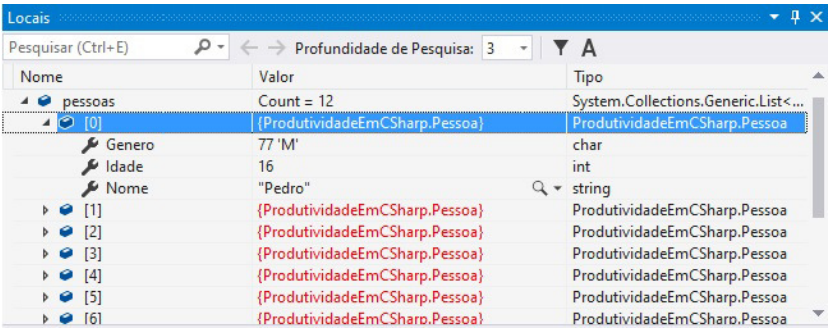


Figura 12.31: Inspecionando objetos na janela Locais

Em projetos escritos em .NET Core 3.0 ou superior, é possível usar os PINs (alfinetes) para fixar membros das classes como favoritos e obter um resultado semelhante ao obtido com o atributo de depuração `DebuggerDisplay` em janelas, como *Locais*, *Automáticos* e *Inspeção*. Veja:

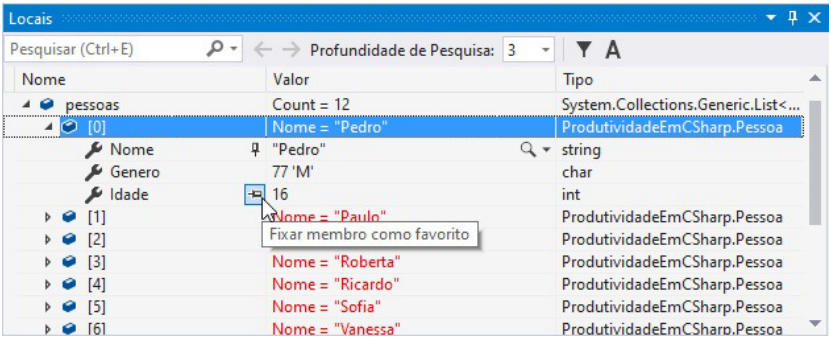


Figura 12.32: Fixando membros como favoritos na janela Locais

O formato numérico padrão nas janelas do depurador é o decimal. Para alterá-lo para hexadecimal, clique com o botão direito na janela e selecione no menu de contexto a opção *Exibição Hexadecimal*.

Para editar os valores das variáveis nas janelas *Automáticos* e *Locais*, efetue um clique duplo no valor atual e digite o novo valor.

Um valor exibido em vermelho nessas janelas indica que o valor foi alterado desde a última avaliação. A alteração pode ter sido realizada pelo próprio desenvolvedor ou ocorrido em uma sessão de depuração anterior.

Pesquisando nas janelas Automáticos, Locais e Inspeção

As janelas de depuração *Automáticos*, *Locais* e *Inspeção* dispõem de uma barra de pesquisa que permite efetuar buscas por palavras-chave nas colunas *Nome*, *Valor* e *Tipo*.

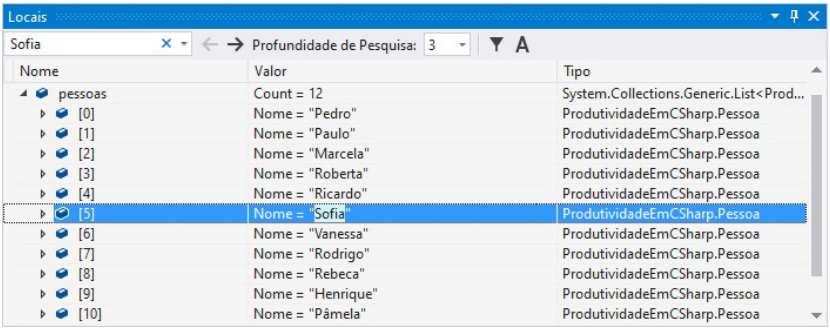


Figura 12.33: Pesquisando na janela Locais

Note que é possível tornar a pesquisa mais ou menos completa, selecionando na lista suspensa *Profundidade de Pesquisa* quantos níveis de profundidade você deseja pesquisar em objetos aninhados.

Utilizando as janelas Inspeção e Inspeção Rápida

Durante a depuração, utilizamos as janelas *Inspeção* e *Inspeção Rápida* (em inglês, *Watch* e *QuickWatch*) para explorar objetos, valores, propriedades e outros objetos aninhados como uma estrutura em árvore. As janelas de inspeção permitem exibir várias variáveis por vez durante a depuração, enquanto a caixa de diálogo *Inspeção Rápida* (QuickWatch) exibe uma única variável por vez e deve ser fechada antes que a depuração possa continuar.

Para ilustrar o uso dessas janelas, vamos partir do exemplo a seguir que calcula o fatorial de um número usando uma função não recursiva:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        public static long Fatorial(int n)
        {
            if (n == 0)
                return 1;
            long valor = 1;
            for (int i = n; i > 0; i--)
            {
                valor *= i;
            }
            return valor;
        }

        static void Main(string[] args)
        {
            long resultado = Fatorial(10);
            Console.WriteLine($"10!: {resultado}");
        }
    }
}
```

Adicione um ponto de interrupção na seguinte linha:

```
long resultado = Fatorial(10);
```

Tecle *F5* para iniciar a depuração e, ao atingir o ponto de interrupção, tecle *F11* para entrar no método `Fatorial` . Para inspecionar os valores das variáveis e parâmetros do método, basta clicar com o botão direito do mouse sobre eles e selecionar no menu de contexto a opção *Adicionar Inspeção*. Isso fará com que a variável ou parâmetro seja exibido na janela *Inspeção 1*. Veja:

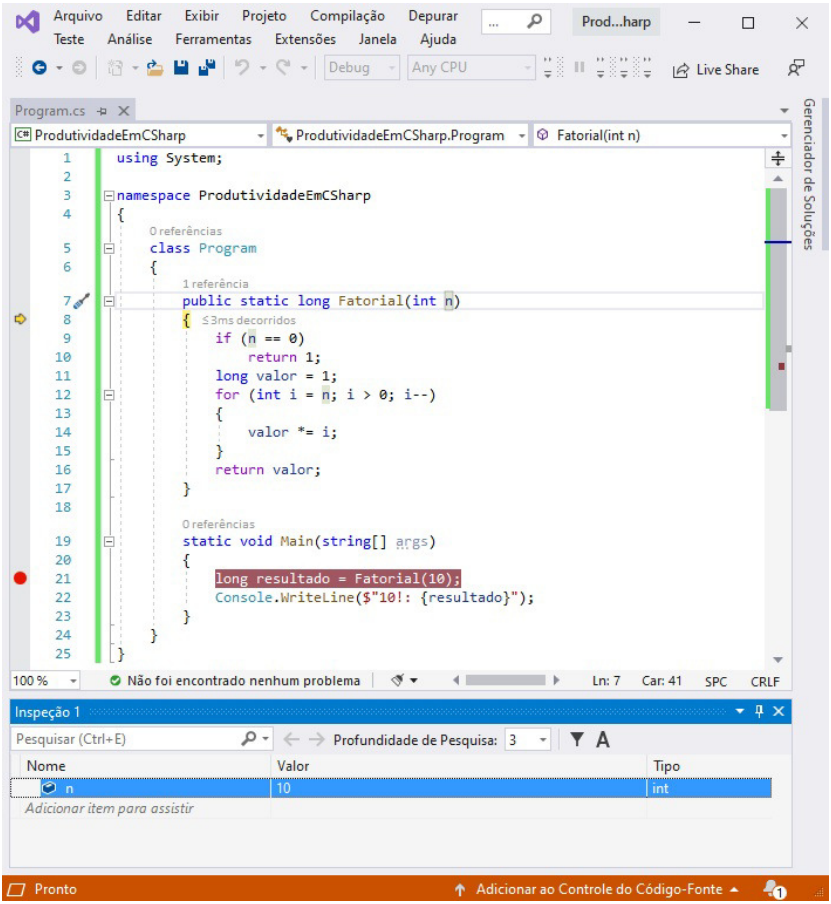


Figura 12.34: Inspecionando o valor de um parâmetro na janela de inspeção

Note que essa janela está numerada porque é possível abrir até quatro janelas de inspeção simultaneamente e observar mais de uma variável em cada uma delas. A maioria dos desenvolvedores se contenta em usar uma única janela de inspeção, mas se desejar abrir janelas extras por alguma necessidade especial ou apenas para testar o recurso, basta clicar no menu *Depurar*, selecionar o submenu *Janelas*, a seguir o submenu *Inspecionar* e, por fim,

selecionar a opção de janela de inspeção desejada.

Ao examinar a janela de inspeção na imagem anterior, repare que na primeira linha livre após a variável não existe uma mensagem *Adicionar item para assistir*. Podemos adicionar manualmente variáveis e expressões válidas nesta janela simplesmente clicando na linha livre e digitando o que desejamos inspecionar. Experimente adicionar a variável `i` durante a depuração do método `Fatorial`. Você verá que, à medida que você avança a depuração iterando pelo loop `for` do método `Fatorial`, os valores das variáveis vão sendo alterados na janela de inspeção.

Uma forma rápida e pouco conhecida de adicionar uma variável ou membro de classe à janela de inspeção é efetuar um duplo clique sobre ele para selecioná-lo no editor de código e arrastá-lo até a janela.

O recurso de arrastar, suportado pela janela de inspeção, é muito flexível e pouco explorado pelos desenvolvedores. Ele nos permite, por exemplo, expandir a coleção `pessoas` do tipo `Pessoa` dentro da janela Inspeção e arrastar um dos seus itens da coleção (`Pessoa[0]`) para baixo a fim de facilitar a sua visualização ou arrastá-lo para uma segunda janela de Inspeção, colocada lado a lado.

Em alguns cenários, você pode desejar comparar as propriedades de um objeto populado do tipo `Pessoa` com uma instância recém-criada do mesmo tipo. Para isso, basta digitar `new Pessoa()` em uma linha da janela de inspeção. Veja:

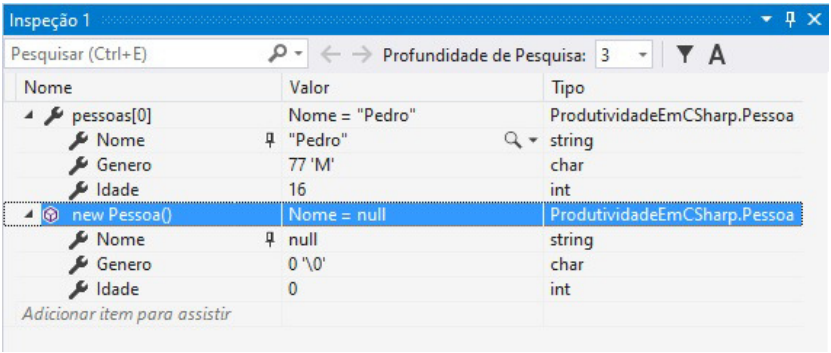


Figura 12.35: Criando uma nova instância da classe na janela de inspeção

Não é possível, entretanto, usar uma instrução como a mostrada a seguir para criar uma nova variável diretamente na janela de inspeção:

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

Para executar essa linha com sucesso, será necessário entrá-la através da janela *Imediata*. Uma vez feito isso, será possível inspecionar a variável `pessoa1` usando a janela de inspeção.

Outra possibilidade a ser explorada pelo desenvolvedor ou desenvolvedora é a capacidade de chamar métodos de um objeto diretamente da janela de inspeção. Supondo que tenhamos um objeto `calc` do tipo `Calculadora` que possui um método `Somar` usado para somar dois valores inteiros, poderíamos digitar na janela de inspeção `calc.Somar(5,6)` e o retorno do método seria mostrado na coluna *Valor*.

Mesmo que a janela de inspeção esteja se tornando mais poderosa e flexível a cada release do Visual Studio, em alguns casos você poderá preferir utilizar a janela *Inspeção Rápida* que nos permite observar uma única variável ou expressão. Para exibi-la,

clique com o botão direito do mouse sobre a variável ou expressão durante a depuração e selecione no menu de contexto a opção *QuickWatch*. A caixa de diálogo *QuickWatch* será exibida:

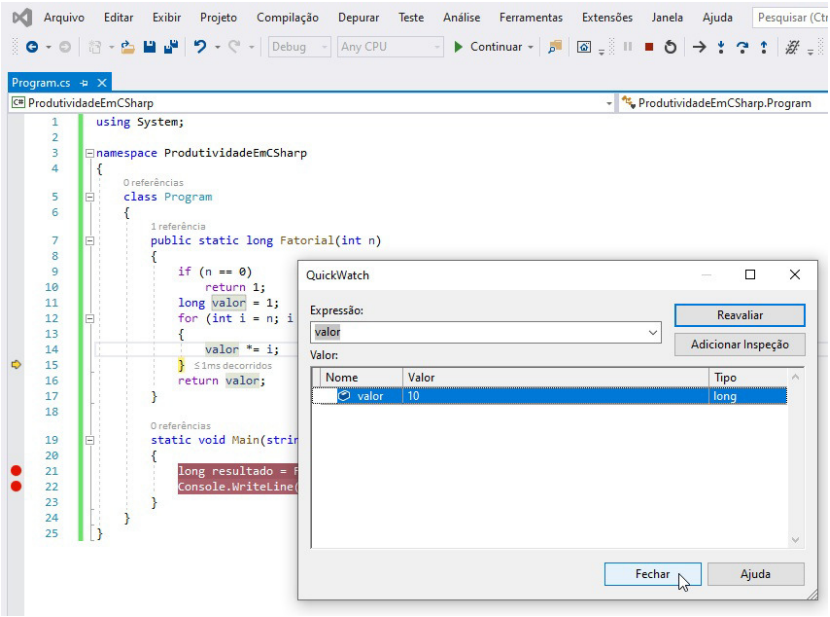


Figura 12.36: Inspecionando o valor de uma variável na janela Inspeção Rápida

Nessa janela, você pode digitar uma expressão, como `valor + n` e clicar no botão *Reavaliar* ou enviar a variável ou expressão atual para a janela de Inspeção clicando no botão *Adicionar Inspeção*. Para avançar com a depuração é necessário clicar no botão *Fechar* para fechar a caixa de diálogo.

Para saber mais sobre as janelas Inspeção e Inspeção Rápida, consulte a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/visualstudio/debugger/watch-and-quickwatch-windows?view=vs-2019

12.9 EXPLORANDO DATATIPS

Ao parar com o mouse sobre variáveis e membros de classe durante a depuração, será mostrada uma *dica de dados* (em inglês, *DataTip*) contendo seu valor atual. Veja na imagem a seguir:

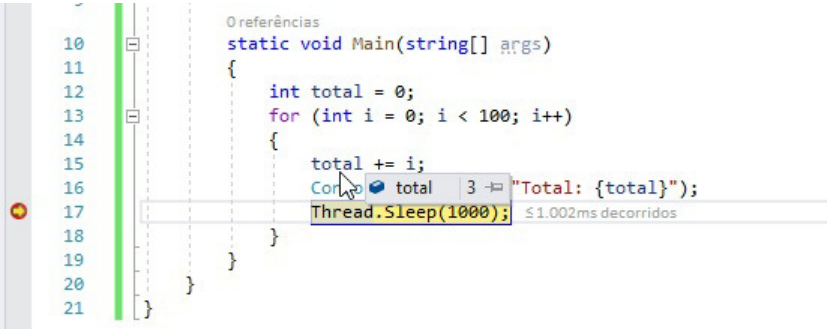


Figura 12.37: Exibindo uma DataTip para uma variável

Para ocultar uma DataTip e inspecionar o que está embaixo, basta manter pressionada a tecla *Ctrl* e ela se tornará temporariamente invisível.

Para alterar o valor da variável usando uma DataTip, efetue um clique no valor atual para editá-lo e digite o novo valor seguido de

Enter. Esse é o modo mais rápido de se atribuir um novo valor a uma variável.

Ao clicarmos com o botão direito do mouse sobre a expressão em uma DataTip, veremos várias opções interessantes no menu de contexto:

- *Copiar* - Copia para a área de transferência tanto a Expressão quanto o Valor (total = 5).
- *Copiar Expressão* - Copia para a área de transferência só a Expressão (total).
- *Copiar Valor* - Copia para a área de transferência só o valor (5).
- *Adicionar Inspeção* - Adiciona expressão à janela *Inspeção*.
- *Adicionar Inspeção Paralela* - Adiciona expressão à janela *Inspeção Paralela*.

Caso deseje permanecer com o valor da variável visível enquanto avança com a depuração, basta fixar a dica de dados no editor. Para isso, clique no ícone de alfinete. Uma vez fixada a dica com o alfinete, é possível clicar com o mouse na DataTip e arrastá-la para o local mais adequado dentro do editor de código. Também é possível fornecer um comentário para uma DataTip. Veja:

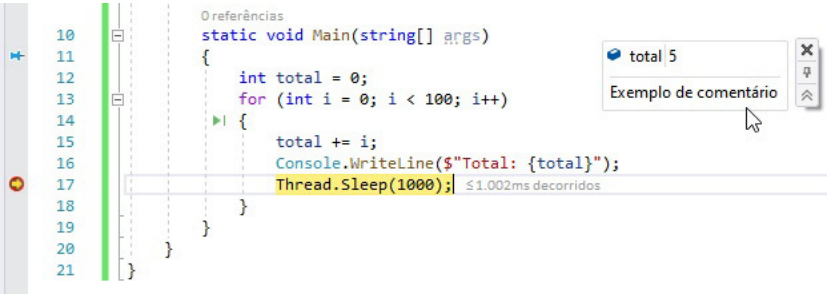


Figura 12.38: Exemplo de DataTip fixada e com um comentário

O suporte à fixação de DataTips é particularmente útil quando passamos pelo mesmo ponto de interrupção várias vezes, como em um loop.

Exportando e importando DataTips

Assim como ocorre com os pontos de interrupção, temos a opção de exportar as DataTips para um arquivo XML que pode ser compartilhado com outros desenvolvedores.

Para exportar as DataTips, execute os seguintes passos:

1. No menu *Depurar*, selecione a opção *Exportar DataTips*.
2. A caixa de diálogo *Exportar DataTips* será exibida. Navegue até o local onde deseja salvar o arquivo XML, digite um nome para o arquivo e selecione *Salvar*.

Para importar as DataTips previamente criadas, execute o roteiro a seguir:

1. No menu *Depurar*, selecione a opção *Importar DataTips*.
2. A caixa de diálogo *Importar DataTips* será exibida. Selecione o arquivo XML contendo as DataTips que você deseja abrir e clique em seguida no botão *Abrir*.

Utilizando visualizadores de dados para inspecionar tipos de dados complexos

A presença de um ícone de lupa ao lado de uma variável ou propriedade em uma DataTip ou em uma das janelas de depuração indica que um ou mais visualizadores estão disponíveis para esse elemento do código. Os visualizadores exibem informações de uma maneira mais significativa e navegável.

Para exibir o elemento usando o visualizador padrão para o tipo de dados, basta clicar no ícone da lupa. Caso existam vários visualizadores disponíveis, é possível selecionar o desejado clicando na seta ao lado do ícone de lupa e escolhendo uma das opções no menu mostrado. Veja:

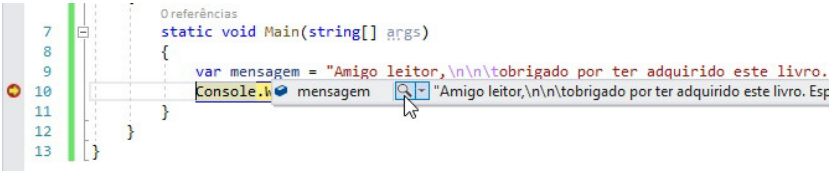


Figura 12.39: Acessando um visualizador

Na imagem a seguir, podemos observar o visualizador de texto exibindo uma string que contém quebras de linha:

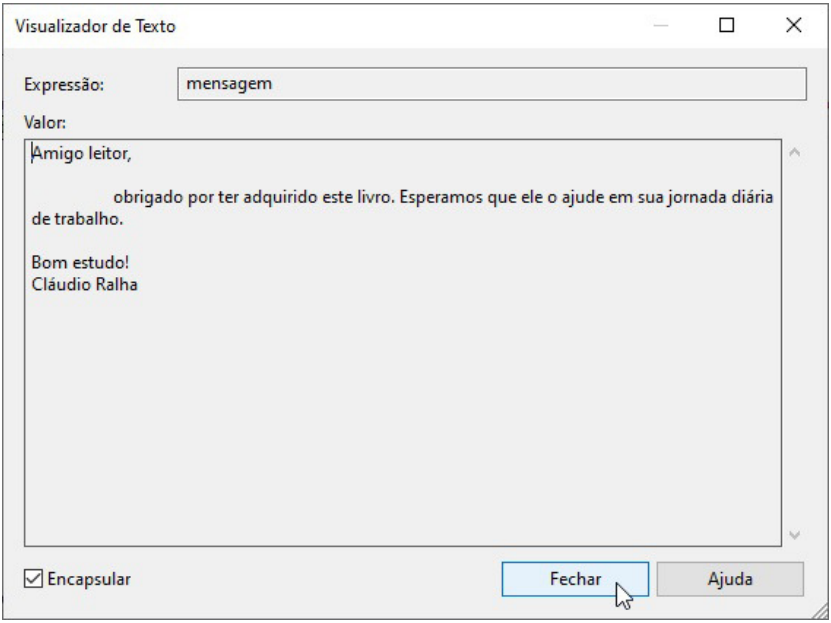


Figura 12.40: Inspecionando o valor da variável mensagem no visualizador de texto

No Visual Studio, temos visualizadores de cadeias de caracteres internos para texto sem formatação, XML, HTML e JSON. Estão disponíveis também visualizadores internos para alguns outros tipos, como objetos DataSet, DataTable e DataView.

Criando visualizadores de dados

Caso nenhum dos visualizadores de dados internos do Visual Studio atenda a sua necessidade, saiba que é possível escrever seu próprio visualizador usando C# ou Visual Basic ou baixar visualizadores extras do Visual Studio Marketplace. Para tanto, pesquise por `tag:Debugger Visualizer` .

> Para criar seus próprios visualizadores, basta seguir os passos descritos em detalhes nas seguintes páginas:
>
> https://docs.microsoft.com/pt-br/visualstudio/debugger/create-custom-visualizers-of-data?view=vs-2019
>
> https://docs.microsoft.com/pt-br/visualstudio/debugger/walkthrough-writing-a-visualizer-in-csharp?view=vs-2019

Capítulo 13

ATRIBUTOS DE DEPURAÇÃO

Ao longo do capítulo anterior, abordamos um grande número de ferramentas que tornam mais simples a tarefa de depuração. As ferramentas mencionadas podem ser usadas sem restrições, uma vez que estão disponíveis em todas as edições do Visual Studio (incluindo a gratuita) e o seu uso não implica alterações no código-fonte dos projetos.

Neste capítulo, mergulharemos um pouco mais nas técnicas de debug e focaremos em *atributos de depuração*, um tipo de notação que pode ser adicionada ao código para tornar a depuração do projeto mais rápida e precisa. A partir desse ponto, será necessário efetuar alterações nos códigos das classes, mas você verá que o resultado justifica o esforço e que a mudança não afetará o código final voltado para release.

Nas próximas seções, você aprenderá a utilizar os atributos `DebuggerStepThrough`, `DebuggerHidden` e `DebuggerNonUserCode` para acelerar o debug do código-fonte pulando partes desnecessárias geradas de forma automática por designers ou já bem testadas.

Em seguida, veremos como empregar os *atributos de exibição do depurador* `DebuggerDisplay`, `DebuggerBrowsable` e `DebuggerTypeProxy`, que nos permitem controlar como os dados serão exibidos. Conforme você pode perceber, todos os atributos de depuração possuem o prefixo `debugger`. Por esse motivo, não é necessário mencionar o prefixo, a exemplo do que fazemos ao omitirmos o sufixo `controller` no nome das controllers em ASP.NET e ASP.NET Core.

Tenha em mente que não existe um jeito certo e único de se depurar. Assim como cada músico profissional é capaz de tocar um instrumento musical de forma única, cada desenvolvedor é capaz de combinar o leque de ferramentas que apresentamos de maneira personalizada a fim de encontrar soluções para os problemas com os quais se depara.

Depurar é um momento especial no desenvolvimento e merece ter seu próprio ritual. Como regra geral, eu costumo reservar sempre o início do dia para corrigir erros em códigos mais complexos, pois é o período em que estamos mais descansados e aquele no qual é possível fugir mais facilmente das distrações. Uma boa playlist de músicas relaxantes e uma xícara de café bem quente são ótimas companhias para estes momentos em que beiramos o Neo desviando das balas da Matrix.

13.1 EMPREGANDO OS ATRIBUTOS DE DEPURAÇÃO STEPTHROUGH, HIDDEN E NONUSERCODE

O C# suporta atributos que nos ajudam a pular algumas partes do código-fonte durante a depuração: `DebuggerStepThrough`,

`DebuggerHidden` e `DebuggerNonUserCode`. Vejamos a função de cada um deles.

Ao decorar o código com o atributo `DebuggerStepThrough`, faremos com que ele seja executado, mas não depurado, mesmo que tenha sido definido um ponto de interrupção dentro do bloco de código. Esse atributo pode ser aplicado em: classe, estrutura, construtor e método.

O atributo `DebuggerHidden` também impede os desenvolvedores de depurar o código e pode ser usado com os seguintes membros: construtor, método e propriedade.

Para ilustrar como utilizar o atributo `DebuggerStepThrough`, vamos partir de um exemplo simples:

```
using System;
using System.Diagnostics;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Metodo1();
        }

        [DebuggerStepThrough]
        public static void Metodo1()
        {
            Console.WriteLine("Aqui a execução não para.");
            Metodo2();
        }

        private static void Metodo2()
        {
            Console.WriteLine("Aqui a execução não para.");
        }
```

```
    }
}
```

Marque dois pontos de interrupção no código conforme mostrado na imagem a seguir. Para criar o ponto de interrupção, basta clicar com o mouse na calha à esquerda da listagem no ponto desejado, ou parar com o cursor de inserção na linha em questão e teclar *F9*. Para executar o programa em modo depuração, basta teclar *F5*.

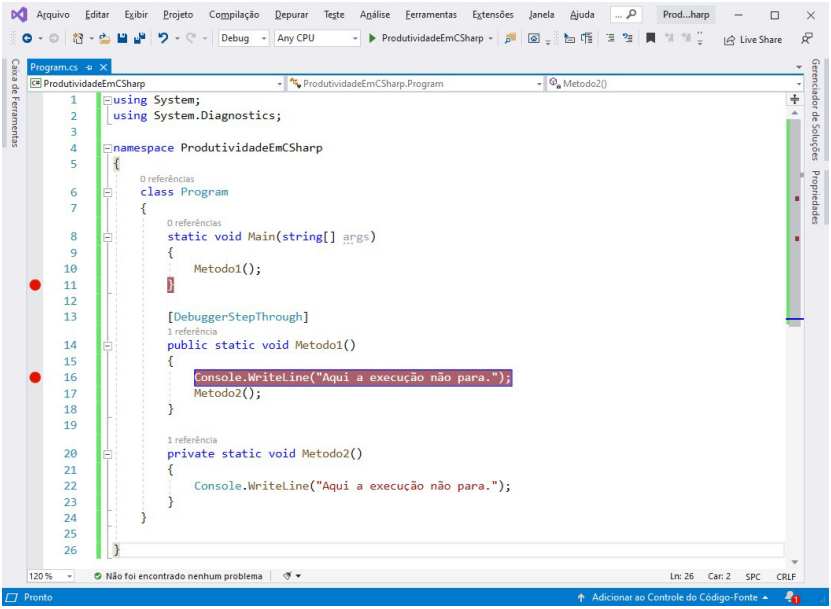


Figura 13.1: Marcando pontos de interrupção no programa de teste

Ao executar o programa em modo depuração, lembre-se de que o ponto de interrupção "não atingível" estará sempre no modo de depuração marcado com um ícone de "ponto de exclamação". Veja:

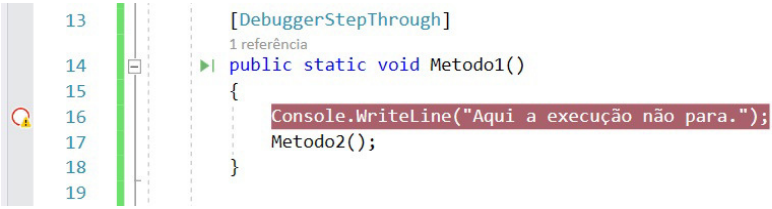


Figura 13.2: Sinalização de um ponto de interrupção não atingível

Repita o teste substituindo o código `[DebuggerStepThrough]` por `[DebuggerHidden]`.

Neste ponto, você provavelmente deve estar curioso sobre as diferenças entre os dois atributos, já que à primeira vista parecem realizar a mesma função. A primeira diferença é que não podemos decorar uma classe ou uma estrutura com o atributo `DebuggerHidden`, só podemos usá-lo com os membros internos. A segunda diferença é a de que o código decorado com `DebuggerStepThrough` será mostrado na janela *Pilha de chamadas* como 'código externo'. O mesmo não ocorre se utilizarmos o atributo `DebuggerHidden`.

Já o atributo `DebuggerNonUserCode` foi criado para sinalizar códigos que não foram criados especificamente pelo usuário, por exemplo, por um *Designer* do Visual Studio de forma automática, e que não devem ser mostrados na janela do depurador para não complicar a experiência de depuração.

Para exibir a janela *Pilha de chamadas* e efetuar testes em seus programas, clique no menu *Depurar*, selecione o submenu *Janelas* e, a seguir, a opção *Pilha de Chamadas*.

Para saber mais sobre os atributos `DebuggerStepThrough`, `DebuggerHidden` e `DebuggerNonUserCode`, consulte respectivamente as seguintes páginas da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/api/system.diagnostics.debuggerstepthroughattribute?view=net-5.0

https://docs.microsoft.com/pt-br/dotnet/api/system.diagnostics.debuggerhiddenattribute?view=net-5.0

https://docs.microsoft.com/pt-br/dotnet/api/system.diagnostics.debuggernonusercodeattribute?view=net-5.0

13.2 CONTROLANDO A EXIBIÇÃO COM O ATRIBUTO DE DEPURAÇÃO DISPLAY

O atributo `DebuggerDisplay` controla como um objeto, propriedade ou campo é exibido nas janelas de variáveis do depurador. Esse atributo pode ser aplicado a assemblies, delegados, propriedades, campos e tipos.

Para ilustrar o seu uso, vamos partir do programa a seguir:

```
using System;

namespace ProdutividadeEmCSharp
{
```

```
    class Livro
    {
        public string Titulo { get; set; }
        public string Autor { get; set; }
        public int AnoLancamento { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var livro = new Livro { Titulo="Produtividade em C#",
 Autor="Cláudio Ralha", AnoLancamento=2019};
            Console.WriteLine($"Título: {livro.Titulo}   Autor: {
livro.Autor}   Ano: {livro.AnoLancamento}");
        }
    }
}
```

Para inspecionar o valor da variável `livro`, vamos executar os seguintes passos:

1. Adicione um *breakpoint* na linha que contém a definição da variável `livro`, clicando à esquerda do número de linha (linha 15 na imagem a seguir).

2. Tecle *F5* para executar o programa com o depurador ativo.

3. Utilize *F10* para avançar linha a linha.

4. Ao ultrapassar a linha com o breakpoint, clique com o botão direito do mouse sobre a variável `livro` e selecione no menu de contexto a opção *Adicionar Inspeção*. A janela *Inspeção 1* será exibida. Veja que a coluna *Valor* referente à variável `livro` não traz nenhuma informação útil:

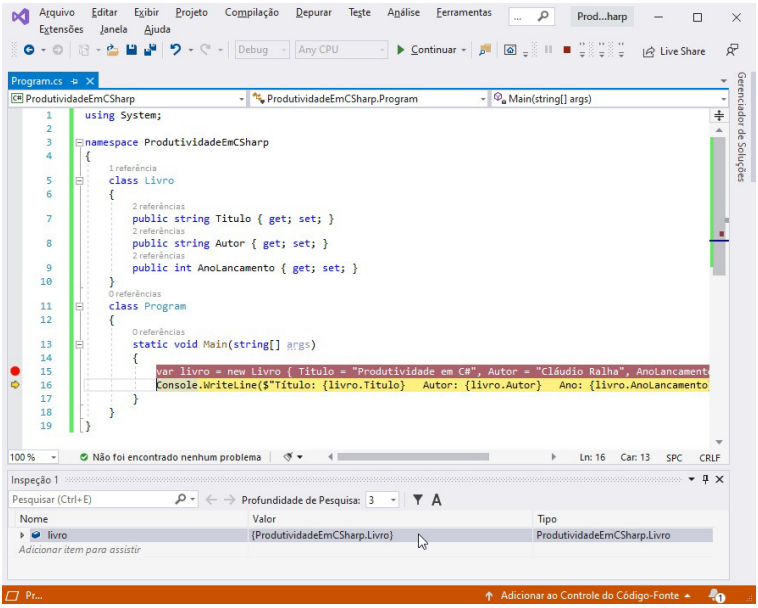


Figura 13.3: Ausência de informação útil na coluna Valor da ferramenta de Inspeção

5. Interrompa a depuração clicando no botão com o quadrado vermelho na barra de botões. Adicione a linha a seguir acima da declaração da classe `Livro` :

```
[DebuggerDisplay("Titulo: {Titulo} - Autor: {Autor}")]
```

E inclua referência para o namespace `System.Diagnostics` no código do arquivo:

```
using System.Diagnostics;
```

Ao repetir o processo de depuração descrito no roteiro anterior, você verá que agora a coluna *Valor* apresenta informação útil para o desenvolvedor sobre o objeto. Confira na imagem a seguir:

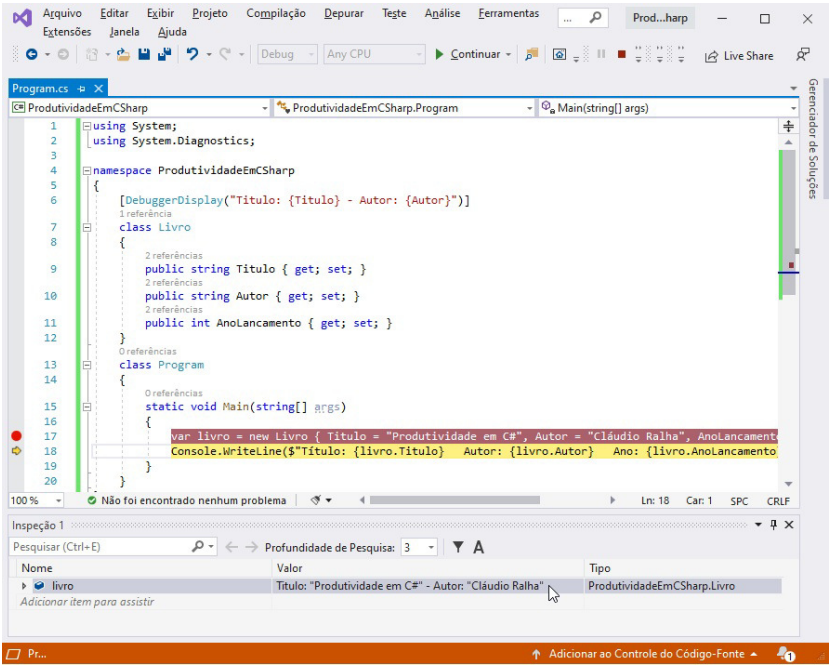


Figura 13.4: Exibindo informações úteis na coluna Valor usando o atributo DebuggerDisplay

O atributo `DebuggerDisplay` possui um único argumento, que é uma cadeia de caracteres a ser exibida na coluna de valor para instâncias do tipo. Essa cadeia de caracteres pode conter chaves (`{` e `}`) e o texto dentro de um par de chaves é avaliado como um campo, propriedade ou método da classe. No exemplo anterior, aplicamos o atributo em nível do tipo, mas também poderíamos tê-lo usado em uma propriedade ou campo específico.

Vale destacar que se você sobrescrever o método `ToString()` da classe, o depurador vai usar o método substituído em vez do padrão que retorna apenas o nome do tipo `{<typeName>}` . Neste caso, você não precisará usar `DebuggerDisplay` . Caso ambos estejam presentes, o atributo `DebuggerDisplay` terá precedência

sobre o método `ToString()` substituído.

Infelizmente, sobrescrever o método `ToString()` de uma classe nem sempre é uma opção viável. Existirão casos em que não desejaremos substituir o método `ToString()` apenas para fins de depuração e outros em que simplesmente não poderemos fazer a sobrescrita, pois a classe é definida em um assembly de terceiros. Para esses cenários, o uso do atributo `DebuggerDisplay` é uma excelente alternativa.

> Para saber mais sobre o atributo `DebuggerDisplay`, consulte as seguintes páginas da documentação oficial:
>
> https://docs.microsoft.com/pt-br/dotnet/api/system.diagnostics.debuggerdisplayattribute?view=net-5.0

13.3 DEFININDO O QUE SERÁ EXIBIDO COM O ATRIBUTO DE DEPURAÇÃO BROWSABLE

O atributo `DebuggerBrowsable` é utilizado para especificar como o campo ou a propriedade deve ser exibida na janela do depurador. O construtor desse atributo usa um dos valores de enumeração `DebuggerBrowsableState`, que especifica um dos seguintes estados:

- `Never`: indica que o membro não é exibido na janela de dados, ou seja, o campo não é exibido quando você expande o tipo delimitador clicando no sinal de adição (+)

da instância de tipo.

- `Collapsed` : indica que o membro é exibido, mas não expandido por padrão. Este é o comportamento padrão.
- `RootHidden` : sendo uma matriz ou uma coleção, esse estado indica que o próprio membro não é mostrado, mas seus objetos constituintes são exibidos.

Vejamos a seguir um exemplo, inicialmente sem o uso do atributo `DebuggerBrowsable` :

```
using System;
using System.Diagnostics;

namespace ProdutividadeEmCSharp
{
    class Endereco
    {
        public string Logradouro { get; set; }
        public string Bairro { get; set; }
        public string CEP { get; set; }
        public string Cidade { get; set; }
        public string Estado { get; set; }
        public string Pais { get; set; }
    }
    class Usuario
    {
        public string Nome { get; set; }
        public string Login { get; set; }
        public string Password { get; set; }
        public Endereco Endereco { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var usuario = new Usuario() {
                                Nome = "Cláudio Ralha",
                                Login = "claudioralha",
                                Password = "naoconto",
```

```
                                Endereco = new Endereco() {
                                    Logradouro = "Rua dos Nerds,
123",
                                    Bairro = "Alto da Boa Vista",
                                    CEP = "18700-000",
                                    Cidade = "Avaré",
                                    Estado = "São Paulo",
                                    Pais = "Brasil"
                                }
            };
            Console.ReadKey();
        }
    }
}
```

Ao inspecionarmos o objeto `usuario` na janela de inspeção do Visual Studio, veremos a seguinte saída:

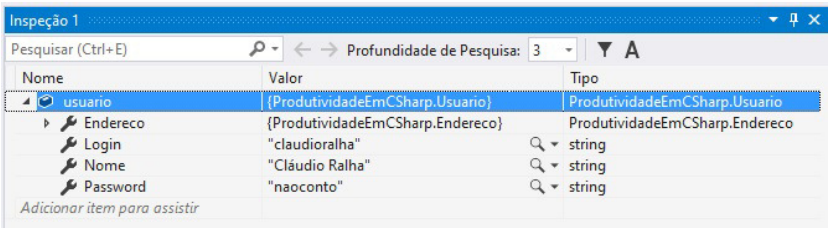


Figura 13.5: Inspecionando as propriedades do objeto usuario

Note que a senha está sendo exibida e que o endereço do usuário aparece colapsado. Vamos agora alterar esse comportamento, modificando o código da classe `Usuario` conforme mostrado na listagem a seguir:

```
class Usuario
{
    public string Nome { get; set; }
    public string Login { get; set; }

    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    public string Password { get; set; }
```

```
    [DebuggerBrowsable(DebuggerBrowsableState.RootHidden)]
    public Endereco Endereco { get; set; }
}
```

Ao inspecionarmos novamente o objeto `usuario`, veremos a seguinte saída na janela de inspeção:

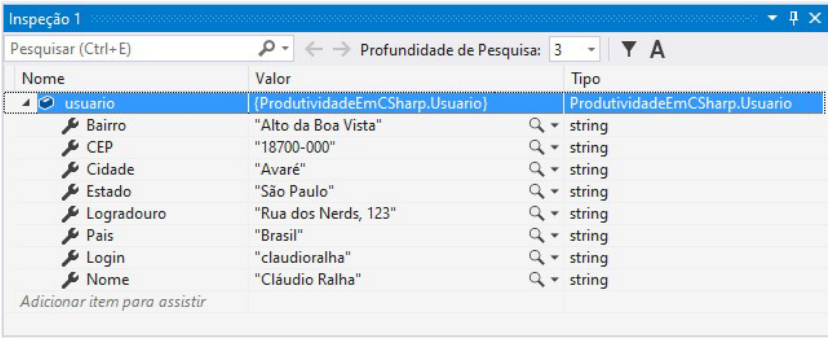


Figura 13.6: Ocultando informações durante a depuração com o atributo DebuggerBrowsable

Quando bem utilizada, esta capacidade de ocultar informações não relevantes dos objetos para fins de depuração em determinados momentos pode tornar o processo de depuração mais produtivo e menos cansativo.

> Para saber mais sobre o atributo `DebuggerBrowsable`, consulte a seguinte página da documentação oficial:
>
> https://docs.microsoft.com/pt-br/dotnet/api/system.diagnostics.debuggerbrowsableattribute?view=net-5.0

13.4 MELHORANDO A VISUALIZAÇÃO DOS DADOS DURANTE A DEPURAÇÃO COM O ATRIBUTO DE DEPURAÇÃO TYPEPROXY

Apesar de o atributo `DebuggerBrowsable` ser bastante útil na maioria dos casos, quando estivermos lidando com tipos que possuem um grande número de propriedades e campos, incluindo propriedades calculadas, ele não representará a melhor solução.

Para lidar com esses cenários mais avançados de depuração, é possível utilizar o atributo `DebuggerTypeProxy` para indicar exatamente o que desejamos visualizar ao inspecionar o objeto. Para tanto, ele recebe em seu construtor uma classe especial criada pelo desenvolvedor contendo apenas o que é relevante.

Para ilustrar como utilizar esse atributo, vamos alterar o programa da seção anterior conforme mostrado a seguir:

```
using System;
using System.Diagnostics;

namespace ProdutividadeEmCSharp
{
    class Endereco
    {
        public string Logradouro { get; set; }
        public string Bairro { get; set; }
        public string CEP { get; set; }
        public string Cidade { get; set; }
        public string Estado { get; set; }
        public string Pais { get; set; }
    }


    class UsuarioTypeProxy
    {
        private readonly Usuario usuario;
```

```
        public UsuarioTypeProxy(Usuario usuario)
        {
            this.usuario = usuario;
        }

        public string Nome => usuario.Nome;
        public string Login => usuario.Login;

        public string Endereco => $"{usuario.Endereco.Logradouro}
 - {usuario.Endereco.Bairro} - {usuario.Endereco.Cidade} - {usuar
io.Endereco.Estado} - {usuario.Endereco.Pais} - CEP: {usuario.End
ereco.CEP}";
    }

    [DebuggerTypeProxy(typeof(UsuarioTypeProxy))]
    class Usuario
    {
        public string Nome { get; set; }
        public string Login { get; set; }

        public string Password { get; set; }

        public Endereco Endereco { get; set; }

    }
    class Program
    {
        static void Main(string[] args)
        {
            var usuario = new Usuario() {
                                Nome = "Cláudio Ralha",
                                Login = "claudioralha",
                                Password = "naoconto",
                                Endereco = new Endereco() {
                                    Logradouro = "Rua dos Nerds,
123",
                                    Bairro = "Alto da Boa Vista",
                                    CEP = "18700-000",
                                    Cidade = "Avaré",
                                    Estado = "São Paulo",
                                    Pais = "Brasil"
                                }
            };
            Console.ReadKey();
        }
```

```
    }
}
```

Observe que a classe `Usuario` foi decorada com o atributo `DebuggerTypeProxy`, que recebeu como parâmetro a classe `UsuarioTypeProxy`, criada para filtrar o que desejamos visualizar durante a depuração do objeto:

```
[DebuggerTypeProxy(typeof(UsuarioTypeProxy))]
```

Ao inspecionar o objeto na janela de depuração do Visual Studio, você verá que agora apenas as propriedades desejadas são mostradas ao expandir `usuario`. Veja:

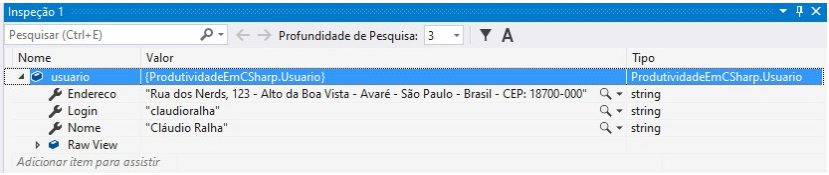


Figura 13.7: Alterando a visualização das propriedades do objeto Usuario com o atributo DebuggerTypeProxy

Note que ainda é possível consultar as demais propriedades, se desejável, expandindo *Raw View*:

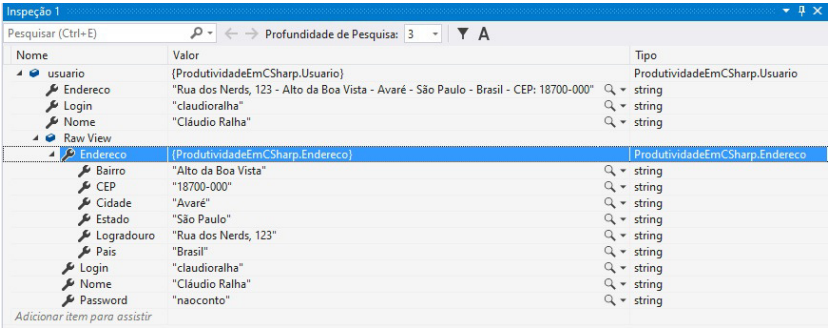


Figura 13.8: Inspecionando as propriedades escondidas pelo atributo DebuggerTypeProxy

Obviamente, criar uma classe como `UsuarioTypeProxy` apenas para fins de depuração só se justifica em cenários complexos para os quais esta abordagem ofereça um ganho real de tempo na depuração.

> Para saber mais sobre o atributo `DebuggerTypeProxy`, consulte a seguinte página da documentação oficial:
>
> https://docs.microsoft.com/pt-br/dotnet/api/system.diagnostics.debuggertypeproxyattribute?view=net-5.0

Conforme você pôde observar ao longo deste capítulo, adicionar atributos de depuração em pontos estratégicos do código pode significar um ganho considerável de tempo no desenvolvimento de um projeto, especialmente no caso de sistemas complexos que envolvem vários desenvolvedores. Em termos práticos, esses atributos funcionam como um tipo de GPS apontando a rota mais rápida para solucionar um problema no código-fonte.

Capítulo 14

COMPILAÇÃO CONDICIONAL

A exemplo de outras linguagens, como o C++, em C# também temos suporte à *compilação condicional*, uma funcionalidade poderosa que utiliza símbolos de pré-processador e diretivas de pré-processamento, como `#define`, `#if`, `#else`, `#elif` e `#endif` para determinar quais partes do código serão compiladas e quais serão excluídas dos conjuntos finais.

A compilação condicional é uma técnica que pode ser usada em um cenário de desenvolvimento entre plataformas para determinar as partes do código que são compiladas especificamente para uma determinada plataforma. Deste modo, é possível compilar condicionalmente qualquer seção de código em C# usando diretivas de pré-processamento. Essas diretivas são instruções especiais para o compilador executadas antes da compilação principal.

Ao longo deste capítulo final, veremos como utilizar diretivas de pré-processamento com constantes definidas pelo usuário e como utilizar as *diretivas de diagnóstico* `#error` e `#warning` para impedir a compilação e sinalizar problemas. Abordaremos a seguir a constante `DEBUG`, que é incluída de forma automática em

modo de depuração, e constantes predefinidas para blocos de código direcionados para estruturas de destino específicas (.NET Framework, .NET Core e .NET Standard) e versões específicas de cada framework. Em seguida, veremos como utilizar o atributo `ConditionalAttribute` como uma alternativa às diretivas condicionais em nossos projetos. O capítulo termina mostrando como utilizar as diretivas `#region` e `#endregion` para organizar melhor o seu código.

14.1 UTILIZANDO DEFINIÇÕES DE SÍMBOLOS E DIRETIVAS DE PRÉ-PROCESSADOR CONDICIONAL

Em C#, os símbolos do pré-processador podem ser definidos ou indefinidos no código durante a compilação. Quando o símbolo que você deseja definir tiver efeitos localizados (afetando apenas um arquivo), é comum criá-lo usando a diretiva de pré-processador `#define` . A diretiva `#define` é seguida pelo nome do símbolo, como mostrado a seguir:

```
#define EdicaoProfissional
```

A declaração `#define` deve aparecer antes de todos os outros códigos em um arquivo, exceto para outras diretivas de pré-processamento. Veja:

```
#define EdicaoProfissional
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
#if EdicaoProfissional
```

```
    const bool SuporteTelefonico = true;
    const bool SuporteViaEmail = true;
#else
        const bool SuporteTelefonico = false;
        const bool SuporteViaEmail = false;
#endif

        static void Main(string[] args)
        {
            Console.WriteLine($"Suporte Telefônico: {SuporteTelef
onico}");
            Console.WriteLine($"Suporte via E-mail: {SuporteViaEm
ail}");
        }
    }
}
```

Note que nenhum valor é aplicado e que a constante definida é case sensitive. A declaração `#define` cria um símbolo que é válido para um único arquivo de código. Isso significa que, se você precisar usar o mesmo símbolo em vários arquivos, terá que definilo mais de uma vez. Tenha em mente que os símbolos não estão relacionados a classes ou outros tipos, portanto, mesmo que você utilize classes parciais, você deverá definir o símbolo em cada arquivo ou usar um símbolo com escopo de projeto para poder utilizá-lo.

Para definir os símbolos do pré-processador com escopo de projeto, é necessário editar as propriedades do projeto no Visual Studio. Execute os seguintes passos:

1. No *Gerenciador de Soluções*, clique com o botão direito do mouse sobre o nome do projeto. No menu de contexto, selecione a opção *Propriedades*.

2. A página de propriedades do projeto será exibida. No menu da esquerda, clique na guia *Build* para ativar a página.

Adicione as constantes desejadas no campo *Símbolos de compilação condicional*, separando cada um com um espaço. Para este exemplo criaremos apenas o símbolo `EdicaoProfissional`. Veja:

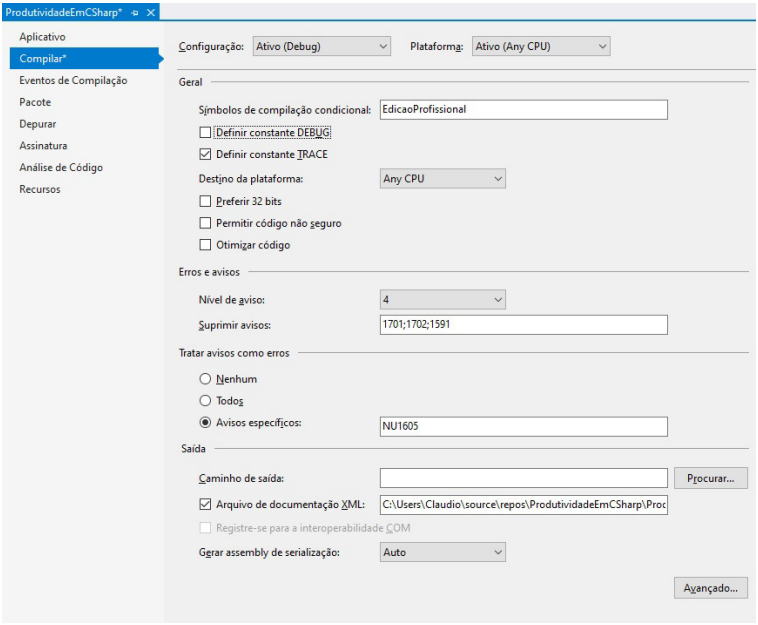


Figura 14.1: Definindo um símbolo de compilação condicional

3. Observe que existe nessa tela uma opção *Definir constante DEBUG* que você também poderá usar para fins de depuração do código. Tecle *Ctrl+s* para salvar a alteração e a seguir feche a página de propriedades do projeto.

Pronto! Isso é tudo que precisamos para ter um símbolo definido com escopo de projeto. Caso precise desligar um símbolo criado com escopo de projeto em um arquivo em particular, basta

utilizar a diretiva `#undefine` da mesma forma que usamos a diretiva `#define`. Exemplo:

```
#undefine EdicaoProfissional
```

Uma vez definidas as constantes em nível de projeto ou de arquivo com a diretiva `#define`, já podemos especificar o que deve ser compilado ou ignorado com base no símbolo de pré-processador que acabamos de definir. Para tanto, utilizamos as diretivas `#if`, `#elif`, `#else` e `#endif` conforme ilustrado no exemplo anterior. Ao ser executado, o exemplo anterior produzirá a seguinte saída (assumindo que você definiu previamente o símbolo `EdicaoProfissional`):

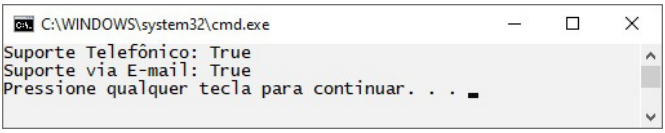


Figura 14.2: Utilizando o símbolo EdicaoProfissional definido previamente

De volta ao editor de código do Visual Studio, observe que o bloco que está desativado será mostrado em cinza, enquanto o bloco ativo terá seu código exibido nas cores normais:

```
7   #if EdicaoProfissional
8       const bool SuporteTelefonico = true;
9       const bool SuporteViaEmail = true;
10  #else
11      const bool SuporteTelefonico = false;
12      const bool SuporteViaEmail = false;
13  #endif
```

Figura 14.3: Exibição do bloco ativo é feito de forma destacada

Caso precise testar mais de um símbolo, é possível incluir uma

ou mais diretivas `#if` conforme mostrado a seguir:

```
#if EdicaoProfissional
    const bool SuporteTelefonico = true;
    const bool SuporteViaEmail = true;
#elif EdicaoStandard
    const bool SuporteTelefonico = false;
    const bool SuporteViaEmail = true;
#else
    const bool SuporteTelefonico = false;
    const bool SuporteViaEmail = false;
#endif
```

14.2 UTILIZANDO AS DIRETIVAS DE DIAGNÓSTICO #ERROR E #WARNING

O C# suporta duas *diretivas de diagnóstico*: `#error` e `#warning`. A `#error` é usada para abortar a compilação, gerando um erro de compilação. Apesar de opcional, como boa prática devemos sempre fornecer uma mensagem de erro ao usarmos essa diretiva. Veja no programa a seguir um exemplo de uso:

```
#define EdicaoProfissional
//#define EdicaoEstudante

using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
#if EdicaoProfissional && EdicaoEstudante
#error O programa não pode ser para profissional e estudante ao m
esmo tempo!
#endif

#if EdicaoProfissional
        const bool SuporteTelefonico = true;
```

```
        const bool SuporteViaEmail = true;
#else
        const bool SuporteTelefonico = false;
        const bool SuporteViaEmail = false;
#endif

        static void Main(string[] args)
        {
            Console.WriteLine($"Suporte Telefônico: {SuporteTelef
onico}");
            Console.WriteLine($"Suporte via E-mail: {SuporteViaEm
ail}");
        }
    }
}
```

Ao examinar o código, note que existe uma linha comentada, na qual definimos a constante `EdicaoEstudante`. Rode a primeira vez da forma como está, em seguida descomente essa linha e tente novamente. Você verá que a linha que contém a diretiva `#error` é sinalizada antes mesmo da compilação, indicando que existe algo errado. Confira na imagem a seguir:

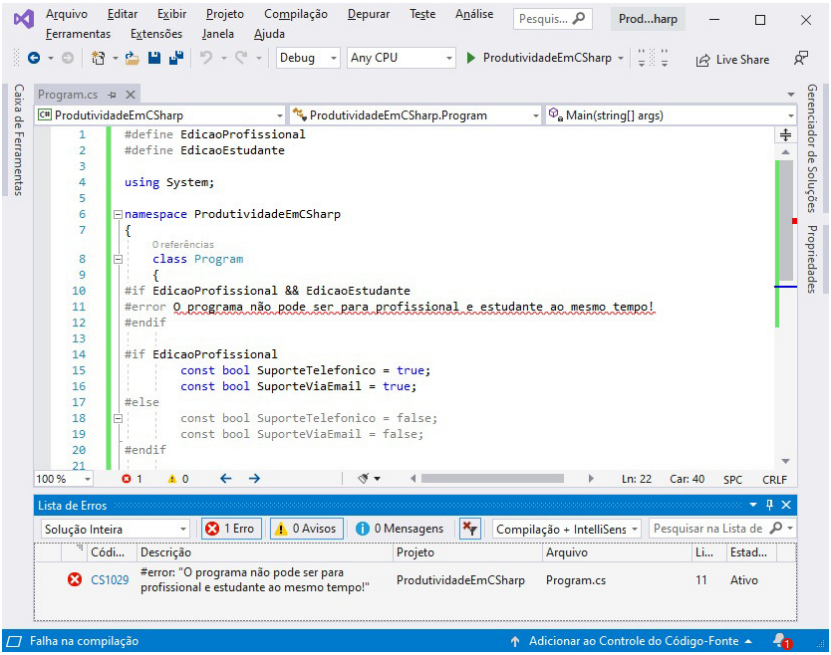


Figura 14.4: Abortando a compilação com a diretiva #error

O erro ocorre nesse caso porque especificamos com a diretiva `#if` que não podemos ter as constantes `EdicaoProfissional` e `EdicaoEstudante` definidas ao mesmo tempo.

O funcionamento da diretiva `#warning` é semelhante ao da `#error`, com a diferença de que a primeira não interrompe a compilação. Essa diretiva gera apenas um *aviso* (em inglês, *warning*).

14.3 UTILIZANDO A CONSTANTE DEBUG

A constante `DEBUG` é inserida de forma automática quando compilamos o programa em modo *Debug* e suprimida quando

selecionamos o modo *Release*.

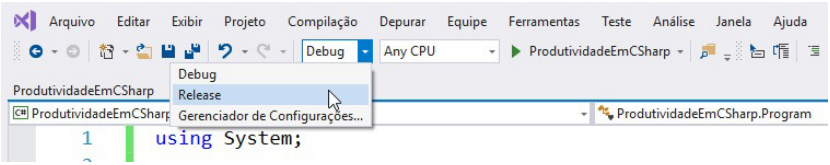


Figura 14.5: Alternando entre os modos Debug e Release

Experimente testar o programa da seção anterior alternando como mostrado a seguir os dois modos para ver a diferença:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
#if DEBUG
    const bool SuporteTelefonico = true;
    const bool SuporteViaEmail = true;
#else
        const bool SuporteTelefonico = false;
        const bool SuporteViaEmail = false;
#endif

        static void Main(string[] args)
        {
            Console.WriteLine($"Suporte Telefônico: {SuporteTelef
onico}");
            Console.WriteLine($"Suporte via E-mail: {SuporteViaEm
ail}");
        }
    }
}
```

Caso precise forçar a definição da constante `DEBUG` em modo *Release* por algum motivo específico, utilize a opção *Definir constante DEBUG* na página de propriedades do projeto:

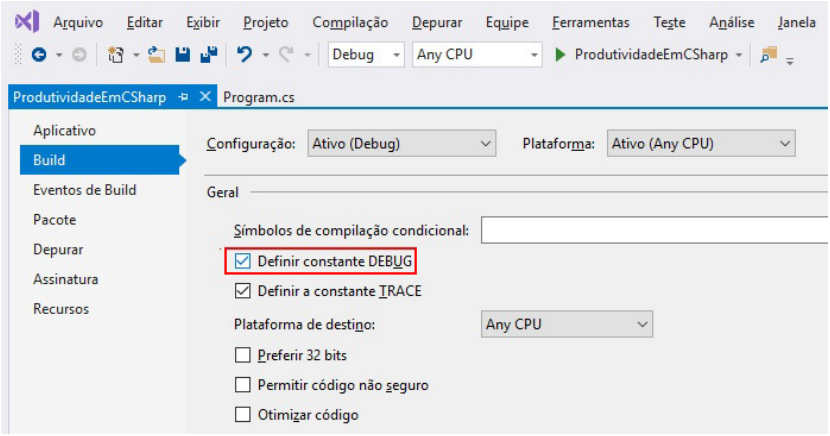


Figura 14.6: Definindo de forma incondicional a constante DEBUG

14.4 EMPREGANDO BLOCOS DE CÓDIGO PARA VERSÕES ESPECÍFICAS DO FRAMEWORK

O sistema de compilação reconhece símbolos de pré-processador predefinidos que representam estruturas de destino diferentes em projetos, o que nos permite criar aplicativos que podem ser direcionados para mais de uma implementação ou versão do .NET.

Confira as constantes definidas até o momento em que este livro foi escrito na tabela a seguir:

Frameworks de destino	Símbolos
.NET Framework	`NETFRAMEWORK` , `NET20` , `NET35` , `NET40` , `NET45` , `NET451` , `NET452` , `NET46` , `NET461` , `NET462` , `NET47` , `NET471` , `NET472` , `NET48`
.NET	`NETSTANDARD` , `NETSTANDARD1_0` , `NETSTANDARD1_1` , `NETSTANDARD1_2` , `NETSTANDARD1_3` , `NETSTANDARD1_4` ,

Standard	`NETSTANDARD1_5`, `NETSTANDARD1_6`, `NETSTANDARD2_0`, `NETSTANDARD2_1`
.NET Core	`NETCOREAPP`, `NETCOREAPP1_0`, `NETCOREAPP1_1`, `NETCOREAPP2_0`, `NETCOREAPP2_1`, `NETCOREAPP2_2`, `NETCOREAPP3_0`, `NETCOREAPP3_1`

O exemplo a seguir ilustra como usar as constantes `NETFRAMEWORK`, `NETSTANDARD` e `NETCOREAPP`:

```
using System;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
#if NETCOREAPP
            Console.WriteLine("Compilado para .NET Core!");
#elif NETFRAMEWORK
            Console.WriteLine("Compilado para .NET Framework!");
#elif NETSTANDARD
            Console.WriteLine("Compilado para .NET Standard!");
#endif
        }
    }
}
```

Ao executar o programa após compilá-lo para .NET Core 3.1, veremos a seguinte saída:

Figura 14.7: Detectando a estrutura de destino usando constantes de compilação

Em termos práticos, a capacidade de identificar as estruturas de destino nos permite utilizar APIs mais recentes, quando

disponíveis.

Para identificar qual estrutura de destino um projeto utiliza, execute os seguintes passos:

1. Clique com o botão direito do mouse sobre o nome do projeto no *Gerenciador de Soluções*. No menu de contexto que será exibido, selecione *Propriedades*.
2. A página de propriedades do projeto será exibida. Selecione a guia *Aplicativo*.
3. Verifique a opção selecionada no campo *Estrutura de Destino*. Veja:

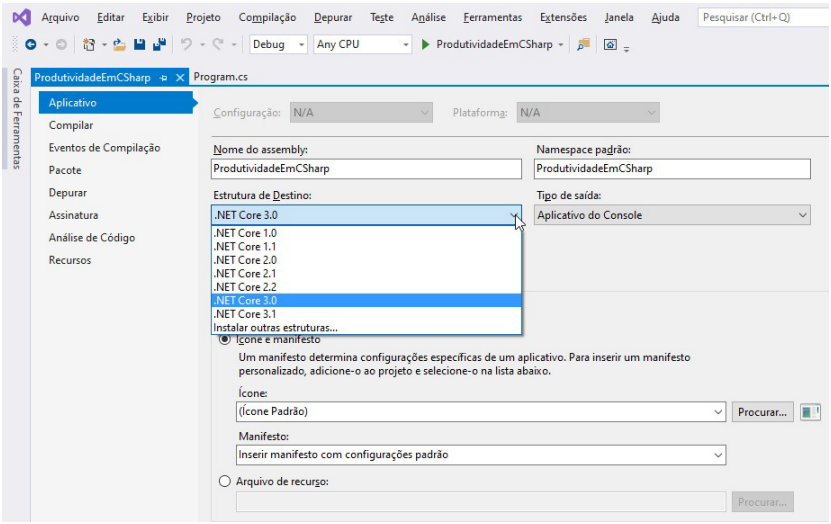


Figura 14.8: Verificando a estrutura de destino na página de propriedades do projeto

14.5 USANDO O ATRIBUTO CONDITIONAL EM UM MÉTODO

Uma alternativa mais moderna e legível ao uso de diretivas de

pré-processamento `#if` é o atributo `ConditionalAttribute`, que nos permite desligar a compilação e a chamada a certos métodos com base nos símbolos que definimos com a diretiva `#define`. Veja a seguir um exemplo bem simples:

```
#define PRODUTIVIDADE
using System;
using System.Diagnostics;

namespace ProdutividadeEmCSharp
{
    class Program
    {
        [Conditional("PRODUTIVIDADE")]
        static void MetodoA()
        {
            Console.WriteLine("Obrigado por adquirir o livro Prod
utividade em C#!");
        }

        [Conditional("DEBUG")]
        static void MetodoB()
        {
            Console.WriteLine("Este livro é fruto de um longo est
udo sobre a linguagem C# desde a sua origem.");
        }

        static void Main(string[] args)
        {
            MetodoA();
            MetodoB();
        }
    }
}
```

Observe que o atributo `ConditionalAttribute` recebe como parâmetro uma literal com o nome do símbolo cuja definição será verificada. Veja também que no início do código-fonte temos apenas a definição da constante `PRODUTIVIDADE`, já que a constante `DEBUG` é gerada de forma automática em modo *Debug*.

Ao ser executado em modo *Debug*, este programa gerará a seguinte saída:

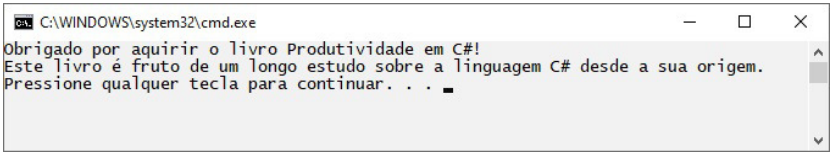


Figura 14.9: Executando o programa em modo Debug

Alterando para modo *Release*, obteremos a saída mostrada na próxima imagem:

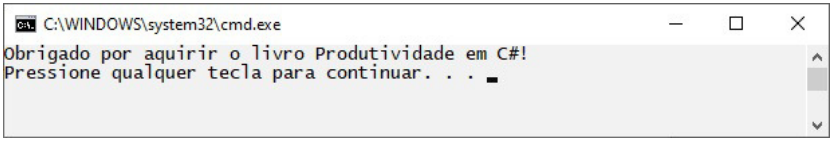


Figura 14.10: Executando o programa em modo Release

Conforme você pode observar, o método `MetodoB` só é executado em modo *Debug*.

Por meio do atributo `ConditionalAttribute` , temos uma forma clara e de fácil manutenção para definir a existência de um método dependendo de uma diretiva de pré-processador. Infelizmente, só podemos definir um método como condicional se ele não retornar um valor, ou seja, se ele retornar void. Obviamente, podemos driblar esta limitação quando necessário usando parâmetros.

Para saber mais sobre o atributo `ConditionalAttribute`, acesse a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/api/system.diagnostics.conditionalattribute?redirectedfrom=MSDN%5C&view=net-5.0

14.6 ATIVANDO O SUPORTE A TIPOS DE REFERÊNCIA ANULÁVEIS

O C# 8.0 introduziu uma distinção entre tipos de referência nulos e não nulos para evitar erros frequentes dos programadores que resultam em exceções do tipo `NullReferenceException`. De forma semelhante aos *tipos de valores anuláveis*, um *tipo de referência anulável* é criado anexando um ponto de interrogação (`?`) ao tipo. Quando estamos trabalhando usando este novo recurso, é possível atribuir um valor nulo apenas ao tipo de referência anulável. Veja:

```
string? titulo = null; // tipo de referência anulável
string autor = "Cláudio Ralha"; // tipo de referência não anuláve
```

O suporte a tipos de referência anuláveis não está ativo por padrão, o que significa que se você tentar usar uma linha de código como a mostrada a seguir:

```
string? titulo = null;
```

Verá a seguinte mensagem de alerta:

A anotação para tipos de referência anuláveis deve ser usada apenas em código dentro de um contexto de anotações '#nullable'.

Para habilitar essa funcionalidade para o projeto inteiro, execute os seguintes passos:

1. Clique com o botão direito do mouse sobre o item do projeto no *Gerenciador de Soluções*. No menu de contexto que será mostrado, selecione *Editar arquivo de projeto*.
2. O arquivo de projeto `.csproj` será aberto. Adicione um elemento `Nullable` ao elemento `PropertyGroup` conforme mostrado a seguir:

```
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
  ...
  <Nullable>enable</Nullable>
</PropertyGroup>
```

```
1  <Project Sdk="Microsoft.NET.Sdk">
2
3    <PropertyGroup>
4      <OutputType>Exe</OutputType>
5      <TargetFramework>netcoreapp3.0</TargetFramework>
6    </PropertyGroup>
7
8    <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
9      <DocumentationFile>C:\Users\Claudio\source\repos\ProdutividadeEmCSharp\Pro
10     <NoWarn>1701;1702;1591</NoWarn>
11     <DefineConstants>TRACE;EdicaoProfissional</DefineConstants>
12     <Nullable>enable</Nullable>
13   </PropertyGroup>
14
15   <ItemGroup>
16     <PackageReference Include="System.ValueTuple" Version="4.5.0" />
17   </ItemGroup>
18
19 </Project>
```

Figura 14.11: Adicionando suporte a tipos de referência anuláveis

Caso deseje ativar o recurso em um único arquivo, utilize a diretiva `#nullable` com o valor `enable` no arquivo deste modo:

```
#nullable enable
string titulo = null; //Isto gera um aviso
```

Com o suporte a tipos de referência anuláveis ativado, qualquer atribuição de valor nulo a tipos de referência não anuláveis provocará um alerta de compilação com a mensagem: `Conversão de literal nula ou possível valor nulo em tipo não anulável`.

O Visual Studio oferece correção para o problema por meio de uma ação rápida. Veja:

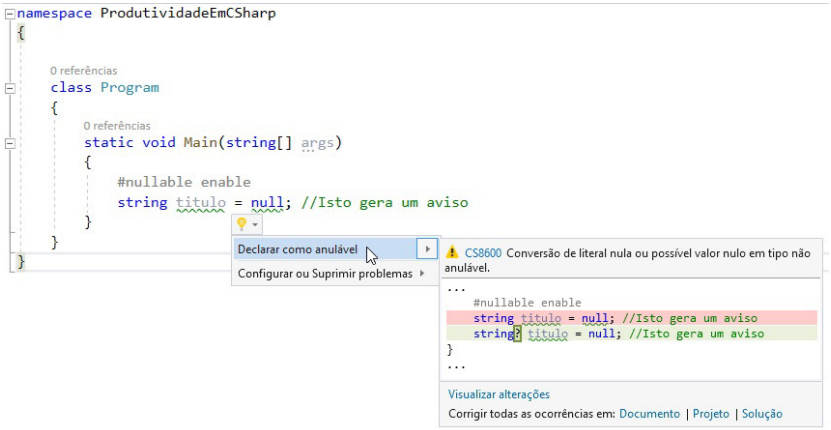


Figura 14.12: Convertendo a declaração para um tipo de referência anulável

As regras passam a ser as seguintes:

- Os tipos de referência não anuláveis não precisam ser verificados como nulos antes que eles sejam desreferenciados.

- Uma verificação nula é necessária para remover o aviso em locais que a ação de desreferenciar uma referência anulável possa gerar uma exceção.

Veja a seguir um exemplo:

```
namespace ProdutividadeEmCSharp
{

    class Program
    {
        static void Main(string[] args)
        {
            #nullable enable
            string? titulo = null;
            int tamanho1 = titulo.Length; //Isto gera aviso
            if (titulo != null)
            {
                int tamanho2 = titulo.Length; //Isto não gera avi
so
            }
        }
    }
}
```

Saiba que este comportamento pode ser modificado usando o *operador null-tolerante* (!) (em inglês, *null-forgiving operator*) incluído no C# 8.0. Ele é útil para suprimir o aviso nos casos em que temos certeza de que a variável anulável não está definida como nula. Exemplo:

```
int tamanho2 = titulo!.Length;
```

Para saber mais sobre o operador *null-forgiving*, consulte a seguinte página da documentação oficial:

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/null-forgiving

14.7 AGRUPANDO OS MEMBROS DAS CLASSES EM REGIÕES

As diretivas de pré-processador `#region` e `#endregion` são usadas para criar regiões no código-fonte.

Uma *região* é uma forma elegante e simples de agrupar os membros de uma classe, enquanto fornece um nome significativo para ela. Desse modo, é possível com um clique de mouse ocultar ou expandir o código contido na região. Para declarar uma região usamos a sintaxe a seguir:

```
#region Nome_da_região

#endregion
```

Vejamos um exemplo:

```
public class Livro
{
    #region Propriedades

    public string Nome { get; set; }

    public int Id { get; }

    public string Log { private get; set; }
```

```
    public string Titulo { get; set; } = "Produtividade em C#";

    public string Autor { get; set; } = "Cláudio Ralha";

    public DateTime DataCadastramento { get; private set; } = Dat
eTime.Now;

    public decimal Preco { get; private set; } = 50;

    #endregion

    #region Métodos

    //TODO: Implementar métodos da classe

    #endregion
}
```

O poder desse recurso está na capacidade de ocultar ou exibir o código interno da região, o que pode ser feito clicando no controle existente à esquerda da declaração da região.

O agrupamento do código em regiões se torna ainda mais valoroso para a organização do código quando descobrimos que é possível declarar regiões aninhadas, ou seja, uma região dentro de outras ou uma região dentro de um método longo. Também é possível selecionar todo o código de uma região colapsada com um clique do mouse sobre o número de linha inicial da região e arrastar o código para uma nova posição dentro do arquivo.

A organização do código de uma classe em regiões pode ser baseada na natureza dos membros, como propriedades, construtores e métodos; ou recorrendo a alguma outra forma de organização que você julgue mais adequada para as classes dos seus projetos (métodos de CRUD, métodos de importação e exportação de dados etc.).

Conforme você pôde observar ao longo deste capítulo, a linguagem C# oferece recursos interessantes de compilação condicional por meio de suas diretivas de pré-processador. Isso é algo bastante explorado por pessoas com background em linguagens, como C, C++ e Object Pascal, mas é pouco explorado por quem desenvolve em C#.

Obviamente, a compilação condicional não será capaz de resolver todos os nossos problemas. Para os casos em que for necessário ativar ou desativar uma funcionalidade em tempo de execução, bastará recorrer a uma abordagem baseada em *variáveis sinalizadoras*.

Capítulo 15

REFERÊNCIAS

Dicas de produtividade para o Visual Studio

https://docs.microsoft.com/pt-br/visualstudio/ide/productivity-tips-for-visual-studio?view=vs-2019

Dicas de acessibilidade e truques do Visual Studio

https://docs.microsoft.com/pt-br/visualstudio/ide/reference/accessibility-tips-and-tricks?view=vs-2019

Documentação da linguagem C#

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/index

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/index

Novidades do C# (separadas por versão)

https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/

Guia de introdução ao Visual Studio

https://visualstudio.microsoft.com/pt-br/vs/getting-started/

Blog do Visual Studio

https://blogs.msdn.microsoft.com/visualstudio/

Blog do .NET

https://devblogs.microsoft.com/dotnet/

Acesso às versões Preview do Visual Studio

https://visualstudio.microsoft.com/pt-br/vs/preview/

Release Notes do Visual Studio 2019

https://docs.microsoft.com/en-us/visualstudio/releases/2019/release-notes

Portal do mestre José Carlos Macoratti

http://www.macoratti.net

Site do autor Cláudio Ralha

https://www.claudioralha.com