



desenvolvedor

// Passo a Passo

Microsoft Visual C# 2013

Intermediário



John Sharp

O autor



JOHN SHARP é o mais importante tecnólogo da Content Master, uma divisão do CM Group Ltd, no Reino Unido. A empresa é especializada em soluções de treinamento avançadas para grandes multinacionais, frequentemente utilizando as tecnologias mais recentes e inovadoras para obter resultados de aprendizado eficientes. John obteve título de distinção em Computação do Imperial College, Londres. Vem desenvolvendo software e cursos de treinamento em redação, guias e livros há mais de 27 anos. Tem ampla experiência em diversas tecnologias, de sistemas de banco de dados e UNIX a aplicativos em C, C++ e C# para o .NET Framework.

Também já escreveu sobre desenvolvimento em Java e JavaScript, e sobre projeto de soluções empresariais utilizando Windows Azure. Além das sete edições do *Microsoft Visual C# Passo a Passo*, escreveu vários outros livros, *incluindo Microsoft Windows Communication Foundation Step By Step* e *J# Core Reference*. Em seu cargo na Content Master, é autor regular da *Microsoft Patterns & Practices*, tendo trabalhado recentemente em guias, como *Building Hybrid Applications in the Cloud on Windows Azure* e *Data Access for Highly Scalable Solutions Using SQL, NoSQL, and Polyglot Persistence*.



S531m Sharp, John.

Microsoft Visual C# 2013 : passo a passo [recurso eletrônico] / John Sharp ; tradução : João Eduardo Nóbrega Tortello ; revisão técnica: Daniel Antonio Callegari. – Porto Alegre : Bookman, 2014.

Editado também como livro impresso em 2014.
ISBN 978-85-8260-210-2

1. Computação - Desenvolvimento de programas. I. Título.

CDU 004.413Visual C#

Catalogação na publicação: Poliana Sanchez de Araujo – CRB-10/2094



John Sharp

Microsoft Visual C# 2013 // Passo a Passo

Tradução:

João Eduardo Nóbrega Tortello

Revisão técnica:

Daniel Antonio Callegari

Doutor em Ciência da Computação e professor da PUCRS
Profissional certificado Microsoft

Versão impressa
desta obra: 2014



2014

Obra originalmente publicada sob o título
Microsoft® Visual C# 2013 Step by Step, de John Sharp

ISBN 978-0-7356-8183-5

Edição original em inglês copyright © 2013 de CM Group, Ltd.

Tradução para a língua portuguesa copyright © 2014, Bookman Companhia Editora Ltda., uma empresa do Grupo A Educação S.A. Tradução publicada e comercializada com permissão da O'Reilly Media, Inc., que detém ou controla todos os direitos de publicação e comercialização da mesma.

Gerente editorial: *Arysinha Jacques Affonso*

Colaboraram nesta edição:

Editora: *Mariana Belloli*

Capa: *Kaéle*, arte sobre capa original

Leitura final: *Bianca Basile*

Editoração: *Techbooks*

Microsoft e todas as marcas listadas em <http://www.microsoft.com/about/legal/en/us/Intellectual-Property/Trademarks/EN-US.aspx> são marcas comerciais registradas do grupo de empresas da Microsoft. Outras marcas mencionadas aqui são marcas comerciais de seus respectivos proprietários.

Os exemplos de empresas, organizações, produtos, nomes de domínio, endereços de correio eletrônico, logotipo, pessoas, lugares ou eventos aqui apresentados são fictícios. Nenhuma associação com qualquer empresa, organização, produto, nome de domínio, endereço de correio eletrônico, logotipo, pessoa, lugar ou eventos reais foi proposital ou deve ser inferido.

Este livro expressa as visões e opiniões dos autores. As informações aqui contidas são fornecidas sem quaisquer garantias expressas, legais ou implícitas. Os autores, a Microsoft Corporation e seus revendedores ou distribuidores não poderão ser responsabilizados por qualquer dano causado, ou supostamente causado, direta ou indiretamente, por este livro.

Reservados todos os direitos de publicação, em língua portuguesa, à
BOOKMAN EDITORA LTDA., uma empresa do GRUPO A EDUCAÇÃO S.A.
Av. Jerônimo de Ornelas, 670 – Santana
90040-340 – Porto Alegre – RS
Fone: (51) 3027-7000 Fax: (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

Unidade São Paulo
Av. Embaixador Macedo Soares, 10.735 – Pavilhão 5 – Cond. Espace Center
Vila Anastácio – 05095-035 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444 – www.grupoa.com.br

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Sumário

Introdução	xvii
Quem deve ler este livro	xviii
Quem não deve ler este livro	xviii
Organização deste livro.....	ix
Encontre o melhor ponto de partida	ix
Convenções e recursos deste livro.....	xx
Requisitos de sistema.....	xi
Exemplos de código.....	xxii
Instale os exemplos de código.....	xxii
Utilize os exemplos de código.....	xxiii
Agradecimentos	xxviii
Suporte técnico	xxix

PARTE I INTRODUÇÃO AO MICROSOFT VISUAL C# E AO MICROSOFT VISUAL STUDIO 2013

Capítulo 1 Bem-vindo ao C#	3
Comece a programar com o ambiente do Visual Studio 2013	3
Escreva seu primeiro programa	8
Namespaces	14
Crie um aplicativo gráfico.....	18
Examine o aplicativo Windows Store	30
Examine o aplicativo WPF	33
Adicione código ao aplicativo gráfico	34
Resumo	38
Referência rápida	38
Capítulo 2 Variáveis, operadores e expressões	39
Instruções	39
Identificadores	40
Identifique palavras-chave	40

Variáveis	41
Nomeie variáveis	41
Declare variáveis.....	42
Tipos de dados primitivos.....	43
Variáveis locais não atribuídas	43
Exiba valores de tipos de dados primitivos	44
Operadores aritméticos.....	52
Operadores e tipos	52
Examine operadores aritméticos	53
Controle a precedência	59
Utilize a associatividade para avaliar expressões.....	60
A associatividade e o operador de atribuição.....	60
Incremente e decremente variáveis.....	61
Prefixo e sufixo	61
Declare variáveis locais implicitamente tipadas.....	62
Resumo	63
Referência rápida	64
Capítulo 3 Como escrever métodos e aplicar escopo	65
Crie métodos.....	65
Declare um método.....	66
Retorne dados de um método.....	67
Chame métodos.....	69
Aplique escopo.....	72
Defina o escopo local	72
Defina o escopo de classe.....	73
Sobrecarregue métodos	74
Escreva métodos.....	74
Parâmetros opcionais e argumentos nomeados.....	83
Define parâmetros opcionais	85
Passe argumentos nomeados.....	85
Resolva ambiguidades com parâmetros opcionais e argumentos nomeados.....	86
Resumo	91
Referência rápida	92

Capítulo 4	Instruções de decisão	93
	Declare variáveis booleanas	93
	Operadores booleanos	94
	Entenda os operadores de igualdade e relacionais	94
	Entenda os operadores lógicos condicionais.....	95
	Curto-circuito	96
	Um resumo da precedência e da associatividade dos operadores	96
	Instruções <i>if</i> para tomar decisões	97
	Entenda a sintaxe da instrução <i>if</i>	97
	Utilize blocos para agrupar instruções.....	98
	Instruções <i>if</i> em cascata	99
	Instruções <i>switch</i>	105
	Entenda a sintaxe da instrução <i>switch</i>	106
	Siga as regras da instrução <i>switch</i>	107
	Resumo	111
	Referência rápida	111
Capítulo 5	Atribuição composta e instruções de iteração	113
	Operadores de atribuição composta.....	113
	Escreva instruções <i>while</i>	115
	Escreva instruções <i>for</i>	121
	Entendendo o escopo da instrução <i>for</i>	123
	Escreva instruções <i>do</i>	123
	Resumo	132
	Referência rápida	133
Capítulo 6	Gerenciamento de erros e exceções	134
	Lide com erros	134
	Teste o código e capture as exceções	135
	Exceções não tratadas.....	136
	Utilize várias rotinas de tratamento <i>catch</i>	137
	Capture múltiplas exceções	138
	Propague exceções	144
	Aritmética verificada e não verificada de números inteiros.....	146
	Escreva instruções verificadas	147
	Escreva expressões verificadas.....	148

Lance exceções	151
Bloco <i>finally</i>	155
Resumo	156
Referência rápida	157

PARTE II O MODELO DE OBJETOS DO C#

Capítulo 7	Criação e gerenciamento de classes e objetos	161
	Classificação	161
	O objetivo do encapsulamento	162
	Defina e utilize uma classe	162
	Controle a acessibilidade	164
	Trabalhe com construtores	165
	Sobrecharge construtores	167
	Dados e métodos <i>static</i>	175
	Crie um campo compartilhado	176
	Crie um campo <i>static</i> utilizando a palavra-chave <i>const</i>	177
	Entenda as classes <i>static</i>	177
	Classes anônimas	180
	Resumo	181
	Referência rápida	182
Capítulo 8	Valores e referências	183
	Copie variáveis de tipo-valor e classes	183
	Valores nulos e tipos <i>nullable</i>	189
	Utilize tipos <i>nullable</i>	190
	Entenda as propriedades dos tipos <i>nullable</i>	191
	Parâmetros <i>ref</i> e <i>out</i>	192
	Crie parâmetros <i>ref</i>	193
	Crie parâmetros <i>out</i>	193
	Como a memória do computador é organizada	195
	Utilize a pilha e o heap	197
	A classe <i>System.Object</i>	198
	Boxing	199
	Unboxing	199

Casting de dados seguro.....	201
O operador <i>is</i>	201
O operador <i>as</i>	202
Resumo	204
Referência rápida	204
Capítulo 9 Como criar tipos-valor com enumerações e estruturas 206	
Enumerações.....	206
Declare uma enumeração.....	207
Utilize uma enumeração.....	207
Escolha valores literais de enumeração	208
Escolha o tipo subjacente de uma enumeração	209
Estruturas.....	211
Declare uma estrutura.....	213
Entenda as diferenças entre estrutura e classe	214
Declare variáveis de estrutura	215
Entenda a inicialização de estruturas	216
Copie variáveis de estrutura.....	220
Resumo	224
Referência rápida	224
Capítulo 10 Arrays 226	
Declare e crie um array	226
Declare variáveis de array.....	226
Crie uma instância de array	227
Preencha e utilize um array	228
Crie um array implicitamente tipado	229
Acesse um elemento individual de um array.....	230
Itere por um array	230
Passe arrays como parâmetros e valores de retorno para um método	232
Copie arrays.....	233
Arrays multidimensionais	235
Crie arrays irregulares	236
Resumo	247
Referência rápida	247

Capítulo 11	Arrays de parâmetros	249
Sobrecarga – uma recapitulação	249	
Argumentos de arrays	250	
Declare um array <i>params</i>	251	
Utilize <i>params object[]</i>	253	
Utilize um array <i>params</i>	254	
Compare arrays de parâmetros e parâmetros opcionais	257	
Resumo	260	
Referência rápida	260	
Capítulo 12	Herança	261
O que é herança?	261	
Herança	262	
A classe <i>System.Object</i> revisitada	264	
Chame construtores da classe base	264	
Atribua classes	265	
Declare métodos <i>new</i>	267	
Declare métodos virtuais	268	
Declare métodos <i>override</i>	269	
Entenda o acesso <i>protected</i>	272	
Métodos de extensão	278	
Resumo	282	
Referência rápida	282	
Capítulo 13	Como criar interfaces e definir classes abstratas	284
Interfaces	284	
Defina uma interface	285	
Implemente uma interface	286	
Referencie uma classe por meio de sua interface	287	
Trabalhe com várias interfaces	288	
Implemente uma interface explicitamente	289	
Restrições das interfaces	290	
Defina e utilize interfaces	291	
Classes abstratas	301	
Métodos abstratos	303	

Classes seladas	303
Métodos selados	303
Implemente e utilize uma classe abstrata	304
Resumo	310
Referência rápida	311
Capítulo 14 Coleta de lixo e gerenciamento de recursos	313
O tempo de vida de um objeto	313
Escreva destrutores	314
Por que utilizar o coletor de lixo?	316
Como funciona o coletor de lixo?	318
Recomendações	318
Gerenciamento de recursos	319
Métodos de descarte	319
Descarte seguro quanto a exceções	320
A instrução <i>using</i> e a interface <i>IDisposable</i>	320
Chame o método <i>Dispose</i> a partir de um destrutor	322
Implemente o descarte seguro quanto a exceções	324
Resumo	332
Referência rápida	333
PARTE III DEFINIÇÃO DE TIPOS EXTENSÍVEIS EM C#	
Capítulo 15 Implementação de propriedades para acessar campos	337
Implemente encapsulamento com métodos	337
O que são propriedades?	339
Utilize propriedades	341
Propriedades somente-leitura	342
Propriedades somente-gravação	342
Acessibilidade de propriedades	343
Restrições de uma propriedade	344
Declare propriedades de interface	345
Substitua métodos por propriedades	347
Como gerar propriedades automáticas	351
Como inicializar objetos com propriedades	353
Resumo	356
Referência rápida	357

Capítulo 16 Indexadores	359
O que é um indexador?	359
Um exemplo que não utiliza indexadores	360
O mesmo exemplo utilizando indexadores	362
Entenda os métodos de acesso do indexador	364
Compare indexadores e arrays	364
Indexadores em interfaces	366
Indexadores em um aplicativo Windows	367
Resumo	374
Referência rápida	375
Capítulo 17 Genéricos	376
O problema do tipo <i>object</i>	376
A solução dos genéricos	380
Classes genéricas <i>versus</i> generalizadas	382
Genéricos e restrições	382
Crie uma classe genérica	383
A teoria das árvores binárias	383
Construa uma classe de árvore binária com genéricos	386
Crie um método genérico	396
Defina um método genérico para criar uma árvore binária	396
Variância e interfaces genéricas	398
Interfaces covariantes	400
Interfaces contravariantes	402
Resumo	404
Referência rápida	404
Capítulo 18 Coleções	406
O que são classes de coleção?	406
A classe de coleção <i>List<T></i>	408
A classe de coleção <i>LinkedList<T></i>	410
A classe de coleção <i>Queue<T></i>	412
A classe de coleção <i>Stack<T></i>	413
A classe de coleção <i>Dictionary< TKey, TValue ></i>	414
A classe de coleção <i>SortedList< TKey, TValue ></i>	415
A classe de coleção <i>HashSet< T ></i>	417
Inicializadores de coleção	418

Os métodos <i>Find</i> , predicados e expressões lambda	419
Compare arrays e coleções.....	421
Utilize classes de coleção para jogar cartas.....	421
Resumo	426
Referência rápida	426
Capítulo 19 Enumeração sobre coleções	428
Enumere os elementos em uma coleção	428
Implemente manualmente um enumerador.....	430
Implemente a interface <i>IEnumerable</i>	434
Implemente um enumerador utilizando um iterador	437
Um iterador simples.....	437
Defina um enumerador para a classe <i>Tree<TItem></i> por meio de um iterador	439
Resumo	441
Referência rápida	442
Capítulo 20 Separação da lógica do aplicativo e tratamento de eventos	443
Delegates.....	444
Exemplos de delegates na biblioteca de classes do .NET Framework..	445
O cenário da fábrica automatizada.....	447
Implemente o sistema de controle da fábrica sem utilizar delegates	447
Implemente a fábrica utilizando um delegate	448
Declare e utilize delegates	451
Expressões lambda e delegates	460
Crie um método adaptador.....	461
As formas das expressões lambda.....	461
Ative notificações por meio de eventos	463
Declare um evento.....	464
Faça a inscrição em um evento	464
Cancela a inscrição em um evento	465
Dispare um evento.....	465
Eventos de interface de usuário.....	466
Utilize eventos	467
Resumo	474
Referência rápida	474

Capítulo 21 Consulta a dados na memória usando expressões de consulta	477
O que é a Language-Integrated Query?	477
Como utilizar a LINQ em um aplicativo C#	478
Selecione dados	480
Filtre dados	482
Ordene, agrupe e agregue dados	483
Junção de dados	485
Utilize operadores de consulta	487
Consulte dados em objetos <i>Tree<TItem></i>	489
LINQ e avaliação postergada	495
Resumo	499
Referência rápida	500
Capítulo 22 Sobrecarga de operadores	502
Operadores	502
Restrições dos operadores	503
Operadores sobrecarregados	503
Crie operadores simétricos	505
Avaliação da atribuição composta	507
Declare operadores de incremento e decremento	508
Como comparar operadores em estruturas e classes	509
Defina pares de operadores	509
Como implementar operadores	511
Operadores de conversão	517
Forneça conversões predefinidas	518
Implemente operadores de conversão definidos pelo usuário	519
Crie operadores simétricos, uma retomada do assunto	520
Escreva operadores de conversão	520
Resumo	523
Referência rápida	523

PARTE IV CONSTRUÇÃO DE APLICATIVOS WINDOWS 8.1 PROFISSIONAIS COM C#

Capítulo 23	Como melhorar o desempenho usando tarefas	527
	Por que fazer multitarefa por meio de processamento paralelo?	527
	O surgimento do processador multinúcleo.....	528
	Como implementar multitarefa com o Microsoft .NET Framework	530
	Tarefas, threads e o <i>ThreadPool</i>	530
	Crie, execute e controle tarefas	531
	Utilize a classe <i>Task</i> para implementar paralelismo	534
	Abstraia tarefas com a classe <i>Parallel</i>	546
	Quando não utilizar a classe <i>Parallel</i>	550
	Cancele tarefas e trate exceções	552
	Mecânica do cancelamento cooperativo.....	552
	Utilize continuações com tarefas canceladas e com falhas.....	566
	Resumo	567
	Referência rápida	568
Capítulo 24	Como melhorar o tempo de resposta empregando operações assíncronas	570
	Implemente métodos assíncronos.....	571
	Definição de métodos assíncronos: o problema	571
	Definição de métodos assíncronos: a solução	574
	Defina métodos assíncronos que retornam valores	580
	Métodos assíncronos e as APIs do Windows Runtime.....	581
	Utilize a PLINQ para paralelizar o acesso declarativo a dados	584
	Utilize a PLINQ para melhorar o desempenho ao iterar sobre uma coleção.....	585
	Cancele uma consulta PLINQ.....	590
	Sincronize acessos simultâneos a dados.....	591
	Bloqueie dados.....	593
	Primitivas de sincronização para coordenar tarefas.....	594
	Cancele a sincronização	596
	Classes de coleção concorrentes	597
	Utilize uma coleção concorrente e um bloqueio para implementar acesso a dados thread-safe	598
	Resumo	608
	Referência rápida	608

Capítulo 25	Implementação da interface do usuário de um aplicativo Windows Store	611
O que é um aplicativo Windows Store?	612	
Use o template Blank App para compilar um aplicativo Windows Store	616	
Implemente uma interface de usuário escalonável	618	
Aplique estilos em uma interface de usuário	650	
Resumo	659	
Referência rápida	660	
Capítulo 26	Exibição e busca de dados em um aplicativo Windows Store	661
Implemente o padrão Model-View-ViewModel	661	
Contratos do Windows 8.1	689	
Resumo	704	
Referência rápida	707	
Capítulo 27	Acesso a um banco de dados remoto em um aplicativo Windows Store	709
Recupere dados de um banco de dados	709	
Insira, atualize e exclua dados por meio de um web service REST	729	
Resumo	747	
Referência rápida	748	
Índice		750

Introdução

Microsoft Visual C# é uma linguagem poderosa e simples, voltada principalmente para os desenvolvedores que criam aplicativos com o Microsoft .NET Framework. Ela herda grande parte dos melhores recursos do C++ e Microsoft Visual Basic e pouco das inconsistências e anacronismos, resultando em uma linguagem mais limpa e lógica. O C# 1.0 foi lançado em 2001. O advento do C# 2.0 com o Visual Studio 2005 introduziu vários recursos novos importantes na linguagem, como genéricos, iteradores e métodos anônimos. O C# 3.0, lançado com o Visual Studio 2008, acrescentou métodos de extensão, expressões lambda e, o mais famoso de todos os recursos, a Language-Integrated Query (LINQ). O C# 4.0, lançado em 2010, ofereceu aprimoramentos que melhoraram sua interoperabilidade com outras linguagens e tecnologias. Esses recursos abrangem o suporte para argumentos nomeados e opcionais, e o tipo *dynamic* (dinâmico), o qual indica que o tempo de execução da linguagem deve implementar a ligação tardia para um objeto. Inclusões importantes no .NET Framework lançado concomitantemente ao C# 4.0 foram as classes e os tipos que constituem a Task Parallel Library (TPL). Com a TPL é possível construir, de modo rápido e fácil, aplicativos altamente escalonáveis para processadores multinúcleo. O C# 5.0 adiciona suporte nativo para processamento assíncrono baseado em tarefas, por meio do modificador de método `async` e do operador `await`.

Outro evento importante da Microsoft foi o lançamento do Windows 8. Essa nova versão do Windows suporta aplicativos altamente interativos que podem compartilhar dados e colaborar entre si, além de se conectar com serviços em execução na nuvem. O ambiente de desenvolvimento fornecido pelo Microsoft Visual Studio 2012 facilitou o uso de todos esses recursos poderosos, e os muitos assistentes e aprimoramentos novos do Visual Studio 2012 podem aumentar muito sua produtividade como desenvolvedor.

Após escutar as opiniões dos desenvolvedores, a Microsoft modificou alguns aspectos do funcionamento da interface do usuário e lançou uma versão de pré-estreia técnica (Technical Preview) do Windows 8.1 contendo essas alterações. Ao mesmo tempo, a Microsoft lançou uma edição de pré-estreia do Visual Studio 2013, contendo alterações adicionais em relação ao Visual Studio 2012 e adicionando novos recursos que ajudam a aumentar ainda mais a produtividade do programador. Embora muitas das atualizações feitas no Visual Studio sejam pequenas e não tenham havido alterações na linguagem C# nessa versão, achamos que as modificações feitas no modo do Windows 8.1 manipular a interface do usuário mereceriam uma atualização gradual semelhante neste livro. O resultado é esta obra.



Nota Este livro se baseia na Technical Preview do Visual Studio 2013. Consequentemente, alguns recursos do IDE podem mudar na versão final do software.

Quem deve ler este livro

Este livro é destinado a desenvolvedores que desejam aprender os conceitos básicos da programação com o C# utilizando o Visual Studio 2013 e o .NET Framework versão 4.5.1. Ao concluir esta obra, você terá um entendimento completo do C# e o terá utilizado para produzir aplicativos ágeis e escalonáveis que podem ser executados no sistema operacional Windows.

É possível construir e executar aplicativos do C# 5.0 no Windows 7, no Windows 8 e no Windows 8.1, embora as interfaces do usuário fornecidas pelo Windows 7 e pelo Windows 8 tenham algumas diferenças significativas. Além disso, o Windows 8.1 modificou algumas partes do modelo de interface do usuário, e os aplicativos projetados para tirar proveito dessas alterações talvez não funcionem no Windows 8. Consequentemente, as Partes I a III deste livro fornecem exercícios e exemplos trabalhados que funcionam no Windows 7, no Windows 8 e no Windows 8.1. A Parte IV se concentra no modelo de desenvolvimento de aplicativos utilizado pelo Windows 8.1, sendo que o material dessa seção fornece uma introdução para a criação de aplicativos interativos para essa nova plataforma.

Quem não deve ler este livro

Esta obra se destina a desenvolvedores iniciantes em C#, mas não totalmente iniciantes em programação. Assim, ela se concentra principalmente na linguagem C#. O livro não tem como objetivo fornecer uma abordagem detalhada da grande quantidade de tecnologias disponíveis para a criação de aplicativos de nível empresarial para Windows, como ADO.NET, ASP.NET, Windows Communication Foundation ou Workflow Foundation. Caso precise de mais informações sobre qualquer um desses itens, pense na possibilidade de ler alguns dos outros títulos da série Passo a Passo.

Organização deste livro

Esta obra está dividida em quatro partes:

- A Parte I, "Introdução ao Microsoft Visual C# e ao Microsoft Visual Studio 2013", fornece uma introdução à sintaxe básica da linguagem C# e ao ambiente de programação do Visual Studio.
- A Parte II, "O modelo de objetos do C#", entra nos detalhes sobre como criar e gerenciar novos tipos com C# e como gerenciar os recursos referenciados por esses tipos.
- A Parte III, "Definição de tipos extensíveis com C#", contém uma abordagem ampliada dos elementos fornecidos pelo C# para a criação de tipos que podem ser reutilizados em vários aplicativos.
- A Parte IV, "Construção de aplicativos profissionais do Windows 8.1 com C#", descreve o modelo de programação do Windows 8.1 e como é possível utilizar C# para criar aplicativos interativos para esse novo modelo.



Nota Embora a Parte IV seja voltada para o Windows 8.1, muitos conceitos descritos nos Capítulos 23 e 24 também são adequados para aplicativos do Windows 8 e do Windows 7.

Encontre o melhor ponto de partida

Este livro foi projetado para ajudá-lo a desenvolver habilidades em várias áreas essenciais. Você pode utilizá-lo se for iniciante em programação ou se estiver migrando de outra linguagem, como C, C++, Java ou Visual Basic. Consulte a tabela a seguir para encontrar seu melhor ponto de partida.

Se você está	Siga estes passos
Começando em programação orientada a objetos	<ol style="list-style-type: none">1. Instale os arquivos de prática conforme descrito na próxima seção, "Exemplos de código".2. Siga os capítulos nas Partes I, II e III sequencialmente.3. Complete a Parte IV conforme seu nível de experiência e interesse.
Familiarizado com linguagens de programação procedural, como C, mas for iniciante em C#	<ol style="list-style-type: none">1. Instale os arquivos de prática conforme descrito na próxima seção, "Exemplos de código". Folheie os cinco primeiros capítulos para obter uma visão geral do C# e do Visual Studio 2013 e, em seguida, concentre-se nos Capítulos 6 a 22.2. Complete a Parte IV conforme seu nível de experiência e interesse.

Se você está	Siga estes passos
Migrando de uma linguagem orientada a objetos como C++ ou Java	<ol style="list-style-type: none"> Instale os arquivos de prática conforme descrito na próxima seção, "Exemplos de código". Folheie os sete primeiros capítulos para obter uma visão geral do C# e do Visual Studio 2013 e, em seguida, concentre-se nos Capítulos 7 a 22. Leia a Parte IV para obter informações sobre como criar aplicativos Windows 8.1 escalonáveis.
Trocando do Visual Basic para o C#	<ol style="list-style-type: none"> Instale os arquivos de prática conforme descrito na próxima seção, "Exemplos de código". Siga os capítulos nas Partes I, II e III sequencialmente. Leia a Parte IV para obter informações sobre como criar aplicativos Windows 8.1. Leia as seções de Referência rápida no final dos capítulos para obter informações sobre questões específicas do C# e construções do Visual Studio 2013.
Consultando o livro após fazer os exercícios	<ol style="list-style-type: none"> Utilize o índice ou o sumário para localizar as informações sobre assuntos específicos. Leia as seções de Referência rápida no final de cada capítulo para encontrar uma revisão sucinta da sintaxe e das técnicas apresentadas no capítulo.

A maioria dos capítulos do livro contém exemplos práticos que permitem a você experimentar os conceitos que acabou de aprender. Independentemente das seções em que queira se concentrar, certifique-se de baixar e instalar os exemplos de aplicativos em seu sistema.

Convenções e recursos deste livro

Este livro usa algumas convenções para tornar as informações legíveis e fáceis de compreender.

- Cada exercício consiste em uma série de tarefas, apresentadas como etapas numeradas (1, 2 e assim por diante) listando cada ação a ser executada para completar o exercício.

- Os elementos em quadros marcados como "Nota" fornecem informações adicionais ou métodos alternativos para completar uma etapa com sucesso.
- Textos que você deve digitar (com exceção dos blocos de código) aparecem em negrito.
- Um sinal de adição (+) entre dois nomes de tecla significa que você deve pressionar essas teclas ao mesmo tempo. Por exemplo, "Pressione Alt+Tab" quer dizer que a tecla Alt deve ser pressionada ao mesmo tempo que a tecla Tab.
- Uma barra vertical entre dois ou mais itens de menu (por exemplo, Arquivo | Fechar) significa que você deve selecionar o primeiro menu ou item de menu e, então, o seguinte e assim por diante.

Requisitos de sistema

Você precisará dos seguintes hardware e software para completar os exercícios deste livro:

- Windows 7 (x86 e x64), Windows 8 (x86 e x64), Windows 8.1 (x86 e x64), Windows Server 2008 R2 SP1 (x64), Windows Server 2012 (x64) ou Windows Server 2012 R2 (x64).



Importante Os templates Windows Store para Visual Studio 2013 não estão disponíveis no Windows 8, Windows 7, Windows Server 2012 ou Windows Server 2008 R2. Se quiser usar esses templates ou fazer os exercícios que produzem aplicativos Windows Store, você deve usar Windows 8.1 ou Windows Server 2012 R2.

- Visual Studio 2013 (qualquer edição, exceto Visual Studio Express para Windows 8.1).



Importante É possível usar Visual Studio Express 2013 para Windows Desktop, mas com esse software você só poderá executar a versão para Windows 7 dos exercícios do livro. Não é possível utilizar esse software para fazer os exercícios da Parte IV desta obra.

- Computador com um processador de 1,6 GHz ou mais rápido (2 GHz recomendado).
- 1 GB (32 bits) ou 2 GB (64 bits) de memória RAM (acrescente 512 MB se estiver executando em uma máquina virtual).
- 10 GB de espaço disponível no disco rígido.
- Unidade de disco rígido de 5400 RPM.

- Placa de vídeo com capacidade para DirectX 9, executando em resolução de tela de 1024 × 768 ou mais; se estiver usando Windows 8.1, recomenda-se uma resolução de 1366 × 768 ou mais.
- Unidade de DVD-ROM (se estiver instalando o Visual Studio a partir de um DVD).
- Conexão com a Internet para baixar software ou os exemplos dos capítulos.

Dependendo de sua configuração de Windows, talvez sejam necessários direitos de Administrador Local para instalar ou configurar o Visual Studio 2013.

Exemplos de código

A maioria dos capítulos do livro contém exercícios com os quais é possível testar interativamente a nova matéria aprendida no livro. Todos os exemplos de projeto, tanto em seus formatos anteriores como posteriores aos exercícios, podem ser baixados em:

www.grupoa.com.br

Cadastre-se gratuitamente no site, encontre a página do livro por meio do campo de busca, acesse a página do livro e clique no link Conteúdo Online para fazer download dos arquivos.



Nota Além dos exemplos de código, seu sistema deve ter o Visual Studio 2013 instalado. Se estiver disponível, instale os pacotes de serviço mais recentes para Windows e Visual Studio.

Instale os exemplos de código

Siga estes passos para instalar os exemplos de código no computador a fim de usá-los com os exercícios do livro.

1. Faça download do arquivo 9780735681835.zip a partir da página do livro no site www.grupoa.com.br
2. Descompacte na sua pasta Documentos (ou em um diretório específico, se preferir) o arquivo 9780735681835.zip que você baixou.

Utilize os exemplos de código

Todos os capítulos explicam quando e como usar os exemplos de código. Quando for o momento de usar um exemplo de código, o livro listará as instruções sobre como abrir os arquivos.

Para quem gosta de conhecer todos os detalhes, segue uma lista dos projetos e das soluções do Visual Studio 2013 contendo exemplos de código, agrupada pelas pastas em que você pode localizá-los. Em diversos casos, os exercícios fornecem arquivos provisórios e versões completas dos mesmos projetos, que você pode utilizar como referência. Os exemplos de código fornecem versões para Windows 7 e para Windows 8.1, e as instruções dos exercícios salientam quaisquer diferenças nas tarefas a serem executadas ou no código que você precisa escrever para esses dois sistemas operacionais. Os projetos concluídos de cada capítulo são armazenados em pastas com o sufixo “- Complete”.



Importante Se você estiver utilizando Windows 8, Windows Server 2012 ou Windows Server 2008 R2, siga as instruções para Windows 7. Se estiver utilizando o Windows Server 2012 R2, siga as instruções para Windows 8.1.

Projeto	Descrição
Capítulo 1	
TextHello	Esse projeto o inicia nas atividades. Guia você passo a passo ao longo do processo de criação de um programa simples que exibe uma saudação baseada em texto.
WPFHello	Exibe a saudação em uma janela, utilizando o Windows Presentation Foundation (WPF).
Capítulo 2	
PrimitiveDataTypes	Demonstra como declarar variáveis de cada um dos tipos primitivos, como atribuir valores a essas variáveis e como exibi-los em uma janela.
MathsOperators	Apresenta os operadores aritméticos (+ – * / %).
Capítulo 3	
Methods	Reexamina o código do projeto anterior e investiga como são empregados os métodos para estruturar o código.
DailyRate	Ensina a escrever e executar seus próprios métodos e a inspecionar passo a passo as chamadas de método utilizando o depurador do Visual Studio 2013.
DailyRate Using Optional Parameters	Mostra como definir um método que aceita parâmetros opcionais e como chamá-lo por meio de argumentos nomeados.

Projeto	Descrição
Capítulo 4	
Selection	Mostra como utilizar uma instrução <i>if</i> em cascata para implementar uma lógica complexa, como comparar a equivalência de duas datas.
SwitchStatement	Utiliza uma instrução <i>switch</i> para converter caracteres em suas representações XML.
Capítulo 5	
WhileStatement	Demonstra uma instrução <i>while</i> que lê o conteúdo de cada linha de um arquivo-fonte e o exibe em uma caixa de texto em um formulário.
DoStatement	Esse projeto utiliza uma instrução <i>do</i> para converter um número decimal em sua representação octal.
Capítulo 6	
MathsOperators	Revisa o projeto MathsOperators do Capítulo 2 e mostra como várias exceções não tratadas podem fazer o programa falhar. As palavras-chave <i>try</i> e <i>catch</i> tornam o aplicativo mais robusto, evitando que ocorram mais falhas.
Capítulo 7	
Classes	Aborda os fundamentos da definição de suas próprias classes, incluindo construtores públicos, métodos e campos privados. Além disso, mostra como criar instâncias de classe utilizando a palavra-chave <i>new</i> e como definir métodos e campos estáticos.
Capítulo 8	
Parameters	Investiga a diferença entre os parâmetros por valor e os parâmetros por referência. Demonstra como utilizar as palavras-chave <i>ref</i> e <i>out</i> .
Capítulo 9	
StructsAndEnums	Define um tipo <i>struct</i> para representar uma data de calendário.
Capítulo 10	
Cards	Mostra como utilizar arrays para modelar mãos de cartas em um jogo de baralho.
Capítulo 11	
ParamsArrays	Demonstra como utilizar a palavra-chave <i>params</i> para criar um único método que aceite todos os argumentos <i>int</i> .

Projeto	Descrição
Capítulo 12	
Vehicles	Cria uma hierarquia simples de classes de veículos utilizando herança. Também demonstra como definir um método virtual.
ExtensionMethod	Mostra como produzir um método de extensão para o tipo <i>int</i> , fornecendo um método que converte um valor inteiro de base 10 em uma base numérica diferente.
Capítulo 13	
Drawing Using Interfaces	Implementa parte de um pacote de desenho gráfico. O projeto utiliza interfaces para definir os métodos que as formas de desenho expõem e implementam.
Drawing Using Abstract Classes	Estende o projeto Drawing Using Interfaces para fatorar a funcionalidade comum de objetos de forma em classes.
Capítulo 14	
GarbageCollectionDemo	Mostra como implementar o descarte de recursos seguro para exceções, usando o padrão <i>Dispose</i> .
Capítulo 15	
Drawing Using Properties	Estende o aplicativo do projeto Drawing Using Abstract Classes, desenvolvido no Capítulo 13, para encapsular dados em uma classe usando propriedades.
AutomaticProperties	Mostra como criar propriedades automáticas para uma classe e utilizá-las para inicializar instâncias da classe.
Capítulo 16	
Indexers	Utiliza dois indexadores: um procura o número de telefone de uma pessoa quando um nome é fornecido e o outro procura o nome de uma pessoa quando um número de telefone é fornecido.
Capítulo 17	
BinaryTree	Mostra como empregar genéricos para criar uma estrutura <i>typesafe</i> que possa conter elementos de qualquer tipo.
BuildTree	Demonstra como utilizar genéricos para implementar um método <i>typesafe</i> que possa receber parâmetros de qualquer tipo.

Projeto	Descrição
Capítulo 18	
Cards	Atualiza o código do Capítulo 10 para mostrar como usar coleções para modelar mãos de cartas em um jogo de baralho.
Capítulo 19	
BinaryTree	Mostra como implementar a interface genérica <i>IEnumerator<T></i> para criar um enumerador para a classe genérica <i>Tree</i> .
IteratorBinaryTree	Utiliza um iterador para gerar um enumerador para a classe genérica <i>Tree</i> .
Capítulo 20	
Delegates	Mostra como desacoplar um método da lógica do aplicativo que o chama, usando um delegado.
Delegates With Event	Mostra como usar um evento para alertar um objeto sobre uma ocorrência significativa e como capturar um evento e realizar o processamento necessário.
Capítulo 21	
QueryBinaryTree	Mostra como utilizar consultas LINQ para recuperar dados de um objeto de árvore binária.
Capítulo 22	
ComplexNumbers	Define um novo tipo que modela números complexos e implementa operadores comuns para esse tipo.
Capítulo 23	
GraphDemo	Gera e exibe um gráfico complexo em um formulário WPF. Utiliza um único thread para efetuar os cálculos.
GraphDemo With Tasks	Versão do projeto GraphDemo que cria várias tarefas para efetuar os cálculos do gráfico simultaneamente.
Parallel GraphDemo	Versão do projeto GraphDemo que usa a classe <i>Parallel</i> para abstrair o processo de criação e gerenciamento de tarefas.
GraphDemo With Cancellation	Demonstra como implementar o cancelamento para interromper tarefas de modo controlado, antes de sua conclusão.
ParallelLoop	Fornece um exemplo de quando você não deve utilizar a classe <i>Parallel</i> para criar e executar tarefas.

Projeto	Descrição
Capítulo 24	
GraphDemo	Versão do projeto GraphDemo do Capítulo 23 que usa a palavra-chave <i>async</i> e o operador <i>await</i> para efetuar os cálculos que geram os dados do gráfico de forma assíncrona.
PLINQ	Apresenta alguns exemplos de como utilizar PLINQ para consultar dados por meio de tarefas paralelas.
CalculatePI	Utiliza um algoritmo de amostragem estatística para calcular uma aproximação de pi. Usa tarefas paralelas.
Capítulo 25	
Customers Without Scalable UI	Utiliza o controle Grid padrão para organizar a interface do usuário do aplicativo Adventure Works Customers. A interface utiliza posicionamento absoluto para os controles e não muda de escala para diferentes resoluções de tela e tamanhos físicos.
Customers With Scalable UI	Utiliza controles Grid aninhados com definições de linha e coluna para permitir seu posicionamento relativo. Essa versão da interface do usuário muda de escala para diferentes resoluções de tela e tamanhos físicos, mas não se adapta bem ao modo de exibição Snapped.
Customers With Adaptive UI	Estende a versão com a interface do usuário escalonável. Utiliza o Visual State Manager para detectar se o aplicativo está sendo executado no modo de exibição Snapped e muda o layout dos controles de forma correspondente.
Customers With Styles	Versão do projeto Customers que utiliza estilos XAML para mudar a fonte e a imagem de fundo exibidas pelo aplicativo.
Capítulo 26	
DataBinding	Utiliza vinculação de dados para exibir na interface do usuário informações de clientes recuperadas de uma fonte de dados. Mostra também como implementar a interface INotifyPropertyChanged para que a interface do usuário possa atualizar as informações dos clientes e enviar essas alterações de volta para a fonte de dados.
ViewModel	Versão do projeto Customers que separa a interface do usuário da lógica que acessa a fonte de dados, implementando o padrão Model-View-ViewModel.

Projeto	Descrição
Search	Implementa o contrato Windows 8.1 Search. O usuário pode procurar clientes pelo nome ou pelo sobrenome.
Capítulo 27	
Web Service	Contém um aplicativo web que fornece um web service ASP.NET Web API, utilizado pelo aplicativo Customers para recuperar dados de clientes de um banco de dados SQL Server. O web service utiliza um modelo de entidade criado com o Entity Framework para acessar o banco de dados.
Updatable ViewModel	Nesta solução, o projeto Customers contém um ViewModel estendido com comandos que permitem à interface do usuário inserir e atualizar informações de clientes usando o WCF Data Service.

Agradecimentos

Apesar de meu nome estar na capa, escrever um livro como este está longe de ser um projeto solitário. Gostaria de agradecer às seguintes pessoas que apoiaram e ajudaram generosamente em todo este exercício um tanto prolongado.

Primeiramente, Russell Jones, que foi o primeiro a me alertar sobre o iminente lançamento das versões de pré-estreia do Windows 8.1 e do Visual Studio 2013. Ele conseguiu acelerar todo o processo de envio desta edição do livro para a gráfica. Sem seus esforços talvez você lesse este livro apenas quando a próxima edição de Windows surgisse.

A seguir, Mike Sumsion e Paul Barnes, meus estimados colegas da Content Master, que realizaram um excelente trabalho de revisão do material das versões originais de cada capítulo, testando meu código e apontando os numerosos erros que eu havia cometido! Acho que agora identifiquei todos eles, mas, é claro, quaisquer erros que restem são de minha inteira responsabilidade.

Além disso, John Mueller, que fez um trabalho notável e muito ágil de revisão técnica desta edição. Sua experiência em escrita e conhecimento das tecnologias aqui abordadas foram extremamente úteis, enriquecendo esta obra.

Evidentemente, assim como muitos programadores, posso entender a tecnologia, mas meu texto nem sempre é tão fluente ou claro como poderia ser. Gostaria de agradecer aos editores por corrigirem minha gramática, meus erros ortográficos e, de modo geral, por tornarem meu material muito mais fácil de entender.

Por fim, gostaria de agradecer à minha esposa e companheira de críquete, Diana, por não franzir muito as sobrancelhas quando eu disse que começaria a trabalhar em uma edição atualizada deste livro. Agora ela já se acostumou com meus murmurios raivosos enquanto depuro código e com os numerosos “oh” que emito ao perceber os erros crassos que cometi.

Suporte técnico

Todos os esforços foram feitos para garantir a exatidão deste livro e do conteúdo complementar que o acompanha. Caso queira fazer comentários ou sugestões, tirar dúvidas ou reportar erros, escreva para secretariaeditorial@grupoa.com.br.

Esta página foi deixada em branco intencionalmente.

PARTE I

Introdução ao Microsoft Visual C# e ao Microsoft Visual Studio 2013

CAPÍTULO 1	Bem-vindo ao C#	3
CAPÍTULO 2	Variáveis, operadores e expressões	39
CAPÍTULO 3	Como escrever métodos e aplicar escopo	65
CAPÍTULO 4	Instruções de decisão	93
CAPÍTULO 5	Atribuição composta e instruções de iteração	113
CAPÍTULO 6	Gerenciamento de erros e exceções	134

Esta página foi deixada em branco intencionalmente.

CAPÍTULO 1

Bem-vindo ao C#

Neste capítulo, você vai aprender a:

- Utilizar o ambiente de programação do Microsoft Visual Studio 2013.
- Criar um aplicativo de console em C#.
- Explicar o objetivo dos namespaces.
- Criar um aplicativo gráfico simples em C#.

Este capítulo apresenta o Visual Studio 2013, o ambiente de programação, e o conjunto de ferramentas projetadas para criar aplicativos para o Microsoft Windows. O Visual Studio 2013 é a ferramenta ideal para escrever código em C#, oferecendo muitos recursos que você vai conhecer à medida que avançar neste livro. Neste capítulo, você vai usar o Visual Studio 2013 para criar alguns aplicativos simples em C# e começar a construir soluções altamente funcionais para Windows.

Comece a programar com o ambiente do Visual Studio 2013

O Visual Studio 2013 é um ambiente de programação rico em recursos que contém a funcionalidade necessária para criar projetos grandes ou pequenos em C# que funcionam no Windows 7, no Windows 8 e no Windows 8.1. Você pode inclusive construir projetos que combinem módulos de diferentes linguagens, como C++, Visual Basic e F#. No primeiro exercício, você vai abrir o ambiente de programação do Visual Studio 2013 e aprender a criar um aplicativo de console.



Nota Um aplicativo de console é executado em uma janela de prompt de comando, em vez de fornecer uma interface gráfica com o usuário (GUI).

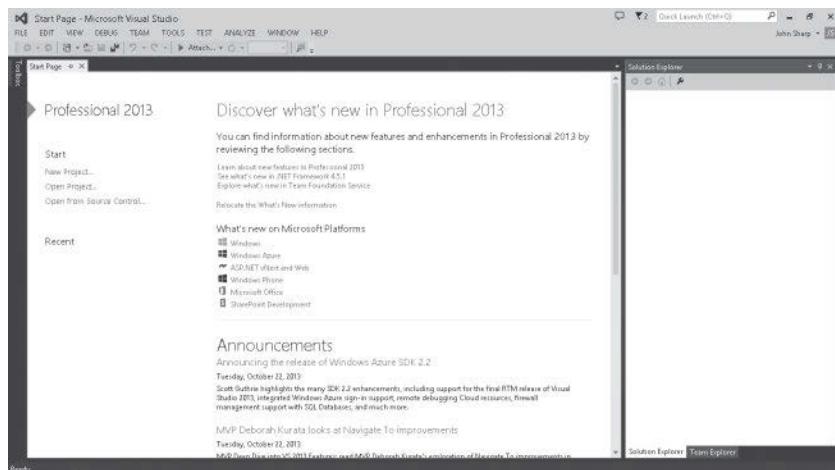
Crie um aplicativo de console no Visual Studio 2013

- Se você estiver utilizando Windows 8.1 ou Windows 8, na tela Iniciar, digite **Visual Studio** e, no painel Resultados da Pesquisa, clique em Visual Studio 2013.



Nota No Windows 8 e no Windows 8.1, para encontrar um aplicativo, você pode digitar o nome dele literalmente (como Visual Studio) em qualquer parte em branco da tela Iniciar, longe de quaisquer tiles. O painel Resultados da Pesquisa aparecerá automaticamente.

O Visual Studio 2013 é iniciado e apresenta a Página Iniciar (Start page), semelhante à seguinte (sua Página Iniciar poderá ser diferente, dependendo da edição de Visual Studio 2013 que estiver usando).



Nota Se você estiver usando o Visual Studio 2013 pela primeira vez, talvez apareça uma caixa de diálogo solicitando a escolha das configurações de ambiente de desenvolvimento padrão. O Visual Studio 2013 pode ser personalizado de acordo com a sua linguagem de desenvolvimento preferida. As seleções padrão das diversas caixas de diálogo e ferramentas do ambiente de desenvolvimento integrado (Integrated Development Environment – IDE) são definidas para a linguagem que você escolher. Na lista, selecione Visual C# Development Settings e clique no botão Start Visual Studio. Após alguns instantes, o IDE do Visual Studio 2013 aparecerá.

- Se estiver usando o Windows 7, execute as seguintes operações para iniciar o Visual Studio 2013:
 - a. Na barra de tarefas do Windows, clique no botão Iniciar, clique em Todos os Programas e, em seguida, clique no grupo de programas Microsoft Visual Studio 2013.
 - b. No grupo de programas Microsoft Visual Studio 2013, clique em Visual Studio 2013.

O Visual Studio 2013 é iniciado e apresenta a Página Iniciar.



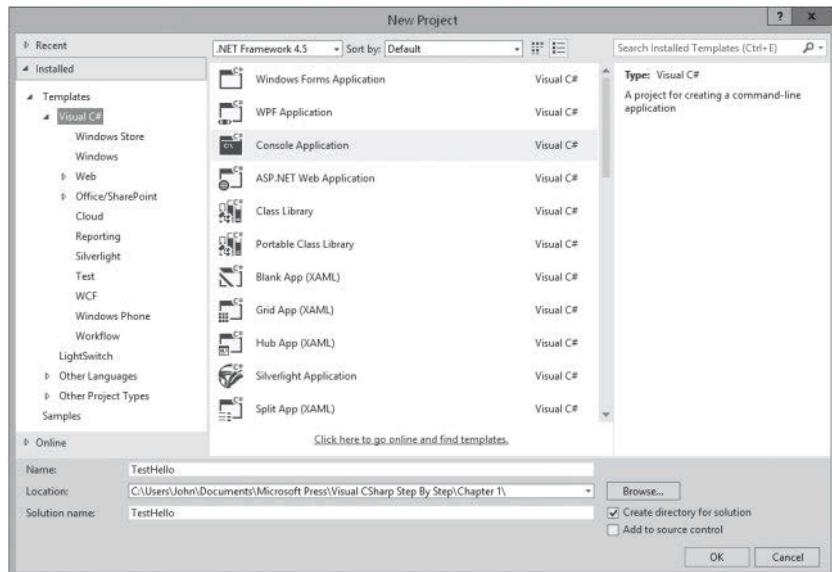
Nota Para não ser repetitivo e economizar espaço, escreverei apenas “Inicie o Visual Studio” quando você precisar abrir o Visual Studio 2013, independentemente do sistema operacional que esteja usando.

■ Siga estes passos para criar um novo aplicativo de console.

- No menu File, aponte para New e então clique em Project.

A caixa de diálogo New Project se abre. Ela lista os templates que você pode utilizar como ponto de partida para construir um aplicativo. A caixa de diálogo categoriza os templates de acordo com a linguagem de programação que você está utilizando e o tipo de aplicativo.

- No painel à esquerda, na seção Templates, clique em Visual C#. No painel central, verifique se a caixa de combinação posicionada no início do painel exibe o texto .NET Framework 4.5 e depois clique no ícone Console Application.



- Na caixa Location, digite **C:\Users\SeuNome\Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. Substitua o texto *SeuNome* nesse caminho pelo seu nome de usuário do Windows.



Nota Para não ser repetitivo e economizar espaço, no restante deste livro vou me referir ao caminho C:\Users\SeuNome\Documentos simplesmente como sua pasta Documentos.

Dica Se a pasta especificada não existir, o Visual Studio 2013 criará uma nova para você.

- d. Na caixa Name, digite **TestHello** (digite sobre o nome existente, ConsoleApplication1).
- e. Certifique-se de que a caixa de seleção Create Directory For Solution está selecionada e clique em OK.

O Visual Studio cria o projeto utilizando o template Console Application e exibe o código básico para o projeto, como na ilustração:

```

TestHello - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
... Start Debug ... Task List
Program.cs > X
Main(string[] args)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

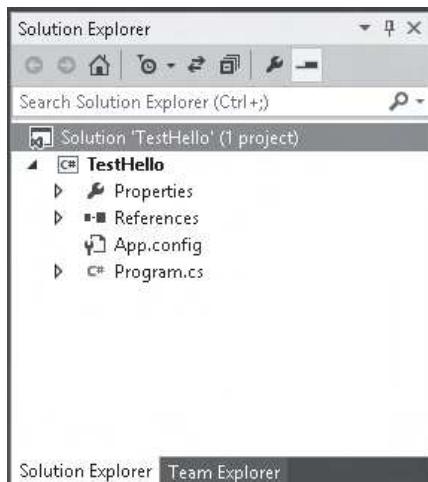
namespace TestHello
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}

```

A barra de menus na parte superior da tela fornece acesso aos recursos que você utilizará no ambiente de programação. Você pode usar o teclado ou o mouse para acessar os menus e os comandos, exatamente como faz em todos os programas baseados em Windows. A barra de ferramentas está localizada abaixo da barra de menus. Ela oferece botões de atalho para executar os comandos utilizados com mais frequência.

A janela Code and Text Editor, que ocupa a parte principal da tela, exibe o conteúdo dos arquivos-fonte. Em um projeto com vários arquivos, quando você edita mais de um deles, cada arquivo-fonte tem uma guia própria com seu nome. Você pode clicar na guia para trazer o arquivo-fonte nomeado para o primeiro plano na janela Code and Text Editor.

O painel Solution Explorer aparece no lado direito da caixa de diálogo:



O Solution Explorer exibe os nomes dos arquivos associados ao projeto, entre outros itens. Você pode clicar duas vezes em um nome de arquivo no painel Solution Explorer para trazer esse arquivo-fonte para o primeiro plano na janela do Code and Text Editor.

Antes de escrever o código, examine os arquivos listados no Solution Explorer, criados pelo Visual Studio 2013 como parte do seu projeto:

- **Solution 'TestHello'** É o arquivo de solução de nível superior. Cada aplicativo contém apenas um arquivo de solução. Uma solução pode conter um ou mais projetos; o Visual Studio 2013 cria o arquivo de solução para ajudar a organizar esses projetos. Se utilizar o Windows Explorer para examinar a pasta Documentos\Microsoft Press\Visual CSharp Step By Step\Chapter 1\TestHello, você verá que o nome real desse arquivo é TestHello.sln.
- **TestHello** É o arquivo de projeto do C#. Cada arquivo de projeto faz referência a um ou mais arquivos que contêm o código-fonte e outros artefatos do projeto, como imagens gráficas. Todos os códigos-fonte de um mesmo projeto devem ser escritos na mesma linguagem de programação. No Windows Explorer, esse arquivo se chama TestHello.csproj e está armazenado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 1\TestHello\TestHello em sua pasta Documentos.
- **Properties** É uma pasta do projeto TestHello. Se for expandida (clique na seta ao lado de Properties), você verá que ela contém um arquivo chamado AssemblyInfo.cs. Esse é um arquivo especial que você pode utilizar para adicionar atributos a um programa, como o nome do autor, a data em que o programa foi escrito, etc. Você pode especificar atributos adicionais para modificar a maneira como o programa é executado. Explicar como esses atributos são utilizados está além dos objetivos deste livro.
- **References** Essa pasta contém as referências às bibliotecas de código compilado que seu aplicativo pode utilizar. Quando o código C# é compilado, ele é convertido em uma biblioteca e recebe um nome exclusivo. No Microsoft .NET Framework, essas bibliotecas são chamadas *assemblies*. Desenvolvedores utilizam assemblies para empacotar funcionalidade útil que escreveram, podendo distri-

bui-los para outros desenvolvedores que queram utilizar esses recursos nos seus aplicativos. Se você expandir a pasta References, verá o conjunto de referências padrão adicionado em seu projeto pelo Visual Studio 2013. Esses assemblies dão acesso a muitos dos recursos normalmente utilizados do .NET Framework e são fornecidos pela Microsoft com o Visual Studio 2013. Você vai aprender sobre muitos desses assemblies à medida que avançar nos exercícios do livro.

- **App.config** É o arquivo de configuração de aplicativo. Ele é opcional e poderá não estar presente todas as vezes. É possível especificar, durante a execução, as configurações que seu aplicativo pode utilizar para modificar seu comportamento, como a versão do .NET Framework a ser utilizada para executar o aplicativo. Você vai aprender mais sobre esse arquivo nos capítulos posteriores deste livro.
- **Program.cs** É um arquivo-fonte do C# exibido na janela Code and Text Editor quando o projeto é criado. Você escreverá seu código para o aplicativo de console nesse arquivo. Ele contém um código que o Visual Studio 2013 fornece automaticamente, o qual será examinado a seguir.

Escreva seu primeiro programa

O arquivo Program.cs define uma classe chamada *Program* que contém um método chamado *Main*. Em C#, todo código executável deve ser definido dentro de um método e todos os métodos devem pertencer a uma classe ou a uma estrutura. Você aprenderá mais sobre classes no Capítulo 7, “Criação e gerenciamento de classes e objetos”, e sobre estruturas, no Capítulo 9, “Como criar tipos-valor com enumerações e estruturas”.

O método *Main* designa o ponto de entrada do programa. Ele deve ser definido como um método estático, da maneira especificada na classe *Program*; caso contrário, o .NET Framework poderá não reconhecê-lo como ponto de partida de seu aplicativo, quando for executado. (Veremos métodos em detalhes no Capítulo 3, “Como escrever métodos e aplicar escopo”, e o Capítulo 7 fornece mais informações sobre os métodos estáticos.)



Importante O C# é uma linguagem que diferencia maiúsculas de minúsculas: Você deve escrever *Main* com *M* maiúsculo.

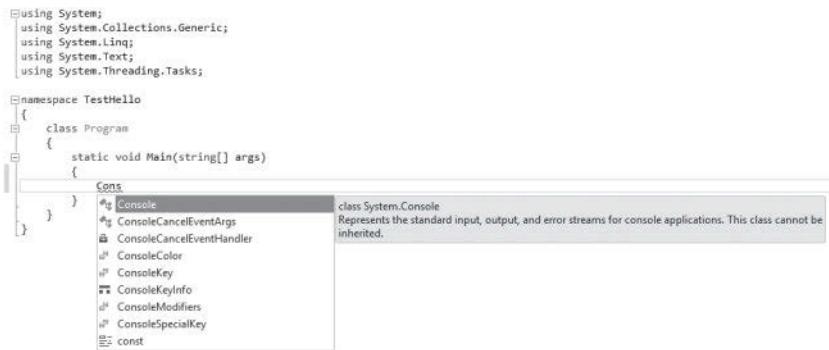
Nos exercícios a seguir, você vai escrever um código para exibir a mensagem “Hello World!” na janela do console, vai compilar e executar seu aplicativo de console Hello World e vai aprender como os namespaces são utilizados para dividir elementos do código.

Escreva o código utilizando o Microsoft IntelliSense

1. Na janela Code and Text Editor que exibe o arquivo Program.cs, coloque o cursor no método *Main* logo após a chave de abertura, {, e pressione Enter para criar uma nova linha.

2. Nessa nova linha, digite a palavra **Console**; esse é o nome de outra classe fornecida pelos assemblies referenciados por seu aplicativo. Ela fornece os métodos para exibir mensagens na janela do console e ler entradas a partir do teclado.

Ao digitar a letra **C** no início da palavra *Console*, uma lista IntelliSense aparecerá.



Essa lista contém todas as palavras-chave válidas do C# e os tipos de dados válidos nesse contexto. Você pode continuar digitando ou rolar pela lista e clicar duas vezes no item **Console** com o mouse. Como alternativa, depois de digitar **Cons**, a lista IntelliSense focalizará automaticamente o item *Console* e você poderá pressionar as teclas Tab ou Enter para selecioná-lo.

Main deve se parecer com isto:

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



Nota *Console* é uma classe interna.

3. Digite um ponto logo após *Console*.

Outra lista IntelliSense aparece, exibindo os métodos, propriedades e campos da classe *Console*.

4. Role para baixo pela lista, selecione *WriteLine* e então pressione Enter. Você também pode continuar a digitar os caracteres **W, r, i, t, e, L**, até *WriteLine* estar selecionado e então pressionar Enter.

A lista IntelliSense é fechada e a palavra *WriteLine* é adicionada ao arquivo-fonte. *Main* deve se parecer com isto:

```
static void Main(string[] args)
{
    Console.WriteLine
}
```

5. Digite um parêntese de abertura, (. Outra dica do IntelliSense aparece.

Essa dica exibe os parâmetros que o método *WriteLine* pode receber. De fato, *WriteLine* é um *método sobrecarregado*, ou seja, a classe *Console* contém mais de um método chamado *WriteLine* – na verdade ela fornece 19 versões diferentes desse método. Cada versão do método *WriteLine* pode ser utilizada para emitir diferentes tipos de dados. (O Capítulo 3 descreve métodos sobrecarregados em mais detalhes.) *Main* deve se parecer com isto:

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```

 **Dica** Você pode clicar nas setas para cima e para baixo na dica para rolar pelas diferentes sobrecargas de *WriteLine*.

6. Digite um parêntese de fechamento,), seguido por um ponto e vírgula, ;.

Main deve se parecer com isto:

```
static void Main(string[] args)
{
    Console.WriteLine();
}
```

7. Mova o cursor e digite a string “**Hello World!**”, incluindo as aspas, entre os parênteses esquerdo e direito depois do método *WriteLine*.

Main deve se parecer com isto:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
}
```

 **Dica** Adquira o hábito de digitar pares de caracteres correspondentes, como parênteses (e) e chaves { e }, antes de preencher seus conteúdos. É fácil esquecer o caractere de fechamento se você esperar para digitá-lo depois de inserir o conteúdo.

Ícones IntelliSense

Quando você digita um ponto depois do nome de uma classe, o IntelliSense exibe o nome de cada membro dessa classe. À esquerda de cada nome de membro há um ícone que representa o tipo de membro. Os ícones mais comuns e seus tipos são:

Ícone	Significado
	Método (Capítulo 3)
	Propriedade (Capítulo 15, "Implementação de propriedades para acessar campos")
	Classe (Capítulo 7)
	Estrutura (Capítulo 9)
	Enumeração (Capítulo 9)
	Método de extensão (Capítulo 12)
	Interface (Capítulo 13, "Como criar interfaces e definir classes abstratas")
	Delegado (Capítulo 17, "Genéricos")
	Evento (Capítulo 17)
	Namespace (próxima seção deste capítulo)

Outros ícones IntelliSense aparecerão à medida que você digitar o código em contextos diferentes.

Muitas vezes, você verá linhas de código contendo duas barras (//) seguidas por um texto comum. Esses são comentários ignorados pelo compilador, mas muito úteis para os desenvolvedores, porque ajudam a documentar o que um programa está fazendo. Considere o seguinte exemplo:

```
Console.ReadLine(); // Espera o usuário pressionar a tecla Enter
```

O compilador pula todo o texto desde as duas barras até o fim da linha. Você também pode adicionar comentários de várias linhas, que iniciam com uma barra normal seguida por um asterisco (*). O compilador pula tudo até localizar um asterisco seguido por barra normal (*), que pode estar várias linhas abaixo. É um estímulo para documentar seu código com o maior número possível de comentários significativos.

Compile e execute o aplicativo de console

1. No menu Build, clique em Build Solution.

Essa ação compila o código C#, resultando em um programa que pode ser executado. A janela Output aparece abaixo da janela Code and Text Editor.



Dica Se a janela Output não aparecer no menu View, clique em Output para exibi-la.

Nessa janela, você deve ver mensagens semelhantes às seguintes, indicando como o programa está sendo compilado.

```
1>----- Build started: Project: TestHello, Configuration: Debug Any CPU -----
1>  TestHello -> C:\Users\John\Documents\Microsoft Press\Visual CSharp Step By
Step\Chapter
1\TestHello\TestHello\bin\Debug\TestHello.exe
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ======
```

Qualquer erro que você cometer aparecerá na janela Error List. A imagem a seguir mostra o que acontece se você esquecer de digitar as aspas de fechamento depois do texto *Hello World* na instrução *WriteLine*. Observe que um único erro às vezes pode causar vários erros de compilador.

The screenshot shows the Microsoft Visual Studio interface. The top part displays the 'Code and Text Editor' window for a file named 'Program.cs'. The code contains a simple 'Hello World' application. The bottom part shows the 'Error List' window, which displays three errors:

Description	File	Line	Column	Project
1 Newline in constant	Program.cs	13	31	TestHello
2 ; expected	Program.cs	13	46	TestHello
3) expected	Program.cs	13	46	TestHello



Dica Para ir diretamente à linha que causou o erro, clique duas vezes em um item na janela Error List. Observe também que o Visual Studio exibe uma linha vermelha ondulada sob qualquer linha de código que não será compilada quando você a inserir.

Se você seguiu as instruções anteriores cuidadosamente, não haverá erro ou aviso algum, e o programa deverá ser compilado com sucesso.

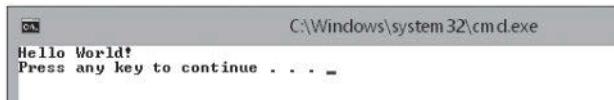


Dica Não há necessidade de salvar o arquivo explicitamente antes de compilá-lo, porque o comando Build Solution o salva automaticamente.

Um asterisco após o nome do arquivo na guia acima da janela Code and Text Editor indica que o arquivo foi alterado após ter sido salvo pela última vez.

2. No menu Debug, clique em Start Without Debugging.

Uma janela de comandos é aberta e o programa é executado. A mensagem "Hello World!" é exibida; o programa espera o usuário pressionar uma tecla, como mostra a ilustração a seguir:

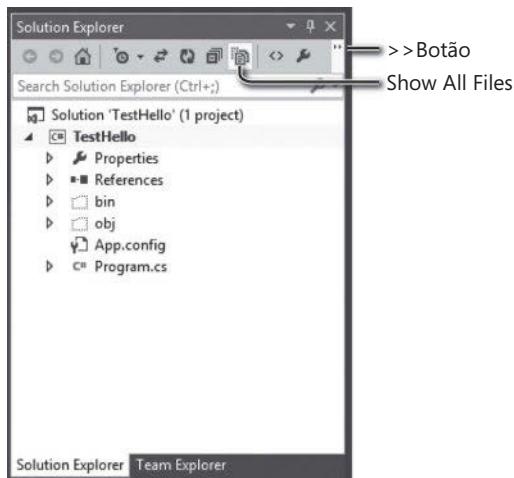


Nota O prompt "Press any key to continue" é gerado pelo Visual Studio sem que você tenha escrito código para fazer isso. Se executar o programa utilizando o comando Start Debugging no menu Debug, o aplicativo será executado, mas a janela de comando fechará imediatamente sem esperar que você pressione uma tecla.

3. Verifique se a janela de comandos que exibe a saída do programa tem o foco (significando que é a janela correntemente ativa) e, em seguida, pressione Enter.

A janela de comandos é fechada e você retorna ao ambiente de programação do Visual Studio 2013.

4. No Solution Explorer, clique no projeto *TestHello* (não na solução) e depois, na barra de ferramentas do Solution Explorer, clique no botão Show All Files. Observe que, para fazer esse botão aparecer, talvez seja necessário clicar no botão na margem direita da barra de ferramentas Solution Explorer.



Entradas chamadas *bin* e *obj* aparecem acima do arquivo *Program.cs*. Essas entradas correspondem diretamente às pastas chamadas *bin* e *obj* na pasta do projeto (Microsoft Press\ VisualCSharp Step By Step\Chapter 1\TestHello\TestHello). O Visual Studio as cria quando você compila seu aplicativo; elas contêm a versão executável do programa e alguns outros arquivos utilizados para compilar e depurar o aplicativo.

5. No Solution Explorer, expanda a entrada *bin*.

Outra pasta chamada Debug é exibida.



Nota Você também poderá ver uma pasta chamada Release.

6. No Solution Explorer, expanda a pasta Debug.

Aparecem diversos outros itens, incluindo um arquivo chamado *TestHello.exe*. Esse é o programa compilado, o qual é o arquivo executado quando você clica em Start Without Debugging no menu Debug. Os outros dois arquivos contêm informações que são utilizadas pelo Visual Studio 2013, se você executar o programa no modo de depuração (quando você clica em Start Debugging no menu Debug).

Namespaces

O exemplo que vimos até aqui é o de um programa muito pequeno. Mas programas pequenos podem crescer bastante. À medida que o programa se desenvolve, duas questões surgem. Primeiro, é mais difícil entender e manter programas grandes do que programas menores. Segundo, mais código normalmente significa mais classes,

com mais métodos, exigindo o acompanhamento de mais nomes. Conforme o número de nomes aumenta, também aumenta a probabilidade de a compilação do projeto falhar porque dois ou mais nomes entram em conflito; por exemplo, você poderia criar duas classes com o mesmo nome. A situação se torna mais complicada quando um programa faz referência a assemblies escritos por outros desenvolvedores que também utilizaram uma variedade de nomes.

Antigamente, os programadores tentavam resolver o conflito prefixando os nomes com algum tipo de qualificador (ou conjunto de qualificadores). Essa não é uma boa solução, pois não é expansível; os nomes tornam-se maiores, e você gasta menos tempo escrevendo o software e mais tempo digitando (há uma diferença), e lendo e relendo nomes longos e incomprensíveis.

Os namespaces ajudam a resolver esse problema criando um contêiner para itens, como classes. Duas classes com o mesmo nome não serão confundidas se elas estiverem em namespaces diferentes. Você pode criar uma classe chamada *Greeting* em um namespace chamado *TestHello*, utilizando a palavra-chave *namespace*, como mostrado a seguir:

```
namespace TestHello
{
    class Greeting
    {
        ...
    }
}
```

Você pode então referenciar a classe *Greeting* como *TestHello.Greeting* em seus programas. Se outro desenvolvedor também criar uma classe *Greeting* em um namespace diferente, como *NewNamespace*, e você instalar o assembly que contém essa classe no seu computador, seus programas ainda funcionarão conforme o esperado, pois usarão a classe *TestHello.Greeting*. Se quiser referenciar a classe *Greeting* do outro desenvolvedor, você deverá especificá-la como *NewNamespace.Greeting*.

É uma boa prática definir todas as suas classes em namespaces, e o ambiente do Visual Studio 2013 segue essa recomendação utilizando o nome do seu projeto como o namespace de nível mais alto. A biblioteca de classes do .NET Framework também segue essa recomendação: toda classe no .NET Framework está situada em um namespace. Por exemplo, a classe *Console* reside no namespace *System*. Isso significa que seu nome completo é, na verdade, *System.Console*.

Porém, se você tivesse que escrever o nome completo de uma classe sempre que ela fosse utilizada, seria melhor prefixar qualificadores ou então atribuir à classe um nome globalmente único, como *SystemConsole*. Felizmente, é possível resolver esse problema com uma diretiva *using* nos seus programas. Se você retornar ao programa *TestHello* no Visual Studio 2013 e examinar o arquivo *Program.cs* na janela Code and Text Editor, notará as seguintes linhas no início do arquivo:

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

Essas linhas são diretivas *using*. Uma diretiva *using* adiciona um namespace ao escopo. No código subsequente, no mesmo arquivo, você não precisa mais qualificar explicitamente os objetos com o namespace ao qual eles pertencem. Os cinco namespaces mostrados contêm classes utilizadas com tanta frequência que o Visual Studio 2013 adiciona essas instruções *using* automaticamente toda vez que você cria um novo projeto. Você pode adicionar outras diretivas *using* no início de um arquivo-fonte, caso precise referenciar outros namespaces.

O exercício a seguir demonstra o conceito dos namespaces com mais detalhes.

Experimente os nomes longos

1. Na janela Code and Text Editor que exibe o arquivo *Program.cs*, transforme em comentário a primeira diretiva *using* na parte superior do arquivo, desta maneira:

```
//using System;
```

2. No menu Build, clique em Build Solution.

A compilação falha e a janela Error List exibe a seguinte mensagem de erro:

The name 'Console' does not exist in the current context.

3. Na janela Error List, clique duas vezes na mensagem de erro.

O identificador que causou o erro é destacado no arquivo-fonte *Program.cs*.

4. Na janela Code and Text Editor, edite o método *Main* para utilizar o nome completo *System.Console*.

Main deve se parecer com isto:

```
static void Main(string[] args)
{
    System.Console.WriteLine("Hello World!");
}
```



Nota Quando você digita o ponto final após *System*, os nomes de todos os itens no namespace *System* são exibidos pelo IntelliSense.

5. No menu Build, clique em Build Solution.

A compilação do projeto deve ser bem-sucedida desta vez. Se não for, certifique-se de que o código *Main* está exatamente como aparece no código precedente e, em seguida, tente recompilar outra vez.

6. Execute o aplicativo para verificar se ele ainda funciona, clicando em Start Without Debugging no menu Debug.

7. Depois que o programa for executado e exibir "Hello World!", na janela do console, pressione Enter para retornar ao Visual Studio 2013.

Namespaces e assemblies

Uma diretiva *using* coloca em escopo os itens de um namespace, e você não precisa qualificar completamente os nomes das classes no seu código. As classes são compiladas em *assemblies*. Um assembly é um arquivo que tem, em geral, a extensão de nome de arquivo .dll, embora programas executáveis com a extensão de nome de arquivo .exe também sejam assemblies.

Um assembly pode conter muitas classes. As classes de biblioteca abrangidas pela biblioteca de classes do .NET Framework, como *System.Console*, são fornecidas nos assemblies instalados no seu computador junto com o Visual Studio. Você descobrirá que a biblioteca de classes do .NET Framework contém milhares de classes. Se todas fossem armazenadas nos mesmos assemblies, estes seriam enormes e difíceis de manter. (Se a Microsoft atualizasse um único método em uma única classe, ela teria de distribuir toda a biblioteca de classes a todos os desenvolvedores!)

Por essa razão, a biblioteca de classes do .NET Framework é dividida em alguns assemblies, agrupados de acordo com a área funcional a que as classes estão relacionadas. Por exemplo, um assembly “básico” (na verdade, chamado *mscorlib.dll*) contém todas as classes comuns, como *System.Console*, e outros assemblies contêm classes para manipular bancos de dados, acessar web services, compilar GUIs e assim por diante. Se quiser utilizar uma classe em um assembly, você deve adicionar ao seu projeto uma referência a ele. Então, pode adicionar instruções *using* ao seu código, colocando em escopo os itens do namespace nesse assembly.

Observe que não há necessariamente uma equivalência 1:1 entre um assembly e um namespace. Um único assembly pode conter classes definidas para muitos namespaces e um único namespace pode abranger vários assemblies. Por exemplo, as classes e itens do namespace *System* são, na verdade, implementados por vários assemblies, incluindo *mscorlib.dll*, *System.dll* e *System.Core.dll*, dentre outros. Isso parece muito confuso agora, mas você logo irá se acostumar.

Ao utilizar o Visual Studio para criar um aplicativo, o template que você seleciona inclui automaticamente referências aos assemblies adequados. Por exemplo, no Solution Explorer do projeto TestHello, expanda a pasta References. Você verá que um aplicativo de console contém automaticamente referências a assemblies chamados *Microsoft.CSharp*, *System*, *System.Core*, *System.Data*, *System.Data.Extensions*, *System.Xml* e *System.Xml.Linq*. Talvez você fique surpreso ao ver que *mscorlib.dll* não está nessa lista. Isso acontece porque todos os aplicativos do .NET Framework devem usar esse assembly, pois ele contém a funcionalidade de tempo de execução fundamental. A pasta References lista somente os assemblies opcionais; é possível adicionar ou remover assemblies dessa pasta, conforme for necessário.

Para acrescentar referências para assemblies adicionais em um projeto, clique com o botão direito do mouse na pasta References e então, no menu de atalho que aparece, clique em Add Reference – você fará isso nos próximos exercícios. Você também pode remover um assembly, clicando nele com o botão direito do mouse na pasta References e, então, clicando em Remove.

Crie um aplicativo gráfico

Até aqui, você usou o Visual Studio 2013 para criar e executar um aplicativo de console básico. O ambiente de programação do Visual Studio 2013 também contém tudo que você precisa para criar aplicativos gráficos para Windows 7, Windows 8 e Windows 8.1. Você pode projetar a interface de usuário (IU) de um aplicativo para Windows de modo interativo. O Visual Studio 2013 então gera as instruções do programa para implementar a interface de usuário que você projetou.

O Visual Studio 2013 fornece duas visualizações de um aplicativo gráfico: a visualização de projeto (*design view*) e a visualização de código (*code view*). Utilize a janela Code and Text Editor para modificar e manter o código e a lógica do programa para um aplicativo gráfico, e a janela Design View para organizar sua interface do usuário. Você pode alternar entre as duas visualizações sempre que quiser.

Nos exercícios a seguir, você aprenderá a criar um aplicativo gráfico utilizando o Visual Studio 2013. Esse programa exibe um formulário simples, contendo uma caixa de texto em que você pode inserir seu nome e um botão que, quando clicado, exibe uma saudação personalizada.



Importante No Windows 7 e no Windows 8, O Visual Studio 2013 fornece dois templates para compilar aplicativos gráficos: o template Windows Forms Application e o template WPF Application. Windows Forms é uma tecnologia que surgiu no .NET Framework versão 1.0. O WPF, ou Windows Presentation Foundation, é uma tecnologia aprimorada que apareceu na versão 3.0 do .NET Framework. O WPF oferece muitos recursos adicionais em relação ao Windows Forms, e você deve considerar o seu uso no lugar do Windows Forms para todos os novos desenvolvimentos para Windows 7.

Também é possível compilar aplicativos Windows Forms e WPF no Windows 8.1. Contudo, o Windows 8 e o Windows 8.1 oferecem um novo tipo de interface do usuário, denominado estilo “Windows Store”. Os aplicativos que utilizam esse estilo de interface são chamados aplicativos Windows Store. O Windows 8 foi projetado para funcionar em uma variedade de hardware, incluindo computadores com telas sensíveis ao toque e tablets ou slates. Esses computadores permitem aos usuários interagir com os aplicativos por meio de gestos baseados em toques — por exemplo, os usuários podem passar o dedo nos aplicativos para movê-los na tela e girá-los ou “apertar” e “alongar” aplicativos para diminuí-los e ampliá-los novamente. Além disso, muitos tablets contêm sensores que detectam a orientação do dispositivo, e o Windows 8 pode passar essa informação para um aplicativo, o qual pode então ajustar a interface do usuário dinamicamente, de acordo com a orientação (pode trocar do modo paisagem para retrato, por exemplo). Se você tiver instalado o Visual Studio 2013 em um computador Windows 8.1, receberá um conjunto adicional de templates para compilar aplicativos Windows Store. *Contudo, esses templates dependem dos recursos fornecidos pelo Windows 8.1; portanto, se você estiver usando o Windows 8, os templates do Windows Store não estarão disponíveis.*

Para satisfazer os desenvolvedores de Windows 7, de Windows 8 e de Windows 8.1, em muitos dos exercícios, forneci instruções para uso dos templates WPF. Se estiver usando o Windows 7 ou o Windows 8, você deverá seguir as instruções do Windows 7. Se quiser usar o estilo de interface de usuário Windows Store, você deve seguir as instruções do Windows 8.1. Evidentemente, você pode seguir as instruções para Windows 7 e para Windows 8 para usar os templates WPF no Windows 8.1, se preferir.

Caso queira mais informações sobre os pormenores de como escrever aplicativos para Windows 8.1, os capítulos finais da Parte IV deste livro fornecem mais detailes e orientações.

Crie um aplicativo gráfico no Visual Studio 2013

- Se estiver usando o Windows 8.1, execute as seguintes operações para criar um novo aplicativo gráfico:
 - a. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
 - b. No menu File, aponte para New e então clique em Project.

A caixa de diálogo New Project se abre.

 - c. No painel da esquerda, na seção Installed Templates, expanda Visual C# (se ainda não estiver expandido) e então clique na pasta Windows Store.
 - d. No painel central, clique no ícone Blank App (XAML).



Nota XAML significa Extensible Application Markup Language, que é a linguagem utilizada por aplicativos Windows Store para definir o layout de sua GUI. Você vai aprender mais sobre XAML à medida que avançar nos exercícios do livro.

- e. Certifique-se de que o campo Location refere-se à pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 1, na pasta Documentos.
- f. Na caixa Name, digite **Hello**.
- g. Na caixa Solution, assegure-se de que Create New Solution está selecionado.

Essa ação cria uma nova solução para armazenar o projeto. A alternativa Add To Solution adiciona o projeto à solução TestHello, mas não é isso que você quer para este exercício.

- h. Clique em OK.

Se essa for a primeira vez que você criou um aplicativo Windows Store, será solicitado a apresentar uma licença de desenvolvedor. Você deve concordar com os termos e condições indicados na caixa de diálogo,

antes de continuar a compilar aplicativos Windows Store. Se estiver de acordo com essas condições, clique em I Agree, como mostrado na ilustração a seguir. Será solicitado que você entre no Windows Live (nesse ponto, é possível criar uma nova conta, se necessário) e uma licença de desenvolvedor será criada e reservada para você.



i. Após a criação do aplicativo, examine a janela Solution Explorer.

Não se engane com o nome do template de aplicativo — embora seja chamado Blank App, na verdade esse template fornece vários arquivos e contém algum código. Por exemplo, se você expandir a pasta MainPage.xaml, encontrará um arquivo C# chamado MainPage.xaml.cs. Esse arquivo é onde você insere o código executado quando a interface do usuário definida pelo arquivo MainPage.xaml é exibida.

j. No Solution Explorer, clique duas vezes em MainPage.xaml.

Esse arquivo contém o layout da interface do usuário. A janela Design View mostra duas representações desse arquivo:

Na parte superior está uma visualização gráfica representando a tela de um computador tablet. O painel inferior contém uma descrição do conteúdo dessa tela em XAML. XAML é uma linguagem tipo XML utilizada por aplicativos Windows Store e WPF para definir o layout de um formulário e seu conteúdo. Se você conhece XML, a XAML deverá lhe parecer familiar.

No próximo exercício, você vai usar a janela Design View para organizar a interface do usuário do aplicativo e vai examinar o código XAML gerado por esse layout.



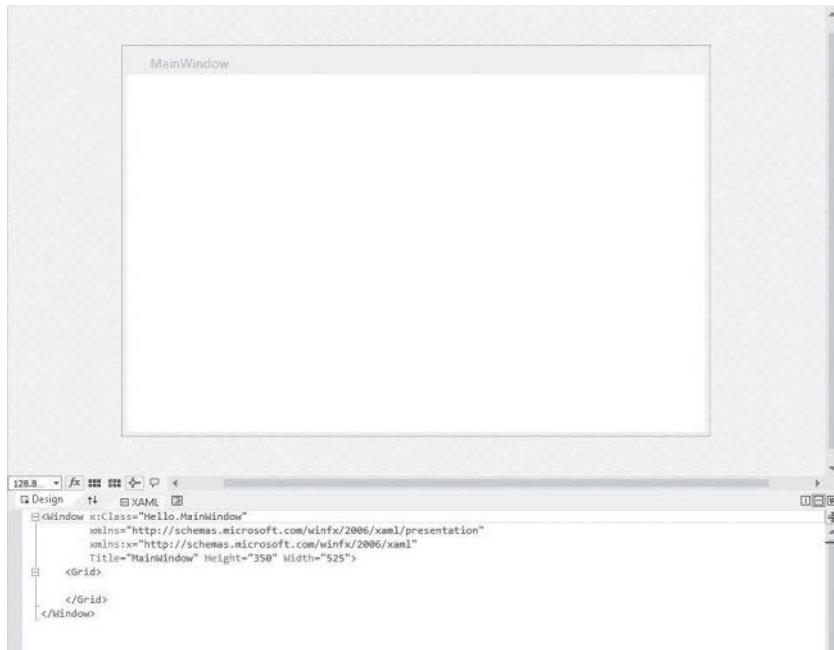
- Se estiver usando o Windows 8 ou o Windows 7, execute as seguintes tarefas:
 - a. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
 - b. No menu File, aponte para New e então clique em Project.

A caixa de diálogo New Project se abre.

 - c. No painel da esquerda, na seção Installed Templates, expanda Visual C# (se ainda não estiver expandido) e então clique na pasta Windows.
 - d. No painel central, clique no ícone WPF Application.
 - e. Certifique-se de que a caixa Location refere-se à pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 1, na pasta Documentos.
 - f. Na caixa Name, digite **Hello**.
 - g. Na caixa Solution, assegure-se de que Create New Solution está selecionado e clique em OK.

O template WPF Application gera menos itens do que o template Windows Store Blank App; ele não contém os estilos gerados pelo template Blank App, pois a funcionalidade incorporada nesses estilos é específica para o Windows 8.1. Contudo, o template WPF Application gera uma janela padrão para seu aplicativo. Como em um aplicativo Windows Store, essa janela é definida com XAML, mas, neste caso, é chamada `MainWindow.xaml` por padrão.

- h.** No Solution Explorer, clique duas vezes em MainWindow.xaml para exibir o conteúdo desse arquivo na janela Design View.



Dica Feche as janelas Output e Error List para dar mais espaço à exibição da janela Design View.

Nota Antes de prosseguirmos, é importante explicarmos alguma terminologia. Em um aplicativo WPF típico, a interface do usuário consiste em uma ou mais *janelas*, mas em um aplicativo Windows Store os itens correspondentes são chamados de *páginas* (rigorosamente falando, um aplicativo WPF também pode conter páginas, mas não quero confundir as coisas neste ponto). Para não ficar repetindo a frase bastante prolixia “janela WPF ou página de aplicativo Windows Store” no livro, vou simplesmente me referir aos dois itens usando o termo geral *formulário*. Entretanto, continuarei usando a palavra *janela* para me referir aos itens do IDE do Visual Studio 2013, como a janela Design View.

Nos próximos exercícios, você vai utilizar a janela Design View para adicionar três controles ao formulário exibido por seu aplicativo e examinar alguns dos códigos C# gerados automaticamente pelo Visual Studio 2013 para implementar esses controles.



Nota Os passos dos próximos exercícios são comuns para o Windows 7, para o Windows 8 e para o Windows 8.1, exceto onde quaisquer diferenças sejam explicitamente indicadas.

Crie a interface do usuário

1. Clique na guia Toolbox exibida à esquerda do formulário na janela Design View. A Toolbox aparece, ocultando parcialmente o formulário, e exibe os vários componentes e controles que você pode colocar em um formulário.
2. Se estiver utilizando o Windows 8.1, expanda a seção Common XAML Controls. Se estiver utilizando o Windows 7 ou o Windows 8, expanda a seção Common WPF Controls. Essa seção exibe uma lista de controles utilizados pela maioria dos aplicativos gráficos.



Dica A seção All XAML Controls (Windows 8.1) ou All WPF Controls (Windows 7 e Windows 8) exibe uma lista mais extensa de controles.

3. Na seção Common XAML Controls ou Common WPF Controls, clique em TextBlock e arraste o controle *TextBlock* para o formulário exibido na janela Design View.



Dica Certifique-se de selecionar o controle *TextBlock* e não o controle *TextBox*. Se accidentalmente você colocar o controle errado em um formulário, pode removê-lo com facilidade, clicando no item no formulário e pressionando Delete.

Um controle *TextBlock* é adicionado ao formulário (você o moverá para o local correto mais adiante), e a Toolbox é ocultada.



Dica Se quiser que a Toolbox permaneça visível, mas não oculte nenhuma parte do formulário, na extremidade direita da barra de título da Toolbox, clique no botão Auto Hide (ele parece um alfinete). A Toolbox aparece permanentemente no lado esquerdo da janela do Visual Studio 2013 e a janela Design View é reduzida para acomodá-la. (Talvez você perca muito espaço se tiver uma tela com baixa resolução.) Clicar no botão Auto Hide mais uma vez fará a Toolbox desaparecer novamente.

4. É provável que o controle *TextBlock* no formulário não esteja exatamente onde você quer. Você pode clicar e arrastar os controles que adicionou a um formulário para repositioná-los. Utilizando essa técnica, mova o controle *TextBlock* para positioná-lo próximo ao canto superior esquerdo do formulário. (O local exato

não é importante para esse aplicativo.) Observe que talvez seja preciso clicar longe do controle e, então, clicar nele novamente, antes que você possa movê-lo na janela Design View.

No painel inferior, a descrição XAML do formulário agora inclui o controle *TextBlock*, junto com propriedades, como sua localização no formulário (controlada pela propriedade *Margin*), o texto padrão exibido por esse controle (na propriedade *Text*), o alinhamento do texto exibido por esse controle (especificado pelas propriedades *HorizontalAlignment* e *VerticalAlignment*) e se o texto deve passar para a próxima linha se ultrapassar a largura do controle *TextWrapping*.

Se você estiver usando Windows 8.1, o código XAML do controle *TextBlock* será parecido com este (seus valores para a propriedade *Margin* poderão ser um pouco diferentes, dependendo de onde você posicionou o controle *TextBlock* no formulário):

```
<TextBlock HorizontalAlignment="Left" Margin="400,200,0,0" TextWrapping="Wrap"
Text="TextBlock" VerticalAlignment="Top"/>
```

Se estiver usando Windows 7 ou Windows 8, o código XAML será praticamente o mesmo, exceto que as unidades utilizadas pela propriedade *Margin* operam em uma escala diferente, devido à resolução maior dos dispositivos Windows 8.1.

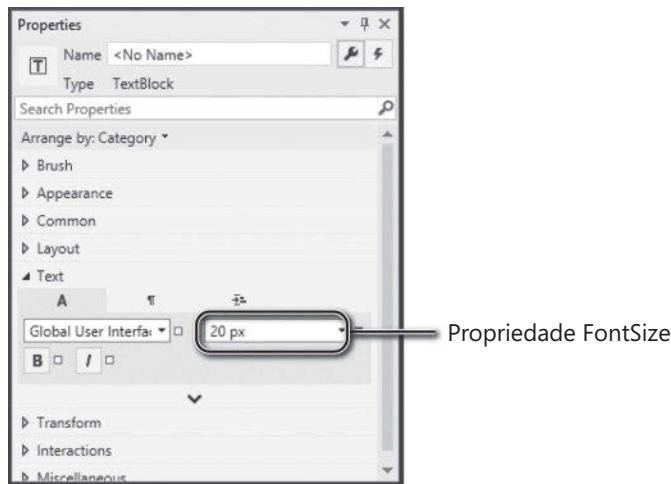
O painel XAML e a janela Design View têm uma relação bilateral entre si. Você pode editar os valores no painel XAML e as alterações serão refletidas na janela Design View. Por exemplo, você pode mudar o local do controle *TextBlock* modificando os valores da propriedade *Margin*.

5. No menu View, clique em Properties Window.

Se já estava aberta, a janela Properties aparece no canto inferior direito da tela, sob o Solution Explorer. É possível especificar as propriedades dos controles usando o painel XAML sob a janela Design View, mas a janela Properties é uma maneira mais prática de modificar as propriedades dos itens em um formulário, assim como outros itens em um projeto.

A janela Properties é sensível ao contexto, exibindo as propriedades do item selecionado. Se clicar no formulário exibido na janela Design View, fora do controle *TextBlock*, você verá que a janela Properties exibe as propriedades de um elemento *Grid*. Se examinar o painel XAML, você verá que o controle *TextBlock* está contido em um elemento *Grid*. Todos os formulários contêm um elemento *Grid* que controla o layout dos itens exibidos – é possível definir layouts tabulares adicionando linhas e colunas ao elemento *Grid*, por exemplo.

- 6.** Na janela Design View, clique no controle *TextBlock*. A janela Properties exibe novamente as propriedades do controle *TextBlock*.
- 7.** Na janela Properties, expanda a propriedade *Text*. Altere a propriedade *FontSize* para **20 px** e, em seguida, pressione Enter. Essa propriedade está localizada ao lado da lista suspensa que contém o nome da fonte, o qual será diferente para o Windows 8.1 (Global User Interface) e para o Windows 7 ou Windows 8 (Segoe UI):



Nota O sufixo **px** indica que o tamanho da fonte é medido em pixels.

8. No painel XAML, abaixo da janela Design View, examine o texto que define o controle *TextBlock*. Se você fizer uma rolagem até o final da linha, deverá ver o texto *FontSize = "20"*. Todas as alterações feitas na janela Properties constarão automaticamente nas definições do XAML e vice-versa.

Digite sobre o valor da propriedade *FontSize* no painel XAML, alterando-o para **24**. O tamanho da fonte do texto do controle *TextBlock* na janela Design View e na janela Properties muda.

9. Na janela Properties, examine as outras propriedades do controle *TextBlock*. Sinta-se livre para fazer testes, alterando-as para ver seus efeitos.

Observe que, à medida que você altera os valores das propriedades, essas propriedades são adicionadas à definição do controle *TextBlock* no painel XAML. Cada controle adicionado a um formulário tem um conjunto de valores de propriedade padrão e esses valores não aparecem no painel XAML, a não ser que você os altere.

10. Altere o valor da propriedade *Text* do controle *TextBlock*, de *TextBlock* para **Please enter your name** (Digite seu nome). Isso pode ser feito editando-se o elemento *Text* no painel XAML ou alterando-se o valor na janela Properties (essa propriedade está localizada na seção Common da janela Properties).

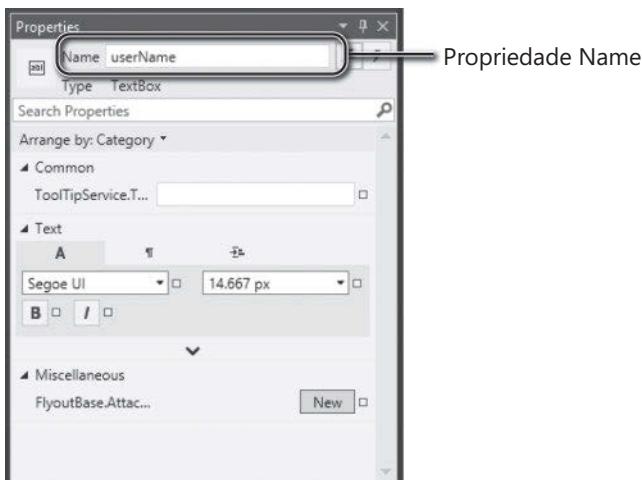
Observe que o texto exibido no controle *TextBlock* na janela Design View muda.

11. Clique no formulário na janela Design View e exiba a Toolbox novamente.
12. Na Toolbox, clique e arraste o controle *TextBox* para o formulário. Mova o controle *TextBox* para posicionará-lo imediatamente abaixo do controle *TextBlock*.



Dica Ao se arrastar um controle em um formulário, indicadores de alinhamento aparecem automaticamente quando o controle torna-se alinhado vertical ou horizontalmente a outros controles. É uma dica visual rápida para você se certificar de que esses controles estão alinhados de modo correto.

13. Na janela Design View, posicione o mouse sobre a borda direita do controle *TextBox*. O cursor do mouse deve mudar para uma seta de duas pontas, indicando que você pode redimensionar o controle. Arraste a borda direita do controle *TextBox* até que ele esteja alinhado com a borda direita do controle *TextBlock* acima; uma guia deverá aparecer quando as duas bordas estiverem alinhadas corretamente.
14. Com o controle *TextBox* ainda selecionado, altere o valor da propriedade *Name* exibida na parte superior da janela Properties, de <No Name> para **userName**, como ilustrado a seguir:



Nota Falaremos mais sobre as convenções de nomes para controles e variáveis no Capítulo 2, "Variáveis, operadores e expressões".

15. Exiba a Toolbox novamente, depois clique e arraste um controle *Button* para o formulário. Posicione o controle *Button* à direita da caixa do controle *TextBox* no formulário, de modo que a parte inferior do botão fique alinhada horizontalmente com a parte inferior da caixa de texto.
16. Na janela Properties, mude a propriedade *Name* do controle *Button* para **ok**, mude a propriedade *Content* (na seção Common) de *Button* para **OK** e pressione Enter. Verifique que a legenda do controle *Button* no formulário muda para exibir o texto *OK*.
17. Se estiver usando Windows 7 ou Windows 8, clique na barra de título do formulário na janela Design View. Na janela Properties, mude a propriedade *Title* (novamente, na seção Common) de *MainWindow* para **Hello**.



Nota Os aplicativos Windows Store não têm barra de título.

18. Se estiver usando Windows 7 ou Windows 8, na janela Design View, clique na barra de título do formulário Hello. Observe que uma alça de redimensionamento (um pequeno quadrado) aparece no canto inferior direito do formulário Hello. Mova o cursor do mouse sobre a alça de redimensionamento. Quando o cursor virar uma seta de duas pontas diagonal, arraste-o para redimensionar o formulário. Pare de arrastar e solte o botão do mouse quando o espaçamento em torno dos controles estiver igual.



Importante Clique na barra de título do formulário Hello e não no contorno da grade dentro do formulário, antes de redimensioná-lo. Se selecionar a grade, você modificará o layout dos controles no formulário, mas não o tamanho do formulário.

O formulário Hello deve ficar parecido com a figura a seguir:



Nota Nos aplicativos Windows Store, as páginas não podem ser redimensionadas da mesma maneira que nos formulários WPF; quando são executados, eles ocupam automaticamente a tela inteira do dispositivo. Contudo, eles podem se adaptar a diferentes resoluções de tela e à orientação do dispositivo, apresentando diferentes visualizações quando são “encaixados”. É fácil ver como seu aplicativo aparece em um dispositivo diferente, clicando em Device Window no menu Design e, então, selecionando as diferentes resoluções de tela disponíveis na lista suspensa Display. Também é possível ver como seu aplicativo aparece no modo retrato ou quando está encaixado, selecionando a orientação Portrait ou a visualização Snapped na lista de visualizações disponíveis.

19. No menu Build, clique em Build Solution e verifique se a compilação do projeto foi bem-sucedida.
20. No menu Debug, clique em Start Debugging.

O aplicativo deve ser executado, exibindo seu formulário. Se você está usando Windows 8.1, o formulário ocupa a tela inteira e aparece deste modo:



Nota Quando um aplicativo Windows Store é executado no modo Debug no Windows 8.1, aparecem dois pares de números nos cantos superior esquerdo e superior direito da tela. Esses números controlam a taxa de redesenho (*frame rate*) e os desenvolvedores podem utilizá-los para determinar quando um aplicativo começa a demorar mais do que devia para responder (possivelmente uma indicação de problemas de desempenho). Eles só aparecem quando um aplicativo é executado no modo Debug. Uma descrição completa do significado desses números está fora dos objetivos deste livro; portanto, você pode ignorá-los por enquanto.

Se você está usando Windows 7 ou Windows 8, o formulário aparece deste modo:



Na caixa de texto, você pode digitar sobre o que está lá, digitar seu nome e clicar em OK, mas nada acontecerá ainda. É necessário adicionar algum código para indicar o que deve acontecer quando o usuário clicar no botão OK, o que faremos em seguida.

21. Retorne ao Visual Studio 2013. No menu DEBUG, clique em Stop Debugging.

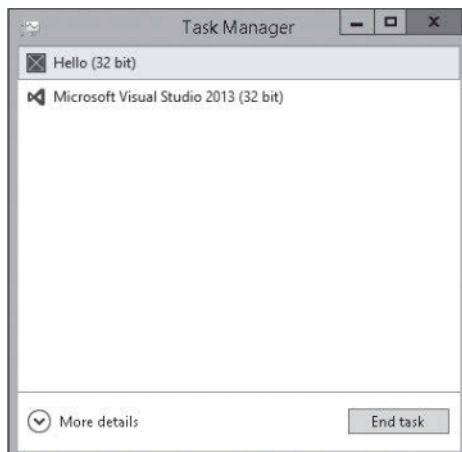
- Se você está usando o Windows 8.1, pressione a tecla Windows+B. Isso deve levá-lo de volta à Área de Trabalho do Windows que está executando o Visual Studio, a partir do qual é possível acessar o menu Debug.

- Se você está usando Windows 7 ou Windows 8, pode trocar diretamente para o Visual Studio. Também é possível clicar no botão de fechamento (o X no canto superior direito do formulário) para fechar o formulário, interromper a depuração e retornar ao Visual Studio.

Como fechar um aplicativo Windows Store

Se você está usando Windows 8.1 e clicou em Start Without Debugging no menu Debug para executar o aplicativo, precisará fechá-lo à força. Isso porque, ao contrário dos aplicativos de console, a vida de um aplicativo Windows Store é gerenciada pelo sistema operacional e não pelo usuário. O Windows 8.1 suspende um aplicativo quando não está sendo exibido e o terminará quando o sistema operacional precisar a liberação dos recursos que ele consome. O modo mais confiável de interromper o aplicativo Hello à força é clicar (ou colocar o dedo, caso você tenha uma tela sensível ao toque) na parte superior da tela e, então, clicar e arrastar (ou deslizar) o aplicativo para a parte inferior, e segurá-lo até que sua imagem se dobre (se você soltar o aplicativo antes da imagem se dobrar, ele continuará sendo executado em segundo plano). Essa ação fecha o aplicativo e o leva de volta à tela Iniciar do Windows, onde você pode retornar ao Visual Studio. Como alternativa, você pode executar as seguintes tarefas:

1. Clique (ou coloque o dedo) no canto superior direito da tela e, então, arraste a imagem do Visual Studio para o meio da tela (ou pressione a tecla Windows+B).
2. Na parte inferior da área de trabalho, clique com o botão direito do mouse na barra de tarefas do Windows e, então, clique em Iniciar Gerenciador de Tarefas.
3. Na janela Gerenciador de Tarefas do Windows, clique no aplicativo Hello e, em seguida, clique em Finalizar Tarefa.



4. Feche a janela Gerenciador de Tarefas do Windows.

Você conseguiu criar um aplicativo gráfico sem escrever uma única linha de código em C#. Esse aplicativo ainda não faz muito (será necessário escrever algum código), mas o Visual Studio 2013 gera uma grande quantidade de código que trata das tarefas de rotina que todos os aplicativos gráficos devem realizar, como abrir e exibir uma janela. Antes de adicionar seu próprio código ao aplicativo, é importante entender o que Visual Studio produziu. A estrutura é um pouco diferente entre um aplicativo Windows Store e um aplicativo WPF, e as seções a seguir resumem esses estilos de aplicativo separadamente.

Examine o aplicativo Windows Store

Se estiver usando Windows 8.1, no Solution Explorer, clique na seta adjacente ao arquivo MainPage.xaml para expandir o nó. O arquivo MainPage.xaml.cs aparece; clique duas vezes nesse arquivo. O código a seguir, do formulário, é exibido na janela Code and Text Editor.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// O template do item Blank Page está documentado em http://go.microsoft.com/fwlink/?LinkId=234238

namespace Hello
{
    /// <summary>
    /// Uma página vazia que pode ser usada sozinha ou acessada dentro de um Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Além de muitas diretivas *using* que colocam no escopo alguns namespaces que a maioria dos aplicativos Windows Store utiliza, o arquivo contém apenas a definição de uma classe chamada *MainPage*. Há um pouco de código para a classe *MainPage*, conhecido como *construtor*, que chama um método denominado *InitializeComponent*. Um construtor é um método especial com o mesmo nome da classe. Ele é executado quando é criada uma instância da classe e pode conter um código para inicializar a instância. Discutiremos sobre construtores no Capítulo 7.

Na realidade, a classe contém muito mais código do que as poucas linhas mostradas no arquivo MainPage.xaml.cs, mas grande parte dele é gerada automaticamente com base na descrição XAML do formulário, e é ocultada. Esse código oculto realiza operações como criar e exibir o formulário e também criar e posicionar os vários controles no formulário.



Dica Você também pode exibir o arquivo do código C# para uma página em um aplicativo Windows Store, clicando em Code no menu View quando a janela Design View estiver exibida.

Você deve estar se perguntando onde está o método *Main* e como o formulário será exibido quando o aplicativo for executado. Lembre-se de que, em um aplicativo de console, *Main* define o ponto em que o programa inicia. Um aplicativo gráfico é um pouco diferente.

No Solution Explorer deve aparecer outro arquivo-fonte chamado App.xaml. Se expandir o nó desse arquivo, você verá outro arquivo, chamado App.xaml.cs. Em um aplicativo Windows Store, o arquivo App.xaml fornece o ponto de entrada no qual o aplicativo começa a executar. Se você clicar duas vezes em App.xaml.cs no Solution Explorer, verá código semelhante a este:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// O template do item Blank Application está documentado em http://go.microsoft.com/fwlink/?LinkId=234227

namespace Hello
{
    /// <summary>
    /// Fornece comportamento específico do aplicativo para complementar a classe Application
    /// padrão.
    /// </summary>
    sealed partial class App : Application
    {
        /// <summary>
        /// Inicializa o objeto aplicativo singleton. Esta é a primeira linha executada
        /// do código escrito e, como tal, é o equivalente lógico de main() ou WinMain().
        /// </summary>
        public App()
        {
            this.InitializeComponent();
            this.Suspending += OnSuspending;
```

```
}

/// <summary>
/// Executado quando o aplicativo é chamado normalmente pelo usuário final. Outros pontos
/// de entrada serão usados quando o aplicativo for chamado para abrir um arquivo
/// específico, para exibir resultados de pesquisa e assim por diante.
/// </summary>
/// <param name="args">Details about the launch request and process.</param>
protected override void OnLaunched(LaunchActivatedEventArgs e)
{

#if DEBUG
    if (System.Diagnostics.Debugger.IsAttached)
    {
        this.DebugSettings.EnableFrameRateCounter = true;
    }
#endif

    Frame rootFrame = Window.Current.Content as Frame;

    // Não repete a inicialização do aplicativo quando a janela já tem conteúdo,
    // apenas garante que ela esteja ativa
    if (rootFrame == null)
    {
        // Cria um Frame para atuar como contexto de navegação e navega para a
primeira página
        rootFrame = new Frame();

        if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            //TODO: carregar estado do aplicativo suspenso anteriormente
        }

        // Coloca o frame na janela atual
        Window.Current.Content = rootFrame;
    }

    if (rootFrame.Content == null)
    {
        // Quando a pilha de navegação não é restaurada, navega para a primeira
        // página, configurando a nova página passando as informações exigidas
        // como parâmetro de navegação
        if (!rootFrame.Navigate(typeof(MainPage), e.Arguments))
        {
            throw new Exception("Failed to create initial page");
        }
    }
    // Garante que a janela atual esteja ativa
    Window.Current.Activate();
}

/// <summary>
/// Chamado quando a execução do aplicativo está sendo suspensa. O estado do aplicativo
/// é salvo sem saber se ele será terminado ou retomado com o conteúdo
/// da memória ainda intacto.
/// </summary>
```

```

    /// </summary>
    /// <param name="sender">The source of the suspend request.</param>
    /// <param name="e">Details about the suspend request.</param>
    private void OnSuspending(object sender, SuspendingEventArgs e)
    {
        var deferral = e.SuspendingOperation.GetDeferral();
        //TODO: salvar o estado do aplicativo e interromper qualquer atividade de segundo
plano
        deferral.Complete();
    }
}
}

```

Grande parte desse código consiste em comentários (as linhas que começam com “///”) e outras instruções que você ainda não precisa entender, mas os principais elementos estão localizados no método *OnLaunched*, realçado em negrito. Esse método é executado quando o aplicativo começa e o código presente nele faz com que o aplicativo crie um novo objeto *Frame*, exiba o formulário *MainPage* nesse quadro (frame) e, então, o ative. Neste estágio, não é necessário compreender completamente o funcionamento desse código ou a sintaxe de qualquer uma dessas instruções, mas é útil reconhecer que é assim que o aplicativo exibe o formulário, quando começa a ser executado.

Examine o aplicativo WPF

Se estiver usando o Windows 7 ou o Windows 8, no Solution Explorer, clique na seta adjacente ao arquivo *MainWindow.xaml* para expandir o nó. O arquivo *MainWindow.xaml.cs* aparece; clique duas vezes nesse arquivo. O código do formulário aparece na janela Code and Text Editor, como mostrado aqui:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace Hello
{
    /// <summary>
    /// Lógica de interação para MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}

```

Esse código parece semelhante ao do aplicativo Windows Store, mas existem algumas diferenças significativas – muitos dos namespaces referenciados pelas diretivas *using* no início do arquivo são diferentes. Por exemplo, os aplicativos WPF utilizam objetos definidos em namespaces que começam com o prefixo *System.Windows*, enquanto os aplicativos Windows Store utilizam objetos definidos em namespaces que começam com *Windows.UI*. Essa diferença não é superficial. Esses namespaces são implementados por diferentes assemblies e os controles e a funcionalidade oferecidos por eles são diferentes entre os aplicativos WPF e Windows Store, embora possam ter nomes semelhantes. Voltando ao exercício anterior, você adicionou controles *TextBlock*, *TextBox* e *Button* ao formulário WPF e ao aplicativo Windows Store. Embora esses controles tenham o mesmo nome em cada estilo de aplicativo, eles são definidos em diferentes assemblies: *Windows.UI.Xaml.Controls* para aplicativos Windows Store e *System.Windows.Controls* para aplicativos WPF. Os controles de aplicativos Windows Store foram especificamente projetados e otimizados para interfaces de toque, enquanto os controles WPF são destinados, em especial, para uso em sistemas voltados para o mouse.

Assim como no código do aplicativo Windows Store, o construtor da classe *MainWindow* inicializa o formulário WPF chamando o método *InitializeComponent*. Novamente, como antes, o código desse método fica oculto e realiza operações como criar e exibir o formulário e também criar e posicionar os vários controles no formulário.

O modo pelo qual um aplicativo WPF especifica o formulário inicial a ser exibido é diferente de um aplicativo Windows Store. Assim como um aplicativo Windows Store, ele estipula um objeto *App* definido no arquivo *App.xaml* para fornecer o ponto de entrada para o aplicativo, mas o formulário a ser exibido é especificado de forma declarada como parte do código XAML, em vez de em forma de programa. Se você clicar duas vezes no arquivo *App.xaml* no Solution Explorer (não em *App.xaml.cs*), poderá examinar a descrição XAML. Há uma propriedade *StartupUri* no código XAML que se refere ao arquivo *MainWindow.xaml*, como mostrado em negrito no exemplo de código a seguir:

```
<Application x:Class="Hello.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        </Application.Resources>
    </Application>
```

Em um aplicativo WPF, a propriedade *StartupUri* do objeto *App* indica o formulário a ser exibido.

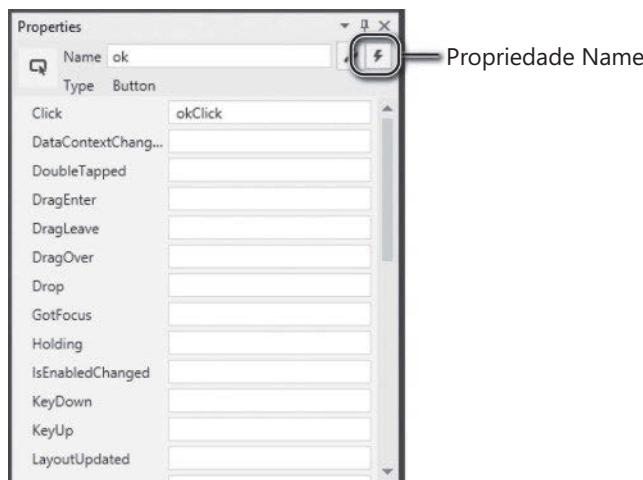
Adicione código ao aplicativo gráfico

Agora que você conhece um pouco da estrutura de um aplicativo gráfico, chegou a hora de escrever código para que seu aplicativo realmente faça alguma coisa.

Escreva o código para o botão OK

1. Na janela Design View, abra o arquivo *MainPage.xaml* (Windows 8.1) ou o arquivo *MainWindow.xaml* (Windows 7 ou Windows 8) – para isso, clique duas vezes em *MainPage.xaml* ou em *MainWindow.xaml* no Solution Explorer.

2. Ainda na janela Design View, clique no botão OK do formulário para selecioná-lo.
3. Na janela Properties, clique no botão Event Handlers for the Selected Element. Esse botão exibe um ícone parecido com um relâmpago, como demonstrado aqui:



A janela Properties exibe uma lista de nomes de evento para o controle *Button*. Um evento indica uma ação significativa que normalmente exige uma resposta, e você pode escrever seu código para executar essa resposta.

4. Na caixa adjacente ao evento *Click*, digite **okClick** e, em seguida, pressione Enter.

O arquivo *MainPage.xaml.cs* (Windows 8.1) ou *MainWindow.xaml.cs* (Windows 7 ou Windows 8) aparece na janela Code and Text Editor e um novo método chamado *okClick* é adicionado à classe *MainPage* ou *MainWindow*. O método é semelhante a este:

```
private void okClick(object sender, RoutedEventArgs e)
{
}
```

Não se preocupe com a sintaxe desse código ainda – você aprenderá tudo sobre métodos no Capítulo 3.

5. Se estiver usando o Windows 8.1, execute as seguintes tarefas:
 - a. Adicione a seguinte diretiva *using*, mostrada em negrito, à lista do início do arquivo (o caractere de reticências [...] indica instruções que foram omitidas por brevidade):

```
using System;
...
using Windows.UI.Xaml.Navigation;
using Windows.UI.Popups;
```

5. Se estiver usando o Windows 8.1, execute as seguintes tarefas:
 - b. Adicione o seguinte código mostrado em negrito ao método *okClick*:

```
void okClick(object sender, RoutedEventArgs e)
{
    MessageDialog msg = new MessageDialog("Hello " + userName.Text);
    msg.ShowAsync();
}
```

Quando compilado, este código irá exibir em "warning" a respeito do uso de um método assíncrono. Não se preocupe com a mensagem; os métodos assíncronos serão explicados no Capítulo 24.

Esse código será executado quando o usuário clicar no botão OK. Novamente, não se preocupe com a sintaxe. Apenas certifique-se de copiar o código exatamente como mostrado; você vai descobrir o que essas instruções significam nos próximos capítulos. O mais importante a entender é que a primeira instrução cria um objeto *MessageDialog* com a mensagem "Hello <SeuNome>", onde <SeuNome> é o nome que você digita no controle *TextBox* do formulário. A segunda instrução exibe o objeto *MessageDialog*, fazendo-o aparecer na tela. A classe *MessageDialog* é definida no namespace *Windows.UI.Popups* e esse é o motivo pelo qual você o adicionou no passo a.

6. Se estiver usando Windows 7 ou Windows 8, basta adicionar ao método *okClick* a única instrução mostrada em negrito:

```
void okClick(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello " + userName.Text);
}
```

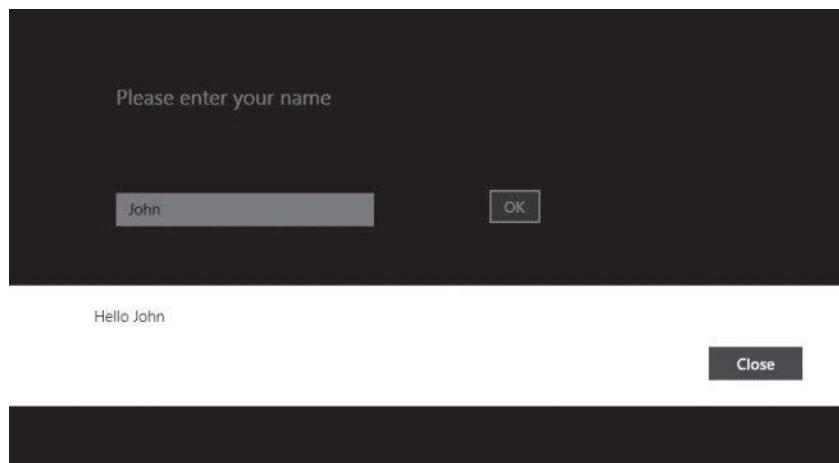
Esse código executa uma função semelhante à função do aplicativo Windows Store, exceto que utiliza uma classe diferente, chamada *MessageBox*. Essa classe é definida no namespace *System.Windows*, o qual já é referenciado pelas diretrizes *using* existentes no início do arquivo; portanto, você não precisa adicioná-lo.

7. Clique na guia *MainPage.xaml* ou na guia *MainWindow.xaml* acima da janela Code and Text Editor para exibir o formulário na janela Design View novamente.
8. No painel inferior que exibe a descrição XAML do formulário, examine o elemento *Button*, mas tenha cuidado para não alterar nada. Observe que agora ele contém um elemento chamado *Click* que se refere ao método *okClick*:

```
<Button x:Name="ok" ... Click="okClick" />
```

9. No menu Debug, clique em Start Debugging.
10. Quando o formulário aparecer, digite seu nome sobre o texto existente na caixa de texto e então clique em OK.

Se você estiver usando o Windows 8.1, aparecerá um diálogo de mensagem no meio da tela, saudando-o pelo seu nome:



Se estiver usando Windows 7 ou Windows 8, aparecerá uma caixa de mensagem exibindo a seguinte saudação:



11. Clique em Close no diálogo de mensagem (Windows 8.1) ou em OK (Windows 7 ou Windows 8) na caixa de mensagem.
12. Volte para o Visual Studio 2013 e, então, no menu Debug, clique em Stop Debugging.

Resumo

Neste capítulo, você viu como é possível utilizar o Visual Studio 2013 para criar, construir e executar aplicativos. Você criou um aplicativo de console que exibe sua saída em uma janela de console e um aplicativo WPF com uma GUI simples.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 2.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes para salvar o projeto.

Referência rápida

Para	Faça isto
Criar um novo aplicativo de console no Visual Studio 2013	No menu File, aponte para New e clique em Project para abrir a caixa de diálogo New Project. No painel à esquerda, em Installed Templates, clique em Visual C#. No painel central, clique em Console Application. Na caixa Location, especifique um diretório para os arquivos de projeto. Digite um nome para o projeto e clique em OK.
Criar um novo aplicativo gráfico Windows Store em branco para Windows 8.1 no Visual Studio 2013	No menu File, aponte para New e clique em Project para abrir a caixa de diálogo New Project. No painel da esquerda, na seção Installed Templates, expanda Visual C# e clique em Windows Store. No painel central, clique em Blank App (XAML). Na caixa Location, especifique um diretório para os arquivos de projeto. Digite um nome para o projeto e clique em OK.
Criar um novo aplicativo gráfico WPF para Windows 7 ou Windows 8 no Visual Studio 2013	No menu File, aponte para New e clique em Project para abrir a caixa de diálogo New Project. No painel da esquerda, na seção Installed Templates, expanda Visual C# e clique em Windows. No painel central, clique em WPF Application. Especifique um diretório para os arquivos do projeto na caixa Location. Digite um nome para o projeto e clique em OK.
Compilar o aplicativo	No menu Build, clique em Build Solution.
Executar o aplicativo no modo Debug	No menu Debug, clique em Start Debugging.
Executar o aplicativo sem depurar	No menu Debug, clique em Start Without Debugging.

CAPÍTULO 2

Variáveis, operadores e expressões

Neste capítulo, você vai aprender a:

- Entender instruções, identificadores e palavras-chave.
- Utilizar variáveis para armazenar informações.
- Trabalhar com tipos de dados primitivos.
- Utilizar operadores aritméticos, como o sinal de adição (+) e o sinal de subtração (-).
- Incrementar e decrementar variáveis.

O Capítulo 1, “Bem-vindo ao C#”, mostrou como utilizar o ambiente de programação do Microsoft Visual Studio 2013 para compilar e executar um programa de console e um aplicativo gráfico. Este capítulo traz os elementos de sintaxe e semântica do Microsoft Visual C#, como instruções, palavras-chave e identificadores. Você vai estudar os tipos primitivos compilados na linguagem C#, assim como as características dos valores armazenados em cada tipo. Além disso, este capítulo também explica como declarar e utilizar variáveis locais (que somente existem dentro de uma função ou outra pequena seção do código). Você vai ser apresentado aos operadores aritméticos que o C# fornece, descobrindo como deve utilizar operadores para manipular valores e aprendendo a controlar expressões com dois ou mais operadores.

Instruções

Instituição é um comando que executa uma ação, como calcular um valor e armazenar o resultado, ou exibir uma mensagem para o usuário. Você combina instruções para criar métodos. Para aprender mais sobre métodos, consulte o Capítulo 3, “Como escrever métodos e aplicar o escopo”, mas, por enquanto, considere um método como uma sequência nomeada de instruções. *Main*, que foi apresentado no capítulo anterior, é um exemplo de método.

As instruções em C# seguem um conjunto bem definido de regras que descrevem seu formato e sua construção. Estas são conhecidas coletivamente como *sintaxe*. (Por outro lado, a especificação *do que* as instruções fazem é conhecida coletivamente como *semântica*.) Uma das regras de sintaxe mais simples e mais importantes do C# diz que você deve terminar todas as instruções com um ponto e vírgula. Por exemplo, o Capítulo 1 demonstrou que, sem o ponto e vírgula de terminação, a instrução a seguir não seria compilada:

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



Dica C# é uma linguagem de “formato livre”, assim, espaços em branco, como um caractere de espaço ou uma nova linha, não têm outro significado a não ser o de serem separadores. Ou seja, você pode dispor as instruções como quiser. Mas deve adotar um estilo consistente e simples de layout para tornar seus programas mais fáceis de ler e entender.

O truque para programar bem em qualquer linguagem é aprender sua sintaxe e semântica e então utilizá-la de maneira natural e idiomática. Essa estratégia facilita a manutenção dos seus programas. À medida que avançar neste livro, você verá exemplos das instruções mais importantes do C#.

Identificadores

Identificadores são os nomes utilizados para distinguir os elementos nos seus programas, como namespaces, classes, métodos e variáveis. (Discutiremos as variáveis em breve.) No C#, você deve seguir as regras de sintaxe abaixo ao escolher os identificadores:

- Você pode utilizar apenas letras (maiúsculas ou minúsculas), dígitos e o caractere de sublinhado.
- Um identificador deve iniciar com uma letra (ou um sublinhado).

Por exemplo, *resultado*, *_placar*, *timeDeFutebol* e *plano9* são identificadores válidos, enquanto *resultado%*, *timeDeFutebol\$* e *9plano* não são.



Importante O C# é uma linguagem que diferencia maiúsculas de minúsculas: *timeDeFutebol* e *TimeDeFutebol* são dois identificadores diferentes.

Identifique palavras-chave

A linguagem C# reserva, para uso próprio, 77 identificadores, os quais não podem ser reutilizados para outros propósitos. Eles são denominados *palavras-chave*, e cada um tem um significado específico. Exemplos de palavras-chave são *class*, *namespace* e *using*. Você aprenderá o significado da maioria das palavras-chave do C# ao longo da leitura deste livro. A seguir está a lista de palavras-chave:

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe

catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

O C# também utiliza os identificadores a seguir. Eles não são específicos ao C#, ou seja, você pode utilizá-los como identificadores em seus próprios métodos, variáveis e classes, mas isso deve ser evitado sempre que possível.

add	get	remove
alias	global	select
ascending	group	set
async	into	value
await	join	var
descending	let	where
dynamic	orderby	yield
from	partial	

Variáveis

Variável é um local de armazenamento que contém um valor. Você pode considerar uma variável como uma caixa na memória do computador que contém informações temporárias. Você deve atribuir a cada variável em um programa um nome não ambíguo que a identifique de forma única no contexto em que é utilizada. Um nome de variável é utilizado para referenciar o valor que ela armazena. Por exemplo, se quiser armazenar o valor do custo de um item em uma loja, você deve criar uma variável chamada *custo* e armazenar o custo do item nela. Se você referenciar a variável *custo*, o valor recuperado será o custo do item armazenado anteriormente.

Nomeie variáveis

Adote uma convenção de nomes que torne claras as variáveis definidas. Isso é especialmente importante se você faz parte de uma equipe de projeto com vários desenvolvedores trabalhando em diferentes partes de um aplicativo; uma convenção de nomes consistente ajuda a evitar confusão e pode reduzir a extensão de erros. A lista a seguir contém algumas recomendações gerais:

- Não inicie um identificador com um sublinhado. Embora isso seja válido em C#, pode limitar a interoperabilidade de seu código com aplicativos compilados em outras linguagens, como Microsoft Visual Basic.
- Não crie identificadores cuja única diferença seja entre maiúsculas e minúsculas. Por exemplo, não crie uma variável chamada *minhaVariavel* e outra chamada *MinhaVariavel* para serem utilizadas ao mesmo tempo, porque será muito fácil confundi-las. Além disso, a definição de identificadores cuja única diferença seja a distinção entre maiúsculas e minúsculas pode limitar a reutilização das classes nos aplicativos desenvolvidos com outras linguagens que não diferem maiúsculas e minúsculas, como o Visual Basic.
- Comece o nome com uma letra minúscula.
- Em um identificador com várias palavras, comece a segunda palavra e as palavras subsequentes com uma letra maiúscula. Isso é chamado de notação *camel* ou *camelCase*.
- Não utilize notação húngara. (Se você for desenvolvedor de Microsoft Visual C++, provavelmente já conhece a notação húngara. Se não souber o que é isso, não se preocupe!)

Por exemplo, *placar*, *timeDeFutebol*, *_placar* e *TimeDeFutebol* são nomes de variáveis válidos, mas apenas os dois primeiros são recomendados.

Declare variáveis

As variáveis armazenam valores. O C# pode armazenar e processar muitos tipos diferentes de valores – inteiros, números de ponto flutuante e sequências de caractere (strings), entre outros. Ao declarar uma variável, você deve especificar o tipo de dado que ela armazenará.

Você declara o tipo e o nome de uma variável em uma instrução de declaração. Por exemplo, a instrução a seguir declara que a variável chamada *age* armazena valores *int* (inteiros). Como sempre, a instrução deve ser terminada com um ponto e vírgula.

```
int age;
```

O tipo de variável *int* é o nome de um dos tipos *primitivos* do C# – *inteiro*, que, como o nome já diz, é um número inteiro. (Você vai aprender sobre os diversos tipos de dados primitivos mais adiante neste capítulo.)



Nota Se você for programador de Visual Basic, deve observar que o C# não permite declarações implícitas de variável. Você deve declarar explicitamente todas as variáveis antes de utilizá-las.

Após ter declarado sua variável, você pode atribuir-lhe um valor. A instrução a seguir atribui o valor de 42 a *age*. Novamente, observe que o ponto e vírgula é obrigatório.

```
age = 42;
```

O sinal de igual (=) é o operador de *atribuição*, que atribui o valor que está a sua direita à variável que está a sua esquerda. Depois dessa atribuição, a variável *age* pode ser utilizada no seu código para referenciar o valor armazenado. A instrução a seguir escreve o valor da variável *age* (42) no console:

```
Console.WriteLine(age);
```



Dica Se você deixar o cursor do mouse sobre uma variável na janela Visual Studio 2013 Code and Text Editor, aparecerá uma dica de tela indicando o tipo da variável.

Tipos de dados primitivos

O C# tem vários tipos predefinidos denominados *tipos de dados primitivos*. A tabela a seguir lista os mais utilizados no C# e o intervalo de valores que podem ser armazenados neles.

Tipo de dado	Descrição	Tamanho (bits)	Intervalo	Exemplo de uso
int	Números inteiros	32	-2^{31} a $2^{31} - 1$	int count; count = 42;
long	Números inteiros (intervalo maior)	64	-2^{63} a $2^{63} - 1$	long wait; wait = 42L;
float	Números de ponto flutuante	32	$\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$	float away; away = 0.42F;
double	Números de ponto flutuante de precisão dupla (mais precisos)	64	$\pm 5.0 \times 10^{-324}$ a $\pm 1.7 \times 10^{308}$	double trouble; trouble = 0.42;
decimal	Valores monetários	128	28 valores significativos	decimal coin; coin = 0.42M;
string	Sequência de caracteres	16 bits por caractere	Não aplicável	string vest; vest = "forty two";
char	Caractere único	16	0 a $2^{16} - 1$	char grill; grill = 'x';
bool	Valor booleano	8	Verdadeiro ou falso	bool teeth; teeth = false;

Variáveis locais não atribuídas

Quando você declara uma variável, ela contém um valor aleatório até que lhe seja atribuído um valor. Esse comportamento era uma grande fonte de erros nos programas C e C++ que criavam uma variável e a utilizavam accidentalmente como fonte de informações antes de ela receber um valor. O C# não permite utilizar uma variável não atribuída. É necessário atribuir um valor a uma variável antes de usá-la; caso contrário, o programa não compilará. Essa exigência é chamada *regra de atribuição definitiva*.

Por exemplo, as instruções a seguir geram a mensagem de erro de tempo de compilação “Use of unassigned local variable ‘age’” porque a instrução `Console.WriteLine` tenta exibir o valor de uma variável não inicializada:

```
int age;  
Console.WriteLine(age); // compile-time error
```

Exiba valores de tipos de dados primitivos

No exercício a seguir, você vai utilizar um programa em C# chamado PrimitiveDataTypes para demonstrar como os vários tipos de dados primitivos funcionam.

Exiba os valores dos tipos de dados primitivos

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. No menu File, aponte para Open e então clique em Project/Solution.
A caixa de diálogo Open Project aparece.
3. Se estiver usando o Windows 8.1, vá até a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows 8.1\PrimitiveDataTypes na sua pasta Documentos. Se estiver usando o Windows 7 ou o Windows 8, vá até a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows 7\PrimitiveDataTypes na sua pasta Documentos.



Nota Para não ser repetitivo e economizar espaço, nos exercícios subsequentes vou simplesmente me referir aos caminhos de soluções usando uma frase da forma \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows X\PrimitiveDataTypes, onde X é 7 ou 8.1, dependendo do sistema operacional que você estiver usando.



Importante Se você estiver executando o Windows 8, lembre-se de usar os projetos e soluções para Windows 7 em todos os exercícios do livro.

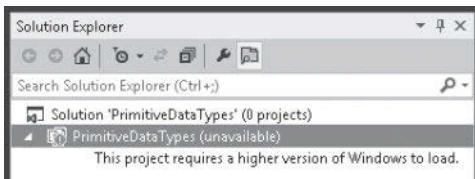
4. Selecione o arquivo de solução PrimitiveDataTypes e clique em Open.
A solução é carregada e o Solution Explorer exibe o projeto PrimitiveDataTypes.



Nota Os nomes dos arquivos de solução têm o sufixo .sln, como em PrimitiveDataTypes.sln. Uma solução pode conter um ou mais projetos. Os arquivos de projeto têm o sufixo .csproj. Se um projeto for aberto em vez de uma solução, o Visual Studio 2013 criará para ele, automaticamente, um novo arquivo de solução. Essa situação pode ser confusa, se você não estiver informado desse detalhe, pois pode resultar na geração acidental de várias soluções para o mesmo projeto.



Dica Certifique-se de abrir o arquivo de solução da pasta correta para seu sistema operacional. Se você tentar abrir uma solução para um aplicativo Windows Store com o Visual Studio 2013 no Windows 7 ou no Windows 8, o projeto não será carregado. Se você expandir o nó do projeto, o Solution Explorer marcará o projeto como indisponível e exibirá a mensagem "This project requires a higher version of Windows to load", como mostrado na imagem a seguir:

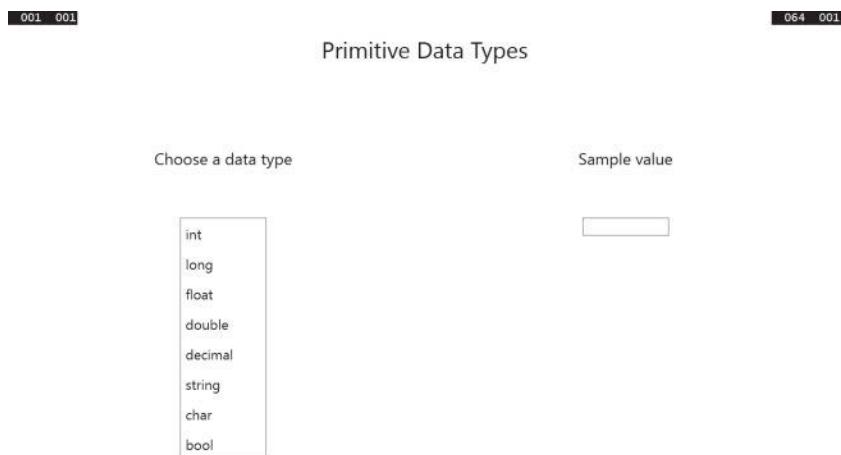


Se isso acontecer, feche a solução e abra a versão da pasta correta.

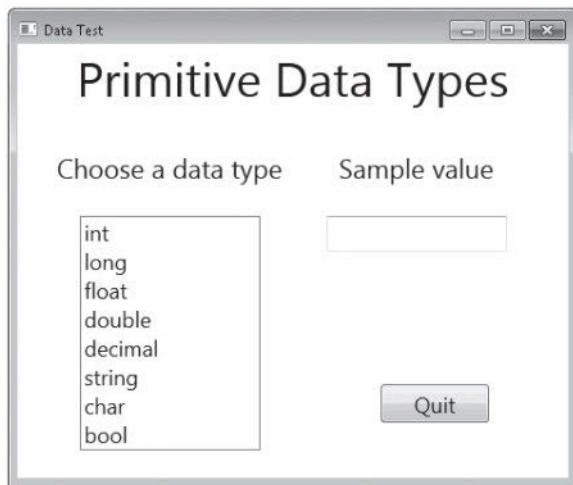
5. No menu Debug, clique em Start Debugging.

Talvez sejam exibidos alguns avisos no Visual Studio. Você pode ignorá-los sem perigo. (Você os corrigirá no próximo exercício.)

Se você está usando o Windows 8.1, a seguinte página será exibida:



Se você está usando o Windows 7 ou o Windows 8, a seguinte janela aparecerá:



6. Na lista Choose A Data Type, clique em *string*.
O valor “forty two” aparece na caixa Sample Value.
7. Novamente, na lista Choose A Data Type, clique no tipo *string*.
O valor “to do” (“a fazer”) aparece na caixa Sample Value, indicando que as instruções para exibir um valor *int* ainda precisam ser escritas.
8. Clique em cada tipo de dado na lista. Confirme que o código para os tipos *double* e *bool* ainda não está implementado.
9. Volte para o Visual Studio 2013 e, então, no menu Debug, clique em Stop Debugging.



Nota Lembre-se de que, no Windows 8.1, você pode pressionar a tecla Windows+B para voltar à área de trabalho do Windows que exibe o Visual Studio 2013.

Se estiver usando o Windows 7 ou o Windows 8, você também pode clicar em Quit para fechar a janela e interromper o programa.

Utilize tipos de dados primitivos no código

1. No Solution Explorer, expanda o projeto PrimitiveDataTypes (se ainda não estiver expandido) e, em seguida, clique duas vezes em MainWindow.xaml.



Nota Para manter as instruções do exercício simples, os formulários nas versões para Windows 8.1 e para Windows 7 do código têm os mesmos nomes.

O formulário do aplicativo aparece na janela Design View.

Sugestão Se sua tela não for grande o suficiente para exibir o formulário inteiro, você pode ampliar e reduzir a janela Design View usando Ctrl+Alt+= e Ctrl+Alt+- ou selecionando o tamanho na lista suspensa de zoom, no canto inferior esquerdo da janela Design View.

2. No painel XAML, role para baixo para localizar a marcação do controle *ListBox*. Esse controle exibe a lista de tipos de dados na parte esquerda do formulário e é parecida com isto (algumas das propriedades foram removidas desse texto):

```
<ListBox x:Name="type" ... SelectionChanged="typeSelectionChanged">
    <ListBoxItem>int</ListBoxItem>
    <ListBoxItem>long</ListBoxItem>
    <ListBoxItem>float</ListBoxItem>
    <ListBoxItem>double</ListBoxItem>
    <ListBoxItem>decimal</ListBoxItem>
    <ListBoxItem>string</ListBoxItem>
    <ListBoxItem>char</ListBoxItem>
    <ListBoxItem>bool</ListBoxItem>
</ListBox>
```

O controle *ListBox* exibe cada tipo de dado como um *ListBoxItem* separado. Quando o aplicativo está em execução, se um usuário clicar em um item na lista, o evento *SelectionChanged* ocorrerá (isso é um pouco como o evento *Clicked* que ocorre quando o usuário clica em um botão, o qual foi demonstrado no Capítulo 1). Você pode ver que, neste caso, o controle *ListBox* chama o método *typeSelectionChanged*. Esse método é definido no arquivo MainWindow.xaml.cs.

3. No menu View, clique em Code.

A janela Code and Text Editor abre, exibindo o arquivo MainWindow.xaml.cs.



Nota Lembre-se de que também é possível usar o Solution Explorer para acessar o código. Clique na seta à esquerda do arquivo MainWindow.xaml para expandir o nó e, então, clique duas vezes em MainWindow.xaml.cs.

4. Na janela Code and Text Editor, localize o método *typeSelectionChanged*.



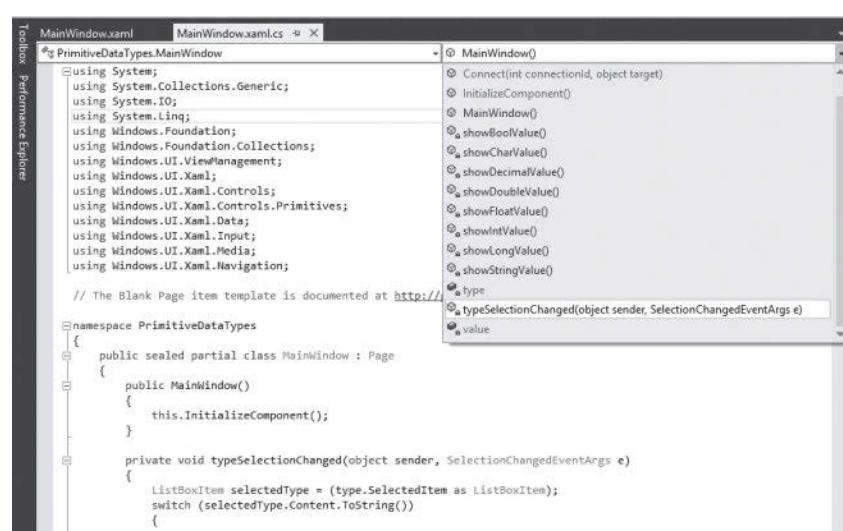
Dica Para localizar um item no seu projeto, no menu Edit, aponte para Find And Replace e clique em Quick Find. Um menu se abre no canto superior esquerdo da janela Code and Text Editor. Na caixa de texto desse menu de atalho, digite o nome do item que você está procurando e, então, clique em Find Next (o símbolo de seta para a direita ao lado da caixa de texto):



Por padrão, a pesquisa não diferencia maiúsculas de minúsculas. Se quiser fazer uma pesquisa que diferencie letras maiúsculas de minúsculas, clique na seta para baixo ao lado do texto a ser pesquisado, clique na seta suspensa à direita da caixa de texto no menu de atalho para exibir as opções adicionais e marque a caixa de seleção Match Case. Se tiver tempo, você pode experimentar as outras opções.

Você também pode pressionar Ctrl+F para exibir a caixa de diálogo Quick Find em vez de utilizar o menu Edit. Da mesma forma, você pode pressionar Ctrl+H para exibir a caixa de diálogo Quick Replace.

Como alternativa ao uso da funcionalidade Quick Find, você também pode localizar os métodos em uma classe utilizando a caixa de lista suspensa de membros da classe, posicionada acima da janela Code and Text Editor, à direita.



A lista suspensa de membros da classe exibe todos os métodos da classe e as variáveis e outros itens que a classe contém. (Você conhecerá mais detalhes sobre esses itens em capítulos posteriores.) Na lista suspensa, clique no método *typeSelectionChanged* e o cursor saltará imediatamente para o método *typeSelectionChanged* na classe.

Se você já programou em outra linguagem, provavelmente pode imaginar como o método *typeSelectionChanged* funciona; caso contrário, o Capítulo 4, “Instruções de decisão”, esclarecerá esse código. No momento, basta entender que, quando o usuário clica em um item no controle *ListBox*, o valor do item é passado para esse método, o qual então utiliza esse valor para determinar o que acontece em seguida. Por exemplo, se o usuário clica no valor *float*, esse método chama outro método, denominado *showFloatValue*.

5. Percorra o código e localize o método *showFloatValue*, que é como este:

```

private void showFloatValue()
{
    float floatVar;
    floatVar = 0.42F;
    value.Text = floatVar.ToString();
}

```

O corpo desse método contém três instruções. A primeira declara uma variável chamada *floatVar* do tipo *float*.

A segunda instrução atribui o valor 0.42F a *floatVar*.



Importante O *F* é um tipo de sufixo especificando que 0.42 deve ser tratado como um valor *float*. Se você esquecer o *F*, o valor 0.42 será tratado como um *double* e seu programa não compilará, porque um valor de um tipo não pode ser atribuído a uma variável de outro tipo sem se escrever código adicional – C# é muito rígido nesse aspecto.

A terceira instrução exibe o valor dessa variável na caixa de texto *Value* no formulário. Essa expressão exige sua atenção. Conforme ilustrado no Capítulo 1, a maneira de exibir um item em uma caixa de texto é configurando a propriedade *Text* (no Capítulo 1, você fez isso usando XAML). Também é possível executar essa tarefa por meio de programa, que é o que está acontecendo aqui. Observe que a propriedade de um objeto é acessada utilizando a mesma notação de ponto que vimos para executar um método. (Lembra-se de *Console.WriteLine* do Capítulo 1?) Além disso, os dados adicionados à propriedade *Text* devem ser uma string e não um número. Se você tentar atribuir um número à propriedade *Text*, seu programa não compilará. Felizmente, o .NET Framework dá alguma ajuda na forma do método *ToString*.

Cada tipo de dado no .NET Framework tem um método *ToString*. A finalidade de *ToString* é converter um objeto na sua representação de string. O método *showFloatValue* utiliza o método *ToString* do objeto *floatVar* da variável *float* para gerar uma versão de string do valor dessa variável. Essa string pode então ser atribuída com segurança à propriedade *Text* da caixa de texto *Value*. Ao criar seus próprios tipos de dados e classes, você pode definir uma implementação própria do método *ToString* para especificar a maneira como sua classe deve ser representada como uma string. Veja como criar suas próprias classes no Capítulo 7, “Criação e gerenciamento de classes e objetos”.

6. Na janela Code and Text Editor, localize o método *showIntValue*:

```
private void showIntValue()
{
    value.Text = "to do";
}
```

O método *showIntValue* é chamado quando você clica no tipo *int* na caixa de listagem.

7. Digite as duas instruções a seguir no início do método *showIntValue*, em uma nova linha depois da chave de abertura, como mostrado em negrito no código a seguir:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = "to do";
}
```

A primeira instrução cria uma variável chamada *intVar* que pode conter um valor *int*. A segunda instrução atribui o valor a essa variável.

8. A instrução original nesse método altera a string “*to do*” para *intVar.ToString()*,

O método agora deve estar exatamente como este:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = intVar.ToString();
}
```

- 9.** No menu Debug, clique em Start Debugging.

O formulário aparece novamente.

- 10.** Na lista Choose A Data Type, selecione o tipo *int*. Confirme se o valor 42 está sendo exibido na caixa de texto Sample Value.

- 11.** Volte para o Visual Studio e, então, no menu Debug, clique em Stop Debugging.

- 12.** Na janela Code and Text Editor, localize o método *showDoubleValue*.

- 13.** Edite o método *showDoubleValue* exatamente como mostrado em negrito no seguinte código:

```
private void showDoubleValue()
{
    double doubleVar;
    doubleVar = 0.42;
    value.Text = doubleVar.ToString();
}
```

Esse código é semelhante ao método *showIntValue*, exceto que cria uma variável chamada *doubleVar* que contém valores *double* e recebe o valor 0.42.

- 14.** Na janela Code and Text Editor, localize o método *showBoolValue*.

- 15.** Edite o método *showBoolValue* exatamente assim:

```
private void showBoolValue()
{
    bool boolVar;
    boolVar = false;
    value.Text = boolVar.ToString();
}
```

Novamente, esse código é semelhante aos exemplos anteriores, exceto que *boolVar* só pode conter um valor booleano, *verdadeiro* ou *falso*. Nesse caso, o valor atribuído é *falso*.

- 16.** No menu Debug, clique em Start Debugging.

- 17.** Na lista Choose A Data Type, selecione os tipos *int*, *double* e *bool*. Em cada um dos casos, verifique se o valor correto é exibido na caixa de texto Sample Value.

- 18.** Volte para o Visual Studio e, então, no menu Debug, clique em Stop Debugging.

Operadores aritméticos

O C# suporta as operações aritméticas que você aprendeu na escola: o sinal de mais (+) para adição, o sinal de menos (-) para subtração, o asterisco (*) para multiplicação e a barra (/) para divisão. Os símbolos +, -, * e / são denominados *operadores* porque “operam” em valores para criar novos valores. No exemplo abaixo, a variável *moneyPaidToConsultant* termina armazenando o produto de 750 (a diária) e de 20 (o número de dias que o consultor trabalhou):

```
long moneyPaidToConsultant;
moneyPaidToConsultant = 750 * 20;
```



Nota Os valores nos quais um operador efetua sua função chamam-se *operандos*. Na expressão 750 * 20, o * é o operador e 750 e 20 são os operandos.

Operadores e tipos

Nem todos os operadores são aplicáveis a todos os tipos de dados. Aqueles que podem ser utilizados em um valor dependem do tipo do valor. Por exemplo, você pode usar todos os operadores aritméticos em valores de tipo *char*, *int*, *long*, *float*, *double* ou *decimal*. Contudo, com exceção do operador de adição, +, os operadores aritméticos não podem ser usados em valores de tipo *string* e nenhum deles pode ser usado com valores de tipo *bool*. Portanto, a instrução a seguir não é permitida porque o tipo *string* não suporta o operador de subtração (não há sentido em subtrair uma *string* de outra):

```
// compile-time error
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

Contudo, você pode utilizar o operador + para concatenar valores de *string*. É preciso ter bastante cuidado, pois isso pode produzir resultados inesperados. Por exemplo, a seguinte instrução escreve “431” (e não “44”) no console:

```
Console.WriteLine("43" + "1");
```



Dica O .NET Framework fornece um método chamado *Int32.Parse* que pode ser utilizado para converter um valor de *string* em um inteiro, se você precisar efetuar cálculos aritméticos em valores armazenados em *strings*.

Você deve estar ciente de que o tipo de resultado de uma operação aritmética depende do tipo dos operandos utilizados. Por exemplo, o valor da expressão 5.0/2.0 é 2.5; o tipo dos dois operandos é *double*, de modo que o tipo do resultado também é *double*. (No C#, os números literais com pontos decimais são sempre *double*, não *float*, para manter o máximo de precisão possível.) Mas o valor da expressão 5/2 é 2. Nesse caso, o tipo de ambos os operandos é *int*; assim, o tipo do resultado também é *int*. O C# sempre arredonda para zero em casos assim. A situação se torna um pouco mais complicada se você misturar os tipos de operandos. Por exemplo, a expressão 5/2.0 consiste em um *int* e um *double*. O compilador do C# detecta a incompatibili-

dade e gera um código que converte o *int* em *double* antes de executar a operação. O resultado da operação é, portanto, um *double* (2.5). Embora funcione, essa prática é considerada ruim.

O C# também suporta um operador aritmético menos conhecido: o operador *resto* ou *módulo*, que é representado pelo sinal de porcentagem (%). O resultado de $x \% y$ é o resto da divisão do valor x pelo valor y . Assim, por exemplo, $9 \% 2$ é 1, porque 9 dividido por 2 é 4, resto 1.



Nota Se você já conhece C ou C++, sabe que nessas linguagens não é possível utilizar o operador resto nos valores *float* ou *double*. Entretanto, C# afrouxa essa regra. O operador resto é válido para todos os tipos numéricos, e o resultado não é necessariamente um inteiro. Por exemplo, o resultado da expressão $7.0 \% 2.4$ é 2.2.

Tipos numéricos e valores infinitos

Há uma ou duas outras características dos números em C# que você precisa conhecer. Por exemplo, o resultado da divisão de qualquer número por zero é infinito, estando fora do intervalo dos tipos *int*, *long* e dos tipos *decimais*; consequentemente, avaliar uma expressão como $5/0$ resulta em um erro. Mas os tipos *double* e *float* têm um valor especial que pode representar valores infinitos, e o valor da expressão $5.0/0.0$ é *Infinity*. A única exceção a essa regra é o valor da expressão $0.0/0.0$. Em geral, se dividir zero por qualquer número, o resultado será zero, mas se dividir algo por zero o resultado será um número infinito. A expressão $0.0/0.0$ resulta em um paradoxo – o valor deve ser zero e infinito ao mesmo tempo. O C# tem outro valor especial para essa situação, chamado *NaN*, que significa “not a number” (não é um número). Portanto, se $0.0/0.0$ for avaliada, o resultado será *NaN*.

NaN e *Infinity* são propagados pelas expressões. Se $10 + NaN$ for avaliado, o resultado será *NaN*, e se $10 + Infinity$ for avaliado, o resultado será *Infinity*. A única exceção a essa regra é quando *Infinity* é multiplicado por 0. O valor da expressão $Infinity * 0$ é 0, embora o valor de $NaN * 0$ seja *NaN*.

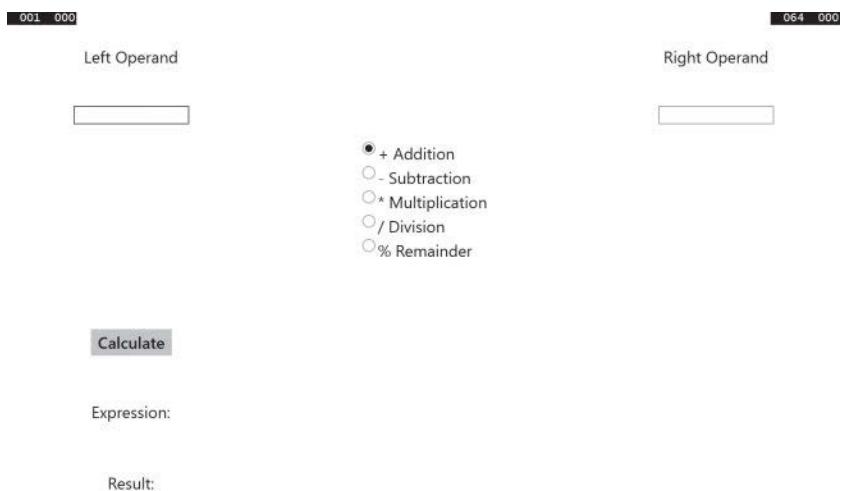
Examine operadores aritméticos

O exercício a seguir demonstra como utilizar os operadores aritméticos em valores *int*.

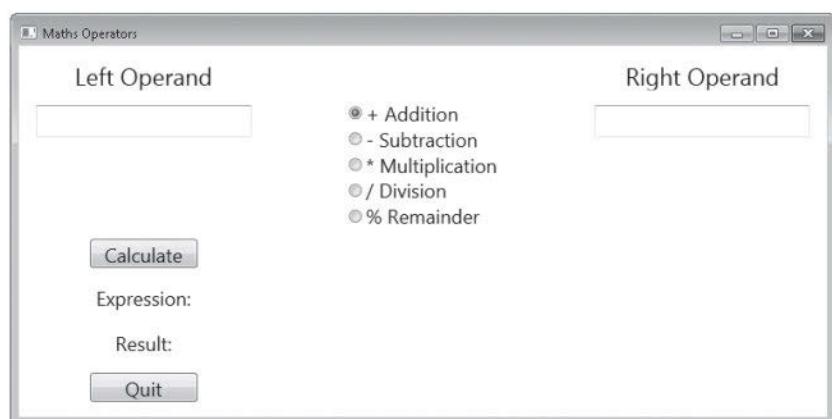
Execute o projeto MathsOperators

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra o projeto MathsOperators, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows X\MathsOperators na sua pasta Documentos.
3. No menu Debug, clique em Start Debugging.

Se você está usando o Windows 8.1, a seguinte página aparecerá:



Se está usando o Windows 7 ou o Windows 8, o seguinte formulário aparecerá:



4. Na caixa Left Operand, digite **54**.
5. Na caixa Right Operand, digite **13**.

Agora você pode aplicar qualquer um dos operadores aos valores das caixas de texto.

6. Clique na opção – Subtraction e, em seguida, clique em Calculate.

O texto na caixa Expression muda para $54 - 13$, mas o valor 0 aparece na caixa Result; claramente, isso está errado.

- Clique na opção / Division e, em seguida, clique em Calculate.

O texto na caixa Expression muda para $54/13$ e, novamente, o número 0 aparece na caixa Result.

- Clique no botão % Remainder e então em Calculate.

O texto na caixa Expression muda para $54 \% 13$; porém, mais uma vez, o número 0 aparece na caixa Result. Teste as outras combinações de números e operadores; você vai descobrir que atualmente todas produzem o valor 0.



Nota Se você digitar um valor não inteiro em uma das caixas de operando, o aplicativo detectará um erro e exibirá a mensagem “Input string was not in a correct format”. Você aprenderá mais sobre como capturar e tratar de erros e exceções no Capítulo 6, “Gerenciamento de erros e exceções”.

- Quando tiver terminado, volte ao Visual Studio e, no menu Debug, clique em Stop Debugging (se estiver usando o Windows 7 ou o Windows 8, também pode clicar em Quit no formulário MathsOperators).

Conforme você pode ter adivinhado, nenhum dos cálculos está atualmente implementado pelo aplicativo MathsOperators. No próximo exercício, você vai corrigir isso.

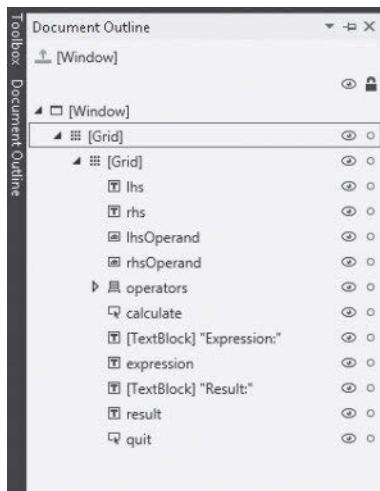
Efetue cálculos no aplicativo MathsOperators

- Exiba o formulário MainWindow.xaml na janela Design View. (No Solution Explorer, no projeto MathsOperators, clique duas vezes no arquivo MainWindow.xaml.)
- No menu View, aponte para Other Windows e clique em Document Outline.

A janela Document Outline é exibida, mostrando os nomes e tipos de controles do formulário. A janela Document Outline é uma maneira simples de localizar e selecionar controles em um formulário complexo. Os controles são organizados hierarquicamente, começando pela *página* (Windows 8.1) ou *janela* (Windows 7 ou Windows 8) que constitui o formulário. Como mencionado no Capítulo 1, uma página de aplicativo Windows Store ou um formulário WPF contém um controle *Grid*, e os outros controles são colocados dentro desse *Grid*. Se você expandir o nó *Grid* na janela Document Outline, os outros controles aparecerão, começando com outro *Grid* (o *Grid* externo atua como uma moldura e o interno contém os controles que você vê no formulário). Se você expandir o *Grid* interno, poderá ver cada um dos controles existentes no formulário.



Nota Na versão para Windows 8.1 do aplicativo, o controle *Grid* externo está envolto em um controle *ScrollViewer*. Esse controle fornece uma barra de rolagem horizontal com a qual o usuário pode rolar a janela que exibe o aplicativo, caso redimensione a janela de exibição.



Se você clicar em qualquer um desses controles, o elemento correspondente será realçado na janela Design View. Do mesmo modo, se você selecionar um controle na janela Design View, o controle correspondente será selecionado na janela Document Outline (para ver isso em ação, fixe a janela Document Outline, cancelando a seleção do botão Auto Hide no canto superior direito da janela Document Outline).

3. No formulário, clique nos dois controles *TextBox* em que o usuário digita os números. Na janela Document Outline, verifique que seus nomes são *LhsOperand* e *rhsOperand*.

Quando o formulário é executado, a propriedade *Text* de cada um desses controles armazena os valores digitados pelo usuário.

4. Na parte inferior do formulário, verifique se o controle *TextBlock* utilizado para exibir a expressão que está sendo avaliada tem o nome *expression* e se o controle *TextBlock* utilizado para exibir o resultado do cálculo tem o nome *result*.

5. Feche a janela Document Outline.

6. No menu View, clique em Code para exibir o código do arquivo MainWindow.xaml.cs na janela Code and Text Editor.

7. Na janela Code and Text Editor, localize o método *addValues*. Ele se parece com este:

```
private void addValues()
{
    int lhs = int.Parse(LhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: somar rhs e lhs e armazenar o resultado em outcome

    expression.Text = LhsOperand.Text + " + " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

A primeira instrução nesse método declara uma variável *int* chamada *lhs* e a inicializa com o inteiro que corresponde ao valor digitado pelo usuário na caixa *LhsOperand*. Lembre-se de que a propriedade *Text* de um controle *TextBox* contém uma string, mas *lhs* é *int*; portanto, é preciso converter essa string em um inteiro antes que ela seja atribuída a *lhs*. O tipo de dado *int* fornece o método *int.Parse*, que faz precisamente isso.

A segunda instrução declara uma variável *int* chamada *rhs* e a inicializa com o valor da caixa *rhsOperand* depois de convertê-lo em um *int*.

A terceira instrução declara uma variável *int* chamada *outcome*.

Em seguida, aparece um comentário dizendo que você precisa somar *rhs* a *lhs* e armazenar o resultado em *outcome*. Esse é o código ausente que precisa ser implementado, o que você vai fazer no próximo passo.

A quinta instrução concatena três strings que indicam o cálculo que está sendo efetuado (utilizando o operador de adição, *+*) e atribui o resultado à propriedade *expression.Text*. Isso faz a string aparecer na caixa *Expression* no formulário.

A última instrução exibe o resultado do cálculo atribuindo-o à propriedade *Text* da caixa *Result*. Lembre-se de que a propriedade *Text* é uma string e de que o resultado do cálculo é um *int*; portanto, você precisa converter o *int* em uma string antes de atribuí-lo à propriedade *Text*. Lembre-se de que é isso que o método *ToString* do tipo *int* faz.

- 8.** Abaixo do comentário no meio do método *addValues*, adicione a instrução a seguir (mostrada em negrito):

```
private void addValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: somar rhs e lhs e armazenar o resultado em outcome
    outcome = lhs + rhs;
    expression.Text = lhsOperand.Text + " + " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

Essa instrução avalia a expressão *lhs + rhs* e armazena o resultado em *outcome*.

- 9.** Examine o método *subtractValues*. Você verá que ele segue um padrão semelhante e é preciso adicionar a instrução para calcular o resultado da subtração de *lhs* por *rhs*, armazenando-o em *outcome*. Adicione a esse método a seguinte instrução (em negrito):

```
private void subtractValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: subtrair rhs de lhs e armazenar o resultado em outcome
    outcome = lhs - rhs;
    expression.Text = lhsOperand.Text + " - " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

- 10.** Examine os métodos *multiplyValues*, *divideValues* e *remainderValues*. Novamente, todos eles têm a instrução crucial que efetua o cálculo ausente especificado. Adicione as instruções apropriadas a esses métodos (mostradas em negrito).

```
private void multiplyValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: multiplicar lhs por rhs e armazenar o resultado em outcome
outcome = lhs * rhs;
    expression.Text = lhsOperand.Text + " * " + rhsOperand.Text;
    result.Text = outcome.ToString();
}

private void divideValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: dividir lhs por rhs e armazenar o resultado em outcome
outcome = lhs / rhs;
    expression.Text = lhsOperand.Text + " / " + rhsOperand.Text;
    result.Text = outcome.ToString();
}

private void remainderValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: calcular o resto após a divisão de lhs por rhs e armazenar o
    // resultado em outcome
outcome = lhs % rhs;
    expression.Text = lhsOperand.Text + " % " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

Teste o aplicativo MathsOperators

1. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
2. Digite **54** na caixa Left Operand, digite **13** na caixa Right Operand, clique no botão + Addition e então clique em Calculate.
O valor 67 deve aparecer na caixa Result.
3. Clique na opção – Subtraction e, em seguida, clique em Calculate. Verifique que o resultado agora é 41.
4. Clique na opção * Multiplication e, em seguida, clique em Calculate. Verifique que o resultado agora é 702.
5. Clique na opção / Division e, em seguida, clique em Calculate. Verifique que o resultado agora é 4.

Em uma situação real, 54/13 dá uma dízima periódica (4,153846...); no entanto, aqui o C# está efetuando uma divisão de inteiros. Quando um inteiro é dividido por outro inteiro, a resposta que você obtém é um inteiro, como explicado anteriormente.

- 6.** Clique na opção % Remainder e, em seguida, clique em Calculate. Verifique que o resultado agora é 2.

Ao se lidar com inteiros, o resto, após a divisão de 54 por 13, é 2; $(54 - ((54/13) * 13))$ é 2. Isso acontece porque, em cada estágio, o cálculo arredonda para um inteiro abaixo. (Meu professor de matemática na escola secundária ficaria horrorizado se soubesse que $(54/13) * 13$ não é igual a 54!)

- 7.** Volte ao Visual Studio e interrompa a depuração (ou clique em Quit, se estiver usando o Windows 7 ou o Windows 8).

Controle a precedência

A *precedência* (ou prioridade) controla a ordem em que os operadores da expressão são avaliados. Considere a expressão a seguir, que utiliza os operadores + e *:

$2 + 3 * 4$

Essa expressão é potencialmente ambígua: qual deve ser efetuada primeiro, a adição ou a multiplicação? A ordem das operações importa porque muda o resultado:

- Se efetuar primeiro a adição e depois a multiplicação, o resultado da adição ($2 + 3$) formará o operando esquerdo do operador *, e o resultado de toda a expressão será $5 * 4 = 20$.
- Se efetuar primeiro a multiplicação e depois a adição, o resultado da multiplicação ($3 * 4$) formará o operando direito do operador +, e o resultado da expressão inteira será $2 + 12 = 14$.

No C#, os operadores multiplicativos (*, / e %) têm precedência sobre os operadores aditivos (+ e -), portanto, em expressões como $2 + 3 * 4$, a multiplicação é efetuada primeiro, seguida pela adição. Portanto, a resposta para $2 + 3 * 4$ é 14.

Parênteses podem ser utilizados para anular a precedência e forçar os operandos a vincular-se aos operadores de maneira diferente. Por exemplo, na expressão a seguir, os parênteses forçam o 2 e o 3 a se vincular ao operador + (produzindo o valor 5), e o resultado dessa soma é o operando esquerdo do operador *, produzindo o valor 20:

$(2 + 3) * 4$



Nota O termo *parênteses* refere-se a (). O termo *chaves* refere-se a {}. O termo *colchetes* refere-se a [].

Utilize a associatividade para avaliar expressões

A precedência dos operadores é apenas metade da história. O que acontece quando uma expressão contém operadores diferentes que têm a mesma precedência? É aí que a *associatividade* se torna importante. Associatividade é a direção (esquerda ou direita) em que os operandos de um operador são avaliados. Considere a expressão a seguir que utiliza os operadores / e *:

```
4 / 2 * 6
```

À primeira vista, essa expressão é potencialmente ambígua. Qual deve ser efetuada primeiro, a divisão ou a multiplicação? A precedência dos dois operadores é a mesma (são ambos multiplicativos), mas a ordem em que são aplicados na expressão é importante, pois dois resultados diferentes podem ser obtidos:

- Se efetuar primeiro a divisão, o resultado da divisão (4/2) formará o operando esquerdo do * operador, e o resultado da expressão inteira será (4/2) * 6 ou 12.
- Se efetuar primeiro a multiplicação, o resultado da multiplicação (2 * 6) formará o operando direito do operador /, e o resultado da expressão inteira será 4 /(2 * 6) ou 4/12.

Nesse caso, a associatividade dos operadores determina como a expressão é avaliada. Ambos os operadores, * e /, associam-se à esquerda, assim, os operandos são calculados da esquerda para a direita. Nesse caso, 4/2 será avaliado antes da multiplicação por 6, que resulta em 12.

A associatividade e o operador de atribuição

No C#, o sinal de igual (=) é um operador. Todos os operadores retornam um valor com base nos seus operandos. O operador de atribuição = não é diferente. Ele aceita dois operandos: o operando à direita é avaliado e então é armazenado no operando à esquerda. O valor do operador de atribuição é o valor que foi atribuído para o operando esquerdo. Por exemplo, na seguinte instrução de atribuição, o valor retornado pelo operador de atribuição é 10, que também é o valor atribuído à variável *myInt*:

```
int myInt;
myInt = 10; // o valor da expressão de atribuição é 10
```

Você pode estar pensando que tudo isso é interessante e esotérico, mas e daí? Bem, como o operador de atribuição retorna um valor, você pode utilizar esse mesmo valor em outra ocorrência da instrução de atribuição, desta maneira:

```
int myInt;
int myInt2;
myInt2 = myInt = 10;
```

O valor atribuído à variável *myInt2* é o valor que foi atribuído a *myInt*. A instrução de atribuição atribui o mesmo valor a ambas as variáveis. Essa técnica é útil se você quer inicializar diferentes variáveis com o mesmo valor. Torna-se claro para qualquer leitor do seu código que todas as variáveis devem ter o mesmo valor:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

A partir dessa discussão, você provavelmente pode deduzir que o operador de atribuição é associado da direita para a esquerda. A atribuição mais à direita ocorre primeiro, e o valor atribuído se propaga pelas variáveis da direita para a esquerda. Se uma das variáveis já tivesse um valor, esse seria sobreescrito pelo valor que está sendo atribuído.

Entretanto, trate essa construção com cautela. Um erro frequentemente cometido por novos programadores de C# é tentar combinar esse uso do operador de atribuição com declarações de variáveis. Por exemplo, você poderia esperar que o código a seguir criasse e inicializasse três variáveis com o mesmo valor (10):

```
int myInt, myInt2, myInt3 = 10;
```

Esse é um código válido do C# (porque é compilado). Ele declara as variáveis *myInt*, *myInt2* e *myInt3*, e inicializa *myInt3* com o valor 10. Contudo, ele não inicializa *myInt* nem *myInt2*. Se você tentar utilizar *myInt* ou *myInt2* em uma expressão como

```
myInt3 = myInt / myInt2;
```

o compilador gerará os seguintes erros:

```
Use of unassigned local variable 'myInt'  
Use of unassigned local variable 'myInt2'
```

Incremente e decremente variáveis

Se quiser somar 1 a uma variável, pode utilizar o operador **+**, como demonstrado aqui:

```
count = count + 1;
```

Mas adicionar 1 a uma variável é tão comum que o C# fornece um operador somente para essa finalidade: o operador **++**. Para incrementar a variável *count* por 1, você pode escrever a instrução a seguir:

```
count++;
```

Da mesma forma, o C# fornece o operador **--**, que pode ser utilizado para subtrair 1 de uma variável, desta maneira:

```
count--;
```

Os operadores **++** e **--** são *unários*, ou seja, eles têm um único operando. Eles compartilham a mesma precedência e ambos são associativos à esquerda.

Prefixo e sufixo

Os operadores de incremento (**++**) e decremento (**--**) fogem do comum, porque você pode colocá-los antes ou depois da variável. Quando o símbolo do operador é colocado antes da variável, chamamos de forma *prefixada* do operador, e quando colocado depois, chamamos de forma *pós-fixada* ou sufixada do operador. Eis alguns exemplos:

```
count++; // incremento pós-fixado
++count; // incremento prefixado
count--; // decremento pós-fixado
--count; // decremento prefixado
```

Utilizar a forma prefixada ou sufixada do operador `++` ou `--` não faz a menor diferença para a variável que está sendo incrementada ou decrementada. Por exemplo, se você escreve `count++`, o valor de `count` aumenta por 1, e se escreve `++count`, o valor de `count` também aumenta por 1. Sabendo isso, você provavelmente poderia perguntar por que há duas maneiras de escrever a mesma coisa. Para entender a resposta, você precisa lembrar que `++` e `--` são operadores e que todos os operadores são utilizados para avaliar uma expressão que tem um valor. O valor retornado por `count++` é o valor de `count` antes do incremento, enquanto o valor retornado por `++count` é o valor de `count` depois que o incremento ocorre. Veja um exemplo:

```
int x;
x = 42;
Console.WriteLine(x++); // x agora é 43, 42 escrito
x = 42;
Console.WriteLine(++x); // x agora é 43, 43 escrito
```

A maneira de lembrar o que cada operando faz é examinar a ordem dos elementos (o operando e o operador) em uma expressão prefixada ou sufixada. Na expressão `x++`, a variável `x` ocorre primeiro; portanto, seu valor é utilizado como o valor da expressão antes de `x` ser incrementada. Na expressão `++x`, o operador ocorre primeiro; portanto, sua operação é executada antes de o valor de `x` ser calculado como o resultado.

Esses operadores são mais utilizados nas instruções `while` e `do`, que serão apresentadas no Capítulo 5, “Atribuição composta e instruções de iteração”. Caso esteja utilizando os operadores de incremento e decremento isoladamente, fique com a forma pós-fixada e seja coerente.

Declare variáveis locais implicitamente tipadas

Vimos anteriormente neste capítulo que uma variável é declarada especificando um tipo de dado e um identificador, assim:

```
int myInt;
```

Também foi mencionado que um valor deve ser atribuído a uma variável antes de se tentar utilizá-la. Você pode declarar e inicializar uma variável na mesma instrução, como ilustrado a seguir:

```
int myInt = 99;
```

Ou assim, supondo que `myOtherInt` seja uma variável do tipo inteiro já inicializada:

```
int myInt = myOtherInt * 99;
```

Agora, lembre-se de que o valor atribuído a uma variável deve ser do mesmo tipo da variável. Por exemplo, você pode atribuir um valor `int` apenas a uma variável `int`. O compilador C# pode calcular rapidamente o tipo de uma expressão utilizada

para inicializar uma variável e indicar se não corresponde ao tipo da variável. Também é possível instruir o compilador C# a deduzir o tipo de uma variável a partir de uma expressão e utilizá-lo ao declarar a variável com a palavra-chave *var* no lugar do tipo, como demonstrado aqui:

```
var myVariable = 99;
var myOtherVariable = "Hello";
```

As variáveis *myVariable* e *myOtherVariable* são conhecidas como variáveis *implicitamente tipadas*. A palavra-chave *var* faz o compilador deduzir o tipo das variáveis a partir dos tipos das expressões utilizadas para inicializá-las. Nesses exemplos, *myVariable* é um *int* e *myOtherVariable* é uma *string*. Contudo, é importante entender que essa é uma conveniência apenas para declarar variáveis e que, depois que uma variável foi declarada, você só pode atribuir valores do tipo inferido a ela – valores *float*, *double* ou *string* não podem ser atribuídos a *myVariable* em um ponto posterior no seu programa, por exemplo. Você também deve entender que só é possível utilizar a palavra-chave *var* quando fornecer uma expressão para inicializar uma variável. A declaração a seguir é ilegal e causará um erro de compilação:

```
var yetAnotherVariable; // Erro – o compilador não pode inferir o tipo
```



Importante Se você já programou em Visual Basic, talvez conheça o tipo *Variant*, que pode ser utilizado para armazenar qualquer tipo de valor em uma variável. É importante frisar que você deve esquecer tudo que já aprendeu sobre variáveis *Variant* ao programar em Visual Basic. Embora as palavras-chave pareçam semelhantes, *var* e *Variant* são totalmente diferentes. Ao declarar uma variável em C# utilizando a palavra-chave *var*, o tipo de valor que você atribui à variável *não pode mudar* em relação àquele utilizado para inicializar a variável.

Se você for purista, provavelmente está cerrando os dentes agora e se perguntando por que cargas d'água os projetistas de uma linguagem elegante como C# permitem a infiltração de um componente como *var*. Afinal, parece uma desculpa para a extrema preguiça dos programadores e pode tornar mais difícil entender o que um programa está fazendo ou rastrear bugs (e pode até mesmo introduzir novos bugs em seu código facilmente). Mas confie no fato de que *var* tem um lugar válido no C#, como veremos ao trabalhar nos capítulos a seguir. Por enquanto, vamos nos ater ao uso de variáveis explicitamente tipadas, exceto quando a tipagem implícita tornar-se uma necessidade.

Resumo

Neste capítulo, você viu como criar e utilizar variáveis e aprendeu sobre alguns tipos de dados comuns, disponíveis para as variáveis no C#. Você conheceu os identificadores e, além disso, usou alguns operadores para construir expressões e aprendeu que a precedência e a associatividade dos operadores determinam o modo como as expressões são avaliadas.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 3.

- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar uma variável	Escreva o nome do tipo de dado, seguido pelo nome da variável, seguido por um ponto e vírgula. Por exemplo: <code>int outcome;</code>
Declarar uma variável e atribuir a ela um valor inicial	Escreva o nome do tipo de dado, seguido pelo nome da variável, seguido pelo operador de atribuição e o valor inicial. Finalize com um ponto e vírgula. Por exemplo: <code>int outcome = 99;</code>
Alterar o valor de uma variável	Escreva o nome da variável à esquerda, seguido pelo operador de atribuição, seguido pela expressão que calcula o novo valor, seguido por um ponto e vírgula. Por exemplo: <code>outcome = 42;</code>
Gerar uma representação de string do valor de uma variável	Chame o método <code>ToString</code> da variável. Por exemplo: <code>int intVar = 42; string stringVar = intVar.ToString();</code>
Converter uma string em um int	Chame o método <code>System.Int32.Parse</code> . Por exemplo: <code>string stringVar = "42"; int intVar = System.Int32.Parse(stringVar);</code>
Anular a precedência de um operador	Utilize parênteses na expressão para explicitar a ordem de avaliação. Por exemplo: <code>(3 + 4) * 5</code>
Atribuir o mesmo valor a diversas variáveis	Use uma instrução de atribuição que liste todas as variáveis. Por exemplo: <code>myInt4 = myInt3 = myInt2 = myInt = 10;</code>
Incrementar ou decrementar uma variável	Utilize o operador <code>++</code> ou <code>--</code> . Por exemplo: <code>count++;</code>

CAPÍTULO 3

Como escrever métodos e aplicar escopo

Neste capítulo, você vai aprender a:

- Declarar e chamar métodos.
- Passar informações para um método.
- Retornar as informações de um método.
- Definir o escopo de classe e local.
- Utilizar o depurador integrado para entrar e sair dos métodos à medida que eles são executados.

No Capítulo 2, “Variáveis, operadores e expressões”, você aprendeu como declarar variáveis e também como criar expressões utilizando operadores. Viu ainda como a precedência e a associatividade controlam a maneira pela qual são avaliadas as expressões com múltiplos operadores. Os métodos são o tema deste capítulo. Você aprenderá como declarar e chamar métodos e também como utilizar os argumentos e parâmetros para transferir informações para um método, e o modo como deve retorná-las com o emprego de uma instrução *return*. Você também saberá, neste capítulo, como entrar e sair dos métodos usando o depurador integrado do Microsoft Visual Studio 2013. Quando for preciso rastrear a execução dos seus métodos, essas informações serão extremamente úteis caso eles não funcionem como o esperado. Por fim, este capítulo ensina a declarar métodos que aceitam parâmetros opcionais e a chamar métodos por meio de argumentos nomeados.

Crie métodos

Um *método* é uma sequência nomeada de instruções. Se você já programou com uma linguagem como C, C++ ou Microsoft Visual Basic, perceberá que um método é muito semelhante a uma função ou a uma sub-rotina. Um método tem um nome e um corpo. O nome do método deve ser um identificador significativo que indique sua finalidade geral (*calcularImpostoDeRenda*, por exemplo). O corpo do método contém as instruções reais a serem executadas quando o método for chamado. Além disso, os métodos podem receber alguns dados para serem processados e retornar informações, que normalmente são o resultado do processamento. Os métodos caracterizam-se como um mecanismo poderoso e fundamental.

Declare um método

A sintaxe para declarar um método C# é:

```
tipoDeRetorno nomeDoMétodo ( listaDeParâmetros )
{
    // as instruções do corpo do método ficam aqui
}
```

Elementos que constituem uma declaração:

- O *tipoDeRetorno* é o nome de um tipo e especifica a informação que o método retorna como resultado do seu processamento. Ele pode ser qualquer tipo, como *int* ou *string*. Se você está escrevendo um método que não retorna um valor, deve utilizar a palavra-chave *void* no lugar do tipo de retorno.
- O *nomeDoMétodo* é o nome utilizado para chamar o método. Os nomes de método seguem as mesmas regras de identificador dos nomes de variáveis. Por exemplo, *addValues* é um nome de método válido, mas *add\$Values* não é. Por enquanto, você deve seguir a convenção camelCase para nomes de métodos; por exemplo, *exibirClientes*.
- A *listaDeParâmetros* é opcional e descreve os tipos e nomes das informações que você pode passar para o método processar. Escreva os parâmetros entre parênteses de abertura e fechamento, (), como se estivesse declarando variáveis, com o nome do tipo seguido pelo nome do parâmetro. Se o método que estiver escrevendo tiver dois ou mais parâmetros, separe-os com vírgulas.
- As instruções do corpo do método são as linhas de código executadas quando o método é chamado. Elas ficam entre chaves de abertura e de fechamento, {}.



Importante Se você programa em C, C++ e Microsoft Visual Basic, deve notar que o C# não suporta métodos globais. Você deve escrever todos os seus métodos dentro de uma classe; caso contrário, seu código não compilará.

Aqui está a definição de um método chamado *addValues* que retorna um resultado *int* e tem dois parâmetros *int* chamados *leftHandSide* e *rightHandSide*:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
    // as instruções do corpo do método ficam aqui
    // ...
}
```



Nota Você deve especificar explicitamente os tipos de quaisquer parâmetros e o tipo de retorno de um método. A palavra-chave *var* não pode ser utilizada.

A seguir, a definição de um método chamado *showResult* que não retorna um valor e tem um único parâmetro *int* chamado *answer*:

```
void showResult(int answer)
{
    // ...
}
```

Observe o uso da palavra-chave *void* para indicar que o método nada retorna.



Importante Caso esteja familiarizado com Visual Basic, observe que o C# não utiliza palavras-chave diferentes para distinguir entre um método que retorna um valor (uma função) e um método que não retorna um valor (um procedimento ou sub-rotina). Você sempre deve especificar um tipo de retorno ou a palavra-chave *void*.

Retorne dados de um método

Para que um método retorne uma informação (ou seja, seu tipo de retorno não é *void*), você deve incluir uma instrução *return* no final do processamento do método. Uma instrução *return* consiste na palavra-chave *return*, seguida por uma expressão especificando o valor retornado e um ponto e vírgula. O tipo da expressão deve ser o mesmo tipo especificado pela declaração do método. Por exemplo, se um método retorna um *int*, a instrução *return* deve retornar um *int*; caso contrário, o programa não compilará. Aqui está um exemplo de método com uma instrução *return*:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
    return leftHandSide + rightHandSide;
}
```

A instrução *return*, em geral, fica no final do método porque o faz terminar e controla os retornos para a instrução que chamou o método, como descrito posteriormente neste capítulo. As instruções que ocorrerem após a instrução *return* não serão executadas (embora o compilador avise sobre esse problema, caso você coloque instruções depois da instrução *return*).

Se não quiser que o método retorne informações (ou seja, seu tipo de retorno é *void*), você pode utilizar uma variação da instrução *return* para causar uma saída imediata do método. Escreva a palavra-chave *return*, seguida imediatamente por um ponto e vírgula. Por exemplo:

```
void showResult(int answer)
{
    // exibe a resposta
    ...
    return;
}
```

Se o método não retornar coisa alguma, você também pode omitir a instrução *return*, porque o método finaliza automaticamente quando a execução chega à chave de fechamento no fim do método. Embora essa prática seja comum, ela nem sempre é considerada um bom estilo de programação.

No exercício a seguir, examinaremos outra versão do projeto MathsOperators do Capítulo 2. Essa versão foi aprimorada pela utilização cuidadosa de alguns pequenos métodos. Dividir código dessa maneira ajuda a torná-lo mais fácil de entender e de manter.

Examine as definições de método

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra o projeto Methods, que está na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 3\Methods na sua pasta Documentos.
3. No menu Debug, clique em Start Debugging.

O Visual Studio 2013 compila e executa o aplicativo. Ele deve parecer igual ao aplicativo do Capítulo 2.

4. Explore o aplicativo e o modo como ele funciona; em seguida, volte ao Visual Studio. No menu Debug, clique em Stop Debugging (ou clique em Quit na janela Methods, se estiver usando o Windows 7 ou o Windows 8).
5. Exiba o código de MainWindow.xaml.cs na janela Code and Text Editor (no Solution Explorer, expanda o arquivo MainWindow.xaml e, então, clique duas vezes em MainWindow.xaml.cs).
6. Na janela Code and Text Editor, localize o método *addValues*, que é como este:

```
private int addValues(int leftHandSide, int rightHandSide)
{
    expression.Text = leftHandSide.ToString() + " + " + rightHandSide.ToString();
    return leftHandSide + rightHandSide;
}
```



Nota Não se preocupe com a palavra-chave *private* no início da definição desse método, por enquanto; você vai aprender o significado dela no Capítulo 7, “Criação e gerenciamento de classes e objetos”.

O método *addValues* contém duas instruções. A primeira exibe o cálculo executado na caixa *expression* do formulário. Os valores dos parâmetros *leftHandSide* e *rightHandSide* são convertidos em strings (utilizando o método *ToString* descrito no Capítulo 2) e concatenados com a versão de string do operador de adição (+).

A segunda instrução utiliza a versão *int* do operador + para somar os valores das variáveis *int leftHandSide* e *rightHandSide* e retorna o resultado dessa operação. Lembre-se de que somar dois valores *int* cria outro valor *int*; portanto, o tipo de retorno do método *addValues* é *int*.

Se examinar os métodos *subtractValues*, *multiplyValues*, *divideValues* e *remainderValues*, você verá que eles seguem um padrão semelhante.

7. Na janela Code and Text Editor, localize o método *showResult*, que é como este:

```
private void showResult(int answer)
{
    result.Text = answer.ToString();
}
```

Esse método contém uma instrução que exibe uma representação em string do parâmetro *answer* na caixa *result*. Ele não retorna um valor, de modo que o tipo desse método é *void*.



Dica Não há um comprimento mínimo para um método. Se um método ajuda a evitar a repetição e a tornar seu programa mais fácil de entender, ele será útil, independentemente do seu tamanho.

Não há também um tamanho máximo para um método, mas é uma boa prática de programação mantê-lo com o menor tamanho possível. Se o método ocupar mais de uma tela, considere a possibilidade de dividi-lo em métodos menores para torná-lo mais legível.

Chame métodos

Os métodos existem para serem chamados! Você chama um método pelo nome para pedir a ele que execute sua tarefa. Se o método precisar de informações (conforme especificado pelos seus parâmetros), você deve fornecê-las. Se o método retorna informações (conforme especificado pelo seu tipo de retorno), você deve providenciar sua captura de alguma maneira.

Especifique a sintaxe de chamada de método

A sintaxe de uma chamada de método em C# é:

```
resultado = nomeDoMétodo (listaDeArgumentos)
```

Descrição dos elementos que constituem uma chamada de método:

- O *nomeDoMétodo* deve corresponder exatamente ao nome do método que você está chamando. Lembre-se, o C# é uma linguagem que faz distinção entre maiúsculas e minúsculas.
- A cláusula *resultado* = é opcional. Se especificada, a variável identificada como *resultado* conterá o valor retornado pelo método. Se o método for *void* (não retorna um valor), você deve omitir a cláusula *resultado* = da instrução. Se você não especificar a cláusula *resultado* = e o método retornar um valor, o método será executado, mas o valor de retorno será descartado.

- A *listaDeArgumentos* fornece as informações que o método aceita. Você deve fornecer um argumento para cada parâmetro, e o valor de cada argumento deve ser compatível com o tipo do seu parâmetro correspondente. Se o método que você está chamando tiver dois ou mais parâmetros, separe os argumentos com vírgulas.



Importante Você deve incluir os parênteses em cada chamada de método, mesmo quando estiver chamando um método sem argumentos.

Para esclarecer esses pontos, examine o método *addValues* novamente:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
}
```

O método *addValues* tem dois parâmetros *int*; portanto, você deve chamá-lo com dois argumentos *int* separados por vírgulas:

```
addValues(39, 3); // ok
```

Você também pode substituir os valores literais 39 e 3 pelos nomes de variáveis *int*. Os valores dessas variáveis são então passados para o método como seus argumentos, como a seguir:

```
int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2);
```

Se você tentar chamar *addValues* de alguma outra maneira, provavelmente não será bem-sucedido, pelas razões descritas nos exemplos abaixo:

```
addValues;           // erro de tempo de compilação, nenhum parêntese
addValues();         // erro de tempo de compilação, falta de argumentos
addValues(39);       // erro de tempo de compilação, falta de argumentos
addValues("39", "3"); // erro de tempo de compilação, tipos de argumentos errados
```

O método *addValues* retorna um valor *int*. Esse valor *int* poderá ser utilizado sempre que um valor *int* puder ser utilizado. Considere estes exemplos:

```
int result = addValues(39, 3); // no lado direito de uma atribuição
showResult(addValues(39, 3)); // como argumento para outra chamada de método
```

O exercício a seguir continua com o aplicativo Methods. Desta vez, você vai examinar algumas chamadas de método.

Examine as chamadas de método

- 1.** Retorne ao projeto Methods. (Esse projeto já estará aberto no Visual Studio 2013, se você estiver continuando do exercício anterior. Se não, abra-o na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 3\Windows X\Methods na sua pasta Documentos.)
- 2.** Exiba o código de MainWindow.xaml.cs, na janela Code and Text Editor.
- 3.** Localize o método *calculateClick* e examine as duas primeiras instruções desse método após a instrução *try* e uma chave de abertura. (Você vai aprender sobre as instruções *try* no Capítulo 6, "Gerenciamento de erros e exceções".)

Essas instruções devem se parecer com isto:

```
int leftHandSide = System.Int32.Parse(lhsOperand.Text);
int rightHandSide = System.Int32.Parse(rhsOperand.Text);
```

Essas duas instruções declaram duas variáveis *int* denominadas *leftHandSide* e *rightHandSide*. Observe como as variáveis são inicializadas. Em ambos os casos é chamado o método *Parse* do tipo *System.Int32*. (*System* é um namespace e *Int32* é o nome do tipo nesse namespace.) Vimos esse método anteriormente – ele recebe um único parâmetro *string* e o converte em um valor *int*. Essas duas linhas de código recebem as entradas do usuário nos controles caixa de texto *lhsOperand* e *rhsOperand* do formulário e as converte em valores *int*.

- 4.** Examine a quarta instrução no método *calculateClick* (após a instrução *if* e outra chave de abertura):

```
calculatedValue = addValues(leftHandSide, rightHandSide);
```

Essa instrução chama o método *addValues*, passando os valores das variáveis *leftHandSide* e *rightHandSide* como argumentos. O valor retornado pelo método *addValues* é armazenado na variável *calculatedValue*.

- 5.** Examine a próxima instrução:

```
showResult(calculatedValue);
```

Essa instrução chama o método *showResult*, passando o valor da variável *calculatedValue* como argumento. O método *showResult* não retorna um valor.

- 6.** Na janela Code and Text Editor, localize o método *showResult* examinado anteriormente.

A única instrução desse método é esta:

```
result.Text = answer.ToString();
```

Observe que a chamada ao método *ToString* utiliza parênteses embora não haja argumentos.



Dica Você pode chamar os métodos que pertencem a outros objetos prefixando o método com o nome do objeto. No exemplo anterior, a expressão *answer.ToString()* chama o método denominado *ToString* pertencente ao objeto denominado *answer*.

Aplique escopo

Para criar métodos, você combina instruções. Você pode criar variáveis facilmente em vários pontos de seu aplicativo. Por exemplo, o método *calculateClick* do projeto Methods cria uma variável *int* chamada *calculatedValue* e atribui a ela o valor inicial zero, como segue:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
    int calculatedValue = 0;
    ...
}
```

Essa variável passa a existir a partir do ponto em que é definida, e as instruções subsequentes no método *calculateClick* podem então utilizá-la. Esse ponto é importante: uma variável só pode ser utilizada depois de ser criada. Quando o método termina, essa variável desaparece e não pode ser usada em outro lugar.

Quando uma variável pode ser acessada em um local específico em um programa, dizemos que ela está no *escopo* desse local. A variável *calculatedValue* tem escopo de método; ela pode ser acessada por todo o método *calculateClick*, mas não fora dele. Também é possível definir variáveis com escopo diferente; por exemplo, definir uma variável fora de um método, mas dentro de uma classe – essa variável pode ser acessada por qualquer método dentro dessa classe. Diz-se que essa variável tem *escopo de classe*.

Ou seja, o escopo de uma variável é simplesmente a região do programa na qual essa variável é utilizada. O escopo se aplica aos métodos e às variáveis. O escopo de um identificador (de uma variável ou método) está vinculado ao local da declaração que introduz o identificador no programa, como você vai aprender a seguir.

Defina o escopo local

As chaves de abertura e fechamento que formam o corpo de um método definem o escopo desse método. Todas as variáveis que você declara dentro do corpo de um método estão no seu escopo; elas desaparecem quando o método termina e só podem ser acessadas pelo código executado dentro desse método. Essas variáveis são denominadas *variáveis locais* porque são locais para o método em que são declaradas; elas não estão no escopo de nenhum outro método.

O escopo das variáveis locais significa que não é possível utilizá-las para compartilhar informações entre métodos. Considere este exemplo:

```
class Example
{
    void firstMethod()
    {
        int myVar;
        ...
    }
    void anotherMethod()
    {
        myVar = 42; // erro - variável fora de escopo
        ...
    }
}
```

Ocorrerá uma falha na compilação desse código porque *anotherMethod* está tentando utilizar a variável *myVar* que não está no escopo. A variável *myVar* só está disponível para as instruções em *firstMethod* que ocorrem depois da linha do código que a declara.

Defina o escopo de classe

As chaves de abertura e fechamento que formam o corpo de uma classe definem o escopo dessa classe. Todas as variáveis que você declara dentro do corpo de uma classe (mas não dentro de um método) estão no escopo dela. O termo apropriado do C# para uma variável definida por uma classe é *field* (campo). Conforme mencionado anteriormente, ao contrário das variáveis locais, os campos podem ser utilizados para compartilhar informações entre métodos. Veja um exemplo:

```
class Example
{
    void firstMethod()
    {
        myField = 42; // ok
        ...
    }
    void anotherMethod()
    {
        myField++; // ok
        ...
    }

    int myField = 0;
}
```

A variável *myField* é definida dentro da classe, mas fora dos métodos *firstMethod* e *anotherMethod*. Portanto, *myField* tem escopo de classe e está disponível para uso por todos os métodos dessa classe.

Há outro ponto a ser observado nesse exemplo. Em um método, você deve declarar uma variável antes de poder utilizá-la. Os campos são um pouco diferentes. Um método pode utilizar um campo antes da instrução que define o campo – o compilador resolve os detalhes para você.

Sobrecarregue métodos

Se dois identificadores têm o mesmo nome e são declarados no mesmo escopo, dizemos que eles estão *sobre carregados*. Um identificador sobre carregado costuma ser um erro capturado como um erro de tempo de compilação. Por exemplo, se você declarar duas variáveis locais com o mesmo nome no mesmo método, o compilador informará um erro. Da mesma forma, se declarar dois campos com o mesmo nome na mesma classe ou dois métodos idênticos na mesma classe, também receberá um erro de tempo de compilação. Talvez não valha a pena mencionar isso, uma vez que tudo que vimos até aqui resulta em um erro de tempo de compilação. Mas há uma maneira útil e importante pela qual você pode sobre carregar um identificador para um método.

Considere o método *WriteLine* da classe *Console*. Você já utilizou esse método para escrever uma string na tela. Mas ao digitar *WriteLine* na janela Code and Text Editor escrevendo em C#, note que o Microsoft IntelliSense oferece 19 opções diferentes! Cada versão do método *WriteLine* tem um conjunto de parâmetros diferente; uma versão não tem parâmetros e simplesmente gera uma linha em branco; outra aceita um parâmetro *bool* e gera uma representação em string desse valor (*True* ou *False*); ainda outra, aceita um parâmetro *decimal* e gera uma string, e assim por diante. Em tempo de compilação, o compilador examina os tipos de argumentos que você está passando e então providencia para que seu aplicativo chame a versão do método que tem o conjunto de parâmetros correspondente. Veja um exemplo:

```
static void Main()
{
    Console.WriteLine("The answer is ");
    Console.WriteLine(42);
}
```

A sobre carga é útil principalmente quando você precisa executar a mesma operação em diferentes tipos de dados ou grupos variados de informações. Você pode sobre carregar um método quando as diferentes implementações têm diferentes conjuntos de parâmetros – isto é, quando elas têm o mesmo nome, mas um número diferente de parâmetros, ou quando os tipos de parâmetro forem diferentes. Quando chama um método, você fornece uma lista de argumentos separados por vírgula; e o número e o tipo dos argumentos são utilizados pelo compilador para selecionar um dos métodos sobre carregados. Mas lembre-se de que, embora possa sobre carregar os parâmetros de um método, você não pode sobre carregar o tipo de retorno de um método. Ou seja, você não pode declarar dois métodos com o mesmo nome cuja diferença seja apenas o seu tipo de retorno. (O compilador é inteligente, mas não tão inteligente.)

Escreva métodos

Nos exercícios a seguir, você vai criar um método que calcula quanto um consultor ganhará por um determinado número de dias de consultoria a uma dada remuneração por dia. Você começará desenvolvendo a lógica do aplicativo e então utilizará o assistente Generate Method Stub para ajudar a escrever os métodos que serão utilizados por essa lógica. Em seguida, você executará esses métodos em um aplicativo de console para ter uma ideia do programa. Por fim, você vai explorar o depurador do Visual Studio 2013 para entrar e sair das chamadas de método à medida que elas são executadas.

Desenvolva a lógica do aplicativo

1. Utilizando o Visual Studio 2013, abra o projeto DailyRate, que está na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 3\Windows X\DailyRate na sua pasta Documentos.
2. No Solution Explorer, no projeto DailyRate, clique duas vezes no arquivo Program.cs para exibir o código do programa na janela Code and Text Editor.

Esse programa é simplesmente um teste para experimentar seu código. Quando o aplicativo começa a executar, ele chama o método *run*. Você pode adicionar o método *run* ao código que deseja testar. (A maneira como o método é chamado exige entendimento das classes, o que veremos no Capítulo 7.)

3. Adicione as seguintes instruções mostradas em negrito ao corpo do método *run*, entre as chaves de abertura e de fechamento:

```
void run()
{
    double dailyRate = readDouble("Enter your daily rate: ");
    int noOfDays = readInt("Enter the number of days: ");
    writeFee(calculateFee(dailyRate, noOfDays));
}
```

O bloco de código que você adicionou ao método *run* chama o método *readDouble* (que você vai escrever em breve) para pedir ao usuário que informe a taxa diária do consultor. A próxima instrução chama o método *readInt* (que você também vai escrever) para obter o número de dias. Por fim, o método *writeFee* (a ser escrito) é chamado para exibir os resultados na tela. Observe que o valor passado para *writeFee* é o valor retornado pelo método *calculateFee* (o último que precisará ser escrito), ao qual é informado o preço por dia e o número de dias, e calcula a taxa total a ser paga.

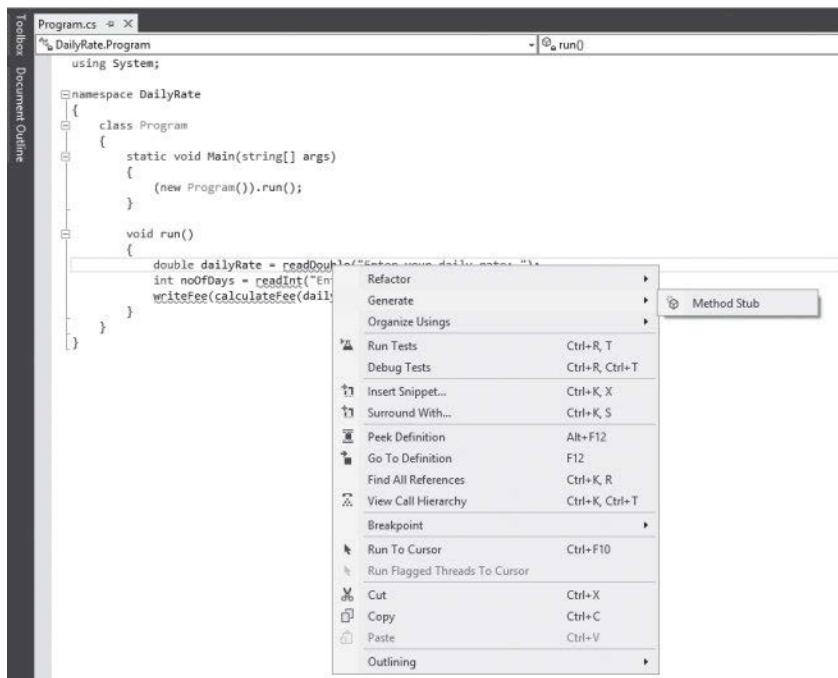


Nota Você ainda não escreveu os métodos *readDouble*, *readInt*, *writeFee* e *calculateFee*; portanto, o IntelliSense não exibirá esses métodos quando você digitar esse código. Não tente compilar o aplicativo ainda – ele falhará.

Escreva os métodos utilizando o assistente Generate Method Stub

1. Na janela Code and Text Editor, no método *run*, clique com o botão direito do mouse na chamada de método *readDouble*.

Um menu de atalho aparece, contendo comandos úteis para gerar e editar código, como mostrado aqui:



2. Nesse menu de atalho, aponte para Generate e clique em Method Stub.

O assistente Generate Method Stub examina a chamada ao método `readDouble`, verifica o tipo dos seus parâmetros e do valor de retorno e gera um método com uma implementação padrão, como mostrado a seguir:

```
private double readDouble(string p)
{
    throw new NotImplementedException();
}
```

O novo método é criado com o qualificador `private`, descrito no Capítulo 7. Atualmente o corpo do método simplesmente lança uma exceção `NotImplementedException`. (As exceções serão descritas no Capítulo 6.) Você vai substituir o corpo pelo seu próprio código no próximo passo.

3. Exclua a instrução `throw new NotImplementedException();` do método `readDouble` e a substitua pelas linhas de código em negrito a seguir:

```
private double readDouble(string p)
{
    Console.WriteLine(p);
    string line = Console.ReadLine();
    return double.Parse(line);
}
```

Esse bloco de código exibe a string da variável *p* na tela. Essa variável é o parâmetro de string que é passado quando o método é chamado; ele contém a mensagem solicitando que o usuário digite a taxa diária.



Nota O método *Console.WriteLine* é semelhante à instrução *Console.WriteLine*; já utilizada nos exercícios anteriores, exceto pelo fato de não gerar um caractere de nova linha depois da mensagem.

O usuário digita um valor, o qual é lido em um tipo *string* utilizando o método *ReadLine* e convertido em um tipo *double* utilizando o método *double.Parse*. O resultado é passado de volta como o valor de retorno da chamada de método.



Nota O método *ReadLine* é companheiro do método *WriteLine*; ele lê a entrada do usuário no teclado, terminando quando o usuário pressiona a tecla Enter. O texto digitado pelo usuário é passado de volta como o valor de retorno. O texto é retornado como um valor de *string*.

4. No método *run*, clique com o botão direito do mouse na chamada ao método *readInt*, aponte para Generate e clique em Method Stub para gerar o método *readInt*.

O método *readInt* é gerado desta maneira:

```
private int readInt(string p)
{
    throw new NotImplementedException();
}
```

5. Substitua a instrução *throw new NotImplementedException()*; no corpo do método *readInt* pelo código em negrito a seguir:

```
private int readInt(string p)
{
    Console.WriteLine(p);
    string line = Console.ReadLine();
    return int.Parse(line);
}
```

Esse bloco de código é semelhante ao código do método *readDouble*. A única diferença é que o método retorna um valor *int*; portanto, a *string* digitada pelo usuário é convertida em um número, utilizando o método *int.Parse*.

6. Clique com o botão direito do mouse na chamada ao método *calculateFee* dentro do método *run*, aponte para Generate e clique em Method Stub.

O método *calculateFee* é gerado desta maneira:

```
private object calculateFee(double dailyRate, int noOfDays)
{
    throw new NotImplementedException();
}
```

Nesse caso, observe que o Visual Studio utiliza os nomes dos argumentos passados para gerar os nomes dos parâmetros. (Evidentemente, você pode alterar os nomes dos parâmetros se eles não forem adequados.) O mais intrigante é o tipo retornado pelo método, que é *object*. O Visual Studio é incapaz de determinar exatamente que tipo de valor deve ser retornado pelo método a partir do contexto em que ele é chamado. O tipo *object* significa apenas uma “coisa”, e você deve alterá-lo para o tipo necessário quando adicionar o código ao método. O Capítulo 7 abordará o tipo *object* com mais detalhes.

7. Mude a definição do método *calculateFee* para que ele retorne um *double*, como mostrado em negrito aqui:

```
private double calculateFee(double dailyRate, int noOfDays)
{
    throw new NotImplementedException();
}
```

8. Substitua o corpo do método *calculateFee* pela instrução em negrito a seguir, que calcula e retorna a remuneração a ser paga, multiplicando os dois parâmetros:

```
private double calculateFee(double dailyRate, int noOfDays)
{
    return dailyRate * noOfDays;
}
```

9. Clique com o botão direito do mouse na chamada ao método *writeFee* dentro do método *run*, clique em Generate e clique em Method Stub.

Observe que o Visual Studio utiliza a definição do método *calculateFee* para concluir que seu parâmetro deve ser *double*. Além disso, a chamada do método não utiliza um valor de retorno, portanto, o tipo do método é *void*:

```
private void writeFee(double p)
{
    ...
}
```



Dica Se você se sentir à vontade com a sintaxe, também pode escrever os métodos digitando-os diretamente na janela Code and Text Editor. Não é necessário utilizar sempre a opção de menu Generate.

- 10.** Substitua o código do corpo do método *writeFee* pela instrução a seguir, que calcula a taxa e adiciona 10% de comissão:

```
Console.WriteLine("The consultant's fee is: {0}", p * 1.1);
```



Nota Essa versão do método *WriteLine* demonstra o uso de uma string de formato simples. O texto *{0}* na string utilizada como o primeiro argumento para o método *WriteLine* é um espaço reservado que é substituído pelo valor da expressão depois da string (*p * 1.1*), quando ela é avaliada em tempo de execução. O uso dessa técnica é preferível às alternativas, como converter o valor da expressão *p * 1.1* em uma string e utilizar o operador + para concatená-la à mensagem.

- 11.** No menu Build, clique em Build Solution.

Refatoração de código

Um recurso muito útil do Visual Studio 2013 é a capacidade de refatorar o código.

Ocasionalmente, você perceberá que está escrevendo o mesmo código (ou semelhante) em mais de um lugar em um aplicativo. Quando isso ocorrer, realize e clique com o botão direito do mouse no bloco de código que você acabou de digitar e, então, no menu Refactor que aparece, clique em Extract Method. A caixa de diálogo Extract Method se abre, solicitando o nome de um novo método que conterá esse código. Digite um nome e clique em OK. O novo método é criado contendo seu código, e o código que você digitou é substituído por uma chamada a esse método. Extract Method também é capaz de identificar se o método deve ter algum parâmetro e retornar um valor.

Teste o programa

- 1.** No menu Debug, clique em Start Without Debugging.

O Visual Studio 2013 compila o programa e o executa. Uma janela de console aparece.

- 2.** No prompt Enter Your Daily Rate, digite **525** e pressione Enter.

- 3.** No prompt Enter the Number of Days, digite **17** e pressione Enter.

O programa escreve a seguinte mensagem na janela de console:

```
The consultant's fee is: 9817.5
```

- 4.** Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2013.

No próximo exercício, você vai utilizar o depurador do Visual Studio 2013 para executar seu programa lentamente. Você vai ver quando cada método é chamado (o que é citado como *stepping into the method*) e como cada instrução *return* transfere o controle de volta ao chamador (também conhecido como *stepping out of the method*).

ou “sair do método”). Ao entrar e sair dos métodos, você pode utilizar as ferramentas da barra de ferramentas Debug. Mas os mesmos comandos também estão disponíveis no menu Debug quando um aplicativo está sendo executado no modo de depuração.

Inspecione os métodos passo a passo utilizando o depurador do Visual Studio 2013

1. Na janela Code and Text Editor, localize o método *run*.
2. Mova o cursor para a primeira instrução do método *run*.


```
double dailyRate = readDouble("Enter your daily rate: ");
```
3. Clique com o botão direito do mouse em qualquer lugar dessa linha e, no menu de atalho que aparece, clique em Run To Cursor.

O programa inicia e é executado até chegar à primeira instrução do método *run* e, então, faz uma pausa. Uma seta amarela na margem esquerda da janela Code and Text Editor indica a instrução atual, e a instrução em si é realçada com um fundo amarelo.

```
Program.cs # X
DailyRate.Program
using System;

namespace DailyRate
{
    class Program
    {
        static void Main(string[] args)
        {
            (new Program()).run();
        }

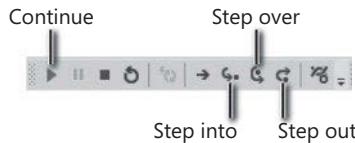
        void run()
        {
            double dailyRate = readDouble("Enter your daily rate: ");
            int noOfDays = readInt("Enter the number of days: ");
            writeFee(calculateFee(dailyRate, noOfDays));
        }

        private void writeFee(double p)
        {
            Console.WriteLine("The consultant's fee is: {0}", p * 1.1);
        }

        private double calculateFee(double dailyRate, int noOfDays)
        {
            return dailyRate * noOfDays;
        }
    }
}
```

4. No menu View, aponte para Toolbars e verifique se a barra de ferramentas Debug está selecionada.

Se ela ainda não estiver visível, a barra de ferramentas Debug é aberta. Ela pode aparecer encaixada com as outras barras de ferramentas. Se não puder ver a barra de ferramentas, tente utilizar o comando Toolbars no menu View para ocultá-la e observe quais botões desaparecem. Então, exiba a barra de ferramentas novamente. A barra de ferramentas Debug é parecida com esta:



5. Na barra de ferramentas Debug, clique no botão Step Into. (É o sétimo botão a partir da esquerda na barra de ferramentas Debug.)

Essa ação faz o depurador entrar no método chamado. O cursor amarelo pula para a chave de abertura no início do método `readDouble`.

6. Clique em Step Into novamente para avançar o cursor até a primeira instrução:

```
Console.WriteLine(p);
```



Dica Você também pode pressionar F11 em vez de clicar várias vezes em Step Into na barra de ferramentas Debug.

7. Na barra de ferramentas Debug, clique em Step Over. (É o oitavo botão a partir da esquerda.)

Essa ação faz o método executar a próxima instrução sem depurá-la (sem entrar nela). A ação é útil principalmente se a instrução chama um método, mas você não quer passar por cada instrução desse método. O cursor amarelo se move para a segunda instrução do método e o programa exibe o prompt Enter Your Daily Rate em uma janela de console, antes de retornar ao Visual Studio 2013. (A janela de console pode estar oculta atrás do Visual Studio.)



Dica Você também pode pressionar F10 em vez de clicar em Step Over na barra de ferramentas Debug.

8. Na barra de ferramentas Debug, clique novamente em Step Over.

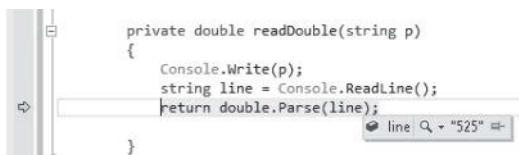
Desta vez, o cursor amarelo desaparece e a janela de console recebe o foco porque o programa está executando o método `Console.ReadLine` e esperando que você digite algo.

9. Digite **525** na janela de console e pressione Enter.

O controle retorna ao Visual Studio 2013. O cursor amarelo aparece na terceira linha do método.

10. Posicione o mouse sobre a referência à variável `line` na segunda ou na terceira linha do método. (Não importa qual delas.)

Uma dica de tela aparece, exibindo o valor atual da variável `line` ("525"). Você pode utilizar esse recurso para verificar se uma variável foi definida com um valor esperado durante a execução passo a passo dos métodos.



- 11.** Na barra de ferramentas Debug, clique em Step Out. (É o nono botão a partir da esquerda.)

Essa ação faz o método atual continuar executando ininterruptamente até o fim. O método *readDouble* termina e o cursor amarelo é colocado de volta na primeira instrução do método *run*. Agora terminou a execução dessa instrução.



Dica Você também pode pressionar Shift+F11 em vez de clicar em Step Out na barra de ferramentas Debug.

- 12.** Na barra de ferramentas Debug, clique em Step Into.

O cursor amarelo se move para a segunda instrução no método *run*:

```
int noOfDays = readInt("Enter the number of days: ");
```

- 13.** Na barra de ferramentas Debug, clique em Step Over.

Desta vez, você optou por executar o método sem entrar nele. A janela de console aparece novamente solicitando o número de dias.

- 14.** Na janela de console, digite **17** e pressione Enter.

O controle volta para o Visual Studio 2013 (talvez seja necessário trazer o Visual Studio para o primeiro plano). O cursor amarelo se move para a terceira instrução do método *run*:

```
writeFee(calculateFee(dailyRate, noOfDays));
```

- 15.** Na barra de ferramentas Debug, clique em Step Into.

O cursor amarelo pula para a chave de abertura no início do método *calculateFee*. Esse método é o primeiro a ser chamado, antes de *writeFee*, porque o valor retornado por esse método é utilizado como o parâmetro para *writeFee*.

- 16.** Na barra de ferramentas Debug, clique em Step Out.

A chamada do método *calculateFee* termina e o cursor amarelo salta para a terceira instrução do método *run*.

- 17.** Na barra de ferramentas Debug, clique em Step Into.

Desta vez, o cursor amarelo pula para a chave de abertura no início do método *writeFee*.

- 18.** Coloque o mouse sobre o parâmetro *p* na definição do método.

O valor de *p*, 8925.0, aparece em uma dica de tela.

- 19.** Na barra de ferramentas Debug, clique em Step Out.

A mensagem “The consultant’s fee is: 9817.5” aparece na janela de console. (Talvez seja necessário trazer a janela de console para o primeiro plano para exibi-la, caso esteja atrás do Visual Studio 2013.) O cursor amarelo retorna à terceira instrução do método *run*.

- 20.** No menu Debug, clique em Continue para fazer o programa continuar executando sem parar em cada instrução.



Dica Se o botão Continue não estiver visível, clique no menu suspenso Add or Remove Buttons que aparece na extremidade da barra de ferramentas Debug e, então, selecione Continue. Agora o botão Continue deverá aparecer. Como alternativa, você pode pressionar F5 para continuar a execução do aplicativo sem depurar.

O aplicativo termina e para de executar. Observe que a barra de ferramentas Debug desaparece quando o aplicativo termina — por padrão, ela só aparece quando um aplicativo está sendo executado no modo de depuração.

Parâmetros opcionais e argumentos nomeados

Você já sabe que, ao definir métodos sobrecarregados, é possível implementar diversas versões de um método, que aceitam diferentes parâmetros. Quando você constrói um aplicativo que utiliza métodos sobrecarregados, o compilador determina quais instâncias específicas de cada método deve usar para atender à chamada de cada método. Esse é um recurso comum de várias linguagens orientadas a objetos, não apenas do C#.

Entretanto, os desenvolvedores podem utilizar outras linguagens e tecnologias que não seguem essas regras para construir aplicativos Windows e componentes. Um recurso importante do C# e de outras linguagens elaboradas para o .NET Framework é a possibilidade de interagir com aplicativos e componentes escritos em outras tecnologias. Uma das principais tecnologias que servem de base para muitos aplicativos Microsoft Windows e serviços executados fora do .NET Framework é o Component Object Model (COM). Na verdade, o Common Language Runtime (CLR) utilizado pelo .NET Framework também é fortemente dependente do COM, assim como o Windows Runtime do Windows 8 e do Windows 8.1. O COM não aceita métodos sobrecarregados; em vez disso, utiliza métodos que admitem parâmetros opcionais. Para facilitar ainda mais a incorporação de bibliotecas COM e componentes em uma solução do C#, esta linguagem também dispõe de suporte para parâmetros opcionais.

Os parâmetros opcionais também são úteis em outras situações. Eles representam uma solução compacta e simples, quando não é possível utilizar sobrecarga porque os tipos dos parâmetros não variam o bastante para permitir que o compilador possa distinguir entre as implementações. Por exemplo, considere o seguinte método:

```
public void DoWorkWithData(int intData, float floatData, int moreIntData)
{
    ...
}
```

O método *DoWorkWithData* aceita três parâmetros: dois *int* e um *float*. Vamos supor que você queira fornecer uma implementação do método *DoWorkWithData* que aceite apenas dois parâmetros: *intData* e *floatData*. Você pode sobrecarregar o método, como demonstrado a seguir:

```
public void DoWorkWithData(int intData, float floatData)
{
    ...
}
```

Se você escrever uma instrução que chama o método *DoWorkWithData*, poderá fornecer dois ou três parâmetros dos tipos adequados, e o compilador usará a informação do tipo para determinar a sobrecarga a ser chamada:

```
int arg1 = 99;
float arg2 = 100.0F;
int arg3 = 101;

DoWorkWithData(arg1, arg2, arg3); // Chama a sobrecarga com três parâmetros
DoWorkWithData(arg1, arg2);      // Chama a sobrecarga com dois parâmetros
```

Entretanto, vamos supor que você queira implementar duas outras versões do método *DoWorkWithData*, que aceitem apenas o primeiro e o terceiro parâmetros. Você poderia experimentar o seguinte:

```
public void DoWorkWithData(int intData)
{
    ...
}

public void DoWorkWithData(int moreIntData)
{
    ...
}
```

O problema aqui é que, para o compilador, essas duas sobrecargas parecem idênticas. A compilação de seu código falhará e gerará o erro “Type ‘*typename*’ already defines a member called ‘*DoWorkWithData*’ with the same parameter types” (o tipo “*nome_do_tipo*” já define um membro chamado “*DoWorkWithData*” com os mesmos tipos de parâmetro). Para entender por que isso acontece, se esse código fosse válido, considere as seguintes instruções:

```
int arg1 = 99;
int arg3 = 101;

DoWorkWithData(arg1);
DoWorkWithData(arg3);
```

Que sobrecarga ou sobrecargas as chamadas ao método *DoWorkWithData* acionariam? O uso de parâmetros opcionais e argumentos nomeados pode ajudar a solucionar esse problema.

Defina parâmetros opcionais

Ao definir um método, você especifica que um parâmetro é opcional fornecendo um valor padrão para o parâmetro. Para indicar um valor padrão, utilize um operador de atribuição. No método *optMethod* mostrado a seguir, o parâmetro *first* é obrigatório porque não especifica um valor padrão, mas os parâmetros *second* e *third* são opcionais:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
    ...
}
```

Você deve especificar todos os parâmetros obrigatórios antes de qualquer parâmetro opcional.

Chame um método que aceita parâmetros opcionais da mesma maneira como você chama qualquer outro método: especifique o nome do método e inclua os argumentos necessários. A diferença em relação aos métodos que aceitam parâmetros opcionais é a possibilidade de omitir os argumentos correspondentes – o método usará o valor padrão quando for executado. No exemplo de código a seguir, a primeira chamada ao método *optMethod* fornece os valores dos três parâmetros. A segunda chamada especifica apenas dois argumentos, e esses valores são aplicados aos parâmetros *first* e *second*. O parâmetro *third* recebe o valor padrão "Hello" quando o método é executado.

```
optMethod(99, 123.45, "World"); // Argumentos fornecidos para os três parâmetros
optMethod(100, 54.321);        // Argumentos fornecidos apenas para os dois primeiros parâmetros
```

Passe argumentos nomeados

Por padrão, o C# utiliza a posição de cada argumento na chamada a um método para determinar os parâmetros aos quais eles se aplicam. Portanto, o segundo exemplo de método mostrado na seção anterior passa os dois argumentos para os parâmetros *first* e *second* no método *optMethod*, porque essa é a sequência na qual eles ocorrem na declaração do método. No C# também é possível especificar parâmetros pelo nome. Esse recurso permite passar os argumentos em uma sequência diferente. Para passar um argumento como um parâmetro nomeado, especifique o nome do parâmetro, seguido por um caractere de dois-pontos e o valor a ser utilizado. Os exemplos a seguir desempenham a mesma função daqueles apresentados na seção anterior, exceto pelo fato de que os parâmetros são especificados por nome:

```
optMethod(first : 99, second : 123.45, third : "World");
optMethod(first : 100, second : 54.321);
```

Os argumentos nomeados permitem que você passe os argumentos em qualquer ordem. Você pode reescrever o código que chama o método *optMethod*, como mostrado aqui:

```
optMethod(third : "World", second : 123.45, first : 99);
optMethod(second : 54.321, first : 100);
```

Esse recurso também torna possível omitir os argumentos. Por exemplo, você pode chamar o método *optMethod* e especificar apenas os valores dos parâmetros *first* e *third* e utilizar o valor padrão para o parâmetro *second*, como a seguir:

```
optMethod(first : 99, third : "World");
```

Além disso, é possível mesclar argumentos posicionais e nomeados. Entretanto, ao utilizar essa técnica, você deve especificar todos os argumentos posicionais antes do primeiro argumento nomeado.

```
optMethod(99, third : "World"); // O primeiro argumento é posicional
```

Resolva ambiguidades com parâmetros opcionais e argumentos nomeados

O uso de parâmetros opcionais e argumentos nomeados pode gerar algumas ambiguidades em seu código. Você deve saber como o compilador resolve essas ambiguidades; caso contrário, seus aplicativos poderão se comportar de modo imprevisto. Vamos supor que você defina o método *optMethod* como um método sobrecarregado, como mostra o exemplo a seguir:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
    ...
}

void optMethod(int first, double second = 1.0, string third = "Goodbye", int fourth
= 100 )
{
    ...
}
```

Esse é um código do C# perfeitamente válido que segue as regras dos métodos sobrecarregados. O compilador pode diferenciar entre os métodos porque eles têm listas de parâmetros diferentes. Entretanto, como demonstrado no exemplo a seguir, pode ocorrer um problema se você tentar chamar o método *optMethod* e omitir algum dos argumentos correspondentes a um ou mais parâmetros opcionais:

```
optMethod(1, 2.5, "World");
```

Mais uma vez, é um código válido, mas ele executa qual versão do método *optMethod*? A resposta é que ele executa a versão que mais se aproxima da chamada ao método, de modo que ele chama o método que aceita três parâmetros, e não a versão que aceita quatro. Isso é justificável; portanto, considere o seguinte:

```
optMethod(1, fourth : 101);
```

Nesse código, a chamada ao método *optMethod* omite os argumentos dos parâmetros *second* e *third*, mas especifica o parâmetro *fourth* pelo nome. Apenas uma versão do método *optMethod* corresponde a essa chamada, de modo que não ocorre qualquer problema. Entretanto, o próximo código vai deixá-lo intrigado:

```
optMethod(1, 2.5);
```

Desta vez, nenhuma das versões do método *optMethod* combina exatamente com a lista de argumentos fornecida. Ambas as versões desse método têm parâmetros opcionais para o segundo, terceiro e quarto argumentos. Então, essa instrução chama a versão do método *optMethod* que aceita três parâmetros e utiliza o valor padrão para o parâmetro *third* ou chama a versão do *optMethod* que aceita quatro parâmetros e utiliza o valor padrão para os parâmetros *third* e *fourth*? A resposta é: nem uma coisa, nem outra. Essa é uma ambiguidade insolúvel e o compilador não permite a compilação do aplicativo. A mesma situação ocorrerá, com o mesmo resultado, se você tentar chamar o método *optMethod* como mostrado em qualquer uma das seguintes instruções:

```
optMethod(1, third : "World");
optMethod(1);
optMethod(second : 2.5, first : 1);
```

No último exercício deste capítulo, você vai praticar a implementação de métodos que aceitam parâmetros opcionais e vai chamá-los por meio de argumentos nomeados. Você também testará exemplos comuns de como o compilador do C# resolve as chamadas a métodos que englobam parâmetros opcionais e argumentos nomeados.

Defina e chame um método que aceita parâmetros opcionais

1. No Visual Studio 2013, abra o projeto DailyRate, que está na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 3\Windows X\DailyRate Using Optional Parameters na pasta Documentos.
2. No Solution Explorer, no projeto DailyRate, clique duas vezes no arquivo Program.cs para exibir o código do programa na janela Code and Text Editor.

Essa versão do aplicativo está vazia, a não ser pelo método *Main* e o esqueleto da versão do método *run*.

3. Na classe *Program*, adicione o método *calculateFee* abaixo do método *run*. Essa é a mesma versão do método implementado no conjunto anterior de exercícios, exceto pelo fato de aceitar dois parâmetros opcionais com valores padrão. O método também imprime uma mensagem que indica a versão chamada do método *calculateFee*. (Nas etapas a seguir, você adicionará as versões sobrecarregadas desse método.)

```
private double calculateFee(double dailyRate = 500.0, int noOfDays = 1)
{
    Console.WriteLine("calculateFee using two optional parameters");
    return dailyRate * noOfDays;
}
```

4. Adicione outra implementação do método *calculateFee* à classe *Program*, como mostrado a seguir. Essa versão aceita um único parâmetro, chamado *dailyRate*, do tipo *double*. O corpo do método calcula e retorna a taxa de um único dia.

```
private double calculateFee(double dailyRate = 500.0)
{
    Console.WriteLine("calculateFee using one optional parameter");
    int defaultNoOfDays = 1;
```

```
        return dailyRate * defaultNoOfDays;
    }
```

5. Adicione uma terceira implementação do método *calculateFee* à classe *Program*. Essa versão não aceita parâmetros e utiliza os valores codificados para a taxa diária e o número de dias.

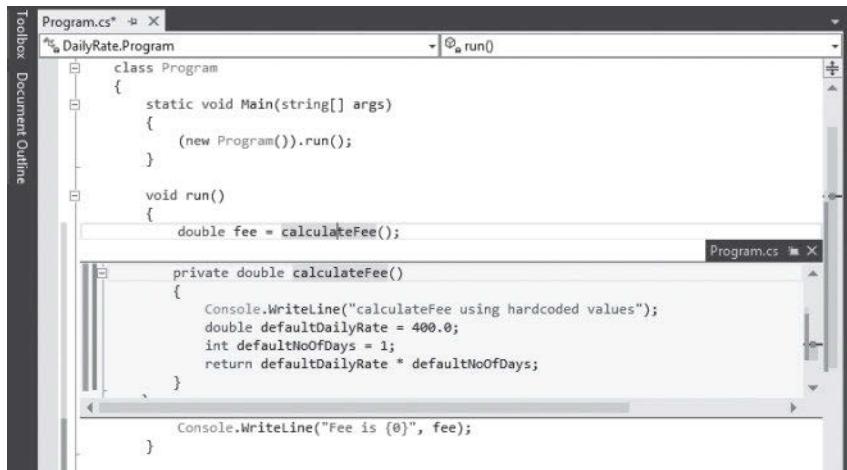
```
private double calculateFee()
{
    Console.WriteLine("calculateFee using hardcoded values");
    double defaultDailyRate = 400.0;
    int defaultNoOfDays = 1;
    return defaultDailyRate * defaultNoOfDays;
}
```

6. No método *run*, adicione as seguintes instruções em negrito, que chamam *calculateFee* e exibem os resultados:

```
public void run()
{
    double fee = calculateFee();
    Console.WriteLine("Fee is {0}", fee);
}
```



Dica É possível ver rapidamente a definição de um método a partir da instrução que o chama. Para isso, clique com o botão direito do mouse na chamada do método e, então, no menu de atalho que aparece, clique em Peek Definition. A imagem a seguir mostra a janela Peek Definition para o método *calculateFee*.



Esse recurso é extremamente útil se seu código está dividido em vários arquivos ou mesmo se está no mesmo arquivo, mas este é muito longo.

7. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

O programa é executado em uma janela de console e exibe as seguintes mensagens:

```
calculateFee using hardcoded values  
Fee is 400
```

O método *run* chamou a versão de *calculateFee* que não aceita parâmetros e não as implementações que aceitam parâmetros opcionais, pois é a versão mais compatível com a chamada do método.

Pressione qualquer tecla para fechar a janela do console e retornar ao Visual Studio.

8. No método *run*, modifique a instrução que chama *calculateFee*, de acordo com o código mostrado em negrito neste exemplo:

```
public void run()  
{  
    double fee = calculateFee(650.0);  
    Console.WriteLine("Fee is {0}", fee);  
}
```

9. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

O programa exibe as seguintes mensagens:

```
calculateFee using one optional parameter  
Fee is 650
```

Desta vez, o método *run* chamou a versão de *calculateFee* que aceita um único parâmetro opcional. Como antes, isso acontece porque essa é a versão que mais se aproxima da chamada do método.

Pressione qualquer tecla para fechar a janela do console e retornar ao Visual Studio.

10. No método *run*, modifique novamente a instrução que chama *calculateFee*:

```
public void run()  
{  
    double fee = calculateFee(500.0, 3);  
    Console.WriteLine("Fee is {0}", fee);  
}
```

11. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

O programa exibe as seguintes mensagens:

```
calculateFee using two optional parameters  
Fee is 1500
```

Como você já previa, com base nos dois casos anteriores, o método *run* chamou a versão de *calculateFee* que aceita dois parâmetros opcionais.

Pressione qualquer tecla para fechar a janela do console e retornar ao Visual Studio.

- 12.** No método *run*, modifique a instrução que chama *calculateFee* e especifique o parâmetro *dailyRate* pelo nome:

```
public void run()
{
    double fee = calculateFee(dailyRate : 375.0);
    Console.WriteLine("Fee is {0}", fee);
}
```

- 13.** No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

O programa exibe as seguintes mensagens:

```
calculateFee using one optional parameter
Fee is 375
```

Como antes, o método *run* chama a versão de *calculateFee* que aceita um único parâmetro opcional. Mudar o código para utilizar um argumento nomeado não altera o modo como o compilador resolve a chamada ao método neste exemplo.

Pressione qualquer tecla para fechar a janela do console e retornar ao Visual Studio.

- 14.** No método *run*, modifique a instrução que chama *calculateFee* e especifique o parâmetro *noOfDays* pelo nome:

```
public void run()
{
    double fee = calculateFee(noOfDays : 4);
    Console.WriteLine("Fee is {0}", fee);
}
```

- 15.** No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

O programa exibe as seguintes mensagens:

```
calculateFee using two optional parameters
Fee is 2000
```

Desta vez, o método *run* chamou a versão de *calculateFee* que aceita dois parâmetros opcionais. A chamada do método omitiu o primeiro parâmetro (*dailyRate*) e especificou o segundo parâmetro pelo nome. Essa versão do método *calculateFee* que aceita dois parâmetros opcionais é a única que coincide com a chamada.

Pressione qualquer tecla para fechar a janela do console e retornar ao Visual Studio.

- 16.** Modifique a implementação do método *calculateFee* que aceita dois parâmetros opcionais. Mude o nome do primeiro parâmetro para *theDailyRate* e atualize a instrução *return*, como mostrado em negrito no código a seguir:

```
private double calculateFee(double theDailyRate = 500.0, int noOfDays = 1)
{
    Console.WriteLine("calculateFee using two optional parameters");
    return theDailyRate * noOfDays;
}
```

- 17.** No método *run*, modifique a instrução que chama *calculateFee* e especifique o parâmetro *theDailyRate* pelo nome:

```
public void run()
{
    double fee = calculateFee(theDailyRate : 375.0);
    Console.WriteLine("Fee is {0}", fee);
}
```

- 18.** No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

O programa exibe as seguintes mensagens:

```
calculateFee using two optional parameters
Fee is 375
```

Quando você especificou a taxa, mas não a diária (passo 13), o método *run* chamou a versão de *calculateFee* que aceita um único parâmetro opcional. Desta vez, o método *run* chamou a versão de *calculateFee* que aceita dois parâmetros opcionais. Nesse caso, o uso de um argumento nomeado mudou o modo como o compilador resolve a chamada do método. Se você especificar um argumento nomeado, o compilador vai comparar o nome do argumento com os nomes dos parâmetros especificados nas declarações de métodos e selecionará o método que possui um parâmetro com um nome correspondente. Se você tivesse especificado o argumento como *aDailyRate: 375.0* na chamada ao método *calculateFee*, o programa não seria compilado, pois nenhuma versão do método tem um parâmetro que combine com esse nome.

Pressione qualquer tecla para fechar a janela do console e retornar ao Visual Studio.

Resumo

Neste capítulo, você aprendeu a definir métodos para implementar um bloco de código nomeado e examinou como passar parâmetros para os métodos e como retornar dados dos métodos. Viu também como chamar um método, passar argumentos e obter um valor de retorno. Além disso, aprendeu a definir métodos sobrecarregados com diferentes listas de parâmetros e constatou que o escopo de uma variável determina onde ela pode ser acessada. Depois, você utilizou o depurador do Visual Studio 2013 para passar pelo código ao longo de sua execução. Por fim, aprendeu a escrever métodos que aceitam parâmetros opcionais e a chamar métodos por meio de parâmetros nomeados.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 4, “Instruções de decisão”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar um método	Escreva o método dentro de uma classe. Especifique o nome do método, a lista de parâmetros e o tipo de retorno, seguidos do corpo do método entre chaves. Por exemplo: int addValues(int leftHandSide, int rightHandSide) { ... }
Retornar um valor de dentro de um método	Escreva uma instrução return dentro do método. Por exemplo: return leftHandSide + rightHandSide;
Retornar de um método antes do seu final	Escreva uma instrução return dentro do método. Por exemplo: return;
Chamar um método	Escreva o nome do método, junto com os argumentos entre parênteses. Por exemplo: addValues(39, 3);
Utilizar o assistente Generate Method Stub	Dê um clique com o botão direito em uma chamada para o método e, então, no menu de atalho, clique em Generate Method Stub.
Exibir a barra de ferramentas Debug	No menu View, aponte para Toolbars e clique em Debug.
Entrar em um método	Na barra de ferramentas Debug, clique em Step Into. ou No menu Debug, clique em Step Into.
Sair de um método	Na barra de ferramentas Debug, clique em Step Out. ou No menu Debug, clique em Step Out.
Especificar um parâmetro opcional para um método	Forneça um valor padrão para o parâmetro na declaração do método. Por exemplo: void optMethod(int first, double second = 0.0, string third = "Hello") { ... }
Passar um argumento de método como parâmetro nomeado	Especifique o nome do parâmetro na chamada do método. Por exemplo: optMethod(first : 100, third : "World");

CAPÍTULO 4

Instruções de decisão

Neste capítulo, você vai aprender a:

- Declarar variáveis booleanas.
- Utilizar os operadores booleanos para criar expressões cujo resultado é verdadeiro ou falso.
- Escrever instruções *if* para tomar decisões baseadas no resultado de uma expressão booleana.
- Escrever instruções *switch* para tomar decisões mais complexas.

O Capítulo 3, “Como escrever métodos e aplicar escopo”, mostrou como agrupar instruções relacionadas em métodos. Também ensinou como utilizar parâmetros para passar informações para um método e como fazer uso das instruções *return* para passar informações a partir de um método. Considera-se uma estratégia necessária a divisão de um programa em um conjunto de métodos distintos, cada um deles projetado para a execução de uma tarefa ou cálculo específico. Uma quantidade considerável de programas precisa resolver problemas complexos. Dividir um programa em métodos auxilia no entendimento desses problemas e ajuda a focar a solução de uma parte a cada vez.

Os métodos do Capítulo 3 são muito diretos, em que cada instrução é executada sequencialmente após o término da anterior. No entanto, para a resolução de diversos problemas do mundo real, também é necessário que você escreva um código que execute diferentes ações e que tomem diferentes caminhos em um método, dependendo das circunstâncias. Este capítulo mostra como realizar essa tarefa.

Declare variáveis booleanas

No mundo da programação em C#, tudo é preto ou branco, certo ou errado, verdadeiro ou falso. Por exemplo, se você criar uma variável inteira chamada *x*, atribuir o valor 99 a *x* e então perguntar se *x* contém o valor 99, a resposta será verdadeiro. Se você perguntar se *x* é menor que 10, a resposta será falso. Esses são exemplos de *expressões booleanas*. Uma expressão booleana sempre é avaliada como verdadeira ou falsa.



Nota As respostas a essas perguntas não são necessariamente definitivas para todas as outras linguagens de programação. Uma variável não atribuída contém um valor indefinido e você não pode, por exemplo, afirmar com precisão que ela é menor que 10. Questões como essas são uma fonte comum de erros nos programas em C e C++. O compilador do Microsoft Visual C# resolve esse problema assegurando que um valor seja sempre atribuído a uma variável antes de examiná-la. Se você tentar examinar o conteúdo de uma variável não atribuída, o programa não compilará.

O Visual C# fornece um tipo de dado chamado *bool*. Uma variável *bool* pode armazenar um dos dois valores: *verdadeiro* ou *falso*. Por exemplo, as três instruções a seguir declaram uma variável *bool* chamada *areYouReady*, atribuem o valor *true* a essa variável e então escrevem seu valor no console:

```
bool areYouReady;
areYouReady = true;
Console.WriteLine(areYouReady); // escreve True no console
```

Operadores booleanos

Um operador booleano é um operador que faz um cálculo cujo resultado é verdadeiro ou falso. O C# tem vários operadores booleanos muito úteis, e o mais simples deles é o operador *NOT*, representado pelo ponto de exclamação (!). O operador ! nega um valor booleano, resultando em valor oposto a esse. No exemplo anterior, se o valor da variável *areYouReady* fosse *true*, o valor da expressão *!areYouReady* seria *falso*.

Entenda os operadores de igualdade e relacionais

Dois operadores booleanos utilizados com frequência são os operadores de igualdade (==) e desigualdade (!=). Esses são operadores binários, os quais permitem determinar se um valor é igual a outro valor de mesmo tipo, produzindo um resultado booleano. A tabela a seguir resume como esses operadores funcionam, utilizando uma variável *int* chamada *age* como exemplo.

Operador	Significado	Exemplo	O resultado se age for 42
==	Igual a	age == 100	falso
!=	Diferente de	age = 0	verdadeiro

Não confunda o operador de *igualdade* == com o operador de *atribuição* =. A expressão *x==y* compara *x* com *y* e tem o valor *true* se os valores forem idênticos. A expressão *x=y* atribui o valor de *y* a *x* e retorna o valor de *y* como resultado.

Os operadores relacionais estão intimamente ligados aos operadores == e !=. Você utiliza esses operadores para descobrir se um valor é menor ou maior que outro do mesmo tipo. A tabela a seguir mostra como utilizar esses operadores.

Operador	Significado	Exemplo	O resultado se age for 42
<	Menor que	age < 21	falso
<=	Menor ou igual a	age <= 18	falso
>	Maior que	age > 16	verdadeiro
>=	Maior ou igual a	age >= 30	verdadeiro

Entenda os operadores lógicos condicionais

O C# também fornece dois outros operadores booleanos binários: o operador lógico AND, representado pelo símbolo `&&`, e o operador lógico OR, representado pelo símbolo `||`. Coletivamente, eles são conhecidos como os operadores lógicos condicionais. Seu propósito é combinar duas expressões ou valores booleanos em um único resultado booleano. Esses operadores são semelhantes aos operadores relacionais e de igualdade pelo fato de que o valor das expressões em que eles aparecem é verdadeiro ou falso, mas diferem pelo fato de que os valores em que eles operam também devem ser verdadeiros ou falsos.

O resultado do operador `&&` será *true* se e somente se as duas expressões booleanas que está avaliando forem *true*. Por exemplo, a instrução a seguir atribuirá o valor *true* a `validPercentage` se e somente se o valor de `percent` for maior ou igual a 0 e o valor de `percent` for menor ou igual a 100:

```
bool validPercentage;
validPercentage = (percent >= 0) && (percent <= 100);
```



Dica Um erro comum dos iniciantes é tentar combinar os dois testes nomeando a variável `percent` somente uma vez, como abaixo:

```
percent >= 0 && <= 100 // essa instrução não compilará
```

O uso de parênteses ajuda a evitar esse tipo de erro e também esclarece o objetivo da expressão. Por exemplo, compare

```
validPercentage = percent >= 0 && percent <= 100;
```

e:

```
validPercentage = (percent >= 0) && (percent <= 100)
```

Ambas as expressões retornam o mesmo valor, porque a precedência do operador `&&` é menor que a precedência dos operadores `>=` e `<=`. Mas a segunda expressão passa seu sentido de maneira mais legível.

O resultado do operador `||` será *true* se pelo menos uma das expressões booleanas que avalia for *true*. O operador `||` é utilizado para determinar se uma expressão de uma combinação de expressões booleanas é *true*. Por exemplo, a instrução a seguir atribuirá o valor *true* a `invalidPercentage` se o valor de `percent` for menor que 0 ou se o valor de `percent` for maior que 100:

```
bool invalidPercentage;
invalidPercentage = (percent < 0) || (percent > 100);
```

Curto-círcuito

Os operadores `&&` e `||` exibem um recurso chamado *curto-círcuito*. Às vezes, não é necessário avaliar os dois operandos ao determinar o resultado de uma expressão lógica condicional. Por exemplo, se o operando esquerdo do operador `&&` for avaliado como *false*, então o resultado da expressão inteira deve ser *false*, independentemente do valor do operando direito. De maneira semelhante, se o valor do operando esquerdo do operador `||` for avaliado como *true*, o resultado da expressão inteira deverá ser *true*, independentemente do valor do operando direito. Nesses casos, os operadores `&&` e `||` pulam a avaliação do operando direito. Eis alguns exemplos:

```
(percent >= 0) && (percent <= 100)
```

Nessa expressão, se o valor de `percent` for menor que 0, a expressão booleana do lado esquerdo de `&&` será avaliada como *false*. Esse valor significa que o resultado de toda a expressão deve ser *false*, e a expressão booleana à direita do operador `&&` não é avaliada.

```
(percent < 0) || (percent > 100)
```

Nessa expressão, se o valor de `percent` for menor que 0, a expressão booleana no lado esquerdo de `||` será avaliada como *true*. Esse valor significa que o resultado da expressão inteira deve ser *true* e a expressão booleana à direita do operador `||` não é avaliada.

Se projetar cuidadosamente as expressões que usam os operadores lógicos condicionais, você poderá aumentar o desempenho do seu código evitando trabalho desnecessário. Coloque expressões booleanas simples que possam ser avaliadas facilmente no lado esquerdo de um operador lógico condicional e as expressões mais complexas no lado direito. Em muitos casos, você perceberá que o programa não precisará avaliar as expressões mais complexas.

Um resumo da precedência e da associatividade dos operadores

A tabela a seguir resume a precedência e a associatividade de todos os operadores sobre os quais você aprendeu até aqui. Os operadores da mesma categoria têm a mesma precedência. Os operadores nas primeiras categorias da tabela têm precedência sobre os operadores nas últimas categorias.

Categoria	Operadores	Descrição	Associatividade
Primários	<code>()</code> <code>++</code> <code>--</code>	Anula a precedência Prefixo de incremento Prefixo de decremeno	Esquerda
Unários	<code>!</code> <code>+</code> <code>-</code> <code>++</code> <code>--</code>	NOT lógico Retorna o valor do operando inalterado Retorna o valor do operando negado Sufixo de incremento Sufixo de decremeno	Esquerda
Multiplicativos	<code>*</code> <code>/</code> <code>%</code>	Multiplicação Divisão Resto da divisão	Esquerda

Categoría	Operadores	Descrição	Associatividade
Aditivos	+	Adição	Esquerda
	-	Subtração	
Relacionais	<	Menor que	Esquerda
	<=	Menor ou igual a	
	>	Maior que	
	>=	Maior ou igual a	
Igualdade	==	Igual a	Esquerda
	!=	Diferente de	
AND condicional	&&	AND condicional	Esquerda
OR condicional		OR condicional	Esquerda
Atribuição	=	Atribui o operando da direta ao da esquerda e retorna o valor que foi atribuído	Direita

Observe que o operador `&&` e o operador `||` têm precedência diferente: `&&` é maior que `||`.

Instruções *if* para tomar decisões

Em um método, se você quiser escolher entre executar duas instruções diferentes com base no resultado de uma expressão booleana, utilize uma instrução *if*.

Entenda a sintaxe da instrução *if*

A sintaxe de uma instrução *if* é a seguinte (*if* e *else* são palavras-chave do C#):

```
if ( booleanExpression )
    statement-1;
else
    statement-2;
```

Se a *expressãoBoolena* for avaliada como *true*, a *instrução-1* será executada; caso contrário, a *instrução-2* será executada. A palavra-chave *else* e a *instrução-2* subsequentes são opcionais. Se não houver uma cláusula *else* e a *expressãoBoolena* for *false*, a execução continuará com o código que vem depois da instrução *if*. Além disso, observe que a expressão booleana deve ser colocada entre parênteses; caso contrário, o código não compilará.

Por exemplo, aqui está uma instrução *if* que incrementa uma variável representando o ponteiro de segundos de um cronômetro. (Os minutos são ignorados por enquanto.) Se o valor da variável *seconds* for 59, ela será redefinida para 0; caso contrário, será incrementada pelo operador `++`:

```
int seconds;
...
if (seconds == 59)
    seconds = 0;
else
    seconds++;
```

Somente expressões booleanas, por favor!

A expressão em uma instrução *if* deve estar entre parênteses. Além disso, ela deve ser uma expressão booleana. Em algumas outras linguagens – principalmente C e C++ –, você pode escrever uma expressão do tipo inteiro, e o compilador discretamente converterá o valor inteiro em *true* (não zero) ou *false* (0). O C# não suporta esse tipo de comportamento, e o compilador reporta um erro se uma expressão desse tipo for escrita.

Se você especificar accidentalmente o operador de atribuição (=) em vez do operador de teste de igualdade (==) em uma instrução *if*, o compilador C# perceberá seu erro e não compilará seu código, como no exemplo a seguir:

```
int seconds;
...
if (seconds = 59) // erro de tempo de compilação
...
if (seconds == 59) // ok
```

As atribuições accidentais são outra fonte de erros comum em programas C e C++, que convertem discretamente o valor atribuído (59) a uma expressão booleana (tudo diferente de zero é considerado verdadeiro), resultando na execução do código após a instrução *if* todas as vezes.

Ocasionalmente, uma variável booleana pode ser utilizada como a expressão para uma instrução *if*, embora ela ainda deva ser incluída entre parênteses, como mostrado neste exemplo:

```
bool inWord;
...
if (inWord == true) // ok, mas não é usado comumente
...
if (inWord) // mais comum e considerado um estilo melhor
```

Utilize blocos para agrupar instruções

Observe que a sintaxe da instrução *if* mostrada anteriormente especifica uma única instrução depois do *if (expressãoBoolena)* e uma única instrução depois da palavra-chave *else*. Às vezes você vai querer executar mais de uma instrução quando uma expressão booleana for verdadeira. Você poderia agrupar as instruções dentro de um novo método e então chamar o novo método, mas uma solução mais simples é agrupar as instruções dentro de um *bloco*. Um bloco é simplesmente uma sequência de instruções agrupadas entre uma chave de abertura e uma de fechamento.

No exemplo a seguir, duas instruções que definem a variável *seconds* como 0 e incrementam a variável *minutes* estão agrupadas em um bloco, e o bloco inteiro será executado se o valor de *seconds* for igual a 59:

```
int seconds = 0;
int minutes = 0;
...
```

```
if (seconds == 59)
{
    seconds = 0;
    minutes++;
}
else
{
    seconds++;
}
```



Importante Se as chaves forem omitidas, o compilador do C# associará apenas a primeira instrução (`seconds = 0;`) à instrução `if`. A instrução subsequente (`minutes++;`) não será reconhecida pelo compilador como parte da instrução `if` quando o programa for compilado. Além disso, quando o compilador alcançar a palavra-chave `else`, ele não a associará à instrução `if` anterior; em vez disso, informará um erro de sintaxe. Portanto, é uma boa prática sempre definir as instruções de cada desvio de uma instrução `if` dentro de um bloco, mesmo que o bloco consista em apenas uma instrução. Isso pode evitar sofrimento posteriormente, caso você queira adicionar mais código.

Um bloco também inicia um novo escopo. As variáveis podem ser definidas dentro de um bloco, mas elas desaparecerão no final do bloco. O fragmento de código a seguir ilustra esse ponto:

```
if (...)

{
    int myVar = 0;
    // myVar pode ser usada aqui
    ...
} // myVar desaparece aqui
else
{
    // myVar não pode ser usada aqui
    ...
}
// myVar não pode ser usada aqui
```

Instruções `if` em cascata

Você pode aninhar instruções `if` dentro de outras instruções `if`. Assim, pode encadear uma sequência de expressões booleanas, que são testadas uma após a outra até que uma delas seja avaliada como `true`. No exemplo a seguir, se o valor de `day` for 0, o primeiro teste será avaliado como `true` e `dayName` receberá a string "Sunday". Se o valor de `day` não for 0, o primeiro teste falhará e o controle passará para a cláusula `else`, que executa a segunda instrução `if` e compara o valor de `day` com 1. A segunda instrução `if` é executada somente se o primeiro teste for `false`. Da mesma forma, a terceira instrução `if` só será executada se o primeiro e o segundo testes forem `false`.

```
if (day == 0)
{
    dayName = "Sunday";
```

```
    }
    else if (day == 1)
    {
        dayName = "Monday";
    }
    else if (day == 2)
    {
        dayName = "Tuesday";
    }
    else if (day == 3)
    {
        dayName = "Wednesday";
    }
    else if (day == 4)
    {
        dayName = "Thursday";
    }
    else if (day == 5)
    {
        dayName = "Friday";
    }
    else if (day == 6)
    {
        dayName = "Saturday";
    }
else
{
    dayName = "unknown";
}
```

No exercício a seguir, você escreverá um método que utiliza uma instrução *if* em cascata para comparar duas datas.

Escreva instruções *if*

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto Selection, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 4\Windows X\Selection na sua pasta Documentos.
3. No menu Debug, clique em Start Debugging.

O Visual Studio 2013 compila e executa o aplicativo. O formulário exibe dois controles *DatePicker*, chamados *firstDate* e *secondDate*. Se estiver usando o Windows 8.1, os dois controles exibirão a data atual.

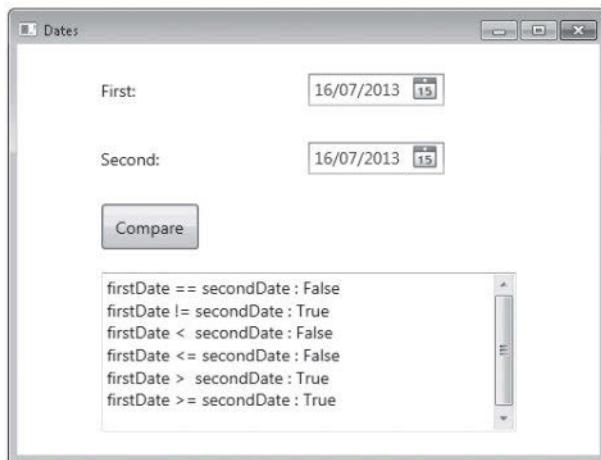
4. Se estiver usando Windows 7 ou Windows 8, clique no ícone de calendário do primeiro controle *DatePicker* e, então, clique na data atual. Repita essa operação para o segundo controle *DatePicker*.
5. Clique em Compare.

O texto a seguir é exibido na caixa de texto na metade inferior da janela:

```
firstDate == secondDate : False
firstDate != secondDate : True
firstDate < secondDate : False
firstDate <= secondDate : False
firstDate > secondDate : True
firstDate >= secondDate : True
```

A expressão booleana *firstDate == secondDate* deve ser *true* porque tanto *first* quanto *second* estão configurados com a data atual. De fato, somente o operador “menor que” e o operador “maior ou igual a” parecem funcionar corretamente. As imagens a seguir mostram as versões para Windows 8.1 e para Windows 7 do aplicativo em execução.





6. Retorne ao Visual Studio 2013. No menu Debug, clique em Stop Debugging (ou simplesmente feche o aplicativo, se estiver usando o Windows 7 ou o Windows 8).
7. Exiba o código de MainWindow.xaml.cs na janela Code and Text Editor.
8. Localize o método *compareClick*, que deve ser parecido com este:

```
private void compareClick(object sender, RoutedEventArgs e)
{
    int diff = dateCompare(first, second);
    info.Text = "";
    show("firstDate == secondDate", diff == 0);
    show("firstDate != secondDate", diff != 0);
    show("firstDate < secondDate", diff < 0);
    show("firstDate <= secondDate", diff <= 0);
    show("firstDate > secondDate", diff > 0);
    show("firstDate >= secondDate", diff >= 0);
}
```

Esse método é executado sempre que o usuário clica no botão Compare do formulário. As variáveis *first* e *second* contêm valores *DateTime*; elas são preenchidas com as datas exibidas nos controles *firstDate* e *secondDate* do formulário em outro lugar no aplicativo. *DateTime* é apenas mais um tipo de dado, como *int* ou *float*, exceto pelo fato de que contém subelementos com os quais é possível acessar as partes individuais de uma data, como ano, mês ou dia.

O método *compareClick* passa os dois valores *DateTime* para o método *dateCompare*. O objetivo desse método é comparar datas e retornar o valor *int* 0 se elas forem iguais, -1 se a primeira data for menor do que a segunda e +1 se a primeira for maior do que a segunda. Uma data é considerada maior do que outra se vem depois dela cronologicamente. Examinaremos o método *dateCompare* no próximo passo.

O método *show* exibe os resultados da comparação no controle caixa de texto *info* na metade inferior do formulário.

9. Localize o método *dateCompare*, que deve ser parecido com este:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    // TO DO
    return 42;
}
```

Atualmente, esse método retorna o mesmo valor sempre que é chamado – em vez de 0, -1 ou +1 –, independentemente dos valores de seus parâmetros. Isso explica por que o aplicativo não funciona conforme o esperado. Você precisa implementar a lógica nesse método para comparar duas datas de modo correto.

10. Remova o comentário `// TO DO` e a instrução `return` do método `dateCompare`.
11. Adicione as seguintes instruções mostradas em negrito ao corpo do método `dateCompare`:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    int result;

    if (leftHandSide.Year < rightHandSide.Year)
    {
        result = -1;
    }
    else if (leftHandSide.Year > rightHandSide.Year)
    {
        result = 1;
    }
}
```

Se a expressão `leftHandSide.Year < rightHandSide.Year` for `true`, a data em `leftHandSide` deve ser anterior à data em `rightHandSide`; portanto, o programa configura a variável `result` como `-1`. Caso contrário, se a expressão `leftHandSide.Year > rightHandSide.Year` for `true`, a data em `leftHandSide` deve ser posterior à data em `rightHandSide`; portanto, o programa configura a variável `result` como `1`.

Se a expressão `leftHandSide.Year < rightHandSide.Year` for `false` e a expressão `leftHandSide.Year > rightHandSide.Year` também for `false`, a propriedade `Year` das duas datas deve ser a mesma; portanto, o programa precisa comparar os meses em cada data.

12. Adicione as instruções a seguir mostradas em negrito ao corpo do método `dateCompare`, depois do código que você inseriu no passo anterior:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Month < rightHandSide.Month)
    {
        result = -1;
    }
    else if (leftHandSide.Month > rightHandSide.Month)
    {
        result = 1;
    }
}
```

Essas instruções seguem uma lógica para comparar meses, semelhante àquela utilizada para comparar anos no passo anterior.

Se a expressão `leftHandSide.Month < rightHandSide.Month` for `false` e a expressão `leftHandSide.Month > rightHandSide.Month` também for `false`, a propriedade `Month` das duas datas deve ser a mesma, assim o programa acaba precisando comparar o dia em cada data.

13. Adicione as seguintes instruções ao corpo do método `dateCompare`, depois do código que você inseriu nos dois passos anteriores:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Day < rightHandSide.Day)
    {
        result = -1;
    }
    else if (leftHandSide.Day > rightHandSide.Day)
    {
        result = 1;
    }
    else
    {
        result = 0;
    }

    return result;
}
```

Você já deve reconhecer o padrão nessa lógica.

Se `leftHandSide.Day < rightHandSide.Day` e `leftHandSide.Day > rightHandSide.Day` forem `false`, o valor nas propriedades `Day` nas duas variáveis deve ser o mesmo. Os valores `Month` e os valores `Year` também devem ser idênticos para que a lógica do programa chegue até esse ponto; portanto, as duas datas devem ser iguais, e o programa configura o valor de `result` como `0`.

A última instrução retorna o valor armazenado na variável `result`.

14. No menu Debug, clique em Start Debugging.

O aplicativo é recompilado e reiniciado. Se estiver usando Windows 7 ou Windows 8, configure os dois controles `DatePicker` com a data atual.

15. Clique em Compare.

O texto a seguir é exibido na caixa de texto:

```
firstDate == secondDate : True
firstDate != secondDate : False
firstDate < secondDate: False
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: True
```

Esses são os resultados corretos para datas idênticas.

16. Se estiver usando Windows 7 ou Windows 8, selecione um mês posterior para o controle *DatePicker secondDate*. Se estiver usando o Windows 8.1, use as setas suspensas para selecionar uma data posterior.
17. Clique em Compare.

O texto a seguir é exibido na caixa de texto:

```
firstDate == secondDate: False
firstDate != secondDate: True
firstDate < secondDate: True
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: False
```

Mais uma vez, esses são os resultados corretos quando a primeira data é anterior à segunda data.

18. Teste algumas outras datas e verifique se os resultados são os esperados. Volte ao Visual Studio 2013 e interrompa a depuração (ou feche o aplicativo, se estiver usando o Windows 7 ou o Windows 8) quando tiver terminado.

Comparação de datas em aplicativos do mundo real

Agora que vimos como utilizar uma série um tanto longa e complicada de instruções *if* e *else*, devo mencionar que essa não é a técnica que você empregaria para comparar datas em um aplicativo real. Se você examinar o método *dateCompare* do exercício anterior, verá que os dois parâmetros, *leftHandSide* e *rightHandSide*, são valores *DateTime*. A lógica escrita só compara a parte da data desses parâmetros, mas eles também contêm um elemento hora que não foi considerado (nem exibido). Para que dois valores *DateTime* sejam considerados iguais, eles não apenas devem ter a mesma data, mas também a mesma hora. Comparar datas e horas é uma operação tão comum que o tipo *DateTime* tem um método interno, chamado *CompareTo*, para fazer justamente isso: ele recebe dois argumentos *DateTime* e os compara, retornando um valor que indica se o primeiro argumento é menor que o segundo, caso em que o resultado será negativo; se o primeiro argumento é maior que o segundo, caso em que o resultado será positivo; ou se os dois argumentos representam a mesma data e hora, caso em que o resultado será 0.

Instruções *switch*

Algumas vezes, ao se escrever uma instrução *if* em cascata, cada uma das instruções *if* parecem iguais, porque todas avaliam uma expressão idêntica. A única diferença é que cada *if* compara o resultado da expressão com um valor diferente. Por exemplo, considere o seguinte bloco de código que utiliza uma instrução *if* para examinar o valor na variável *day* e calcular qual é o dia da semana:

```
if (day == 0)
{
    dayName = "Sunday";
```

```

    }
    else if (day == 1)
    {
        dayName = "Monday";
    }
    else if (day == 2)
    {
        dayName = "Tuesday";
    }
    else if (day == 3)
    {
        ...
    }
    else
    {
        dayName = "Unknown";
    }
}

```

Nessas situações, normalmente é possível reescrever a instrução *if* em cascata como uma instrução *switch*, para tornar o programa mais eficiente e legível.

Entenda a sintaxe da instrução *switch*

A sintaxe de uma instrução *switch* é a seguinte (*switch*, *case* e *default* são palavras-chave):

```

switch ( expressãoDeControle )
{
    case expressãoConstante :
        instruções
        break;
    case expressãoConstante :
        instruções
        break;
    ...
    default :
        instruções
        break;
}

```

A *expressãoDeControle*, que deve ser colocada entre parênteses, é avaliada uma vez. O controle passa então para o bloco de código identificado pela *expressãoConstante*, cujo valor é igual ao resultado da *expressãoDeControle*. (O identificador da *expressãoConstante* também é chamado de rótulo *case*.) A execução prossegue até a instrução *break* e, então, a instrução *switch* termina e o programa continua na primeira instrução depois da chave de fechamento da instrução *switch*. Se nenhum dos valores da *expressãoConstante* for igual ao valor da *expressãoDeControle*, as instruções abaixo do rótulo *default* opcional serão executadas.



Nota Cada valor da *expressãoConstante* deve ser único; assim, a *expressãoDeControle* só corresponderá a um deles. Se o valor da *expressãoDeControle* não corresponder a nenhum valor da *expressãoConstante* e não houver um rótulo *default*, a execução do programa continuará na primeira instrução após a chave de fechamento da instrução *switch*.

Portanto, você pode reescrever a instrução *if* em cascata anterior como a instrução *switch* a seguir:

```
switch (day)
{
    case 0 :
        dayName = "Sunday";
        break;
    case 1 :
        dayName = "Monday";
        break;
    case 2 :
        dayName = "Tuesday";
        break;
    ...
    default :
        dayName = "Unknown";
        break;
}
```

Siga as regras da instrução *switch*

A instrução *switch* é muito útil, mas, infelizmente, nem sempre você poderá utilizá-la como deseja. Todas as instruções *switch* que você escrever devem obedecer às seguintes regras:

- A instrução *switch* só pode ser utilizada em certos tipos de dados, como *int*, *char* ou *string*. Com qualquer outro tipo (incluindo *float* e *double*), você deve utilizar uma instrução *if*.
- Os rótulos *case* devem ser expressões constantes, como *42*, se o tipo de dado da instrução *switch* for *int*, '*'4'*', se for *char* ou "*'42'*", se for *string*. Se for necessário calcular valores dos rótulos *case* em tempo de execução, utilize uma instrução *if*.
- Os rótulos *case* devem ser expressões únicas. Ou seja, dois rótulos *case* não podem ter o mesmo valor.
- Você pode especificar que deseja executar as mesmas instruções para mais de um valor fornecendo uma lista de rótulos de caso sem nenhuma instrução no meio, caso em que o código do rótulo final na lista é executado para todas as instruções *case* nessa lista. Mas se um rótulo tiver uma ou mais instruções associadas, a execução não poderá prosseguir (*fall-through*) para os rótulos subsequentes; nesse caso, o compilador gerará um erro. O fragmento de código a seguir ilustra esses pontos:

```
switch (trumps)
{
    case Hearts :
    case Diamonds : // Fall-through allowed - não há nenhum código entre os rótulos
```

```

color = "Red";      // Código executado para Hearts e Diamonds
break;
case Clubs :
    color = "Black";
case Spades :        // Erro – código entre rótulos
    color = "Black";
    break;
}

```



Nota A instrução *break* é a maneira mais comum de parar um *fall-through*, mas você também pode usar uma instrução *return* para sair do método que contém a instrução *switch* ou uma instrução *throw*, para gerar uma exceção e abortar a instrução *switch*. A instrução *throw* será descrita no Capítulo 6, “Gerenciamento de erros e exceções”.

Regras de *fall-through* da instrução *switch*

Como não é possível passar accidentalmente de um rótulo *case* para outro se houver algum código no meio, você pode reorganizar livremente as seções de uma instrução *switch* sem afetar o significado (incluindo o rótulo *default* que, por convenção, em geral – mas não obrigatoriamente – é posicionado como o último rótulo).

Os programadores de C e C++ devem notar que a instrução *break* é obrigatória para cada *case* em uma instrução *switch* (mesmo o *case* padrão). Há uma razão para isso: em programas C ou C++, é comum a instrução *break* ser esquecida, permitindo que a execução prossiga (*faça fall-through*) para o próximo rótulo, originando erros difíceis de descobrir.

Se você quiser, pode simular o *fall-through* do C/C++ no C#, usando a instrução *goto* para ir para a instrução *case* seguinte ou para o rótulo *default*. Mas, em geral, o uso de *goto* não é recomendável, e este livro não demonstra como fazê-lo.

No próximo exercício, você completará um programa que lê os caracteres de uma string e mapeia cada caractere para sua representação XML. Por exemplo, o caractere de sinal de menor, <, tem um significado especial em XML (ele é utilizado para formar elementos). Se houver dados que contenham esse caractere, eles deverão ser convertidos na entidade de texto < para que um processador de XML saiba que são dados e não parte de uma instrução XML. Regras semelhantes se aplicam ao sinal de maior (>) e aos caracteres “e” comercial (&), aspa única (‘) e aspa dupla (“). Você escreverá uma instrução *switch* que testa o valor do caractere e captura os caracteres XML especiais como rótulos *case*.

Escreva instruções *switch*

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra o projeto SwitchStatement, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 4\Windows X\SwitchStatement na sua pasta Documentos.

3. No menu Debug, clique em Start Debugging.

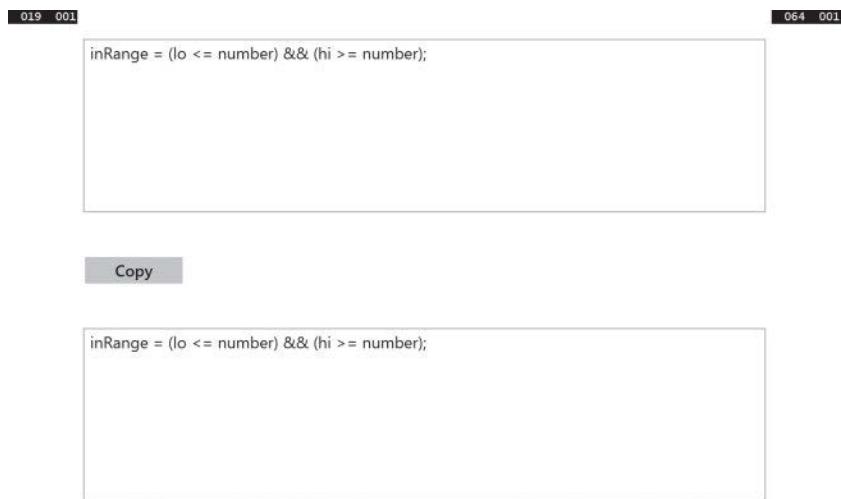
O Visual Studio 2013 compila e executa o aplicativo. O aplicativo exibe um formulário contendo duas caixas de texto separadas por um botão Copy.

4. Digite o seguinte texto de exemplo na caixa de texto superior.

inRange = (lo <= number) && (hi >= number);

5. Clique em Copy.

A instrução é copiada *ipsis litteris* para a caixa de texto inferior e não ocorre qualquer tradução dos caracteres <, & ou >, como se vê na captura de tela a seguir, que mostra a versão para Windows 8.1 do aplicativo.



6. Retorne ao Visual Studio 2013 e interrompa a depuração.
7. Exiba o código de MainWindow.xaml.cs na janela Code and Text Editor e localize o método *copyOne*.

O método *copyOne* copia o caractere especificado como o parâmetro de entrada no final do texto exibido na caixa de texto inferior. No momento, *copyOne* contém uma instrução *switch* com uma única ação *default*. Nos próximos passos, você modificará essa instrução *switch* para converter os caracteres que são significativos em XML para seu mapeamento XML. Por exemplo, o caractere < será convertido na string <.

- 8.** Adicione as instruções mostradas em negrito a seguir à instrução *switch*, depois da chave de abertura da instrução e imediatamente antes do rótulo *default*:

```
switch (current)
{
    case '<' :
        target.Text += "&lt;";
        break;
    default:
        target.Text += current;
        break;
}
```

Se o caractere que está sendo copiado for um sinal de menor (<), o código anterior acrescentará em seu lugar a string "<" ao texto que está sendo gerado.

- 9.** Adicione as seguintes instruções à instrução *switch*, depois da instrução *break* que você recém adicionou e acima do rótulo *default*:

```
case '>' :
    target.Text += "&gt;";
    break;
case '&' :
    target.Text += "&amp;";
    break;
case '\"' :
    target.Text += "&#34;";
    break;
case '\'' :
    target.Text += "&#39;";
    break;
```



Nota A aspa única (') e a aspa dupla (") têm um significado especial em C# – elas são utilizadas para delimitar constantes de caracteres e de string. A barra invertida (\) no final dos dois rótulos *case* é um caractere de escape que faz o compilador C# tratar esses caracteres como literais, em vez de como delimitadores.

- 10.** No menu Debug, clique em Start Debugging.
- 11.** Digite o texto a seguir na caixa de texto superior.

```
inRange = (lo <= number) && (hi >= number);
```

12. Clique em Copy.

A instrução é copiada na caixa de texto inferior. Desta vez, cada caractere submete-se ao mapeamento XML implementado na instrução *switch*. A caixa de texto de destino exibe o seguinte texto:

inRange = (lo <= number) && (hi >= number);

13. Teste outras strings e verifique se todos os caracteres especiais (<, >, &, " e ') são tratados corretamente.

14. Volte ao Visual Studio e interrompa a depuração (ou simplesmente feche o aplicativo, se estiver usando o Windows 7 ou o Windows 8).

Resumo

Neste capítulo, você conheceu as expressões e variáveis booleanas. Aprendeu a usar expressões booleanas com instruções *if* e *switch* para tomar decisões em seus programas e combinou expressões booleanas por meio de operadores booleanos.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 5, “Atribuição composta e instruções de iteração”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto	Exemplo
Determinar se dois valores são equivalentes	Utilize o operador == ou o operador !=.	answer == 42
Comparar o valor de duas expressões	Utilize o operador <, <=, > ou >=.	age >= 21
Declarar uma variável booleana	Utilize a palavra-chave bool como o tipo da variável.	bool inRange;
Criar uma expressão booleana que seja verdadeira somente se duas condições forem ambas verdadeiras	Utilize o operador &&.	inRange = (lo <= number) && (number <= hi);
Criar uma expressão booleana que seja verdadeira se uma de duas condições for verdadeira	Utilize o operador .	outOfRange = (number < lo) (hi < number);
Executar uma instrução se uma condição for verdadeira	Utilize uma instrução if.	if (inRange) process();

Para	Faça isto	Exemplo
Executar mais de uma instrução se uma condição for verdadeira	Utilize uma instrução if e um bloco.	<pre>if (seconds == 59) { seconds = 0; minutes++; }</pre>
Associar diferentes instruções a diferentes valores de uma expressão de controle	Utilize uma instrução switch.	<pre>switch (current) { case 0: ... break; case 1: ... break; default : ... break; }</pre>

CAPÍTULO 5

Atribuição composta e instruções de iteração

Neste capítulo, você vai aprender a:

- Atualizar o valor de uma variável utilizando os operadores de atribuição composta.
- Escrever instruções de iteração (ou repetição) *while*, *for* e *do*.
- Inspecionar uma instrução *do* passo a passo e ver como os valores de variáveis mudam.

O Capítulo 4, “Instruções de decisão”, mostrou como utilizar as construções *if* e *switch* a fim de executar instruções seletivamente. Este capítulo vai mostrar como usar várias instruções de iteração (*loop*) com a finalidade de executar uma ou mais instruções repetidamente.

Ao escrever as instruções de iteração, em geral, é preciso controlar o número de iterações que serão executadas. Isso é feito utilizando-se uma variável que atualiza seu valor a cada iteração feita e que para o processo quando a variável atinge um valor específico. Para auxiliar na simplificação desse processo, você aprenderá sobre operadores de atribuição especiais que precisam ser usados para a atualização do valor de uma variável nessas circunstâncias.

Operadores de atribuição composta

Você já sabe como usar os operadores matemáticos para criar novos valores. Por exemplo, a instrução a seguir utiliza o operador (+) para exibir no console um valor que é 42 unidades maior que a variável *answer*.

```
Console.WriteLine(answer + 42);
```

Você também aprendeu como usar as instruções de atribuição para alterar o valor de uma variável. A instrução a seguir usa o operador de atribuição (=) para alterar o valor da variável *answer* para 42:

```
answer = 42;
```

Se você quiser somar 42 ao valor de uma variável, pode combinar o operador de atribuição com o operador de adição. Por exemplo, a instrução a seguir adiciona 42 à variável *answer*. Depois da execução dessa instrução, o valor de *answer* será 42 unidades maior que o valor anterior:

```
answer = answer + 42;
```

Embora essa instrução funcione, você provavelmente nunca verá um programador experiente escrever um código assim. Adicionar um valor a uma variável é tão comum que o C# oferece um modo de executar essa tarefa de maneira mais rápida, utilizando o operador `+=`. Para adicionar 42 a *answer*, escreva esta instrução:

```
answer += 42;
```

Utilize essa notação para combinar qualquer operador aritmético com o operador de atribuição, como mostra a tabela a seguir. Esses operadores são conhecidos como *operadores de atribuição composta*.

Não escreva isto	Escreva isto
<code>variável = variável * número;</code>	<code>variável *= número;</code>
<code>variável = variável / número;</code>	<code>variável /= número;</code>
<code>variável = variável % número;</code>	<code>variável %= número;</code>
<code>variável = variável + número;</code>	<code>variável += número;</code>
<code>variável = variável - número;</code>	<code>variável -= número;</code>



Dica Os operadores de atribuição composta compartilham a mesma precedência e associatividade à direita dos operadores de atribuição simples correspondentes.

O operador `+=` também funciona em strings; ele anexa uma string ao final de outra. Por exemplo, o código a seguir exibe "Hello John" no console:

```
string name = "John";
string greeting = "Hello ";
greeting += name;
Console.WriteLine(greeting);
```

Você não pode utilizar outro operador de atribuição composta em strings.



Dica Utilize os operadores de incremento (`++`) e decremento (`--`) em vez de um operador de atribuição composta ao incrementar ou decrementar uma variável por 1. Por exemplo, substitua

```
count += 1;
```

por:

```
count++;
```

Escreva instruções *while*

Você utiliza uma instrução *while* para executar uma instrução repetidamente enquanto alguma condição se mantiver verdadeira. A sintaxe de uma instrução *while* é esta:

```
while ( booleanExpression )
    statement
```

A expressão booleana (que deve ser colocada entre parênteses) é avaliada e, se for verdadeira, a instrução é executada e a expressão booleana é então avaliada novamente. Se a expressão se mantiver verdadeira, a instrução será repetida e então a expressão booleana será avaliada ainda mais uma vez. Esse processo continua até que a expressão booleana seja avaliada como falsa; nesse ponto a instrução *while* termina. A execução continua então com a primeira instrução depois da instrução *while*. Uma instrução *while* compartilha as seguintes semelhanças sintáticas com uma instrução *if* (na verdade, a sintaxe é idêntica, só muda a palavra-chave):

- A expressão deve ser uma expressão booleana.
- A expressão booleana deve ser escrita entre parênteses.
- Se a expressão booleana for avaliada como falsa na primeira avaliação, a instrução não será executada.
- Se quiser executar duas ou mais instruções sob o controle de uma instrução *while*, você deve utilizar chaves para agrupar essas instruções em um bloco.

Observe uma instrução *while* que escreve os valores de 0 a 9 no console. Note que, assim que a variável *i* atinge o valor 10, a instrução *while* termina e o código presente no bloco de instruções não é executado:

```
int i = 0;
while (i < 10)
{
    Console.WriteLine(i);
    i++;
}
```

Todas as instruções *while* devem terminar em algum ponto. Um erro comum de iniciantes é esquecerem de incluir uma instrução para fazer a expressão booleana ser, por fim, avaliada como falsa e terminar o loop, o que resulta em um programa que é executado de modo contínuo. No exemplo, a instrução *i++*; desempenha esse papel.



Nota A variável *i* no loop *while* controla o número de iterações que ele executa. Essa expressão é comum e a variável que executa essa função é, às vezes, chamada de variável *sentinela*. Também é possível criar loops aninhados (um loop dentro de outro) e, nesses casos, é comum estender esse padrão de nomeação para usar as letras *j*, *k* e até *l* como nomes das variáveis sentinelas utilizadas para controlar as iterações nesses loops.

Dica Como nas instruções *if*, recomenda-se sempre utilizar um bloco com uma instrução *while*, mesmo que o bloco contenha apenas uma instrução. Assim, se depois você decidir adicionar mais instruções ao corpo da construção *while*, será claro que deve adicioná-las no bloco. Se você não fizer isso, somente a primeira instrução imediatamente após a expressão booleana na construção *while* será executada como parte do loop, resultando em erros difíceis de encontrar, como este:

```
int i = 0;
while (i < 10)
    Console.WriteLine(i);
    i++;
```

Esse código itera indefinidamente, exibindo um número infinito de zeros, pois somente a instrução *Console.WriteLine* — e não a instrução *i++* — é executada como parte da construção *while*.

No exercício a seguir, você escreverá um loop *while* para iterar pelo conteúdo de um arquivo de texto, uma linha de cada vez, e escreverá cada linha em uma caixa de texto de um formulário.

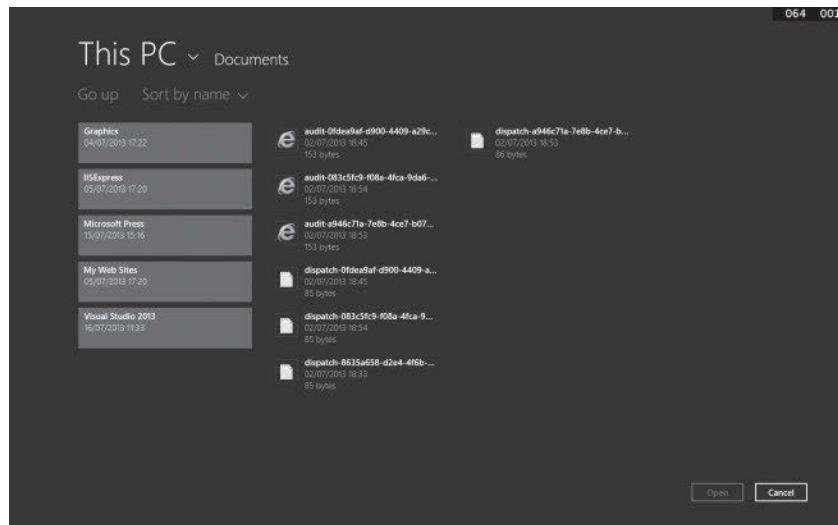
Escreva uma instrução while

1. Utilizando o Microsoft Visual Studio 2013, abra o projeto WhileStatement, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 5\Windows X\WhileStatement na sua pasta Documentos.
2. No menu Debug, clique em Start Debugging.

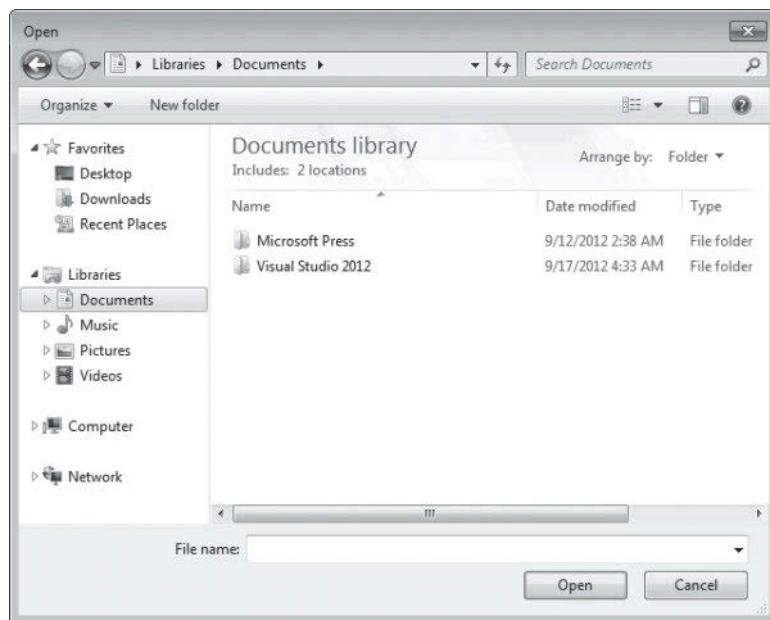
O Visual Studio 2013 compila e executa o aplicativo. O aplicativo é um visualizador simples de arquivo de texto que você pode utilizar para selecionar um arquivo de texto e exibir o conteúdo.

3. Clique em Open File.

Se estiver usando o Windows 8.1, o selecionador de arquivos Open aparecerá e exibirá os arquivos da pasta Documentos, como se vê na imagem a seguir (a lista de arquivos e pastas pode ser diferente em seu computador).



Se estiver usando o Windows 7, será exibida a caixa de diálogo Open, como esta:



Não importa o sistema operacional que esteja utilizado, com esse recurso é possível ir até uma pasta e selecionar um arquivo para exibir.

4. Abra a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 5\Windows X\WhileStatement\ WhileStatement na sua pasta Documentos.
5. Selecione o arquivo MainWindow.xaml.cs e clique em Open.

O nome do arquivo, MainWindow.xaml.cs, aparece na caixa de texto da parte superior do formulário, mas o conteúdo do arquivo não aparece na caixa de texto grande. Isso ocorre porque você ainda não implementou o código que lê e exibe o conteúdo do arquivo. Você adicionará essa funcionalidade nos próximos passos.

6. Volte ao Visual Studio 2013 e interrompa a depuração (ou feche o aplicativo, se estiver usando o Windows 7 ou o Windows 8).
7. Exiba o código do arquivo MainWindow.xaml.cs na janela Code and Text Editor e localize o método *openFileClick*.

Esse método é chamado quando o usuário clica no botão Open para selecionar um arquivo na caixa de diálogo Open. A maneira como esse método é implementado é diferente nas diversas versões do aplicativo. Neste ponto, não é necessário entender os detalhes exatos do funcionamento desse método – basta aceitar o fato de que ele solicita um arquivo para o usuário (com uma janela *FileOpenPicker* ou *OpenFileDialog*) e abre o arquivo selecionado para leitura. (Na versão para Windows 7, esse método simplesmente exibe a janela *OpenFileDialog* e, quando o usuário seleciona um arquivo, o método *openFileDialogFileOk* é executado; portanto, é esse método que realmente abre o arquivo para leitura.)

Mas as duas últimas instruções no método *openFileClick* (Windows 8.1) ou *openFileDialogFileOk* (Windows 7) são importantes. Na versão para Windows 8.1, o código é como este:

```
TextReader reader = new StreamReader(inputStreamAsStreamForRead());
displayData(reader);
```

A primeira instrução declara uma variável *TextReader* chamada *reader*. *TextReader* é uma classe disponibilizada pelo Microsoft .NET Framework que pode ser utilizada para ler fluxos de caracteres a partir de fontes como arquivos. Ela está localizada no namespace *System.IO*. Essa instrução torna os dados do arquivo especificado pelo usuário no *FileOpenPicker*, disponíveis para o objeto *TextReader*, o qual pode então ser utilizado para ler os dados do arquivo. A última instrução chama um método denominado *displayData*, passando *reader* como parâmetro para esse método. O método *displayData* lê os dados utilizando o objeto *reader* e os exibe na tela (ou fará isso, quando você tiver escrito o código necessário).

Na versão para Windows 7 do código, as instruções correspondentes são como segue:

```
TextReader reader = src.OpenText();
displayData(reader);
```

A variável *src* é um objeto *FileInfo* preenchido com informações sobre o arquivo selecionado pelo usuário com a janela *OpenFileDialog*. *FileInfo* é outra classe encontrada no .NET Framework e fornece o método *OpenText* para abrir um arquivo para leitura. A primeira instrução abre o arquivo selecionado pelo usuário para que a variável *reader* possa recuperar o conteúdo desse arquivo. Como

na versão para Windows 8.1 do código, a segunda instrução chama o método *displayData*, passando *reader* como parâmetro.

8. Examine o método *displayData*. Ele se parece com este nas duas versões:

```
private void displayData(TextReader reader)
{
    // TODO: adicionar o loop while aqui
}
```

Pode-se ver que, fora o comentário, esse método está atualmente vazio. É aí que você precisa adicionar o código para buscar e exibir os dados.

9. Substitua o comentário *// TODO: adicionar um loop while aqui* pela seguinte instrução:

```
source.Text = "";
```

A variável *source* refere-se à caixa de texto grande no formulário. A configuração de sua propriedade *Text* como uma string vazia ("") limpa todo o texto que é exibido atualmente nessa caixa de texto.

10. Adicione a seguinte instrução depois da linha anterior que você adicionou ao método *displayData*:

```
string line = reader.ReadLine();
```

Essa instrução declara uma variável *string* chamada *line* e chama o método *reader.ReadLine* para ler a primeira linha do arquivo nessa variável. Esse método retorna a próxima linha de texto do arquivo ou um valor especial chamado *null*, quando não há mais linhas para ler.

11. Adicione as seguintes instruções ao método *displayData*, depois do código que você acabou de inserir:

```
while (line != null)
{
    source.Text += line + '\n';
    line = reader.ReadLine();
}
```

Esse é um loop *while* que itera pelo arquivo uma linha por vez até que não haja linha alguma disponível.

A expressão booleana no início do loop *while* examina o valor da variável *line*. Se não for *null*, o corpo do loop exibirá a linha de texto anexando-a à propriedade *Text* da caixa de texto *source*, junto com um caractere de nova linha ('\n' – o método *ReadLine* do objeto *TextReader* exclui os caracteres de nova linha à medida que lê cada linha, portanto, o código precisa adicioná-lo novamente). O loop *while* lê então a próxima linha de texto antes de realizar a próxima iteração. O loop *while* termina quando não há mais texto para ler no arquivo, e o método *ReadLine* retorna um valor *null*.

- 12.** Se estiver usando o Windows 8.1, digite a seguinte instrução depois da chave de fechamento no fim do loop `while`:

```
reader.Dispose();
```

Se estiver usando o Windows 7 ou o Windows 8, digite a seguinte instrução:

```
reader.Close();
```

Essas instruções liberam os recursos associados ao arquivo e o fecham. Essa é uma boa prática, pois permite que outros aplicativos utilizem o arquivo, além de liberar memória e outros recursos utilizados para acessar o arquivo.

- 13.** No menu Debug, clique em Start Debugging.
14. Quando o formulário aparecer, clique em Open File.
15. No selecionador de arquivos Open ou na caixa de diálogo Open File, abra a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 5\Windows X\WhileStatement\WhileStatement na sua pasta Documentos, selecione o arquivo MainWindow.xaml.cs e clique em Open.



Nota Não tente abrir um arquivo que não contenha texto. Se você tentar abrir um programa executável ou um arquivo gráfico, por exemplo, o aplicativo simplesmente exibirá uma representação de texto da informação binária presente nesse arquivo. Se o arquivo for grande, poderá travar o aplicativo, exigindo que você o termine à força.

Desta vez, o conteúdo do arquivo selecionado aparece na caixa de texto – você deve reconhecer o código que esteve editando. A imagem a seguir mostra a versão para Windows 8.1 do aplicativo em execução; a versão para Windows 7 funciona da mesma maneira:

```

013 003
064 001

Open File
ileStatement - Complete\WhileStatement\MainWindow.xaml.cs

}

private void displayData(TextReader reader)
{
    source.Text = "";
    string line = reader.ReadLine();
    while (line != null)
    {
        source.Text += line + '\n';
        line = reader.ReadLine();
    }
    reader.Dispose();
}
}

```

16. Role pelo texto na caixa de texto e localize o método *displayData*. Verifique que esse método contém o código que você acabou de adicionar.
17. Volte ao Visual Studio e interrompa a depuração (ou feche o aplicativo, se estiver usando o Windows 7).

Escreva instruções *for*

No C#, a maioria das instruções *while* tem a seguinte estrutura geral:

```
inicialização
while (expressão booleana)
{
    instrução
    atualização da variável de controle
}
```

No C#, a instrução *for* fornece uma versão mais formal desse tipo de construção, combinando a inicialização, a expressão booleana e o código que atualiza a variável de controle. A instrução *for* é útil, pois é muito mais difícil sair accidentalmente do código que inicializa ou atualiza a variável de controle, de modo que é menos provável que você escreva código contendo loops infinitos. Observe a sintaxe da instrução *for*:

```
for (inicialização; expressão booleana; atualização da variável de controle)
    instrução
```

A instrução que forma o corpo da construção *for* pode ser uma única linha de código ou um bloco de código colocado entre chaves.

Você pode reformular o loop *while* mostrado anteriormente, que exibe os inteiros de 0 a 9, como o loop *for* a seguir:

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

A inicialização ocorre apenas uma vez, bem no início do loop. Portanto, se a expressão booleana for avaliada como *true*, a instrução será executada. A atualização da variável de controle ocorre e então a expressão booleana é reavaliada. Se a condição ainda for *true*, a instrução é executada novamente, a variável de controle é atualizada, a expressão booleana é avaliada mais uma vez e assim por diante.

Observe que a inicialização ocorre apenas uma vez e que a instrução no corpo do loop sempre é executada antes que a atualização se realize e que a atualização acontece antes de a expressão booleana ser reavaliada.



Dica Como na construção *while*, é considerada uma boa prática usar sempre um bloco de código, mesmo que o corpo do loop *for* seja composto de apenas uma instrução. Caso mais instruções sejam adicionadas ao corpo do loop *for* posteriormente, essa estratégia ajudará a garantir que o código seja sempre executado como parte de cada iteração.

Você pode omitir qualquer uma das três partes de uma instrução *for*. Se você omitir a expressão booleana, ela assumirá *true* por padrão; portanto, a instrução *for* a seguir é executada indefinidamente:

```
for (int i = 0; ; i++)
{
    Console.WriteLine("somebody stop me!");
}
```

Se você omitir as partes da inicialização e atualização, terá um loop *while* escrito de forma estranha:

```
int i = 0;
for (; i < 10; )
{
    Console.WriteLine(i);
    i++;
}
```



Nota As partes inicialização, expressão booleana e atualização da variável de controle de uma instrução *for* sempre devem ser separadas por ponto e vírgula, mesmo quando omitidas.

Também é possível fornecer várias inicializações e várias atualizações em um loop *for*. (Contudo, você pode ter somente uma expressão booleana.) Para conseguir isso, separe as várias inicializações e atualizações com vírgulas, como mostrado no exemplo a seguir:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
    ...
}
```

Como um último exemplo, observe o loop *while* do exercício anterior reescrito como um loop *for*:

```
for (string line = reader.ReadLine(); line != null; line = reader.ReadLine())
{
    source.Text += line + '\n';
}
```

Entendendo o escopo da instrução *for*

Talvez você tenha notado que é possível declarar uma variável na parte da inicialização de uma instrução *for*. Essa variável tem o escopo definido para o corpo da instrução *for* e desaparece quando a instrução *for* termina. Essa regra tem duas consequências importantes. Em primeiro lugar, você não pode utilizar essa variável após a instrução *for* ter terminado, porque ela não estará mais em escopo. Veja o exemplo:

```
for (int i = 0; i < 10; i++)
{
    ...
}
Console.WriteLine(i); // erro de tempo de compilação
```

Segundo, você pode escrever duas ou mais instruções *for* que reutilizam o mesmo nome de variável, porque cada variável está em um escopo diferente, como no código a seguir:

```
for (int i = 0; i < 10; i++)
{
    ...
}
for (int i = 0; i < 20; i += 2) // ok
{
    ...
}
```

Escreva instruções *do*

As instruções *while* e *for* testam suas expressões booleanas no início do loop. Isso significa que, se a expressão é avaliada como *false* no primeiro teste, o corpo do loop não é executado – nem mesmo uma vez. A instrução *do* é diferente: sua expressão booleana é avaliada após cada iteração e, portanto, o corpo sempre é executado ao menos uma vez.

A sintaxe da instrução *do* é a seguinte (não esqueça o ponto e vírgula final):

```
do
    instrução
while (expressãoBooleana);
```

Se o corpo do loop contiver mais de uma instrução, você deve utilizar um bloco de *instruções* (se não fizer isso, o compilador informará um erro de sintaxe). Aqui está uma versão do exemplo que escreve os valores de 0 a 9 no console, desta vez construída utilizando uma instrução *do*:

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
}
while (i < 10);
```

As instruções *break* e *continue*

No Capítulo 4, você viu como utilizar a instrução *break* para sair de uma instrução *switch*. Uma instrução *break* também pode ser utilizada para sair do corpo de uma instrução de iteração. Quando você sai de um loop, ele encerra imediatamente e a execução continua na primeira instrução após o loop. Nem a atualização nem a condição de continuação do loop são executadas novamente.

Por outro lado, a instrução *continue* faz o programa executar imediatamente a próxima iteração do loop (depois de avaliar de novo a expressão booleana). Observe uma versão do exemplo anterior que escreve os valores de 0 a 9 no console, desta vez utilizando as instruções *break* e *continue*:

```
int i = 0;
while (true)
{
    Console.WriteLine("continue " + i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

Esse código é medonho. Muitas diretrizes de programação recomendam a utilização cautelosa da instrução *continue* ou simplesmente não utilizá-la, porque ela está muitas vezes associada a um código difícil de entender. O comportamento de *continue* também é muito sutil. Por exemplo, se você executar uma instrução *continue* de dentro de uma instrução *for*, a parte da atualização será executada antes da execução da próxima iteração do loop.

No exercício a seguir, você vai escrever uma instrução *do* para converter um número inteiro decimal positivo na sua representação de string em notação octal. O programa se baseia no seguinte algoritmo, fundamentado em um procedimento matemático conhecido:

```
armazene o número decimal na variável dec
faça o seguinte
    divida dec por 8 e armazene o resto
    defina dec com o quociente do passo anterior
enquanto dec não é igual a zero
combine os valores armazenados para o resto em cada cálculo, em ordem inversa
```

Por exemplo, vamos supor que você queira converter o número decimal 999 em octal. Execute as seguintes etapas:

1. Divida 999 por 8. O quociente é 124 e o resto é 7.
2. Divida 124 por 8. O quociente é 15 e o resto é 4.
3. Divida 15 por 8. O quociente é 1 e o resto é 7.
4. Divida 1 por 8. O quociente é 0 e o resto é 1.

- 5.** Combine os valores calculados para o resto em cada etapa, em ordem inversa. O resultado é 1747. Essa é a representação octal do valor decimal 999.

Escreva uma instrução *do*

1. Utilizando Visual Studio 2013, abra o projeto DoStatement, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 5\Windows X\DoStatement na sua pasta Documentos.
2. Exiba o formulário MainWindow.xaml na janela Design View.

O formulário contém uma caixa de texto chamada *number*, na qual o usuário pode digitar um número decimal. Quando o usuário clicar no botão Show Steps, a representação octal do número inserido será gerada. A caixa de texto à direita, chamada *steps*, mostra os resultados de cada estágio do cálculo.

3. Exiba o código de MainWindow.xaml.cs na janela Code and Text Editor. Localize o método *showStepsClick*.

Esse método é executado quando o usuário clica no botão Show Steps do formulário. Atualmente, ele está vazio.

4. Adicione ao método *showStepsClick* as instruções que aparecem em negrito a seguir:

```
private void showStepsClick(object sender, RoutedEventArgs e)
{
    int amount = int.Parse(number.Text);
    steps.Text = "";
    string current = "";
}
```

A primeira instrução converte o valor da string na propriedade *Text* da caixa de texto *number* em um tipo *int* usando o método *Parse* do tipo *int* e o armazena em uma variável local, chamada *amount*.

A segunda instrução limpa o texto exibido na caixa de texto inferior, definindo sua propriedade *Text* como uma string vazia.

A terceira instrução declara uma variável *string* chamada *current* e a inicializa como a string vazia. Você vai usar essa string para armazenar os dígitos gerados em cada iteração do loop utilizado para converter o número decimal em sua representação octal.

5. Adicione a seguinte instrução *do*, que aparece em negrito, ao método *showStepsClick*:

```
private void showStepsClick(object sender, RoutedEventArgs e)
{
    int amount = int.Parse(number.Text);
    steps.Text = "";
    string current = "";

    do
    {
```

```

        int nextDigit = amount % 8;
        amount /= 8;
        int digitCode = '0' + nextDigit;
        char digit = Convert.ToChar(digitCode);
        current = digit + current;
        steps.Text += current + "\n";
    }
    while (amount != 0);
}

```

O algoritmo efetua repetidamente aritmética de inteiros para dividir a variável *amount* por 8 e determinar o resto. O resto depois de cada divisão sucessiva constitui o próximo dígito na string que está sendo construída. Por fim, quando *amount* é reduzida a 0, o loop termina. Observe que o corpo deve ser executado ao menos uma vez. Esse comportamento é exatamente o que é necessário, porque mesmo o número 0 tem um dígito octal.

Examine o código de forma mais minuciosa; você verá que a primeira instrução executada pelo loop *do* é esta:

```
int nextDigit = amount % 8;
```

Essa instrução declara uma variável *int* chamada *nextDigit* e a inicializa como o resto da divisão do valor em *amount* por 8. Isso será um número em algum lugar entre 0 e 7.

A próxima instrução no loop *do* é

```
amount /= 8;
```

Essa é uma instrução de atribuição composta e equivale a escrever *amount* = *amount* / 8;. Se o valor de *amount* for 999, o valor de *amount* depois da execução dessa instrução será 124.

A próxima instrução é esta:

```
int digitCode = '0' + nextDigit;
```

Essa expressão exige uma pequena explicação. Os caracteres têm um código único, de acordo com o conjunto de caracteres utilizado pelo sistema operacional. Nos conjuntos de caracteres frequentemente utilizados pelo sistema operacional Microsoft Windows, o código para o caractere "0" tem o valor inteiro 48. O código para o caractere "1" é 49, o código para o caractere "2" é 50 e assim por diante até o código para o caractere "9", que tem o valor inteiro 57. No C# é possível tratar um caractere como um inteiro e efetuar aritmética nele, mas, quando você faz isso, o C# utiliza o código do caractere como o valor. Portanto, a expressão '0' + *nextDigit* na verdade resultará em um valor em algum lugar entre 48 e 55 (lembre-se de que *nextDigit* estará entre 0 e 7), correspondente ao código para o dígito octal equivalente.

A quarta instrução no loop *do* é

```
char digit = Convert.ToChar(digitCode);
```

Essa instrução declara uma variável *char* chamada *digit* e a inicializa para o resultado da chamada do método *Convert.ToChar(digitCode)*. O método *Convert.ToChar* recebe um inteiro que contém um código de caractere e retorna o caracte-

tere correspondente. Assim, por exemplo, se *digitCode* tiver o valor 54, *Convert.ToString(digitCode)* retornará o caractere "6".

Resumindo, as três primeiras instruções no loop *do* determinaram o caractere que representa o dígito octal menos significativo (mais à direita) correspondente ao número digitado pelo usuário. A tarefa seguinte é incluir esse dígito no início da string a ser apresentada na saída, como segue:

```
current = digit + current;
```

A próxima instrução no loop *do* é esta:

```
steps.Text += current + "\n";
```

Essa instrução adiciona à caixa de texto *steps* a string que contém os dígitos produzidos até agora para a representação octal do número. A instrução também inclui um caractere de nova linha, para que cada estágio da conversão apareça em uma linha separada na caixa de texto.

Por fim, a condição na cláusula *while* no fim do loop é avaliada:

```
while (amount != 0);
```

Como o valor de *amount* ainda não é 0, o loop faz mais uma iteração.

No exercício final deste capítulo, você utilizará o depurador do Visual Studio 2013 para inspecionar passo a passo a instrução *do* anterior, para ajudá-lo a entender como ela funciona.

Inspecione passo a passo a instrução *do*

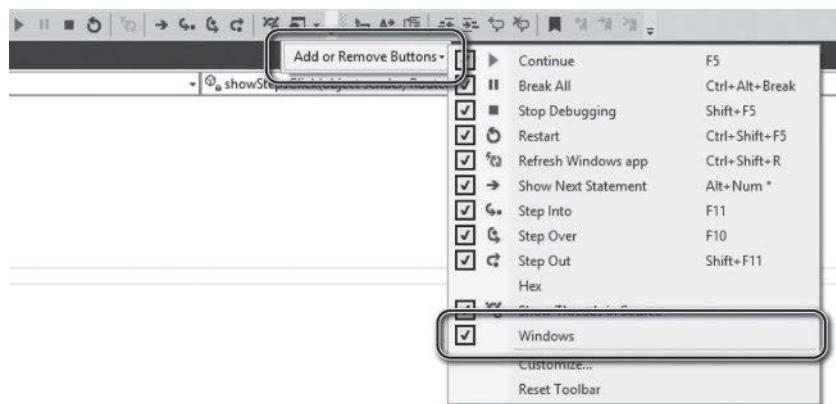
1. Na janela Code and Text Editor que exibe o arquivo *MainWindow.xaml.cs*, mova o cursor para a primeira instrução do método *showStepsClick*:

```
int amount = int.Parse(number.Text);
```

2. Clique com o botão direito do mouse em qualquer lugar da primeira instrução e, no menu de atalho que aparece, clique em Run To Cursor.
3. Quando o formulário aparecer, digite **999** na caixa de texto *number* à esquerda e, em seguida, clique em Show Steps.

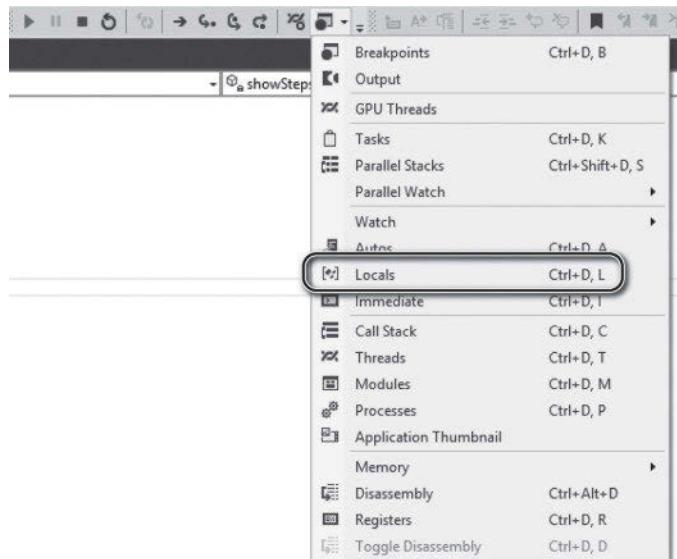
O programa para e você é colocado no modo de depuração do Visual Studio 2013. Uma seta amarela na margem esquerda da janela Code and Text Editor e o realce amarelo do código indicam a instrução atual.

4. Exiba a barra de ferramentas Debug, se ainda não estiver visível. No menu View, aponte para Toolbars e clique em Debug.
5. Na barra de ferramentas Debug, clique na seta suspensa, aponte para Add Or Remove Buttons e, em seguida, selecione Windows, como mostrado na imagem a seguir:



Essa ação adiciona o botão Breakpoints Window à barra de ferramentas.

6. Na barra de ferramentas Debug, clique no botão Breakpoints Window e então em Locals.



A janela Locals aparece (se ainda não estiver aberta). Essa janela exibe o nome, o valor e o tipo das variáveis locais no método atual, incluindo a variável local *amount*. Observe que o valor de *amount* no momento é 0:

The screenshot shows the Visual Studio IDE interface. The top part is the code editor with the file `MainWindow.xaml.cs` open. The code implements a `DoStatement` class with a `MainWindow` constructor and a `showStepsClick` method. The `showStepsClick` method takes an `object sender` and a `RoutedEventArgs e` as parameters. It initializes variables `amount`, `steps`, and `current`. It then enters a `do` loop where it calculates the next digit of `amount` using `amount % 8`, updates `amount` using `amount /= 8`, converts the digit to a character using `Convert.ToChar(digitCode)`, and appends it to `current`. The `steps` text is updated with `current` followed by a new line. The loop continues until `amount` is zero. The bottom part of the screenshot shows the `Locals` window with the following data:

Name	Type	Value
this	DoStatement.MainWindow	{DoStatement.MainWindow}
sender	Windows.UI.Xaml.Controls.Button	{Windows.UI.Xaml.Controls.Button}
amount	int	0
current	string	null

7. Na barra de ferramentas Debug, clique no botão Step Into.

O depurador executa a seguinte instrução:

```
int amount = int.Parse(number.Text);
```

O valor de `amount` na janela Locals muda para 999 e a seta amarela se move para a próxima instrução.

8. Clique novamente em Step Into.

O depurador executa esta instrução:

```
steps.Text = "";
```

Essa instrução não afeta a janela Locals porque `steps` é um campo do formulário e não uma variável local. A seta amarela se move para a próxima instrução.

9. Clique em Step Into.

O depurador executa a instrução mostrada aqui:

```
string current = "";
```

A seta amarela se move para a chave de abertura no início do loop *do*. O loop *do* contém três variáveis locais próprias: *nextDigit*, *digitCode* e *digit*. Observe que essas variáveis locais aparecem na janela Locals – o valor das três variáveis é inicialmente configurado como 0.

10. Clique em Step Into.

A seta amarela se move para a primeira instrução dentro do loop *do*.

11. Clique em Step Into.

O depurador executa a seguinte instrução:

```
int nextDigit = amount % 8;
```

O valor de *nextDigit* na janela Locals muda para 7. Esse é o resto depois da divisão de 999 por 8.

12. Clique em Step Into.

O depurador executa esta instrução:

```
amount /= 8;
```

O valor de *amount* muda para 124 na janela Locals.

13. Clique em Step Into.

O depurador executa esta instrução:

```
int digitCode = '0' + nextDigit;
```

O valor de *digitCode* na janela Locals muda para 55. Esse é o código do caractere "7" (48 + 7).

14. Clique em Step Into.

O depurador continua nesta instrução:

```
char digit = Convert.ToChar(digitCode);
```

O valor de *digit* muda para "7" na janela Locals. A janela Locals mostra valores *char* utilizando tanto o valor numérico subjacente (nesse caso, 55) como também a representação em caractere ("7").

Observe que, na janela Locals, o valor da variável *current* ainda é "".

15. Clique em Step Into.

O depurador executa a seguinte instrução:

```
current = current + digit;
```

O valor de *current* muda para "7" na janela Locals.

16. Clique em Step Into.

O depurador executa a instrução mostrada aqui:

```
steps.Text += current + "\n";
```

Essa instrução exibe o texto "7" na caixa de texto *steps*, seguido por um caractere de nova linha para fazer a saída subsequente ser exibida na próxima linha na caixa de texto. (O formulário atualmente está oculto atrás do Visual Studio; portanto, você não será capaz de vê-lo.) O cursor se desloca para a chave de fechamento no final do loop *do*.

17. Clique em Step Into.

A seta amarela se move para a instrução *while* para avaliar se o loop *do* foi concluído ou se deve continuar para outra iteração.

18. Clique em Step Into.

O depurador executa esta instrução:

```
while (amount != 0);
```

O valor de *amount* é 124 e a expressão *124!=0* é avaliada como *true*; portanto, o loop *do* faz outra iteração. A seta amarela retorna à chave de abertura no início do loop *do*.

19. Clique em Step Into.

A seta amarela se move novamente para a primeira instrução dentro do loop *do*.

20. Clique repetidamente em Step Into para investigar as três iterações seguintes do loop *do* e observe como os valores das variáveis mudam na janela Locals.**21.** No fim da quarta iteração do loop, o valor de *amount* agora é 0 e o valor de *current* é "1747". A seta amarela está na condição *while* no final do loop *do*:

```
while (amount != 0);
```

Como o valor de *amount* agora é 0, a expressão *amount!=0* é avaliada como *false* e o loop *do* termina.

22. Clique em Step Into.

O depurador executa a seguinte instrução:

```
while (amount != 0);
```

Conforme previsto, o loop *do* termina e a seta amarela se move para a chave de fechamento no final do método *showStepsClick*.

23. No menu Debug, clique em Continue.

O formulário aparece, exibindo as quatro etapas utilizadas para criar uma representação octal de 999: 7, 47, 747 e 1747.



24. Retorne ao Visual Studio 2013. No menu Debug, clique em Stop Debugging (ou feche o aplicativo, se estiver usando o Windows 7 ou o Windows 8).

Resumo

Neste capítulo, você aprendeu a utilizar os operadores de atribuição composta para atualizar variáveis numéricas. Você viu como é possível utilizar as instruções *while*, *for* e *do* para executar o código várias vezes, enquanto uma condição booleana for *true*.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 6, "Gerenciamento de erros e exceções".
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Adicionar uma quantidade a uma variável	Utilize o operador de adição composto. Por exemplo: <code>variable += amount;</code>
Subtrair uma quantidade de uma variável	Utilize o operador de subtração composto. Por exemplo: <code>variable -= amount;</code>
Executar uma ou mais instruções zero ou mais vezes, enquanto uma condição for verdadeira	Utilize uma instrução while. Por exemplo: <code>int i = 0; while (i < 10) { Console.WriteLine(i); i++; }</code> Como alternativa, utilize uma instrução for. Por exemplo: <code>for (int i = 0; i < 10; i++) { Console.WriteLine(i); }</code>
Executar repetidamente as instruções uma ou mais vezes	Utilize uma instrução do. Por exemplo: <code>int i = 0; do { Console.WriteLine(i); i++; } while (i < 10);</code>

CAPÍTULO 6

Gerenciamento de erros e exceções

Neste capítulo, você vai aprender a:

- Tratar exceções utilizando as instruções *try*, *catch* e *finally*.
- Controlar o overflow de números inteiros utilizando as palavras-chave *checked* e *unchecked*.
- Levantar exceções a partir de seus métodos utilizando a palavra-chave *throw*.
- Garantir que o código sempre execute, mesmo após a ocorrência de uma exceção, utilizando um bloco *finally*.

Até aqui, você aprendeu as principais instruções do C# necessárias para a execução das tarefas comuns, como escrever métodos, declarar variáveis, utilizar operadores para criar valores, escrever instruções *if* e *switch* para a execução de um código de modo seletivo e escrever instruções *while*, *for* e *do* para executar código repetidamente. Os capítulos anteriores, entretanto, não consideraram a possibilidade (ou probabilidade) de alguma coisa dar errado.

Garantir que o código sempre funcione conforme o esperado é muito difícil. As falhas podem acontecer por inúmeros motivos, e muitos deles estão além do seu controle como programador. Todos os aplicativos que você escrever devem ter a capacidade de detectar falhas e tratar delas elegantemente, executando as ações corretivas adequadas ou, se isso não for possível, informando as razões da falha de modo claro para o usuário. Neste último capítulo da Parte I, você vai aprender como o C# usa exceções para sinalizar uma falha e como utilizar as instruções *try*, *catch* e *finally* para capturar e lidar com os erros que essas exceções representam.

Por fim, você terá uma base sólida a respeito de todos os elementos fundamentais do C#, sobre a qual desenvolverá seu conhecimento na Parte II.

Lide com erros

Às vezes, coisas ruins acontecem e isso faz parte da vida. Pneus furam, baterias descarregam, ferramentas nunca ficam onde você as deixou e os usuários de seus aplicativos se comportam de maneira imprevisível. No mundo dos computadores, os discos rígidos estragam, outros aplicativos em execução no mesmo computador em que seu programa é executado consomem toda a memória disponível de modo descontrolado, conexões de rede sem fio desaparecem no momento mais inoportuno e até fenômenos naturais, como uma descarga elétrica, podem ter um impacto, se causarem queda de energia ou uma falha da rede. Erros podem ocorrer em praticamente

qualquer estágio da execução de um programa e muitos deles podem nem mesmo ser falha de seu aplicativo; portanto, como detectá-los e tentar se recuperar deles?

Ao longo dos anos, vários mecanismos foram criados. Uma estratégia típica, adotada por sistemas mais antigos, como o UNIX, envolvia determinar que o sistema operacional definisse uma variável global especial sempre que um método falhasse. Então, depois de cada chamada a um método, você verificava a variável global para ver se o método havia sido bem-sucedido. O C# e a maioria das outras linguagens modernas orientadas a objetos não tratam erros dessa maneira; é trabalhoso demais. Em vez disso, elas utilizam exceções. Se quiser escrever programas robustos em C#, você precisa conhecer as exceções.

Teste o código e capture as exceções

Os erros podem acontecer a qualquer momento, e o uso de técnicas tradicionais para adicionar manualmente um código de detecção de erro em torno de cada instrução é complicado, lento e propenso a erros. Você também pode perder de vista o fluxo principal de um aplicativo se cada instrução exigir uma lógica enrolada de tratamento de erros para gerenciar cada possível erro que ocorra em cada estágio. Felizmente, o C# facilita separar o código de tratamento do código que implementa a lógica principal de um programa, utilizando as exceções e as rotinas de tratamento de exceção. Para escrever programas compatíveis com exceções, você precisa fazer duas coisas:

- Escrever o código dentro de um bloco *try* (*try* é uma palavra-chave do C#). Quando o código executa, ele tenta executar todas as instruções no bloco *try* e, se nenhuma das instruções gerar uma exceção, todas serão executadas, uma após a outra, até a conclusão. Mas, se uma condição de erro ocorrer, a execução sai do bloco *try* e vai até outro fragmento de código projetado para capturar e tratar a exceção – uma rotina de tratamento *catch* (em inglês, *catch handler*).
- Escrever uma ou mais rotinas de tratamento *catch* (*catch* é outra palavra-chave do C#) imediatamente após o bloco *try* para tratar todas as condições de erro possíveis. Uma rotina de tratamento *catch* é concebida para capturar e tratar um tipo de exceção específico e você pode ter várias rotinas de tratamento *catch* depois de um bloco *try*, cada uma projetada para interceptar e processar uma exceção específica para que você possa fornecer diferentes rotinas de tratamento para os diferentes erros que possam surgir no bloco *try*. Se qualquer uma das instruções dentro do bloco *try* causar um erro, o runtime lançará uma exceção. O runtime examina então as rotinas de tratamento *catch* após o bloco *try* e transfere o controle diretamente para a primeira rotina de tratamento correspondente.

Observe um exemplo de código em um bloco *try* que tenta converter para valores inteiros as strings digitadas por um usuário em algumas caixas de texto de um formulário, chamar um método para calcular um valor e gravar o resultado em outra caixa de texto. Converter uma string em um número inteiro exige que a sequência contenha um conjunto de dígitos válidos e não alguma string arbitrária. Se a sequência contém caracteres inválidos, o método *int.Parse* lança uma *FormatException* e a execução é transferida para a rotina de tratamento *catch* correspondente. Quando a rotina de tratamento *catch* termina, o programa continua na primeira instrução após a rotina de tratamento. Observe que, se não houver uma rotina de tratamento que corresponda à exceção, diz-se que a exceção é não tratada (essa situação será descrita em breve).

```

try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (FormatException fEx)
{
    // Trata a exceção
    ...
}

```

Uma rotina de tratamento *catch* emprega uma sintaxe similar à utilizada por um parâmetro de método para especificar a exceção a ser capturada. No exemplo anterior, quando *FormatException* é lançada, a variável *fEx* é preenchida com um objeto que contém os detalhes da exceção.

O tipo *FormatException* possui várias propriedades que você pode examinar para determinar a causa exata da exceção. Muitas dessas propriedades são comuns a todas as exceções. Por exemplo, a propriedade *Message* contém uma descrição textual do erro que provocou a exceção. Você pode utilizar essas informações ao tratar a exceção, talvez gravando os detalhes em um arquivo de log ou exibindo uma mensagem significativa para o usuário e, depois, solicitando que ele tente novamente, por exemplo.

Exceções não tratadas

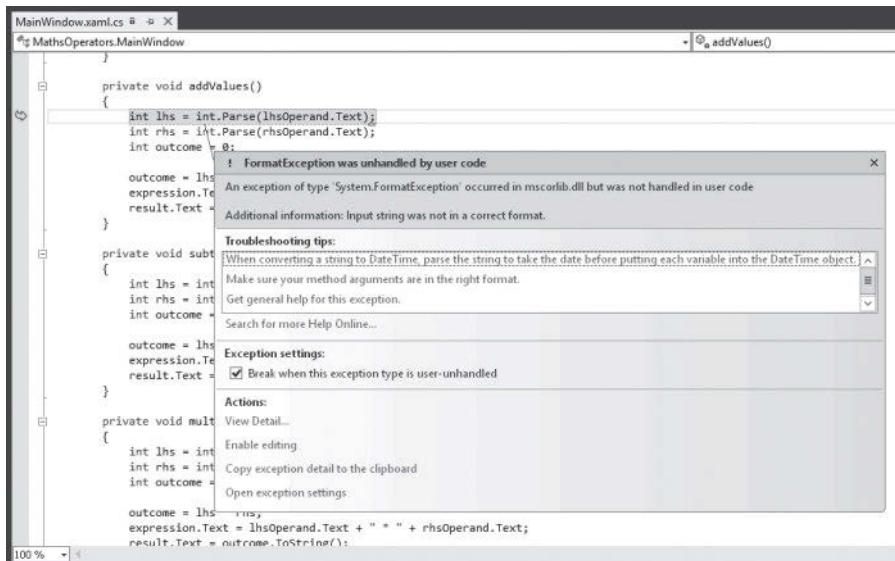
O que acontece se um bloco *try* lança uma exceção e não há uma rotina de tratamento *catch* correspondente? No exemplo anterior, é possível que a caixa de texto *lhsOperand* contenha a representação em string de um número inteiro válido, mas o inteiro representado esteja fora do intervalo de números inteiros válidos suportado pelo C# (por exemplo, "2147483648"). Nesse caso, a instrução *int.Parse* lança uma *OverflowException* que não será capturada pela rotina de tratamento *catch* de *FormatException*. Se isso ocorrer e o bloco *try* for parte de um método, este se encerrará imediatamente e a execução retornará ao método chamador. Se o método chamador utiliza um bloco *try*, o runtime tenta localizar e executar a rotina de tratamento *catch* correspondente para esse bloco *try*. Se o método chamador não utiliza um bloco *try* ou não há uma rotina de tratamento *catch* correspondente, o método chamador encerra imediatamente e a execução retorna ao seu chamador, onde o processo é repetido. Se uma rotina de tratamento *catch* correspondente é por fim encontrada, a rotina é executada e a execução continua na primeira instrução após a rotina de tratamento *catch* no método de captura.



Importante Observe que, após a captura de uma exceção, a execução continua no método que contém o bloco *catch* que a capturou. Se a exceção ocorreu em um método além daquele que contém a rotina de tratamento *catch*, o controle *não* retorna ao método que causou a exceção.

Se, após retornar pela cascata de métodos chamadores, o runtime for incapaz de encontrar uma rotina de tratamento *catch* correspondente, o programa terminará com uma exceção não tratada.

Você pode examinar facilmente as exceções geradas por seu aplicativo. Se você estiver executando o aplicativo no Microsoft Visual Studio 2013 no modo de depuração (isto é, você selecionou Start Debugging no menu Debug para executar o aplicativo) e uma exceção ocorrer, será exibida uma caixa de diálogo semelhante à da imagem a seguir e o aplicativo fará uma pausa para que você determine a causa da exceção:



O aplicativo é paralisado na instrução que causou a exceção e você entra no depurador. Você pode examinar e alterar os valores de variáveis e pode analisar seu código a partir do ponto em que a exceção ocorreu, utilizando a barra de ferramentas Debug e as várias janelas de depuração.

Utilize várias rotinas de tratamento *catch*

A discussão anterior destacou como diferentes erros lançam diferentes tipos de exceção para representar diferentes tipos de falhas. Para lidar com essas situações, você pode fornecer várias rotinas de tratamento *catch*, uma após a outra, como neste caso:

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (FormatException fEx)
{
    //...
}
catch (OverflowException oEx)
{
    //...
}
```

Se o código no bloco *try* lançar uma exceção *FormatException*, as instruções no bloco *catch* para a exceção *FormatException* serão executadas. Se o código lançar uma exceção *OverflowException*, o bloco *catch* para a exceção *OverflowException* será executado.



Nota Se o código no bloco *catch* de *FormatException* gerar uma exceção *OverflowException*, o bloco *catch* de *OverflowException* adjacente não será executado. Em vez disso, a exceção se propagará para o método que chamou o código, como descrito anteriormente nesta seção.

Capture múltiplas exceções

O mecanismo de captura de exceções fornecido pelo C# e pelo Microsoft .NET Framework é bem abrangente. O .NET Framework define vários tipos de exceções, e qualquer programa que você escreva poderá lançar a maioria delas. É bastante improvável que você queira escrever rotinas de tratamento *catch* para cada exceção possível que seu código possa lançar — lembre-se de que seu aplicativo deve ser capaz de tratar de exceções que você nem mesmo considerou ao escrevê-lo! Então, como você assegura que seus programas capturam e tratam todas as possíveis exceções?

A resposta a essa pergunta está na maneira como as diferentes exceções estão relacionadas entre si. As exceções são organizadas em famílias chamadas *hierarquias de herança*. (Você aprenderá sobre herança no Capítulo 12, “Herança”.) *FormatException* e *OverflowException* pertencem a uma família chamada *SystemException*, assim como várias outras exceções. *SystemException* é um membro de uma família maior, chamada *Exception*, que é a bisavó de todas as exceções. Se você capturar *Exception*, a rotina de tratamento capturará todas as exceções possíveis que possam ocorrer.



Nota A família *Exception* inclui uma ampla variedade de exceções, muitas delas planejadas para serem usadas por várias partes do .NET Framework. Algumas dessas exceções são um pouco esotéricas, mas ainda assim é útil entender como capturá-las.

O próximo exemplo mostra como capturar todas as possíveis exceções:

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (Exception ex) // esta é uma rotina de tratamento catch geral
{
    //...
}
```



Dica Se quiser capturar a exceção *Exception*, você pode omitir seu nome na rotina de tratamento *catch*, porque ela é a exceção padrão:

```
catch
{
    // ...
}
```

Mas isso não é recomendado. O objeto exceção passado para a rotina de tratamento *catch* contém informações úteis referentes à exceção, as quais não são facilmente acessíveis ao se utilizar essa versão da construção *catch*.

Neste ponto, há uma última pergunta que você deve estar fazendo: o que acontece se a mesma exceção corresponder a várias rotinas de tratamento *catch* no final de um bloco *try*? Se você capturar *FormatException* e *Exception* em duas rotinas de tratamento diferentes, qual delas será executada? (Ou ambas serão executadas?)

Quando ocorre uma exceção, o runtime utiliza a primeira rotina de tratamento correspondente à exceção que encontra e as outras são ignoradas. Isso significa que, se você colocar uma rotina de tratamento para *Exception* antes de uma rotina de tratamento para *FormatException*, a rotina de tratamento de *FormatException* nunca será executada. Portanto, você deve colocar as rotinas de tratamento *catch* mais específicas acima de uma rotina de tratamento *catch* geral, depois de um bloco *try*. Se as rotinas de tratamento *catch* específicas não correspondem à exceção, a rotina de tratamento *catch* geral corresponderá.

Nos exercícios a seguir, você vai ver o que acontece quando um aplicativo lança uma exceção não tratada e, em seguida, vai escrever um bloco *try*, além de capturar e tratar de uma exceção.

Observe como o Windows relata exceções não tratadas

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
 2. Abra a solução MathsOperators, localizada na pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 6\Windows X\MathsOperators` na sua pasta Documentos.
- Essa é uma versão do programa do Capítulo 2, “Variáveis, operadores e expressões”, que demonstra os diferentes operadores aritméticos.
3. No menu Debug, clique em Start Without Debugging.



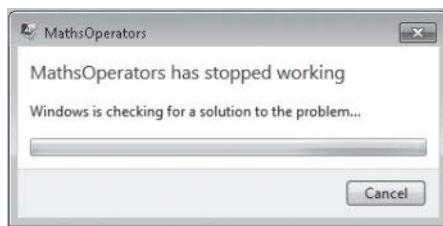
Nota Para este exercício, certifique-se de executar o aplicativo sem depurar, mesmo que esteja usando o Windows 8.1.

O formulário aparece. Agora você digitará na caixa Left Operand um texto que causará uma exceção. Essa operação demonstrará a falta de robustez da versão atual do programa.

4. Digite **John** na caixa Left Operand, digite **2** na caixa Right Operand, clique no botão + Addition e então clique em Calculate.

Essa ação aciona o tratamento de erros do Windows. Se estiver utilizando o Windows 8.1, o aplicativo terminará e você voltará para a tela Iniciar.

Se estiver utilizando a versão para Windows 7 do código, você deverá ver a seguinte caixa de mensagem:

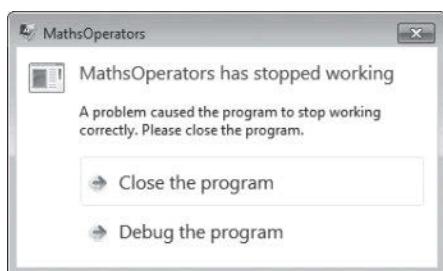


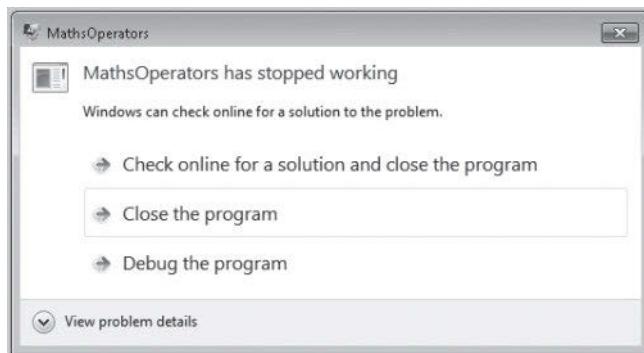
Após um curto espaço de tempo, essa caixa é seguida por outra caixa de diálogo reportando uma exceção não tratada:



Se clicar em Debug, você poderá lançar uma nova instância do Visual Studio sobre seu programa no modo de depuração, mas, por hora, clique em Close Program.

Você poderá ver uma das seguintes versões dessa caixa de diálogo, dependendo de como configurou o informe de problemas no painel de controle.





Se uma dessas caixas de diálogo aparecer, clique em Close The Program.

Além disso, deve ser exibida uma caixa de diálogo com a mensagem "Do you want to send information about the problem?". O Windows pode reunir informações sobre os aplicativos com falha e enviá-las para a Microsoft. Se essa caixa de diálogo aparecer, clique em Cancel.

Agora que você viu como o Windows captura e relata exceções não tratadas, o próximo passo é tornar o aplicativo mais robusto, tratando de entradas inválidas e impedindo a ocorrência de exceções não tratadas.

Escreva um bloco de instruções *try/catch*

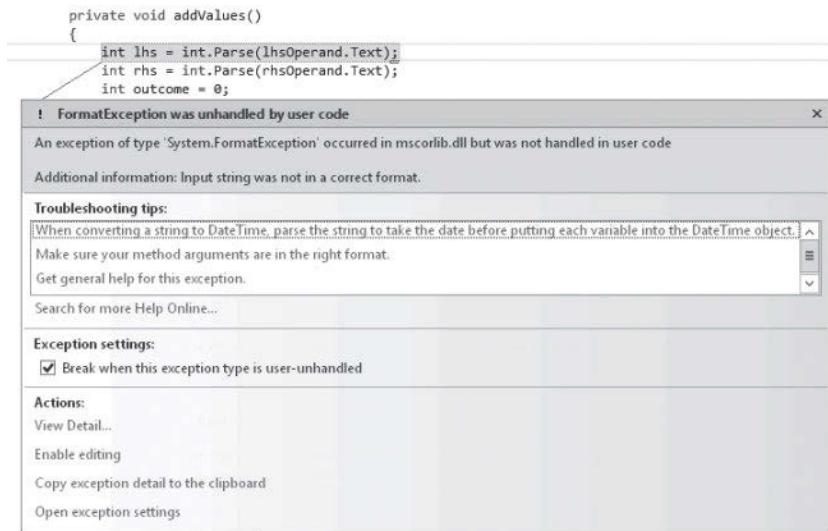
1. Retorne ao Visual Studio 2013.
2. No menu Debug, clique em Start Debugging.
3. Quando o formulário aparecer, digite **John** na caixa Left Operand, digite **2** na caixa Right Operand, clique no botão + Addition e então clique em Calculate.

Essa entrada deve causar a mesma exceção que ocorreu no exercício anterior, exceto que, agora, você está executando no modo de depuração, de maneira que o Visual Studio captura a exceção e a relata.



Nota Se aparecer uma caixa de mensagem informando que o modo break falhou porque o arquivo App.g.i.cs não pertence ao projeto que está sendo depurado, basta clicar em OK. Quando a caixa de mensagem desaparecer, a exceção será exibida.

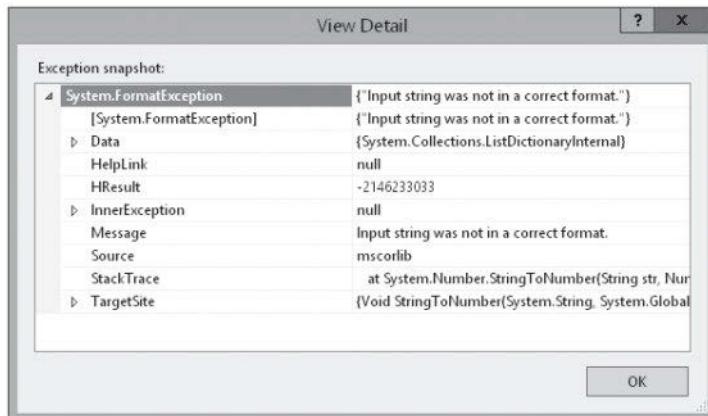
4. O Visual Studio exibe o seu código e destaca a instrução que causou a exceção, juntamente com uma caixa de diálogo que descreve essa exceção. Neste caso, a informação é a seguinte: "Input string was not in a correct format."



Você pode ver que a exceção foi lançada pela chamada a `int.Parse` dentro do método `addValues`. Esse método, porém, é incapaz de processar o texto "John" em um número válido.

5. Na caixa de diálogo de exceção, clique em View Detail.

Outra caixa de diálogo se abre, na qual é possível ver mais informações sobre a exceção. Se você expandir `System.FormatException`, poderá ver esta informação:





Dica Algumas exceções resultam de outras lançadas anteriormente. A exceção relatada pelo Visual Studio é apenas a última nesse encadeamento, mas, em geral, são as exceções anteriores que destacam a causa real do problema. Você pode examinar essas exceções anteriores expandindo a propriedade *InnerException* na caixa de diálogo View Detail. As exceções internas podem ter mais exceções internas, e você pode ir se aprofundando até encontrar uma exceção com a propriedade *InnerException* configurada como *null* (como mostrado na imagem anterior). Nesse ponto, você atingiu a exceção inicial e normalmente é essa exceção que precisa ser corrigida.

6. Clique em OK na caixa de diálogo View Detail e, então, no Visual Studio, no menu Debug, clique em Stop Debugging.
7. Exiba o código do arquivo MainWindow.xaml.cs na janela Code and Text Editor e localize o método *addValues*.
8. Adicione um bloco *try* (incluindo as chaves) em torno das instruções dentro desse método, junto com uma rotina de tratamento *catch* para a exceção *FormatException*, como mostrado no texto em negrito aqui:

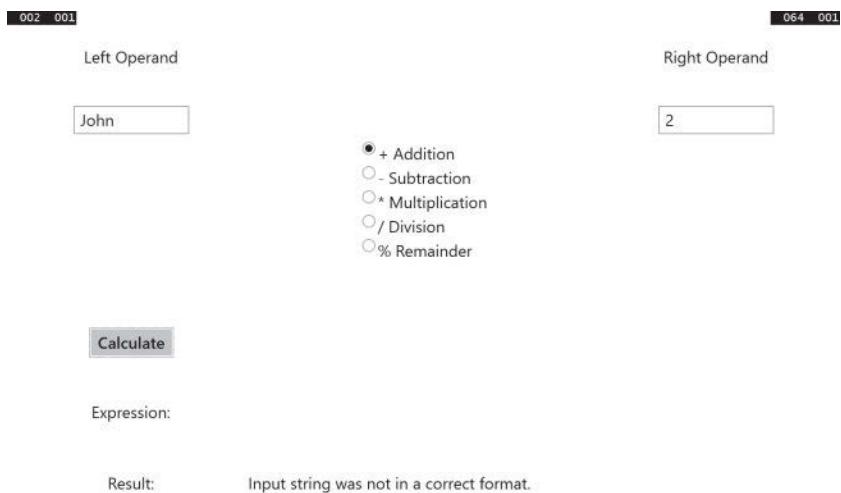
```
try
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;

    outcome = lhs + rhs;
    expression.Text = lhsOperand.Text + " + " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
catch (FormatException fEx)
{
    result.Text = fEx.Message;
}
```

Se ocorrer uma exceção *FormatException*, a rotina de tratamento *catch* exibirá na caixa de texto *result*, na parte inferior do formulário, o texto contido na propriedade *Message* da exceção.

9. No menu Debug, clique em Start Debugging.
10. Quando o formulário aparecer, digite **John** na caixa Left Operand, digite **2** na caixa Right Operand, clique no botão + Addition e então clique em Calculate.

A rotina de tratamento *catch* captura com sucesso a *FormatException* e a mensagem "Input string was not in a correct format" é escrita na caixa de texto Result. O aplicativo agora está um pouco mais robusto.



- 11.** Substitua John pelo número **10**, digite **Sharp** na caixa Right Operand e, então, clique em Calculate.

O bloco *try* encerra as instruções que processam as duas caixas de texto, de modo que a mesma rotina de tratamento de exceção trata os erros de entrada de usuário em ambas as caixas de texto.

- 12.** Na caixa Left Operand, substitua Sharp por **20**, clique no botão + Addition e então clique em Calculate.

Agora o aplicativo funciona como previsto e exibe o valor 30 na caixa Result.

- 13.** Na caixa Left Operand, substitua 10 por **John** e, então, clique no botão – Subtraction.

O Visual Studio aciona o depurador e relata novamente uma exceção *FormatException*. Desta vez, o erro ocorreu no método *subtractValues*, o qual não contém o processamento de *try/catch* necessário.

- 14.** No menu Debug, clique em Stop Debugging.

Propague exceções

A adição de um bloco *try/catch* ao método *addValues* tornou o método mais robusto, mas é preciso aplicar o mesmo tratamento de exceção aos outros métodos: *subtractValues*, *multiplyValues*, *divideValues* e *remainderValues*. O código para cada uma dessas rotinas de tratamento de exceção provavelmente será muito parecido, resultando na escrita do mesmo código em cada método. Cada um desses métodos é chamado pelo método *calculateClick* quando o usuário clica no botão Calculate. Portanto, para evitar a duplicação do código de tratamento de exceção, faz sentido transferi-lo para o método *calculateClick*. Se ocorrer uma *FormatException* em qualquer um dos métodos *subtractValues*, *multiplyValues*, *divideValues* e *remainderValues*, ela será propagada

retroativamente para tratamento no método *calculateClick*, conforme descrito na seção “Exceções não tratadas”, anteriormente neste capítulo.

Propague uma exceção de volta para o método chamador

1. Exiba o código do arquivo MainWindow.xaml.cs na janela Code and Text Editor e localize o método *addValues*.
2. Remova o bloco *try* e a rotina de tratamento *catch* do método *addValues* e retorne-o ao seu estado original, como mostrado no seguinte código:

```
private void addValues()
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int outcome = 0;

    outcome = lhs + rhs;
    expression.Text = lhsOperand.Text + " + " + rhsOperand.Text
    result.Text = outcome.ToString();
}
```

3. Localize o método *calculateClick*. Adicione a esse método o bloco *try* e a rotina de tratamento *catch* mostrados em negrito no exemplo a seguir:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
    try
    {
        if ((bool)addition.IsChecked)
        {
            addValues();
        }
        else if ((bool)subtraction.IsChecked)
        {
            subtractValues();
        }
        else if ((bool)multiplication.IsChecked)
        {
            multiplyValues();
        }
        else if ((bool)division.IsChecked)
        {
            divideValues();
        }
        else if ((bool)remainder.IsChecked)
        {
            remainderValues();
        }
    }
    catch (FormatException fEx)
    {
        result.Text = fEx.Message;
    }
}
```

4. No menu Debug, clique em Start Debugging.
5. Quando o formulário aparecer, digite **John** na caixa Left Operand, digite **2** na caixa Right Operand, clique no botão + Addition e então clique em Calculate.

Como antes, a rotina de tratamento *catch* captura com sucesso a exceção *FormatException* e a mensagem "Input string was not in a correct format" é escrita na caixa de texto Result. Contudo, lembre-se de que, na verdade, a exceção foi lançada no método *addValue*, mas capturada pela rotina de tratamento do método *calculateClick*.

6. Clique no botão – Subtraction e, em seguida, clique em Calculate.

Desta vez, o método *subtractValues* causa a exceção, mas ela é propagada de volta para o método *calculateClick* e tratada da mesma maneira que antes.

7. Teste os botões * Multiplication, / Division e % Remainder e verifique se a exceção *FormatException* é capturada e tratada corretamente.
8. Retorne ao Visual Studio e interrompa a depuração.



Nota A decisão de capturar explicitamente as exceções não tratadas em um método dependerá da natureza do aplicativo que você está compilando. Em alguns casos, faz sentido capturar exceções o mais próximo possível do ponto em que elas ocorrem. Em outras situações, é mais útil deixar que uma exceção se propague de volta ao método que invocou a rotina e lançou a exceção e tratar o erro ali.

Aritmética verificada e não verificada de números inteiros

O Capítulo 2 discutiu como utilizar os operadores aritméticos binários, como + e *, em tipos de dados primitivos, como *int* e *double*. Vimos também que os tipos de dados primitivos têm um tamanho fixo. Por exemplo, um *int* do C# tem 32 bits. Como *int* tem um tamanho fixo, você sabe exatamente o intervalo de valores que ele pode armazenar: de -2147483648 a 2147483647.



Dica Se quiser referenciar o valor mínimo ou máximo de *int* no código, utilize a propriedade *int.MinValue* ou *int.MaxValue*.

O tamanho fixo de um tipo *int* cria um problema. Por exemplo, o que acontece se você adicionar 1 a um *int* cujo valor é atualmente 2147483647? A resposta é que depende de como o aplicativo é compilado. Por padrão, o compilador C# gera um código que permite ao cálculo ter um overflow ("estouro") silencioso e você obtém uma resposta errada. (Na verdade, o cálculo excede para o valor inteiro negativo e o resultado gerado é -2147483648.) O motivo desse comportamento é o desempenho: a aritmética de números inteiros é uma operação comum em quase todos os progra-

mas e adicionar a sobrecarga de verificar o overflow em cada expressão de números inteiros pode levar a um desempenho muito deficiente. Em muitos casos, o risco é aceitável porque você sabe (ou espera!) que seus valores *int* não atinjam seus limites. Se não gostar dessa estratégia, você pode ativar a verificação de overflow.



Dica Você pode ativar e desativar a verificação de overflow no Visual Studio 2013 definindo as propriedades do projeto. No Solution Explorer, clique em *SeuProjeto* (onde *SeuProjeto* é o nome real de seu projeto). No menu Project, clique em *SeuProjeto Properties*. Na caixa de diálogo de propriedades do projeto, clique na guia Build. Clique no botão Advanced no canto inferior direito da página. Na caixa de diálogo Advanced Build Settings, marque ou desmarque a caixa de seleção Check For Arithmetic Overflow/Underflow.

Independentemente de como você compila um aplicativo, é possível utilizar as palavras-chave *checked* e *unchecked* para ativar e desativar seletivamente a verificação de overflow aritmético de inteiros nas partes de um aplicativo que você julgar necessário. Essas palavras-chave redefinem a opção de compilador especificada para o projeto.

Escreva instruções verificadas

Uma instrução verificada é um bloco precedido pela palavra-chave *checked*. Toda a aritmética de números inteiros em uma instrução verificada sempre lança uma *OverflowException* se um cálculo de inteiros no bloco sofrer overflow, como mostrado neste exemplo:

```
int number = int.MaxValue;
checked
{
    int willThrow = number++;
    Console.WriteLine("this won't be reached");
}
```



Importante Somente a aritmética de inteiros diretamente dentro do bloco *checked* está sujeita à verificação de overflow. Por exemplo, se uma das instruções verificadas for uma chamada de método, a verificação não se aplicará ao código que executa o método chamado.

Você também pode utilizar a palavra-chave *unchecked* para criar uma instrução de bloco *unchecked*. Toda a aritmética de inteiros em um bloco *unchecked* não é verificada e nunca lança uma *OverflowException*. Por exemplo:

```
int number = int.MaxValue;
unchecked
{
    int wontThrow = number++;
    Console.WriteLine("this will be reached");
}
```

Escreva expressões verificadas

Você também pode utilizar as palavras-chave *checked* e *unchecked* para controlar a verificação de overflow em expressões de números inteiros, precedendo apenas a expressão entre parênteses com a palavra-chave *checked* ou *unchecked*, como mostrado neste exemplo:

```
int wontThrow = unchecked(int.MaxValue + 1);
int willThrow = checked(int.MaxValue + 1);
```

Os operadores compostos (como `+ =` e `- =`) e os operadores de incremento, `++`, e de decremento, `--`, são operadores aritméticos e podem ser controlados com as palavras-chave *checked* e *unchecked*. Lembre-se de que, $x += y$; é o mesmo que $x = x + y$.



Importante Você não pode usar as palavras-chave *checked* e *unchecked* para controlar a aritmética de ponto flutuante (não inteiro). As palavras-chave *checked* e *unchecked* só se aplicam à aritmética de inteiros utilizando tipos de dados como `int` e `long`. A aritmética de ponto flutuante nunca lança uma *OverflowException* – nem mesmo quando você divide por 0.0. (Lembre-se, do Capítulo 2, que o .NET Framework tem uma representação de ponto flutuante especial para infinito.)

No próximo exercício, você verá como executar a aritmética verificada ao utilizar o Visual Studio 2013.

Utilize expressões verificadas

1. Retorne ao Visual Studio 2013.
 2. No menu Debug, clique em Start Debugging.
- Agora, você tentará multiplicar dois valores grandes.
3. Digite **9876543** na caixa Left Operand, digite **9876543** na caixa Right Operand, clique no botão * Multiplication e então clique em Calculate.

O valor **-1195595903** aparece na caixa Result do formulário. Esse é um valor negativo, que não pode estar correto. Esse valor é o resultado de uma operação de multiplicação que silenciosamente excedeu o limite de 32 bits do tipo `int`.

4. Retorne ao Visual Studio e interrompa a depuração.
5. Na janela Code and Text Editor que exibe `MainWindow.xaml.cs`, localize o método `multiplyValues`, que deve ser como este:

```
private void multiplyValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
```

```
        outcome = lhs * rhs;
        expression.Text = lhsOperand.Text + " * " + rhsOperand.Text;
        result.Text = outcome.ToString();
    }
```

A instrução `outcome = lhs * rhs;` contém a operação de multiplicação que causa overflow silenciosamente.

6. Edite essa instrução para que o valor do cálculo seja verificado, desta maneira:

```
outcome = checked(lhs * rhs);
```

A multiplicação agora é verificada e lançará uma `OverflowException`, em vez de retornar silenciosamente a resposta errada.

7. No menu Debug, clique em Start Debugging.

8. Digite **9876543** na caixa Left Operand, digite **9876543** na caixa Right Operand, clique no botão * Multiplication e então clique em Calculate.

O Visual Studio aciona o depurador e relata que a multiplicação resultou em uma exceção `OverflowException`. Agora, você precisa adicionar uma rotina de tratamento para capturar essa exceção e tratar dela de forma mais elegante do que apenas falhar com um erro.

9. No menu Debug, clique em Stop Debugging.

10. Na janela Code and Text Editor que exibe o arquivo `MainWindow.xaml.cs`, localize o método `calculateClick`.

11. Adicione a rotina de tratamento `catch` a seguir (mostrada em negrito), imediatamente após a rotina de tratamento `catch` de `FormatException` existente nesse método:

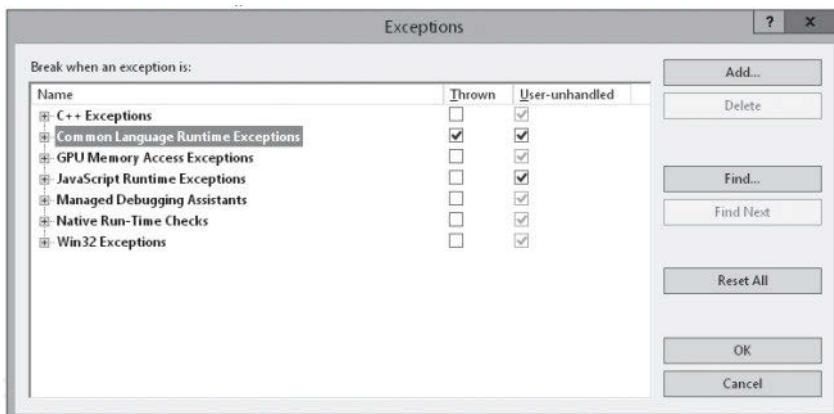
```
private void calculateClick(object sender, RoutedEventArgs e)
{
    try
    {
        ...
    }
    catch (FormatException fEx)
    {
        result.Text = fEx.Message;
    }
    catch (OverflowException oEx)
    {
        result.Text = oEx.Message;
    }
}
```

A lógica dessa rotina de tratamento `catch` é a mesma da rotina de tratamento `catch` de `FormatException`. Mas ainda vale a pena manter essas duas rotinas de tratamento separadas, em vez de simplesmente escrever uma rotina de tratamento `catch` de `Exception` genérica, porque talvez você queira tratar essas exceções de maneira diferente no futuro.

12. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
13. Digite **9876543** na caixa Left Operand, digite **9876543** na caixa Right Operand, clique no botão * Multiplication e então clique em Calculate.
A segunda rotina de tratamento *catch* captura com sucesso a *OverflowException* e exibe a mensagem "Arithmetic operation resulted in an overflow" na caixa Result.
14. Retorne ao Visual Studio e interrompa a depuração.

Tratamento de exceção e o Visual Studio Debugger

Por padrão, o Visual Studio Debugger só interrompe um aplicativo que está sendo depurado e relata as exceções não tratadas. Às vezes, é interessante depurar as próprias rotinas de tratamento de exceção e, nesse caso, você precisa seguir as exceções quando elas são lançadas pelo aplicativo, antes de serem capturadas. Essa funcionalidade pode ser habilitada facilmente. No menu Debug, clique em Exceptions. Na caixa de diálogo Exceptions, selecione a coluna Thrown de Common Language Runtime Exceptions e, em seguida, clique em OK:



Agora, quando ocorrerem exceções, como *OverflowException*, o Visual Studio acionará o depurador e você poderá utilizar o botão Step da barra de ferramentas Debug para percorrer passo a passo a rotina de tratamento *catch*. Se não quiser capturar todas as exceções Common Language Runtime (CLR) dessa maneira, você pode ser mais seletivo. Se expandir o nó Common Language Runtime Exceptions, poderá ver as diferentes categorias de exceções que podem ocorrer (elas estão organizadas por namespace). Se expandir qualquer namespace, você poderá ver as exceções individuais disponíveis e poderá selecionar cada uma delas individualmente.

Lance exceções

Suponha que você esteja implementando um método chamado *monthName* que aceita um único argumento *int* e retorna o nome do mês correspondente. Por exemplo, *monthName(1)* retorna “January”, *monthName(2)* retorna “February” e assim por diante. A pergunta é: o que o método deve retornar se o argumento inteiro for menor que 1 ou maior que 12? A melhor resposta é que o método não deve retornar coisa alguma – ele deve lançar uma exceção. As bibliotecas de classes do .NET Framework contêm uma grande quantidade de classes de exceção projetadas especificamente para situações desse tipo. Na maioria das vezes, você achará que uma dessas classes descreve sua condição excepcional. (Se não, você pode criar sua própria classe de exceção de modo fácil, mas antes disso precisa conhecer um pouco mais da linguagem C#.) Nesse caso, a classe *ArgumentOutOfRangeException* existente no .NET Framework serve perfeitamente. Você pode lançar uma exceção utilizando a instrução *throw*, como mostrado no exemplo a seguir:

```
public static string monthName(int month)
{
    switch (month)
    {
        case 1 :
            return "January";
        case 2 :
            return "February";
        ...
        case 12 :
            return "December";
        default :
            throw new ArgumentOutOfRangeException("Bad month");
    }
}
```

A instrução *throw* precisa de um objeto exceção para ser lançada. Esse objeto contém os detalhes da exceção, incluindo qualquer mensagem de erro. Esse exemplo utiliza uma expressão que cria um novo objeto *ArgumentOutOfRangeException*. O objeto é inicializado com uma string que preenche sua propriedade *Message* utilizando um construtor. Os construtores serão abordados detalhadamente no Capítulo 7, “Criação e gerenciamento de classes e objetos”.

Nos exercícios a seguir, você modificará o projeto MathsOperators para lançar uma exceção se o usuário tentar efetuar um cálculo sem especificar um operador.



Nota Esse exercício é um pouco artificial, pois qualquer bom projeto de aplicativo forneceria um operador padrão, mas o objetivo desse aplicativo é ilustrar esse ponto.

Lance uma exceção

1. Retorne ao Visual Studio 2013.
2. No menu Debug, clique em Start Debugging.

3. Digite **24** na caixa Left Operand, digite **36** na caixa Right Operand e então clique em Calculate.

Nada aparece nas caixas Expression e Result. O fato de você não ter selecionado uma opção de operador não fica claro imediatamente. Seria útil escrever uma mensagem de diagnóstico na caixa Result.

4. Retorne ao Visual Studio e interrompa a depuração.
5. Na janela Code and Text Editor que exibe MainWindow.xaml.cs, localize e examine o método *calculateClick*, que deve ser como este:

```
private int calculateClick(object sender, RoutedEventArgs e)
{
    try
    {
        if ((bool)addition.IsChecked)
        {
            addValues();
        }
        else if ((bool)subtraction.IsChecked)
        {
            subtractValues();
        }
        else if ((bool)multiplication.IsChecked)
        {
            multiplyValues();
        }
        else if ((bool)division.IsChecked)
        {
            divideValues();
        }
        else if ((bool)remainder.IsChecked)
        {
            remainderValues();
        }
    }
    catch (FormatException fEx)
    {
        result.Text = fEx.Message;
    }
    catch (OverflowException oEx)
    {
        result.Text = oEx.Message;
    }
}
```

Os campos *addition*, *subtraction*, *multiplication*, *division* e *remainder* são os botões de opção que aparecem no formulário. Cada botão tem uma propriedade chamada *IsChecked* que indica se o usuário a selecionou. *IsChecked* é uma propriedade booleana nullable que tem o valor *true* se o botão estiver selecionado ou *false*, caso contrário (você vai aprender mais sobre os valores nullable no Capítulo 8, "Valores e referências"). A instrução *if* em cascata examina cada botão para descobrir qual deles está selecionado. (Os botões de opção são mutuamente exclusivos; portanto, o usuário pode selecionar no máximo um botão de opção.) Se não houver botão selecionado, não haverá instruções *if* verdadeiras e os métodos de cálculo não serão chamados.

Você pode tentar resolver o problema adicionando mais uma instrução *else* à cascata *if-else*, para escrever uma mensagem na caixa de texto *result* do formulário, mas uma solução melhor é separar a detecção e a sinalização de um erro da captura e do tratamento desse erro.

6. Adicione outra instrução *else* no final da lista de instruções *if-else* e lance uma *InvalidOperationException*, como mostrado em negrito a seguir:

```
if ((bool)addition.IsChecked)
{
    addValues();
}

...
else if ((bool)remainder.IsChecked)
{
    remainderValues();
}
else
{
    throw new InvalidOperationException("No operator selected");
}
```

7. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
8. Digite **24** na caixa Left Operand, digite **36** na caixa Right Operand e então clique em Calculate.

O Visual Studio detecta que seu aplicativo lançou uma exceção *InvalidOperationException* e uma caixa de diálogo de exceção é aberta. Seu aplicativo lançou uma exceção, mas o código não a captura ainda.

9. No menu Debug, clique em Stop Debugging.

Agora que escreveu uma instrução *throw* e verificou que ela lança uma exceção, você escreverá uma rotina de tratamento *catch* para capturar essa exceção.

Capture a exceção

1. Na janela Code and Text Editor que exibe o arquivo MainWindow.xaml.cs, adicione a seguinte rotina de tratamento *catch*, mostrada em negrito, imediatamente abaixo das duas rotinas de tratamento *catch* existentes no método *calculateClick*:

```
...
catch (FormatException fEx)
{
    result.Text = fEx.Message;
}
catch (OverflowException oEx)
{
    result.Text = oEx.Message;
}
catch (InvalidOperationException ioEx)
{
    result.Text = ioEx.Message;
}
```

Esse código captura a *InvalidOperationException* que é lançada quando não há botão de operador selecionado.

2. No menu Debug, clique em Start Debugging.
3. Digite **24** na caixa Left Operand, digite **36** na caixa Right Operand e então clique em Calculate.

A mensagem “no operator selected” aparece na caixa Result.



Nota Se o Visual Studio Debugger for acionado, você provavelmente habilitou o Visual Studio para capturar exceções ao serem lançadas, conforme descrito anteriormente. Se isso acontecer, no menu Debug, clique em Continue. Quando terminar este exercício, lembre-se de desabilitar no Visual Studio a captura de exceções CLR quando são lançadas!

4. Retorne ao Visual Studio e interrompa a depuração.

Agora o aplicativo está muito mais robusto. Mas ainda podem surgir várias exceções que não são capturadas e farão o aplicativo falhar. Por exemplo, se você tentar dividir por 0, uma *DivideByZeroException* não tratada será lançada. (Divisão de inteiro por 0 lança uma exceção, diferentemente da divisão de ponto flutuante por 0.) Uma maneira de resolver esse problema é escrever um número ainda maior de rotinas de tratamento *catch* dentro do método *calculateClick*. Outra solução é adicionar no final da lista de rotinas de tratamento *catch* uma rotina de tratamento *catch* geral que capture *Exception*. Isso capturará todas as exceções inesperadas que possam ter sido esquecidas ou que possam ocorrer como resultado de circunstâncias realmente incomuns.



Nota O uso de uma rotina de tratamento genérica para capturar a exceção *Exception* não é justificativa para omitir a captura de exceções específicas. Quanto mais preciso você possa ser no tratamento de exceções, mais fácil será manter seu código e identificar as causas de qualquer problema obscuro ou recorrente. Só utilize a exceção *Exception* em casos realmente... bem, excepcionais. Para os propósitos do próximo exercício, a exceção “dividir por zero” cai nessa categoria. Contudo, tendo-se estabelecido que essa exceção é uma possibilidade diferenciada em um aplicativo profissional, uma boa prática seria adicionar uma rotina de tratamento para a exceção *DivideByZeroException* ao aplicativo.

Capture exceções não tratadas

1. Na janela Code and Text Editor que exibe o arquivo *MainWindow.xaml.cs*, adicione a seguinte rotina de tratamento *catch* no final da lista de rotinas de tratamento *catch* existentes no método *calculateClick*:

```
catch (Exception ex)
{
    result.Text = ex.Message;
}
```

Essa rotina de tratamento *catch* capturará todas as exceções não tratadas até aqui, qualquer que seja seu tipo específico.

- 2.** No menu Debug, clique em Start Debugging.

Agora você vai tentar efetuar alguns cálculos conhecidos por provocar exceções e confirmar se elas são tratadas corretamente.

- 3.** Digite **24** na caixa Left Operand, digite **36** na caixa Right Operand e então clique em Calculate.

Confirme que a mensagem de diagnóstico “no operator selected” ainda é exibida na caixa Result. Essa mensagem foi gerada pela rotina de tratamento *InvalidOperationException*.

- 4.** Digite **John** na caixa de texto Left Operand, clique no botão + Addition e depois clique em Calculate.

Confirme que a mensagem de diagnóstico “Input string was not in a correct format” é exibida na caixa Result. Essa mensagem foi gerada pela rotina de tratamento *FormatException*.

- 5.** Digite **24** na caixa Left Operand, digite **0** na caixa Right Operand, clique no botão / Division e então clique em Calculate.

Confirme que a mensagem de diagnóstico “Attempted to divide by zero” é exibida na caixa Result. Essa mensagem foi gerada pela rotina de tratamento *Exception* geral.

- 6.** Experimente outras combinações de valores e verifique que as condições de exceção são tratadas sem fazer o aplicativo falhar. Quando tiver terminado, retorne ao Visual Studio e interrompa a depuração.

Bloco **finally**

É importante lembrar que, quando uma exceção é lançada, ela altera o fluxo da execução no programa. Isso significa que você não pode garantir que uma instrução será sempre executada quando a instrução anterior terminar, porque a instrução anterior poderá lançar uma exceção. Lembre-se de que, nesse caso, após a execução da rotina de tratamento *catch*, o fluxo de controle é retomado na próxima instrução do bloco que contém essa rotina e não na instrução imediatamente após o código que lançou a exceção.

Observe o exemplo a seguir, adaptado do código do Capítulo 5, “Atribuição composta e instruções de iteração”. É muito fácil supor que a chamada a *reader.Dispose()* sempre ocorrerá quando o loop *while* terminar (se estiver usando o Windows 7 ou o Windows 8, substitua *reader.Dispose* por *reader.Close* nesse exemplo). Afinal de contas, é o que está no código.

```
TextReader reader = ...;
...
string line = reader.ReadLine();
while (line != null)
{
    ...
    line = reader.ReadLine();
}
reader.Dispose();
```

Algumas vezes, o fato de uma instrução específica não ser executada não é problema, mas em muitas ocasiões isso pode ser um grande problema. Se a instrução libera um recurso que foi adquirido em uma instrução anterior, então a falha na execução dessa instrução resultará na retenção do recurso. Este exemplo é precisamente o caso: quando um arquivo é aberto para leitura, essa operação adquire um recurso (um handle de arquivo) e você deve garantir a chamada de `reader.Dispose` para liberar o recurso (`reader.Close` chama `reader.Dispose` para fazer isso no Windows 7 e no Windows 8). Se você não fizer isso, cedo ou tarde não terá handles de arquivos suficientes e será incapaz de abrir mais arquivos. Se achar handles de arquivos muito triviais, pense, em vez disso, nas conexões de banco de dados.

A maneira de garantir que uma instrução seja sempre executada, quer uma exceção seja lançada ou não, é escrever essa instrução em um bloco `finally`. Um bloco `finally` ocorre imediatamente após um bloco `try` ou imediatamente após a última rotina de tratamento `catch`, depois de um bloco `try`. Desde que o programa entre no bloco `try` associado a um bloco `finally`, o bloco `finally` sempre será executado, mesmo que uma exceção ocorra. Se uma exceção for lançada e capturada localmente, a rotina de tratamento de exceção será executada primeiro, seguida pelo bloco `finally`. Se a exceção não for capturada localmente (ou seja, o runtime precisará pesquisar a lista de métodos chamadores para descobrir uma rotina de tratamento), o bloco `finally` será executado primeiro. Em qualquer caso, o bloco `finally` sempre é executado.

A solução para o problema da instrução `reader.Close` é a seguinte:

```
TextReader reader = ...;
...
try
{
    string line = reader.ReadLine();
    while (line != null)
    {
        ...
        line = reader.ReadLine();
    }
}
finally
{
    if (reader != null)
    {
        reader.Dispose();
    }
}
```

Mesmo que ocorra uma exceção durante a leitura do arquivo, o bloco `finally` garante que a instrução `reader.Dispose` sempre seja executada. Você verá outra maneira de tratar dessa situação no Capítulo 14, "Coleta de lixo e gerenciamento de recursos".

Resumo

Neste capítulo, você aprendeu a capturar e tratar exceções por meio das construções `try` e `catch`. Você viu como é possível ativar e desativar a verificação de overflow de números inteiros por meio das palavras-chave `checked` e `unchecked`. Aprendeu a lançar uma exceção se seu código detectar uma situação excepcional e examinou como

utilizar um bloco *finally* para garantir que o código crucial seja executado, mesmo se ocorrer uma exceção.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 7.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Capturar uma exceção específica	<p>Escreva uma rotina e tratamento catch que capture a classe de exceção específica. Por exemplo:</p> <pre>try { ... } catch (FormatException fEx) { ... }</pre>
Garantir que a aritmética de inteiros seja sempre verificada quanto a overflow	<p>Use a palavra-chave checked. Por exemplo:</p> <pre>int number = Int32.MaxValue; checked { number++; }</pre>
Lançar uma exceção	<p>Utilize uma instrução throw. Por exemplo:</p> <pre>throw new FormatException(source);</pre>
Capturar todas as exceções em uma única rotina de tratamento catch	<p>Escreva uma rotina de tratamento catch que capture Exception. Por exemplo:</p> <pre>try { ... } catch (Exception ex) { ... }</pre>
Garantir que algum código sempre seja executado, mesmo se uma exceção for lançada	<p>Escreva o código dentro de um bloco finally. Por exemplo:</p> <pre>try { ... } finally { // sempre executa }</pre>

Esta página foi deixada em branco intencionalmente.

PARTE II

O modelo de objetos do C#

CAPÍTULO 7	Criação e gerenciamento de classes e objetos	161
CAPÍTULO 8	Valores e referências	183
CAPÍTULO 9	Como criar tipos-valor com enumerações e estruturas	206
CAPÍTULO 10	Arrays.....	226
CAPÍTULO 11	Arrays de parâmetros	249
CAPÍTULO 12	Herança	261
CAPÍTULO 13	Como criar interfaces e definir classes abstratas.....	284
CAPÍTULO 14	Coleta de lixo e gerenciamento de recursos	313

Esta página foi deixada em branco intencionalmente.

CAPÍTULO 7

Criação e gerenciamento de classes e objetos

Neste capítulo, você vai aprender a:

- Definir uma classe contendo um conjunto de métodos e itens de dados relacionados.
- Controlar a acessibilidade de membros utilizando as palavras-chave *public* e *private*.
- Criar objetos utilizando a palavra-chave *new* para chamar um construtor.
- Escrever e chamar seus próprios construtores.
- Criar métodos e dados que podem ser compartilhados por todas as instâncias da mesma classe utilizando a palavra-chave *static*.
- Explicar como criar classes anônimas.

O Microsoft Windows Runtime do Windows 8 e Windows 8.1, junto com o Microsoft .NET Framework disponível no Windows 7, no Windows 8 e no Windows 8.1, contêm muitas classes, e você já utilizou muitas delas, inclusive *Console* e *Exception*. As classes apresentam um mecanismo conveniente para modelar as entidades manipuladas pelos aplicativos. Uma *entidade* pode representar um item específico, como um cliente, ou algo mais abstrato, como uma transação. Parte do processo do projeto de qualquer sistema está concentrada na determinação das entidades importantes para os processos implementados pelo sistema e na execução de uma análise para ver quais informações essas entidades necessitam armazenar e quais operações devem executar. Você armazena as informações de uma classe como campos e usa métodos para a implementação das operações que uma classe pode realizar.

Classificação

Classe é a raiz da palavra *classificação*. Ao projetar uma classe, você sistematicamente organiza as informações e o comportamento em uma entidade com significado. Essa organização é um ato de classificação e é algo que todos fazem – não apenas os programadores. Por exemplo, todos os carros compartilham comportamentos comuns (eles podem ser dirigidos, parados, acelerados, etc.) e atributos comuns (eles têm um volante, um motor, etc.). As pessoas utilizam a palavra *carro* para significar um objeto que compartilha esses comportamentos e atributos comuns. Desde que todos concordem com o que a palavra significa, esse sistema funcionará bem e você pode expressar ideias complexas, mas precisas, de maneira concisa. Sem a classificação, é difícil imaginar como as pessoas poderiam pensar ou se comunicar.

Como a classificação está profundamente arraigada na maneira como pensamos e nos comunicamos, faz sentido tentar escrever programas classificando os diferentes conceitos inerentes a um problema e sua solução e então modelar essas classes em uma linguagem de programação. Isso é exatamente o que você pode fazer com linguagens modernas de programação orientada a objetos, como o Microsoft Visual C#.

O objetivo do encapsulamento

O *encapsulamento* é um princípio importante durante a definição de classes. A ideia é que um programa que utiliza uma classe não precisa se preocupar com o modo como essa classe realmente funciona internamente; o programa simplesmente cria uma instância de uma classe e chama os métodos dessa classe. Desde que esses métodos façam o que se propõem a fazer, o programa não se preocupa com a maneira como eles são implementados. Por exemplo, ao chamar o método *Console.WriteLine*, você não quer se incomodar com todos os detalhes complicados de como a classe *Console* organiza fisicamente os dados a serem escritos na tela. Uma classe talvez precise manter todos os tipos de informações internas para executar seus vários métodos. Essas atividades e informações de estado adicionais são ocultas do programa que está utilizando a classe. Portanto, o encapsulamento é, às vezes, chamado de ocultamento de informação. O encapsulamento na realidade tem dois objetivos:

- Combinar os métodos e dados dentro de uma classe; ou seja, dar suporte à classificação.
- Controlar a acessibilidade de métodos e dados; ou seja, controlar o uso da classe.

Defina e utilize uma classe

No C#, você utiliza a palavra-chave *class* para definir uma nova classe. Os dados e os métodos da classe ocorrem no corpo da classe entre um par de chaves. A seguir está uma classe do C# chamada *Circle* que contém um método (para calcular a área do círculo) e uma parte de dados (o raio do círculo):

```
class Circle
{
    int radius;

    double Area()
    {
        return Math.PI * radius * radius;
    }
}
```



Nota A classe *Math* contém métodos para efetuar cálculos matemáticos e campos contendo constantes matemáticas. O campo *Math.PI* contém o valor 3.14159265358979323846, que é uma aproximação do valor de pi.

O corpo de uma classe contém métodos comuns (como *Area*) e campos (como *radius*). Lembre-se de que as variáveis em uma classe são chamadas de *campos*. O Capítulo 2, “Variáveis, operadores e expressões”, mostrou como declarar variáveis e o Capítulo 3, “Como escrever métodos e aplicar escopo”, demonstrou como escrever métodos, de modo que quase não há sintaxe nova aqui.

Você pode utilizar a classe *Circle* de modo semelhante a como usa outros tipos já encontrados. Você cria uma variável especificando *Circle* como seu tipo e inicializa a variável com algum dado válido. Veja um exemplo:

```
Circle c;           // Cria uma variável Circle
c = new Circle();  // Inicializa a variável
```

Um aspecto que merece destaque nesse código é o uso da palavra-chave *new*. Antes, ao inicializar uma variável como *int* ou *float*, você simplesmente atribuiu um valor a ela:

```
int i;
i = 42;
```

Você não pode fazer o mesmo com variáveis do tipo classe. Uma razão é que o C# não fornece uma sintaxe para atribuir valores literais de classe às variáveis. Você não pode escrever uma instrução como esta:

```
Circle c;
c = 42;
```

Afinal, o que significaria um *Circle* igual a 42? Outra razão diz respeito à maneira como a memória para variáveis do tipo classe é alocada e gerenciada pelo runtime – isso será discutido com mais detalhes no Capítulo 8, “Valores e referências”. Por enquanto, basta aceitar que a palavra-chave *new* cria uma nova instância de uma classe, normalmente chamada de *objeto*.

Mas você pode atribuir diretamente uma instância de uma classe a outra variável do mesmo tipo, assim:

```
Circle c;
c = new Circle();
Circle d;
d = c;
```

Mas isso não é tão simples e direto quanto parece ser à primeira vista, por razões que abordaremos no Capítulo 8.



Importante Não confunda os termos *classe* e *objeto*. Uma classe é a definição de um tipo. Um objeto é uma instância desse tipo, criada quando o programa é executado. Vários objetos podem ser instâncias da mesma classe.

Controle a acessibilidade

Surpreendentemente, a classe *Circle* não tem, hoje, qualquer utilidade prática. Por padrão, quando você encapsula seus métodos e dados dentro de uma classe, a classe forma um limite para o mundo externo. Campos (como *radius*) e métodos (como *Area*) definidos na classe podem ser vistos por outros métodos dentro da classe, mas não pelo mundo externo – eles são privados para a classe. Assim, embora seja possível criar um objeto *Circle* em um programa, não é possível acessar seu campo *radius* ou chamar seu método *Area*, razão pela qual a classe não é muito útil – ainda! Mas você pode modificar a definição de um campo ou método com as palavras-chave *public* e *private* para controlar se ele pode ou não ser acessado de fora:

- Dizemos que um método ou campo é privado se ele é acessível somente a partir de dentro da classe. Para declarar que um método ou campo é privado, você escreve a palavra-chave *private* antes da sua declaração. Conforme anunciado antes, esse é de fato o padrão, mas é uma boa prática determinar explicitamente que campos e métodos são privados para evitar qualquer confusão.
- Dizemos que um método ou campo é público se ele é acessível tanto de dentro quanto de fora da classe. Para declarar que um método ou campo é público, você escreve a palavra-chave *public* antes da sua declaração.

Veja a classe *Circle* novamente. Desta vez, *Area* é declarada como um método público e *radius* é declarado como um campo privado:

```
class Circle
{
    private int radius;

    public double Area()
    {
        return Math.PI * radius * radius;
    }
}
```



Nota Se você é programador de C++, note que não há um sinal de dois-pontos após as palavras-chave *public* ou *private*. Você precisa repetir a palavra-chave para cada declaração de campo e método.

Embora *radius* seja declarado como um campo privado e não esteja acessível fora da classe, *radius* estará acessível a partir de dentro da classe *Circle*. O método *Area* está dentro da classe *Circle*; portanto, o corpo de *Area* tem acesso a *radius*. Entretanto, a classe ainda é de valor limitado, pois não há como inicializar o campo *radius*. Para corrigir isso, você utiliza um construtor.



Dica De modo diferente das variáveis declaradas em um método, os campos em uma classe são automaticamente inicializados como *0*, *false* ou *null*, dependendo do seu tipo. Mas ainda é uma boa prática fornecer uma maneira explícita de inicializar os campos.

Convenção de nomes e acessibilidade

Muitas organizações têm seu próprio estilo, que solicitam aos desenvolvedores seguir, ao escreverem código. Parte desse estilo muitas vezes envolve regras para dar nomes a identificadores, e o objetivo dessas regras em geral é ajudar na manutenção do código. As recomendações a seguir são razoavelmente comuns e relacionam-se às convenções de nomes para campos e métodos com base na acessibilidade dos membros da classe; contudo, o C# não impõe essas regras:

- Os identificadores que são *public* devem iniciar com uma letra maiúscula. Por exemplo, *Area* começa com *A* (não com *a*) porque é *public*. Esse sistema é conhecido como esquema de nomes *PascalCase* (porque foi utilizado principalmente na linguagem Pascal).
- Os identificadores que não são *public* (que incluem as variáveis locais) devem começar com uma letra minúscula. Por exemplo, *radius* começa com *r* (não com *R*) porque é *private*. Esse sistema é conhecido como *camelo* (*camelCase*).



Nota Algumas organizações utilizam o sistema camelo somente para métodos e adotam a convenção de que os nomes dos campos privados começam com um caractere de sublinhado, como *_radius*. Contudo, os exemplos deste livro utilizam o sistema camelo para métodos e campos privados.

Só há uma exceção a essa regra: os nomes de classes devem iniciar com uma letra maiúscula e os construtores devem corresponder exatamente ao nome de suas classes; portanto, um construtor *private* deve iniciar com uma letra maiúscula.



Importante Não declare dois membros de classe *public* cujos nomes diferem apenas pelo uso de maiúsculas e minúsculas. Se fizer isso, os desenvolvedores que utilizam outras linguagens que não fazem distinção entre letras maiúsculas e minúsculas (como o Microsoft Visual Basic) talvez não possam utilizar sua classe na solução deles.

Trabalhe com construtores

Quando você utiliza a palavra-chave *new* para criar um objeto, o runtime precisa construir esse objeto utilizando a definição da classe. O runtime precisa se apropriar de uma parte da memória do sistema operacional, preenchê-la com os campos definidos pela classe e, então, chamar o construtor para executar qualquer inicialização necessária.

Um *construtor* é um método especial executado automaticamente quando você cria uma instância de uma classe. Ele tem o mesmo nome da classe e pode receber parâmetros, mas não pode retornar um valor (nem mesmo *void*). Toda classe deve ter um construtor. Se você não escrever um, o compilador irá gerar automaticamente um construtor padrão para você. (Mas o construtor padrão gerado pelo compilador na realidade não faz coisa alguma.) Você pode escrever seu próprio construtor padrão facilmente. Basta adicionar um método público que não retorna um valor e dar a ele o mesmo nome da classe. O exemplo a seguir mostra a classe *Circle* com um construtor padrão que inicializa o campo *radius* como 0:

```
class Circle
{
    private int radius;

    public Circle() // construtor padrão
    {
        radius = 0;
    }

    public double Area()
    {
        return Math.PI * radius * radius;
    }
}
```



Nota No jargão do C#, o construtor *padrão* é um construtor que não recebe parâmetros. Não importa se é gerado pelo compilador ou escrito por você; ainda assim ele é o construtor padrão. Você também pode escrever construtores não padrão (construtores que *recebem* parâmetros), como veremos na próxima seção, “Sobrecarregue construtores”.

Nesse exemplo, o construtor está marcado como *public*. Se essa palavra-chave *for* omitida, o construtor será privado (exatamente como qualquer outro método e campo). Se o construtor for privado, ele não poderá ser utilizado fora da classe, o que lhe impede de criar objetos *Circle* a partir dos métodos que não fazem parte da classe *Circle*. Assim, você pode achar que os construtores privados não são tão valiosos. Eles realmente têm suas utilidades, mas estas estão fora do objetivo da discussão atual.

Tendo adicionado um construtor público, você agora pode utilizar a classe *Circle* e empregar seu método *Area*. Observe como você utiliza a notação de ponto para chamar o método *Area* em um objeto *Circle*:

```
Circle c;
c = new Circle();
double areaOfCircle = c.Area();
```

Sobreconstrutores

Você quase já terminou, só falta um detalhe. Agora você pode declarar uma variável *Circle*, utilizá-la para referenciar um objeto *Circle* recém-criado e então chamar seu método *Area*. Mas há um último problema. A área de todos os objetos *Circle* sempre será 0, porque o construtor padrão define o raio como 0 e ele permanece em 0; o campo *radius* é privado e não há como alterar seu valor depois que ele é inicializado. Um construtor é apenas um tipo especial de método e – como todos os métodos – pode ser sobreconstruído. Assim como existem várias versões do método *Console.WriteLine* e cada uma recebe parâmetros diferentes, também é possível escrever diferentes versões de um construtor. Assim, você pode adicionar outro construtor à classe *Circle*, com um parâmetro especificando o raio a ser usado, como este:

```
class Circle
{
    private int radius;

    public Circle() // construtor padrão
    {
        radius = 0;
    }

    public Circle(int initialRadius) // construtor sobreconstruído
    {
        radius = initialRadius;
    }

    public double Area()
    {
        return Math.PI * radius * radius;
    }
}
```



Nota A ordem dos construtores em uma classe é irrelevante; você pode definir construtores na ordem que achar melhor.

Você pode então utilizar esse construtor ao criar um novo objeto *Circle*, como a seguir:

```
Circle c;
c = new Circle(45);
```

Quando você compila o aplicativo, o compilador deduz qual construtor deve chamar com base nos parâmetros especificados para o operador *new*. Neste exemplo, você passou um *int*; portanto, o compilador gera o código que chama o construtor que recebe um parâmetro *int*.

Fique atento a um importante recurso da linguagem C#: se você escrever seu próprio construtor para uma classe, o compilador não gerará um construtor padrão. Portanto, se você escreveu um construtor que aceita um ou mais parâmetros e também quiser um construtor padrão, você mesmo terá de escrever o construtor padrão.

Classes parciais

Uma classe pode conter vários métodos, campos e construtores, assim como outros itens discutidos nos próximos capítulos. Uma classe altamente funcional pode tornar-se muito grande. Com o C#, é possível dividir o código-fonte para uma classe em arquivos separados de modo que você possa organizar a definição de uma classe grande em partes menores, mais fáceis de gerenciar. Esse recurso é usado pelo Microsoft Visual Studio 2013 para aplicativos Windows Presentation Foundation (WPF) e aplicativos Windows Store, em que o código-fonte que o desenvolvedor pode editar é mantido em um arquivo separado do código que é gerado pelo Visual Studio sempre que o layout de um formulário for alterado.

Ao dividir uma classe em vários arquivos, você define as partes da classe usando a palavra-chave *partial* em cada arquivo. Por exemplo, se a classe *Circle* fosse dividida entre dois arquivos chamados *circ1.cs* (contendo os construtores) e *circ2.cs* (contendo os métodos e campos), o conteúdo de *circ1.cs* seria este:

```
partial class Circle
{
    public Circle() // construtor padrão
    {
        this.radius = 0;
    }

    public Circle(int initialRadius) // construtor sobreescrito
    {
        this.radius = initialRadius;
    }
}
```

O conteúdo de *circ2.cs* seria semelhante a este:

```
partial class Circle
{
    private int radius;

    public double Area()
    {
        return Math.PI * this.radius * this.radius;
    }
}
```

Ao compilar uma classe que foi dividida em arquivos separados, você deve fornecer todos os arquivos para o compilador.

No próximo exercício, você vai declarar uma classe que modela um ponto no espaço bidimensional. Essa classe conterá dois campos privados para as coordenadas x e y de um ponto e fornecerá os construtores para inicializar esses campos. Você criará instâncias da classe usando a palavra-chave *new* e chamando os construtores.

Escreva os construtores e crie os objetos

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra o projeto Classes, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 7\Windows X\Classes na sua pasta Documentos.
3. No Solution Explorer, clique duas vezes no arquivo Program.cs para exibi-lo na janela Code and Text Editor.
4. Na classe *Program*, localize o método *Main*.

O método *Main* chama o método *doWork*, inserido dentro de um bloco *try* e seguido por uma rotina de tratamento *catch*. Com esse bloco *try/catch*, você pode escrever no método *doWork* o código que em geral entraria em *Main*, tendo a certeza de que ele irá capturar e tratar qualquer exceção. Atualmente, o método *doWork* não contém nada, a não ser um comentário *// TODO*:



Dica Os comentários TODO são frequentemente utilizados pelos desenvolvedores como um lembrete de que há um trecho de código a ser revisto, muitas vezes contendo a descrição do trabalho a ser feito, como *// TODO: implementar o método doWork*. O Visual Studio reconhece essa forma de comentário, e eles podem ser localizados rapidamente em qualquer lugar de um aplicativo, com a janela Task List. Para exibir essa janela, no menu View, clique em Task List. Por padrão, a janela Task List é aberta abaixo da janela Code and Text Editor. Na caixa de lista suspensa, na parte superior dessa janela, selecione Comments. Todos os comentários TODO serão listados. Você pode então clicar duas vezes em qualquer um desses comentários para ir diretamente ao código correspondente, o qual será exibido na janela Code and Text Editor.

The screenshot shows the Visual Studio 2013 interface. The main window displays the code for the *Program.cs* file. The code includes a *doWork* method with a *TODO* comment and a *Main* method containing a *try/catch* block. Below the editor, the Task List window is open, showing two tasks: one for the *doWork* method and another for the *TODO* comment in the *doWork* method. The Task List window has a header "Comments" and a table with columns for Description, File, and Line.

Description	File	Line
TODO:	Point.cs	13
TODO:	Program.cs	15

- Exiba o arquivo Point.cs na janela Code and Text Editor.

Esse arquivo define uma classe chamada *Point*, a qual você utilizará para representar a localização de um ponto no espaço bidimensional, definido por um par de coordenadas *x* e *y*. No momento, a não ser por outro comentário *// TODO*, a classe *Point* está vazia.

- Retorne ao arquivo Program.cs. Na classe *Program*, edite o corpo do método *doWork* e substitua o comentário *// TODO* pela instrução a seguir:

```
Point origin = new Point();
```

Essa instrução cria uma nova instância da classe *Point* e chama seu construtor padrão.

- No menu Build, clique em Build Solution.

O código é compilado sem erro porque o compilador gera o código para um construtor padrão para a classe *Point*. Mas você não pode ver o código C# desse construtor porque o compilador não gera qualquer instrução na linguagem fonte.

- Retorne à classe *Point* no arquivo Point.cs. Substitua o comentário *// TODO* por um construtor *public* que aceita dois argumentos *int* chamados *x* e *y* e que chama o método *Console.WriteLine* para exibir os valores desses argumentos no console, como mostrado no texto em negrito no exemplo de código a seguir:

```
class Point
{
    public Point(int x, int y)
    {
        Console.WriteLine("x:{0}, y:{1}", x, y);
    }
}
```



Nota Lembre-se de que o método *Console.WriteLine* usa *{0}* e *{1}* como espaços reservados. Na instrução mostrada, *{0}* será substituído pelo valor de *x* e *{1}* será substituído pelo valor de *y*, quando o programa for executado.

- No menu Build, clique em Build Solution.

O compilador agora informa um erro:

```
'Classes.Point' does not contain a constructor that takes 0 arguments
```

A chamada ao construtor padrão no método *doWork* agora é inválida, porque não há mais um construtor padrão. Você escreveu seu próprio construtor para a classe *Point*; portanto, o compilador não gerará o construtor padrão. Agora você corrigirá isso escrevendo seu próprio construtor padrão.

- 10.** Edite a classe *Point*, adicionando um construtor padrão *public* que chama *Console.WriteLine* para escrever a string “*Default constructor called*” no console, como mostrado em negrito no exemplo a seguir. Agora a classe *Point* deve ser semelhante a isto:

```
class Point
{
    public Point()
    {
        Console.WriteLine("Default constructor called");
    }

    public Point(int x, int y)
    {
        Console.WriteLine("x:{0}, y:{1}", x, y);
    }
}
```

- 11.** No menu Build, clique em Build Solution.

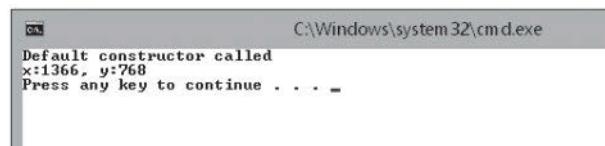
O programa agora deve ser compilado com sucesso.

- 12.** No arquivo Program.cs, edite o corpo do método *doWork*. Declare uma variável chamada *bottomRight* do tipo *Point* e inicialize-a como um novo objeto *Point* utilizando o construtor com dois argumentos, como mostrado em negrito no código a seguir. Forneça os valores 1366 e 768, que representam as coordenadas do canto inferior direito da tela com base na resolução 1366 × 768 (uma resolução comum de muitos dispositivos tablet para Windows 8 e Windows 8.1). O método *doWork* deve agora ser semelhante a este:

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
}
```

- 13.** No menu Debug, clique em Start Without Debugging.

O programa compila e executa, escrevendo as seguintes mensagens no console:



- 14.** Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2013.

Agora você adicionará dois campos *int* à classe *Point* para representar as coordenadas *x* e *y* de um ponto e modificará os construtores para inicializar esses campos.

- 15.** Edite a classe *Point* no arquivo Point.cs, adicionando dois campos *private* chamados *x* e *y*, do tipo *int*, como mostrado em negrito no código a seguir. Agora a classe *Point* deve ser semelhante a isto:

```
class Point
{
    private int x, y;

    public Point()
    {
        Console.WriteLine("default constructor called");
    }

    public Point(int x, int y)
    {
        Console.WriteLine("x:{0}, y:{1}", x, y);
    }
}
```

Você editará o segundo construtor *Point* para inicializar os campos *x* e *y* com os valores dos parâmetros *x* e *y*. Há uma armadilha potencial quando se faz isso. Se você não tiver cuidado, o construtor ficará igual a este:

```
public Point(int x, int y) // Não digite isso!
{
    x = x;
    y = y;
}
```

Embora o código seja compilado, essas instruções parecem ambíguas. Como o compilador sabe que na instrução *x* = *x*; o primeiro *x* é o campo e o segundo *x* é o parâmetro? A resposta é que ele não sabe! Um parâmetro de método com o mesmo nome do campo oculta o campo para todas as instruções no método. Tudo o que esse código realmente faz é atribuir os parâmetros a eles mesmos; ele não modifica os campos. Isso é exatamente o que não queremos.

A solução é utilizar a palavra-chave *this* para qualificar quais variáveis são parâmetros e quais são campos. Colocar o prefixo *this* na variável significa "o campo neste objeto".

- 16.** Modifique o construtor *Point* que recebe dois parâmetros, substituindo a instrução *Console.WriteLine* pelo seguinte código mostrado em negrito:

```
public Point(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

- 17.** Edite o construtor *Point* padrão para inicializar os campos *x* e *y* com -1, como no texto em negrito. Observe que, embora não haja parâmetros para causar confusão, ainda é uma boa prática qualificar as referências de campo com *this*:

```
public Point()
{
    this.x = -1;
    this.y = -1;
}
```

- 18.** No menu Build, clique em Build Solution. Confirme se o código compila sem erros ou alertas. (Você pode executá-lo, mas ele ainda não produz saída.)

Os métodos que pertencem a uma classe e que operam em dados que pertencem a uma instância específica de uma classe são chamados *métodos de instância*. (Você vai aprender sobre outros tipos de métodos mais adiante neste capítulo.) No exercício a seguir, você escreverá um método de instância para a classe *Point*, chamado *DistanceTo*, o qual calcula a distância entre dois pontos.

Escreva e chame métodos de instância

- 1.** No projeto Classes no Visual Studio 2013, adicione à classe *Point* o método de instância público chamado *DistanceTo* a seguir, depois dos construtores. O método aceita um único argumento *Point* chamado *other* e retorna um *double*.

O método *DistanceTo* deve ser semelhante a este:

```
class Point
{
    ...
    public double DistanceTo(Point other)
    {
    }
}
```

Nos próximos passos, você adicionará código ao corpo do método de instância *DistanceTo* para calcular e retornar a distância entre o objeto *Point* que está sendo utilizado para fazer a chamada e o objeto *Point* passado como parâmetro. Para fazer isso, você deve calcular a diferença entre as coordenadas *x* e as coordenadas *y*.

- 2.** No método *DistanceTo*, declare uma variável local *int* chamada *xDiff* e inicialize-a com a diferença entre *this.x* e *other.x*, como mostrado em negrito a seguir:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
}
```

- 3.** Declare outra variável *int* local chamada *yDiff* e inicialize-a com a diferença entre *this.y* e *other.y*, como mostrado aqui no texto em negrito:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
}
```



Nota Embora os campos *x* e *y* sejam privados, outras instâncias da mesma classe ainda podem acessá-los. É importante entender que o termo *private* opera no nível da classe e não no nível do objeto; dois objetos que são instâncias da mesma classe podem acessar dados privados uns dos outros, mas objetos que são instâncias de outra classe, não podem.

Para calcular a distância, utilize o teorema de Pitágoras e calcule a raiz quadrada da soma dos quadrados de *xDiff* e *yDiff*. A classe *System.Math* fornece o método *Sqrt* que você pode utilizar para calcular raízes quadradas.

4. Declare uma variável chamada *distance* de tipo *double* e utilize-a para conter o resultado do cálculo que acabamos de descrever.

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
    double distance = Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
}
```

5. Adicione a instrução *return* ao final do método *DistanceTo* e retorne o valor na variável *distance*:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
    double distance = Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
    return distance;
}
```

Agora você testará o método *DistanceTo*.

6. Retorne ao método *doWork* na classe *Program*. Após as instruções que declaram e inicializam as variáveis *Point origin* e *bottomRight*, declare uma variável chamada *distance* do tipo *double*. Inicialize essa variável *double* com o resultado obtido pela chamada ao método *DistanceTo* no objeto *origin*, passando o objeto *bottomRight* para ele como argumento.

O método *doWork* deve agora ser semelhante a este:

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
    double distance = origin.DistanceTo(bottomRight);
}
```



Nota O Microsoft IntelliSense deve exibir o método *DistanceTo* quando você digitar o caractere de ponto após *origin*.

7. Adicione ao método *doWork* outra instrução que escreve o valor da variável *distance* no console utilizando o método *Console.WriteLine*.

O método *doWork* deve ser semelhante a este:

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
    double distance = origin.DistanceTo(bottomRight);
    Console.WriteLine("Distance is: {0}", distance);
}
```

8. No menu Debug, clique em Start Without Debugging.
9. Confirme que o valor 1568.45465347265 é escrito na janela do console e, em seguida, pressione Enter para fechar o aplicativo e voltar ao Visual Studio 2013.

Dados e métodos static

No exercício anterior, você utilizou o método *Sqrt* da classe *Math*. Da mesma forma, ao examinar a classe *Circle*, você leu o campo *PI* da classe *Math*. Se pensar sobre isso, a maneira como você chamou o método *Sqrt* ou leu o campo *PI* foi um pouco estranha. Você chamou o método e leu o campo na própria classe, não em um objeto do tipo *Math*. É como tentar escrever *Point.DistanceTo* em vez de *origin.DistanceTo* no código que você adicionou no exercício anterior. Portanto, o que está acontecendo e como isso funciona?

Você notará com frequência que nem todos os métodos pertencem a uma instância de uma classe; eles são métodos utilitários, pois fornecem uma função útil que é independente de qualquer instância da classe específica. O método *Sqrt* serve apenas como um exemplo. Se *Sqrt* fosse um método de instância de *Math*, você teria de criar um objeto *Math* para chamar *Sqrt*:

```
Math m = new Math();
double d = m.Sqrt(42.24);
```

Isso seria inconveniente. O objeto *Math* não participaria do cálculo da raiz quadrada. Todos os dados de entrada que *Sqrt* necessita são fornecidos na lista de parâmetros e o resultado é retornado para o chamador utilizando o valor de retorno do método. Os objetos não são realmente necessários aqui; portanto, forçar *Sqrt* em uma instância ‘camisa de força’ não é uma boa ideia.



Nota Além do método *Sqrt* e do campo *PI*, a classe *Math* contém vários outros métodos matemáticos utilitários, como *Sin*, *Cos*, *Tan* e *Log*.

Em C#, todos os métodos devem ser declarados dentro de uma classe. Mas se declarar um método ou um campo como *static*, você pode chamar o método ou acessar o campo utilizando o nome da classe. Não há necessidade de instância. Veja como o método *Sqrt* da classe *Math* é declarado:

```
class Math
{
    public static double Sqrt(double d)
    {
        ...
    }
    ...
}
```

Um método estático não depende de uma instância da classe e não pode acessar um campo ou método de instância definido na classe; ele só utiliza os campos e outros métodos marcados como *static*.

Crie um campo compartilhado

A definição de um campo como estático torna possível criar uma única instância de um campo que é compartilhada entre todos os objetos criados a partir de uma única classe. (Campos não estáticos são locais para cada instância de um objeto.) No exemplo a seguir, o campo *static NumCircles* na classe *Circle* é incrementado pelo construtor *Circle* toda vez que um novo objeto *Circle* é criado:

```
class Circle
{
    private int radius;
    public static int NumCircles = 0;

    public Circle() // default constructor
    {
        radius = 0;
        NumCircles++;
    }

    public Circle(int initialRadius) // overloaded constructor
    {
        radius = initialRadius;
        NumCircles++;
    }
}
```

Todos os objetos *Circle* compartilham a mesma instância do campo *NumCircles*; portanto, a instrução *NumCircles++*; incrementa os mesmos dados toda vez que uma nova instância é criada. Observe que você não pode prefixar *NumCircles* com a palavra-chave *this*, pois *NumCircles* não pertence a um objeto específico.

Você pode acessar o campo *NumCircles* fora da classe, especificando a classe *Circle*, em vez de um objeto *Circle*, como no exemplo a seguir:

```
Console.WriteLine("Number of Circle objects: {0}", Circle.NumCircles);
```



Nota Convém lembrar que os métodos *static* também são chamados de métodos *de classe*. Mas os campos *static* não são em geral chamados de campos *de classe*; eles são chamados apenas de campos *static* (ou, eventualmente, de variáveis *static*).

Crie um campo static utilizando a palavra-chave *const*

Prefixando o campo com a palavra-chave *const*, você pode declarar que um campo é estático, mas que seu valor nunca pode mudar. A palavra-chave *const* é uma abreviação de *constante*. Um campo *const* não utiliza a palavra-chave *static* na sua declaração, mas mesmo assim é estático. Contudo, por razões cujas explicações estão fora dos objetivos deste livro, você só pode declarar um campo como *const* quando esse campo é um tipo numérico (como *int* ou *double*), uma string ou uma enumeração (você vai aprender sobre enumerações no Capítulo 9, “Como criar tipos-valor com enumerações e estruturas”). Por exemplo, veja como a classe *Math* declara *PI* como um campo *const*:

```
class Math
{
    ...
    public const double PI = 3.14159265358979323846;
}
```

Entenda as classes static

Outro recurso da linguagem C# é a capacidade de declarar uma classe como *static*. Uma classe *static* só pode conter membros *static*. (Todos os objetos que você cria utilizando essa classe compartilham uma única cópia desses membros.) O objetivo de uma classe *static* é puramente atuar como um contêiner de campos e métodos utilitários. Uma classe *static* não pode conter dados ou métodos de instância e não faz sentido tentar criar um objeto de uma classe *static* usando o operador *new*. De fato, você não pode criar uma instância de um objeto que utiliza uma classe *static* utilizando *new*, mesmo se quiser fazer isso. (O compilador informará um erro se você tentar.) Se você precisar fazer alguma inicialização, uma classe *static* poderá ter um construtor padrão, desde que ele também seja declarado como *static*. Qualquer outro tipo de construtor é ilegal e será reportado como tal pelo compilador.

Se você estivesse definindo sua própria versão da classe *Math*, contendo apenas membros *static*, ela poderia se parecer com esta:

```
public static class Math
{
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Sqrt(double x) {...}
    ...
}
```



Nota A classe *Math* real não é definida assim, porque na verdade ela tem alguns métodos de instância.

No exercício final deste capítulo, você adicionará um campo *private static* à classe *Point* e inicializará o campo como 0. Você incrementará essa contagem nos dois construtores. Por fim, você escreverá um método *public static* para retornar o valor desse campo *private static*. Com esse campo, você pode descobrir quantos objetos *Point* foram criados.

Escreva membros *static* e chame métodos *static*

1. No Visual Studio 2013, exiba a classe *Point* na janela Code and Text Editor.
2. Adicione um campo *private static* chamado *objectCount* do tipo *int* à classe *Point*, imediatamente antes dos construtores. Inicialize-o como 0 ao declará-lo, da seguinte maneira:

```
class Point
{
    ...
    private static int objectCount = 0;
    ...
}
```



Nota Ao declarar um campo como *objectCount*, você pode escrever as palavras-chave *private* e *static* em qualquer ordem. Contudo, a ordem preferida é *private* em primeiro lugar, *static* em segundo.

3. Adicione uma instrução aos dois construtores *Point* para incrementar o campo *objectCount*, como mostrado em negrito no exemplo de código a seguir.

A classe *Point* deve se parecer com isto:

```
class Point
{
    private int x, y;
    private static int objectCount = 0;

    public Point()
    {
        this.x = -1;
        this.y = -1;
        objectCount++;
    }

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
        objectCount++;
    }
    public double DistanceTo(Point other)
    {
        int xDiff = this.x - other.x;
        int yDiff = this.y - other.y;
        return Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
    }
}
```

Toda vez que um objeto é criado, seu construtor é chamado. Desde que você incremente o *objectCount* em cada construtor (incluindo o construtor padrão), *objectCount* armazenará o número de objetos criados até aqui. Essa estratégia só funciona porque *objectCount* é um campo *static* compartilhado. Se *objectCount* fosse um campo de instância, cada objeto teria seu próprio campo *objectCount* pessoal que seria definido como 1.

A questão agora é: como os usuários da classe *Point* podem descobrir quantos objetos *Point* foram criados? No momento, o campo *objectCount* é *private* e não está disponível fora da classe. Uma solução precária seria tornar o campo *objectCount* publicamente acessível. Essa estratégia quebraria o encapsulamento da classe e, então, você não teria qualquer garantia de que seu valor estaria correto, porque qualquer coisa poderia alterar o valor no campo. Uma ideia muito melhor é fornecer um método *public static* que retorne o valor do campo *objectCount*. Isso é o que você fará agora.

4. Adicione à classe *Point* um método *public static* chamado *ObjectCount*, que retorna um *int* mas não recebe parâmetros. Nesse método, retorne o valor do campo *objectCount*, como em negrito aqui:

```
class Point
{
    ...
    public static int ObjectCount()
    {
        return objectCount;
    }
}
```

5. Exiba a classe *Program* na janela Code and Text Editor. Adicione uma instrução ao método *doWork* para escrever na tela o valor retornado a partir do método *ObjectCount* da classe *Point*, como mostrado no texto em negrito no exemplo de código a seguir.

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
    double distance = origin.distanceTo(bottomRight);
    Console.WriteLine("Distance is: {0}", distance);
    Console.WriteLine("Number of Point objects: {0}", Point.ObjectCount());
}
```

O método *ObjectCount* é chamado referenciando *Point*, o nome da classe e não o nome de uma variável *Point* (como *origin* ou *bottomRight*). Como dois objetos *Point* foram criados quando *ObjectCount* foi chamado, o método deve retornar o valor 2.

6. No menu Debug, clique em Start Without Debugging.

Confirme que a mensagem “Number of Point objects: 2” foi escrita na janela do console (após a mensagem que exibe o valor da variável *distance*).

7. Pressione Enter para terminar o programa e retorne para o Visual Studio 2013.

Classes anônimas

Uma *classe anônima* é uma classe que não tem nome. Isso parece bastante estranho, mas é bem útil em algumas situações que veremos mais adiante neste livro, especialmente ao se utilizar expressões de consulta (query). (Você aprenderá sobre expressões de consulta no Capítulo 20, “Separação da lógica do aplicativo e tratamento de eventos”.) Por enquanto, você terá de acreditar que elas são úteis.

Você cria uma classe anônima simplesmente utilizando a palavra-chave *new* e um par de chaves que definem os campos e valores que você quer que a classe contenha, assim:

```
myAnonymousObject = new { Name = "John", Age = 47 };
```

Essa classe contém dois campos públicos chamados *Name* (inicializado com a string “John”) e *Age* (inicializado com o inteiro 47). O compilador infere os tipos dos campos a partir dos tipos de dados que você especifica para inicializá-los.

Ao se definir uma classe anônima, o compilador gera um nome próprio para a classe, mas ele não informará qual é esse nome. Portanto, as classes anônimas suscitam um enigma potencialmente interessante: se você não sabe o nome da classe, como poderá criar um objeto do tipo apropriado e atribuir uma instância da classe a ele? No exemplo de código mostrado antes, qual deve ser o tipo da variável *myAnonymousObject*? A resposta é que você não sabe – esse é o propósito das classes anônimas! Mas isso não é um problema se você declarar *myAnonymousObject* como uma variável implicitamente tipada utilizando a palavra-chave *var*, desta maneira:

```
var myAnonymousObject = new { Name = "John", Age = 47 };
```

Lembre-se de que a palavra-chave *var* faz o compilador criar uma variável do mesmo tipo da expressão utilizada para inicializá-la. Nesse caso, o tipo da expressão é o nome que o compilador gera para a classe anônima.

Você pode acessar os campos no objeto utilizando a conhecida notação de ponto, como demonstrado aqui:

```
Console.WriteLine("Name: {0} Age: {1}", myAnonymousObject.Name, myAnonymousObject.Age);
```

Você pode até mesmo criar outras instâncias da mesma classe anônima, mas com valores diferentes, como no seguinte:

```
var anotherAnonymousObject = new { Name = "Diana", Age = 46 };
```

O compilador do C# utiliza os nomes, os tipos, o número e a ordem dos campos para determinar se duas instâncias de uma classe anônima têm o mesmo tipo. Nesse caso, as variáveis *myAnonymousObject* e *anotherAnonymousObject* têm o mesmo número de campos, com o mesmo nome e tipo, na mesma ordem; portanto, as duas variáveis são instâncias da mesma classe anônima. Isso significa que você pode realizar instruções de atribuição como esta:

```
anotherAnonymousObject = myAnonymousObject;
```



Nota Esteja ciente de que essa instrução de atribuição talvez não realize o que você espera. Você aprenderá mais sobre como atribuir variáveis de objeto no Capítulo 8.

Há muitas restrições quanto ao conteúdo de uma classe anônima. Por exemplo, as classes anônimas só podem conter campos públicos, todos esses campos precisam ser inicializados, eles não podem ser estáticos e você não pode definir método algum para elas. Você vai usar classes anônimas periodicamente neste livro e, à medida que fizer isso, vai aprender mais sobre elas.

Resumo

Neste capítulo, você viu como pode definir novas classes. Você aprendeu que, por padrão, os campos e os métodos de uma classe são privados e inacessíveis ao código fora da classe, mas que pode utilizar a palavra-chave *public* para expor campos e métodos para o mundo exterior. Você viu como utilizar a palavra-chave *new* para criar uma nova instância de uma classe e como definir construtores que podem inicializar instâncias de classes. Por último, você examinou a implementação de campos e métodos estáticos, para fornecer dados e operações que independem de qualquer instância específica de uma classe.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 8.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar uma classe	Escreva a palavra-chave <code>class</code> , seguida pelo nome da classe, seguida por uma chave de abertura e uma de fechamento. Os métodos e campos da classe são declarados entre as chaves de abertura e fechamento. Por exemplo:
	<pre>class Point { ... }</pre>
Declarar um construtor	Escreva um método cujo nome seja o mesmo nome da classe e que não tenha qualquer tipo de retorno (nem mesmo <code>void</code>). Por exemplo:
	<pre>class Point { public Point(int x, int y) { ... } }</pre>
Chamar um construtor	Use a palavra-chave <code>new</code> e especifique o construtor com um conjunto de parâmetros apropriados. Por exemplo:
	<pre>Point origin = new Point(0, 0);</pre>
Declarar um método static	Escreva a palavra-chave <code>static</code> antes da declaração do método. Por exemplo:
	<pre>class Point { public static int ObjectCount() { ... } }</pre>
Chamar um método static	Escreva o nome da classe, seguida de um ponto, seguido do nome do método. Por exemplo:
	<pre>int pointsCreatedSoFar = Point.ObjectCount();</pre>
Declarar um campo static	Use a palavra-chave <code>static</code> antes do tipo do campo. Por exemplo:
	<pre>class Point { ... private static int objectCount; }</pre>
Declarar um campo const	Escreva a palavra-chave <code>const</code> antes da declaração do campo e omita a palavra-chave <code>static</code> . Por exemplo:
	<pre>class Math { ... public const double PI = ...; }</pre>
Acessar um campo static	Escreva o nome da classe, seguido de um ponto, seguido do nome do método. Por exemplo:
	<pre>double area = Math.PI * radius * radius;</pre>

CAPÍTULO 8

Valores e referências

Neste capítulo, você vai aprender a:

- Explicar as diferenças entre um tipo-valor e um tipo-referência.
- Modificar a maneira como os argumentos são passados como parâmetros de métodos utilizando as palavras-chave *ref* e *out*.
- Converter um valor em uma referência usando boxing.
- Converter uma referência de volta em um valor usando unboxing e casting.

O Capítulo 7, “Criação e gerenciamento de classes e objetos”, apresentou a declaração de suas classes e a criação de objetos por meio da palavra-chave *new*. Também foi possível ver como fazer a inicialização de um objeto por meio de um construtor. Neste capítulo, você vai aprender a diferença entre as características dos tipos primitivos – como *int*, *double* e *char* – e as características dos tipos de classe.

Copie variáveis de tipo-valor e classes

A maioria dos tipos primitivos do C#, como *int*, *float*, *double* e *char* (mas não *string*, por motivos que serão abordados em breve) é coletivamente chamada de *tipos-valor*. Esses tipos têm tamanho fixo e, quando você declara uma variável como um tipo-valor, o compilador gera o código que aloca um bloco de memória grande o suficiente para conter um valor correspondente. Por exemplo, declarar uma variável *int* faz o compilador alocar 4 bytes de memória (32 bits). Uma instrução que atribui um valor (como 42) a *int* faz o valor ser copiado para esse bloco de memória.

Os tipos classe, como *Circle* (descrito no Capítulo 7), são tratados de maneira diferente. Quando você declara uma variável *Circle*, o compilador *não* gera um código que aloca um bloco de memória grande o suficiente para armazenar *Circle*; tudo o que ele faz é alocar uma pequena parte da memória que possa armazenar o endereço de (ou referência a) outro bloco de memória que contém *Circle*. (Um endereço específico a localização de um item na memória.) A memória para o objeto *Circle* real só é alocada quando a palavra-chave *new* é utilizada para criar o objeto. Uma classe é um exemplo de *tipo-referência*. Tipos-referência contêm referências a blocos de memória. Para escrever programas C# eficazes, que usem plenamente o Microsoft .NET Framework, você deve saber a diferença entre tipos-valor e tipos-referência.



Nota Em C#, o tipo *string* é na verdade uma classe. Isso porque não há um tamanho padrão para uma *string* (diferentes *strings* podem conter diferentes números de caracteres) e é bem mais eficiente alocar memória para elas dinamicamente, quando o programa é executado, do que estaticamente, em tempo de compilação. A descrição de tipos-referência, como as classes, deste capítulo, também se aplica ao tipo *string*. Na verdade, no C#, a palavra-chave *string* é apenas um alias para a classe *System.String*.

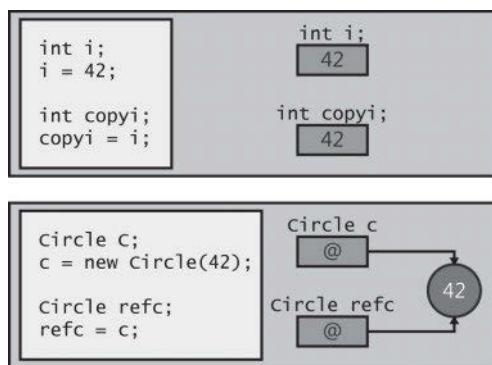
Considere a situação em que você declara uma variável chamada *i* como um *int* e atribui a ela o valor 42. Se declarar outra variável chamada *copyi* como um *int* e então atribuir *i* a *copyi*, *copyi* conterá o mesmo valor que *i* (42). Contudo, embora *copyi* e *i* contenham o mesmo valor, há dois blocos de memória contendo o valor 42: um bloco para *i* e outro bloco para *copyi*. Se você modificar o valor de *i*, o valor de *copyi* não será alterado. Vejamos isso em código:

```
int i = 42;      // declara e inicializa i
int copyi = i; /* copyi contém uma cópia dos dados em i;
                 i e copyi contêm ambas o valor 42 */
i++;           /* um incremento em i não tem efeito sobre copyi;
                 agora i contém 43, mas copyi ainda contém 42
```

O efeito de declarar uma variável *c* como um tipo de classe, como *Circle*, é muito diferente. Quando você declara *c* como *Circle*, *c* pode se referir a um objeto *Circle*; o valor real armazenado por *c* é o endereço de um objeto *Circle* na memória. Se você declarar mais uma variável, chamada *refc* (também como *Circle*), e atribuir *c* a *refc*, *refc* terá uma cópia do mesmo endereço que *c*; ou seja, existe apenas um objeto *Circle* e, agora, tanto *refc* como *c* se referem a ele. Vejamos o exemplo em código:

```
Circle c = new Circle(42);
Circle refc = c;
```

A figura a seguir ilustra ambos os exemplos. O sinal (@) nos objetos *Circle* representa uma referência que armazena um endereço na memória:



Essa diferença é muito importante. Em particular, ela significa que o comportamento dos parâmetros do método depende de eles serem tipos-valor ou tipos-referência. Você vai explorar essa diferença no próximo exercício.

Cópia de tipos-referência e privacidade de dados

Se você realmente quiser copiar o conteúdo de um objeto *Circle*, *c*, para outro objeto *Circle*, *refc*, em vez de apenas copiar a referência, deve fazer *refc* referenciar uma nova instância da classe *Circle* e então copiar os dados campo por campo, de *c* para *refc*, assim:

```
Circle refc = new Circle();
refc.radius = c.radius; // Não tente isto
```

Mas se qualquer membro da classe *Circle* for privado (como o campo *radius*), você não poderá copiar esses dados. Em vez disso, poderia tornar os dados dos campos privados acessíveis, expondo-os como propriedades, e então utilizar essas propriedades para ler os dados de *c* e copiá-los em *refc*. Você aprenderá a fazer isso no Capítulo 15, “Implementação de propriedades para acessar campos”.

Como alternativa, uma classe poderia fornecer um método *Clone* que retornasse outra instância da mesma classe, mas preenchesse com os mesmos dados. O método *Clone* teria acesso aos dados privados de um objeto e poderia copiar esses dados diretamente para a outra instância da mesma classe. Por exemplo, o método *Clone* da classe *Circle* poderia ser definido assim:

```
class Circle
{
    private int radius;
    // Construtores e outros métodos omitidos
    ...
    public Circle Clone()
    {
        // Cria um novo objeto Circle
        Circle clone = new Circle();

        // Copia dados privados de this para clone
        clone.radius = this.radius;

        // Retorna o novo objeto Circle contendo os dados copiados
        return clone;
    }
}
```

Essa estratégia é natural se todos os dados privados consistirem em valores, mas se um ou mais campos forem eles próprios tipos-referência (por exemplo, a classe *Circle* poderia ser estendida para conter um objeto *Point* do Capítulo 7, indicando a posição do *Circle* em um gráfico), esses tipos-referência também precisariam fornecer um método *Clone*; caso contrário, o método *Clone* da classe *Circle* simplesmente copiaria uma referência para esses campos. Esse é um processo conhecido como *cópia profunda*. A estratégia alternativa, onde o método *Clone* simplesmente copia referências, é conhecida como *cópia rasa*.

O exemplo de código anterior também apresenta uma questão interessante: como `private` é dado privado? Vimos anteriormente que a palavra-chave `private` torna um campo ou método inacessível fora de uma classe. No entanto, isso não significa que ele possa ser acessado por apenas um objeto. Se você criar dois objetos da mesma classe, cada um poderá acessar os dados privados do outro. Isso parece curioso, mas na verdade, métodos como `Clone` dependem dessa característica. A instrução `clone.radius = this.radius;` só funciona porque o campo privado `radius` no objeto `clone` é acessível dentro da instância atual da classe `Circle`. Assim, `private` significa na verdade “privado para a classe” e não “privado para um objeto”. Mas não confunda `private` com `static`. Se você simplesmente declara um campo como `private`, cada instância da classe recebe seus próprios dados. Se um campo é declarado como `static`, cada instância da classe compartilha os mesmos dados.

Utilize parâmetros por valor e parâmetros por referência

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
 2. Abra o projeto Parameters, localizado na pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 8\Windows X\Parameters` na sua pasta Documentos.
- O projeto contém três arquivos de código C#: `Pass.cs`, `Program.cs` e `WrappedInt.cs`.
3. Exiba o arquivo `Pass.cs` na janela Code and Text Editor.

Esse arquivo define uma classe chamada `Pass` que atualmente está vazia, a não ser por um comentário `// TODO`:

 **Dica** Lembre-se de que é possível utilizar a janela Task List para localizar todos os comentários `TODO` em uma solução.

4. Adicione um método `public static` chamado `Value` à classe `Pass`, substituindo o comentário `// TODO`: Esse método deve aceitar um único parâmetro `int` (um tipo-valor) chamado `param` e ter um tipo de retorno `void`. O corpo do método `Value` deve simplesmente atribuir o valor 42 a `param`, como mostrado em negrito no exemplo de código a seguir.

```
namespace Parameters
{
    class Pass
    {
        public static void Value(int param)
        {
            param = 42;
        }
    }
}
```



Nota O motivo de você estar definindo esse método como estático é manter o exercício simples. Você pode chamar o método *Value* diretamente na classe *Pass*, em vez de primeiro ter de criar um novo objeto *Pass*. Os princípios ilustrados neste exercício se aplicam exatamente da mesma maneira aos métodos de instância.

5. Exiba o arquivo-fonte Program.cs na janela Code and Text Editor e localize o método *doWork* da classe *Program*.

O método *doWork* é chamado pelo método *Main* quando o programa começa a executar. Conforme explicado no Capítulo 7, a chamada de método vem dentro de um bloco *try* e é seguida por uma rotina de tratamento *catch*.

6. Adicione quatro instruções ao método *doWork* para executar as seguintes tarefas:
 - a. Declarar uma variável local *int* chamada *i* e inicializá-la como 0.
 - b. Escrever o valor de *i* no console utilizando *Console.WriteLine*.
 - c. Chamar *Pass.Value*, passando *i* como argumento.
 - d. Escrever novamente o valor de *i* no console.

Com as chamadas a *Console.WriteLine* antes e depois da chamada a *Pass.Value*, você pode ver se a chamada a *Pass.Value* realmente modifica o valor de *i*. O método *doWork* concluído deve ser exatamente como este:

```
static void doWork()
{
    int i = 0;
    Console.WriteLine(i);
    Pass.Value(i);
    Console.WriteLine(i);
}
```

7. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.
8. Confirme se o valor “0” foi escrito duas vezes na janela do console.

A instrução de atribuição dentro do método *Pass.Value*, que atualiza o parâmetro e o define com 42, utiliza uma cópia do argumento passado e o argumento original *i* permanece completamente inalterado.

9. Pressione a tecla Enter para fechar o aplicativo.
- Você verá agora o que acontece quando passa um parâmetro *int* que está inserido dentro de uma classe.
10. Exiba o arquivo WrappedInt.cs na janela Code and Text Editor. Esse arquivo contém a classe *WrappedInt*, a qual está vazia, a não ser por um comentário // TODO:.

- 11.** Adicione um campo de instância *public* chamado *Number* do tipo *int* à classe *WrappedInt*, como mostrado em negrito no código a seguir:

```
namespace Parameters
{
    class WrappedInt
    {
        public int Number;
    }
}
```

- 12.** Exiba o arquivo *Pass.cs* na janela Code and Text Editor. Adicione um método *public static* chamado *Reference* à classe *Pass*. Esse método deve aceitar um único parâmetro *WrappedInt* chamado *param* e ter um tipo de retorno *void*. O corpo do método *Reference* deve atribuir 42 a *param.Number*, como mostrado aqui:

```
public static void Reference(WrappedInt param)
{
    param.Number = 42;
}
```

- 13.** Exiba o arquivo *Program.cs* na janela Code and Text Editor. Transforme em comentário o código existente no método *doWork* e adicione mais quatro instruções para executar as seguintes tarefas:

- Declarar uma variável local *WrappedInt* chamada *wi* e inicializá-la com um novo objeto *WrappedInt* chamando o construtor padrão.
- Escrever o valor de *wi.Number* no console.
- Chamar o método *Pass.Reference*, passando *wi* como argumento.
- Escrever o valor de *wi.Number* novamente no console.

Como antes, com as chamadas a *Console.WriteLine*, você pode ver se a chamada a *Pass.Reference* modifica o valor de *wi.Number*. O método *doWork* agora deve estar assim (as novas instruções estão destacadas em negrito):

```
static void doWork()
{
    // int i = 0;
    // Console.WriteLine(i);
    // Pass.Value(i);
    // Console.WriteLine(i);

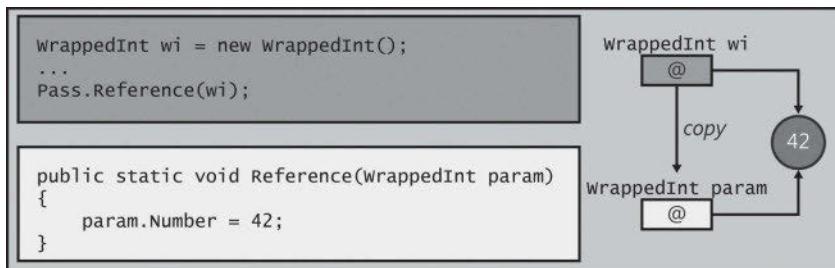
    WrappedInt wi = new WrappedInt();
    Console.WriteLine(wi.Number);
    Pass.Reference(wi);
    Console.WriteLine(wi.Number);
}
```

- 14.** No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

Desta vez, os dois valores exibidos na janela de console correspondem ao valor de *wi.Number* antes e depois da chamada ao método *Pass.Reference*. Você deve ver que os valores 0 e 42 são exibidos.

- Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2013.

Para explicar o que o exercício anterior demonstra, o valor de *wi.Number* é inicializado como 0 pelo construtor gerado pelo compilador. A variável *wi* contém uma referência ao objeto *WrappedInt* recém-criado (que contém um *int*). A variável *wi* é então copiada como um argumento para o método *Pass.Reference*. Como *WrappedInt* é uma classe (um tipo-referência), *wi* e *param* referenciam o mesmo objeto *WrappedInt*. Qualquer alteração feita ao conteúdo do objeto, por meio da variável *param* no método *Pass.Reference*, é visível utilizando a variável *wi* quando o método é concluído. O diagrama a seguir ilustra o que acontece quando um objeto *WrappedInt* é passado como um argumento para o método *Pass.Reference*:



Valores nulos e tipos nullable

Ao se declarar uma variável, sempre é uma boa ideia inicializá-la. Com tipos-valor, é comum ver código como este:

```
int i = 0;
double d = 0.0;
```

Lembre-se de que, para inicializar uma variável de referência como uma classe, você pode criar uma nova instância da classe e atribuir a variável de referência ao novo objeto, assim:

```
Circle c = new Circle(42);
```

Tudo isso está correto, mas e se você na verdade não quiser criar um novo objeto? Talvez o propósito da variável seja simplesmente armazenar uma referência para um objeto existente em algum ponto posterior em seu programa. No exemplo de código a seguir, a variável *Circle copy* é inicializada, mas depois a ela é atribuída uma referência a outra instância da classe *Circle*:

```
Circle c = new Circle(42);
Circle copy = new Circle(99); // Algum valor aleatório, para inicializar copy
...
copy = c; // copy e c referenciam o mesmo objeto
```

Depois de atribuir `c` a `copy`, o que acontece ao objeto `Circle` original com um raio de 99 que você utilizou para inicializar `copy`? Nada mais o referencia. Nessa situação, o runtime pode reivindicar a memória, realizando uma operação conhecida como *coleta de lixo*, cujos detalhes você conhecerá no Capítulo 14, “Coleta de lixo e gerenciamento de recursos”. O importante a entender agora é que a coleta de lixo é uma operação potencialmente demorada; você não deve criar objetos que nunca são utilizados, pois isso desperdiça tempo e recursos.

Você poderia argumentar que, se uma variável vai receber uma referência a outro objeto em algum ponto em um programa, não faz sentido inicializá-la. Mas isso é uma péssima prática de programação, que pode levar a problemas no seu código. Por exemplo, você se encontrará inevitavelmente na situação em que quer referenciar uma variável para um objeto somente se essa variável ainda não contiver uma referência, como mostra o seguinte exemplo de código:

```
Circle c = new Circle(42);
Circle copy; // Não inicializada !!!
...
if (copy == null) // só atribui a copy se não estiver inicializada, mas o que acontece aqui?
{
    copy = c; // copy e c referenciam o mesmo objeto
    ...
}
```

O propósito da instrução `if` é testar a variável `copy` para ver se ela foi inicializada, mas com qual valor você deve comparar essa variável? A resposta é utilizar um valor especial chamado `null`.

No C#, você pode atribuir o valor `null` a qualquer variável-referência. O valor `null` simplesmente significa que a variável não referencia objeto algum na memória. Você pode utilizá-lo desta forma:

```
Circle c = new Circle(42);
Circle copy = null; // Inicializada
...
if (copy == null)
{
    copy = c; // copy e c referenciam o mesmo objeto
    ...
}
```

Utilize tipos *nullable*

O valor `null` é útil para inicializar tipos-referência. Às vezes, você precisa de um valor equivalente para tipos-valor, mas `null` é ele próprio uma referência; assim, não é possível atribuí-lo a um tipo-valor. A seguinte instrução é, portanto, inválida no C#:

```
int i = null; // inválido
```

Mas o C# define um modificador que pode ser utilizado para declarar se uma variável é um tipo-valor *nullable*. Um tipo-valor *nullable* comporta-se de maneira semelhante ao tipo-valor original, mas você pode atribuir o valor `null` a ele. Use o ponto de interrogação (?) para indicar que um tipo-valor é *nullable*, assim:

```
int? i = null; // válido
```

É possível determinar se uma variável *nullable* contém *null* testando-a da mesma maneira que um tipo-referência:

```
if (i == null)
    ...
```

Você pode atribuir uma expressão do tipo-valor apropriado diretamente a uma variável *nullable*. Todos os exemplos a seguir são válidos:

```
int? i = null;
int j = 99;
i = 100;      // Copia um tipo-valor constante em um tipo nullable
i = j;        // Copia um tipo-valor variável em um tipo nullable
```

Você deve observar que o contrário não é verdadeiro. Você não pode atribuir uma variável *nullable* a uma variável de tipo-valor normal. Portanto, dadas as definições das variáveis *i* e *j* do exemplo anterior, a instrução a seguir não é permitida:

```
j = i;      // Inválido
```

Isso faz sentido, se você considerar que a variável *i* pode conter *null* e que *j* é um tipo-valor que não pode conter *null*. Isso também significa que você não pode utilizar uma variável *nullable* como um parâmetro para um método que espera receber um tipo-valor normal. Se você se lembra, o método *Pass.Value* do exercício anterior espera um parâmetro normal *int*; portanto, a seguinte chamada de método não compilará:

```
int? i = 99;
Pass.Value(i);    // Erro do compilador
```

Entenda as propriedades dos tipos *nullable*

Um tipo *nullable* expõe um par de propriedades que você pode utilizar para determinar se o tipo realmente tem um valor não *null* e qual é esse valor. A propriedade *HasValue* indica se um tipo *nullable* contém um valor ou é *null* e você pode recuperar o valor de um tipo *nullable* não *null* lendo a propriedade *Value*, desta maneira:

```
int? i = null;
...
if (!i.HasValue)
{
    // Se i é null, atribui o valor 99 a ele
    i = 99;
}
else
{
    // Se i é não null, então exibe seu valor
    Console.WriteLine(i.Value);
}
```

O Capítulo 4, “Instruções de decisão”, ensinou que o operador NOT (!) nega um valor booleano. Esse fragmento de código testa a variável *nullable* *i* e, se ela não tiver um valor (for *null*), atribui a essa variável o valor 99; do contrário, exibe o valor da variável. Nesse exemplo, utilizar a propriedade *HasValue* não traz benefício algum em relação a testar se um valor é *null* diretamente. Além disso, ler a propriedade *Value* é uma maneira tediosa de ler o conteúdo da variável. Mas essas deficiências aparentes são causadas pelo fato de que *int?* é um tipo *nullable* muito simples. Você pode criar

tipos-valor mais complexos e utilizá-los para declarar variáveis nullable em que as vantagens da utilização das propriedades *HasValue* e *Value* tornam-se mais aparentes. Veremos alguns exemplos no Capítulo 9, “Como criar tipos-valor com enumeração e estruturas”.



Nota A propriedade *Value* de um tipo nullable é somente de leitura. Você pode utilizar essa propriedade para ler o valor de uma variável, mas não para modificá-la. Para atualizar uma variável nullable, utilize uma instrução de atribuição comum.

Parâmetros *ref* e *out*

Em geral, quando você passa um argumento para um método, o parâmetro correspondente é inicializado com uma cópia do argumento. Isso é verdade independentemente de o parâmetro ser um tipo-valor (como um *int*), um tipo nullable (como *int?*) ou um tipo-referência (como um *WrappedInt*). Esse arranjo significa que é impossível qualquer alteração no parâmetro afetar o valor do argumento passado. Por exemplo, no código a seguir, o valor apresentado no console é 42 e não 43. O método *doIncrement* incrementa uma *cópia* do argumento (*arg*) e *não* o argumento original, como demonstrado aqui:

```
static void doIncrement(int param)
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(arg);
    Console.WriteLine(arg); // escreve 42, não 43
}
```

No exercício anterior, você viu que, se o parâmetro para um método é um tipo-referência, qualquer alteração feita utilizando esse parâmetro modifica os dados referenciados pelo argumento passado por ele. O ponto-chave é este: embora os dados referenciados tenham mudado, o argumento passado como parâmetro não mudou — ele ainda referencia o mesmo objeto. Em outras palavras, embora seja possível modificar o objeto que o argumento referencia, não é possível modificar o argumento propriamente dito (por exemplo, para defini-lo a fim de referenciar um objeto completamente diferente). Na maioria das vezes, essa garantia é muito útil e pode ajudar a reduzir o número de erros em um programa. Eventualmente, porém, você pode querer escrever um método que de fato precise modificar um argumento. O C# fornece as palavras-chave *ref* e *out* para isso.

Crie parâmetros *ref*

Se você utilizar a palavra-chave *ref* como prefixo de um parâmetro, o compilador do C# gerará código que passa uma referência ao argumento real, em vez de uma cópia do argumento. Ao utilizar um parâmetro *ref*, tudo o que você fizer ao parâmetro também será feito ao argumento original, porque o parâmetro e o argumento referenciam o mesmo dado. Ao passar um argumento como um parâmetro *ref*, você também deve prefixar o argumento com a palavra-chave *ref*. Essa sintaxe fornece uma indicação visual útil para o programador de que o argumento pode mudar. Veja novamente o exemplo anterior, desta vez modificado para utilizar a palavra-chave *ref*.

```
static void doIncrement(ref int param) // usando ref
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(ref arg);    // usando ref
    Console.WriteLine(arg); // escreve 43
}
```

Desta vez, o método *doIncrement* recebe uma referência ao argumento original, em vez de uma cópia; portanto, qualquer alteração feita pelo método utilizando essa referência, muda o argumento original. Essa é a razão de o valor 43 ser exibido no console.

Lembre-se de que o C# impõe a regra de que você deve atribuir um valor a uma variável, antes que possa lê-la. Essa regra também se aplica aos argumentos de método: você não pode passar um valor não inicializado como argumento para um método, mesmo que o argumento seja definido como *ref*. Por exemplo, no programa a seguir, *arg* não é inicializada; portanto, esse código não será compilado. Essa falha ocorre porque a instrução *param++*; dentro do método *doIncrement* é, na verdade, um alias para a instrução *arg++*; e essa operação só é permitida se *arg* tiver um valor definido:

```
static void doIncrement(ref int param)
{
    param++;
}

static void Main()
{
    int arg;           // não inicializada
    doIncrement(ref arg);
    Console.WriteLine(arg);
}
```

Crie parâmetros *out*

O compilador verifica se o parâmetro *ref* recebeu um valor, antes de chamar o método. Mas pode haver ocasiões em que você queira que o próprio método inicialize o parâmetro. Você pode fazer isso com a palavra-chave *out*.

A palavra-chave *out* é sintaticamente semelhante à palavra-chave *ref*. Você pode utilizar a palavra-chave *out* como prefixo do parâmetro para que o parâmetro se torne um alias para o argumento. Assim como ao utilizar *ref*, tudo o que você faz no parâmetro também é feito no argumento original. Ao passar um argumento para um parâmetro *out*, você também deve prefixar o argumento com a palavra-chave *out*.

A palavra-chave *out* é uma abreviação de *output*. Quando você passa um parâmetro *out* para um método, o método *deve* atribuir um valor a ele antes de terminar ou retornar, como mostrado no exemplo a seguir:

```
static void doInitialize(out int param)
{
    param = 42; // Inicializa param antes de terminar
}
```

O exemplo a seguir não compila porque *doInitialize* não atribui um valor a *param*:

```
static void doInitialize(out int param)
{
    // Não faz nada
}
```

Uma vez que um parâmetro *out* deve receber um valor do método, você pode chamar o método sem inicializar seu argumento. Por exemplo, o código a seguir chama *doInitialize* para inicializar a variável *arg*, que é então exibida no console:

```
static void doInitialize(out int param)
{
    param = 42;
}

static void Main()
{
    int arg;           // não inicializada
    doInitialize(out arg); // válido
    Console.WriteLine(arg); // escreve 42
}
```

Examinaremos parâmetros *ref* no próximo exercício.

Utilize parâmetros *ref*

1. Retorne ao projeto Parameters no Visual Studio 2013.
2. Exiba o arquivo Pass.cs na janela Code and Text Editor.
3. Edite o método *Value* para aceitar seu parâmetro como um parâmetro *ref*.

O método *Value* deve se parecer com este:

```
class Pass
{
    public static void Value(ref int param)
    {
        param = 42;
    }
    ...
}
```

4. Exiba o arquivo Program.cs na janela Code and Text Editor.
5. Transforme em comentário as quatro primeiras instruções. Observe que a terceira instrução do método `doWork`, `Pass.Value(i)`; mostra um erro. Isso porque o método `Value` agora espera um parâmetro `ref`. Edite essa instrução de modo que a chamada do método `Pass.Value` passe seu argumento como um parâmetro `ref`.



Nota Deixe as quatro instruções que criam e testam o objeto `WrappedInt` no estado em que se encontram.

O método `doWork` deve agora ser semelhante a este:

```
class Program
{
    static void doWork()
    {
        int i = 0;
        Console.WriteLine(i);
        Pass.Value(ref i);
        Console.WriteLine(i);
        ...
    }
}
```

6. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.
Desta vez, os dois primeiros valores escritos na janela de console são 0 e 42. Esse resultado mostra que a chamada ao método `Pass.Value` modificou o argumento `i` com sucesso.
7. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2013.



Nota Você pode usar os modificadores `ref` e `out` nos parâmetros de tipo-referência assim como nos parâmetros de tipo-valor. O efeito é exatamente o mesmo: o parâmetro torna-se um alias para o argumento.

Como a memória do computador é organizada

Os computadores utilizam a memória para armazenar os programas que estão sendo executados e os dados que esses programas utilizam. Para entender as diferenças entre os tipos-valor e os tipos-referência, é útil entender como os dados são organizados na memória.

Sistemas operacionais e runtimes (ambientes de execução) de linguagens, como os utilizados pelo C#, em geral, dividem a memória utilizada para armazenar dados em duas áreas separadas, cada uma gerenciada de uma maneira distinta. Essas duas áreas são tradicionalmente chamadas *pilha* (*stack*) e *heap*. Pilha e heap servem para propósitos diferentes, os quais estão descritos aqui:

- Quando você chama um método, a memória necessária para seus parâmetros e suas variáveis locais é sempre adquirida da pilha. Quando o método termina (seja porque retornou, seja porque lançou uma exceção), a memória adquirida para os parâmetros e variáveis locais é automaticamente liberada de volta para a pilha e fica disponível para ser reutilizada quando outro método for chamado. Os parâmetros de método e as variáveis locais na pilha têm uma vida útil bem definida: eles nascem quando o método começa e desaparecem assim que o método termina.



Nota Na verdade, a mesma vida útil se aplica às variáveis definidas em qualquer bloco de código colocado entre chaves de abertura e fechamento. No exemplo de código a seguir, a variável *i* é criada quando o corpo do loop *while* começa, mas desaparece quando o loop *while* termina e a execução continua após a chave de fechamento:

```
while (...)  
{  
    int i = ...; // i é criada na pilha aqui  
    ...  
}  
// i desaparece da pilha aqui
```

- Quando você cria um objeto (uma instância de uma classe) utilizando a palavra-chave *new*, a memória necessária para compilar o objeto é sempre adquirida do heap. Você viu que o mesmo objeto pode ser referenciado de vários lugares utilizando variáveis de referência. Quando a última referência a um objeto desaparece, a memória utilizada pelo objeto torna-se disponível para ser reutilizada (embora ela possa não ser utilizada imediatamente). O Capítulo 14 inclui uma discussão mais detalhada de como a memória heap é empregada. Portanto, os objetos criados no heap têm vida útil mais indeterminada; um objeto é criado com a palavra-chave *new*, mas só desaparece em algum ponto após a última referência a ele ser removida.



Nota Todos os tipos-valor são criados na pilha. Todos os tipos-referência (objetos) são criados no heap (embora a referência em si esteja na pilha). Tipos nullable na verdade são tipos-referência e são criados no heap.

Os nomes *pilha* e *heap* têm origem na maneira como o runtime gerencia a memória:

- A memória de pilha é organizada como uma pilha de caixas sobrepostas umas sobre as outras. Quando um método é chamado, cada parâmetro é colocado em uma caixa que é disposta na parte superior da pilha. Cada variável local é igualmente atribuída a uma caixa, e esta é colocada no topo da pilha de caixas. Quando um método termina, pode-se considerar que todas as caixas são removidas da pilha.

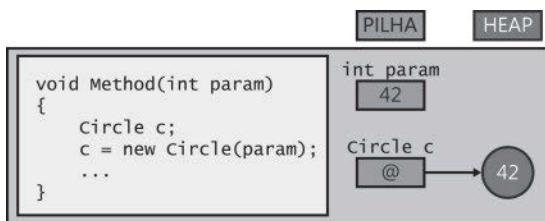
- A memória heap é literalmente um “monte” de caixas espalhadas por uma sala, em vez de empilhadas ordenadamente umas sobre as outras. Cada caixa tem um rótulo indicando se está em uso ou não. Quando um novo objeto é criado, o runtime procura uma caixa vazia e a aloca para o objeto. A referência à caixa é armazenada em uma variável local na pilha. O runtime monitora o número de referências a cada caixa. (Lembre-se de que duas variáveis podem referenciar o mesmo objeto). Quando a última referência desaparece, o runtime marca a caixa como fora de uso e, em algum ponto no futuro, esvaziará a caixa e a disponibilizará para reutilização.

Utilize a pilha e o heap

Agora vamos examinar o que acontece quando o método *Method* a seguir é chamado:

```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    ...
}
```

Suponha que o argumento passado para *param* seja o valor 42. Quando o método é chamado, um bloco de memória (grande o suficiente para um *int*) é alocado na pilha e inicializado com o valor 42. Quando o fluxo do programa entra no método, outro bloco de memória, grande o suficiente para armazenar uma referência (um endereço de memória), também é alocado da pilha, mas permanece não inicializado. (Isso serve para a variável *Circle*, *c*). Em seguida, outra parte da memória grande o suficiente para um objeto *Circle* é alocada do heap. Isso é o que faz a palavra-chave *new*. O construtor *Circle* é executado para converter essa memória bruta do heap em um objeto *Circle*. Uma referência a esse objeto *Circle* é armazenada na variável *c*. A figura a seguir ilustra a situação:



Neste ponto, você já deve ter notado duas coisas:

- Embora o objeto esteja armazenado no heap, a referência ao objeto (a variável *c*) está armazenada na pilha.
- A memória heap não é infinita. Se a memória heap estiver esgotada, o operador *new* lançará uma exceção *OutOfMemoryException* e o objeto não será criado.



Nota O construtor *Circle* também poderá lançar uma exceção. Se ele o fizer, a memória alocada para o objeto *Circle* será reivindicada e o valor retornado pelo construtor será *null*.

Quando o método termina, os parâmetros e variáveis locais saem do escopo. A memória adquirida para *c* e *param* são automaticamente liberadas na pilha. O runtime nota que o objeto *Circle* não é mais referenciado e, mais tarde, providenciará para que sua memória seja reivindicada pelo heap. (Consulte o Capítulo 14.)

A classe *System.Object*

Um dos tipos-referência mais importantes no .NET Framework é a classe *Object* no namespace *System*. Para compreender completamente o significado da classe *System.Object* é necessário que você entenda herança, que será descrita no Capítulo 12, "Herança". Por enquanto, simplesmente aceite que todas as classes são tipos especializados da classe *System.Object* e que você pode utilizar *System.Object* para criar uma variável que pode referenciar qualquer tipo-referência. *System.Object* é uma classe tão importante que o C# fornece a palavra-chave *object* como um alias de *System.Object*. No seu código, você pode utilizar *object* ou pode escrever *System.Object* – eles significam exatamente a mesma coisa.

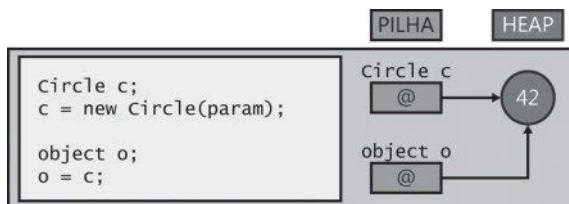


Dica Utilize a palavra-chave *object* em vez de *System.Object*. Ela é mais direta e coerente com outras palavras-chave que são sinônimos para classes (como *string* para *System.String* e algumas outras que serão abordadas no Capítulo 9).

No exemplo a seguir, as variáveis *c* e *o* referenciam o mesmo objeto *Circle*. O fato de que o tipo de *c* é *Circle* e o tipo de *o* é *object* (o alias de *System.Object*) na prática fornece duas visões diferentes do mesmo item na memória.

```
Circle c;
c = new Circle(42);
object o;
o = c;
```

O diagrama a seguir ilustra como as variáveis *c* e *o* referenciam o mesmo item no heap.

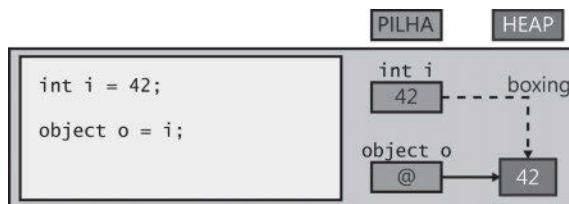


Boxing

Conforme você acabou de ver, as variáveis do tipo *object* podem referenciar qualquer item de qualquer tipo-referência. Mas as variáveis do tipo *object* também podem referenciar um tipo-valor. Por exemplo, as duas instruções a seguir inicializam a variável *i* (do tipo *int*, um tipo-valor) como 42 e, então, inicializam a variável *o* (do tipo *object*, um tipo-referência) como *i*:

```
int i = 42;
object o = i;
```

A segunda instrução exige uma pequena explicação para se compreender o que realmente está acontecendo. Lembre-se de que *i* é um tipo-valor e existe na pilha. Se a referência dentro de *o* referenciasse diretamente *i*, ela referenciaría a pilha. Mas todas as referências devem referenciar objetos no heap; criar referências a itens na pilha pode comprometer seriamente a robustez do runtime e criar uma potencial brecha de segurança; logo, isso não é permitido. Portanto, o runtime aloca uma parte da memória a partir do heap, copia o valor do inteiro *i* para essa parte da memória e faz o objeto *o* referenciar essa cópia. Essa cópia automática de um item da pilha para o heap é chamada de *boxing*. A figura a seguir mostra o resultado:



Importante Se você modificar o valor original da variável *i*, o valor no heap referenciado por meio de *o* não mudará. Da mesma forma, se você modificar o valor no heap, o valor original da variável não será alterado.

Unboxing

Como uma variável do tipo *object* pode referenciar uma cópia na forma boxed de um valor, é razoável permitir que você acesse o valor boxed por meio da variável. Talvez você suponha que possa acessar o valor *int* na forma boxed que a variável *o* referencia, utilizando uma instrução de atribuição simples, como esta:

```
int i = o;
```

Mas se tentar essa sintaxe, você receberá um erro de tempo de compilação. Se pensar no assunto, é muito lógico que você não possa utilizar a sintaxe `int i = o;`. Afinal, `o` pode estar referenciando qualquer coisa, e não apenas um `int`. Considere o que aconteceria no código a seguir se essa instrução fosse permitida:

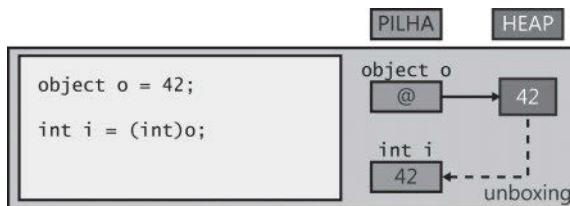
```
Circle c = new Circle();
int i = 42;
object o;

o = c; // o referencia um círculo
i = o; // o que é armazenado em i?
```

Para obter o valor da cópia boxed, você precisa utilizar o que é conhecido como *casting*. Essa é uma operação que verifica se é seguro converter um item de um tipo em outro, antes de realmente fazer a cópia. Você coloca o nome do tipo como prefixo da variável `object` entre parênteses, como neste exemplo:

```
int i = 42;
object o = i; // faz boxing
i = (int)o; // compila normalmente
```

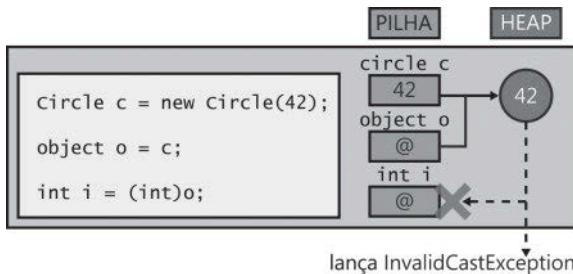
O efeito desse casting é sutil. O compilador nota que você especificou o tipo `int` no casting. Em seguida, o compilador gera um código para verificar o que `o` realmente referencia em tempo de execução. Poderia ser absolutamente qualquer coisa. Só porque seu casting diz que `o` referencia um `int`, isso não significa que ele de fato faz isso. Se `o` realmente referencia um `int` na forma boxed e tudo coincide, o casting é bem-sucedido e o código gerado pelo compilador extrai o valor do `int` na forma boxed e o copia em `i`. (Neste exemplo, o valor na forma boxed é armazenado em `i`.) Isso é chamado *unboxing*. O diagrama a seguir mostra o que está acontecendo:



Por outro lado, se `o` não referencia um valor `int` na forma boxed, há uma incompatibilidade de tipos, fazendo o casting falhar. O código gerado pelo compilador lança uma exceção `InvalidCastException` em tempo de execução. Veja o exemplo de um casting para uma operação de unboxing que falha:

```
Circle c = new Circle(42);
object o = c;           // não faz boxing porque Circle é uma variável de referência
int i = (int)o;         // compila normalmente, mas lança uma exceção em tempo de execução
```

O diagrama a seguir ilustra esse caso:



Você utilizará boxing e unboxing em exercícios posteriores. Lembre-se de que boxing e unboxing são operações caras devido à quantidade de verificação exigida e à necessidade de alocar memória heap adicional. O boxing tem suas utilidades, mas o uso imprudente pode prejudicar seriamente o desempenho de um programa. Você verá uma alternativa ao boxing no Capítulo 17, "Genéricos".

Casting de dados seguro

Utilizando um casting, você pode especificar que, *em sua opinião*, os dados referenciados por um objeto têm um tipo específico e que é seguro referenciar o objeto utilizando esse tipo. A expressão-chave aqui é "em sua opinião". O compilador do C# não verificará se esse é o caso, mas o runtime sim. Se o tipo de objeto na memória não corresponder ao casting, o runtime lançará uma *InvalidCastException*, como descrito na seção anterior. Você deve estar preparado para capturar essa exceção e tratá-la apropriadamente, se ela ocorrer.

Mas capturar uma exceção e tentar se recuperar dela, caso o tipo de um objeto não seja aquele que você esperava, é uma estratégia bastante inepta. O C# fornece dois operadores bem mais úteis que podem ajudar a fazer um casting de uma maneira muito mais elegante, os operadores *is* e *as*.

O operador *is*

Utilize o operador *is* para verificar se o tipo de um objeto é aquele que você espera, desta maneira:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
if (o is WrappedInt)
{
    WrappedInt temp = (WrappedInt)o; // Isso é seguro; o é um WrappedInt
    ...
}
```

O operador *is* aceita dois operandos: uma referência a um objeto à esquerda e o nome de um tipo à direita. Se o tipo do objeto referenciado no heap tiver o tipo especificado, *is* será avaliado como *true*; caso contrário, será avaliado como *false*. O código anterior tenta fazer o casting da referência à variável *object o* somente se ele souber que o casting será bem-sucedido.

O operador *as*

O operador *as* desempenha um papel semelhante a *is*, mas de uma maneira ligeiramente mais abreviada. Você utiliza o operador *as* desta maneira:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
WrappedInt temp = o as WrappedInt;
if (temp != null)
{
    ... // O casting foi bem-sucedido
}
```

Como ocorre com o operador *is*, o operador *as* recebe um objeto e um tipo como seus operandos. O runtime tenta fazer o casting do objeto para o tipo especificado. Se o casting for bem-sucedido, o resultado será retornado e, neste exemplo, ele é atribuído à variável *WrappedInt temp*. Se o casting for malsucedido, o operador *as* será avaliado como o valor *null* e atribuirá isso a *temp*.

Há um pouco mais sobre operadores *is* e *as* do que descrito aqui e o Capítulo 12 os discutirá com mais detalhes.

Ponteiros e código inseguro

Esta seção serve apenas para sua informação e dirige-se aos desenvolvedores que conhecem C ou C++. Se você é iniciante em programação, sinta-se livre para pular esta seção.

Se você já desenvolveu programas em linguagens como C ou C++, deve estar familiarizado com grande parte da discussão deste capítulo sobre referências a objetos. Embora nem o C nem o C++ tenham tipos-referência explícitos, as duas linguagens têm uma construção que fornece uma funcionalidade semelhante: um ponteiro.

Um *ponteiro* é uma variável que armazena o endereço ou uma referência a um item na memória (no heap ou na pilha). Uma sintaxe especial é usada para identificar uma variável como um ponteiro. Por exemplo, a instrução a seguir declara a variável *pi* como um ponteiro para um número inteiro:

```
int *pi;
```

Embora a variável *pi* seja declarada como um ponteiro, na verdade ela não apontará para lugar algum até que você a inicialize. Por exemplo, para fazer *pi* apontar para a variável do tipo inteiro *i*, você pode usar as instruções a seguir e o operador de endereço (*&*), o que retorna o endereço de uma variável:

```
int *pi;
int i = 99;
...
pi = &i;
```

Você pode acessar e modificar o valor mantido na variável *i* por meio da variável ponteiro *pi*, como mostrado aqui:

```
*pi = 100;
```

Esse código atualiza o valor da variável *i* para 100, uma vez que *pi* aponta para a mesma posição da memória que a variável *i*.

Um dos principais problemas que os desenvolvedores que aprendem C e C++ encontram é entender a sintaxe usada pelos ponteiros. O operador `*` tem pelo menos dois significados (além de ser o operador aritmético da multiplicação) e sempre há uma grande confusão sobre quando usar `&` em vez de `*`. A outra questão com os ponteiros é a facilidade em apontar para algo inválido ou simplesmente esquecer-se de apontar para algo, e então tentar referenciar esse algo. O resultado será lixo ou um programa que falhará com um erro, porque o sistema operacional detecta uma tentativa de acessar um endereço inválido na memória. Também há toda uma série de falhas de segurança em muitos sistemas existentes que resultam de um gerenciamento inadequado dos ponteiros; alguns ambientes (não o Microsoft Windows) falham em impor a verificação de que um ponteiro não referencia a memória pertencente a outro processo, abrindo a possibilidade de que dados confidenciais sejam comprometidos.

As variáveis de referência foram adicionadas ao C# para evitar todos esses problemas. Se realmente quiser, você pode continuar utilizando ponteiros no C#, mas deve marcar o código como *unsafe*. A palavra-chave *unsafe* pode ser usada para marcar um bloco de código ou um método inteiro, como mostrado aqui:

```
public static void Main(string [] args)
{
    int x = 99, y = 100;
    unsafe
    {
        swap (&x, &y);
    }
    Console.WriteLine("x is now {0}, y is now {1}", x, y);
}

public static unsafe void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Quando compilar programas que contêm um código inseguro, você deve especificar a opção Allow Unsafe Code ao compilar o projeto. Para fazer isso, no Solution Explorer, clique com o botão direito do mouse no projeto e, então, no menu de atalho que aparece, clique em Properties. Na janela Properties, clique na guia Build, selecione Allow Unsafe Code e então, no menu File, clique em Save All.

Um código inseguro também afeta a maneira como a memória é gerenciada. Objetos criados em código inseguro são chamados de não gerenciados. Embora não seja comum, você poderá se deparar com algumas situações que exigem acessar a memória dessa maneira, especialmente se estiver escrevendo código que precisa executar algumas operações de baixo nível do Windows.

No Capítulo 14, vamos ver com mais detalhes as implicações do uso de código que acessa memória não gerenciada.

Resumo

Neste capítulo, você aprendeu algumas diferenças importantes entre tipos-valor, que armazenam seus valores diretamente na pilha, e tipos-referência, que referenciam indiretamente seus objetos no heap. Também aprendeu a utilizar as palavras-chave `ref` e `out` nos parâmetros de método para obter acesso aos argumentos. Você viu como a atribuição de um valor (por exemplo, o `int 42`) a uma variável da classe `System.Object` cria uma cópia boxed do valor no heap e então faz a variável `System.Object` referenciar essa cópia. Viu também como a atribuição de uma variável de um tipo-valor (como um `int`) a uma variável da classe `System.Object` copia o (ou faz o unbox do) valor na classe `System.Object` para a memória utilizada pelo `int`.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 9.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Copiar uma variável de tipo-valor	Basta fazer a cópia. Como a variável é um tipo-valor, você terá duas cópias da mesma variável. Por exemplo: <code>int i = 42; int copyi = i;</code>
Copiar uma variável de tipo-referência	Basta fazer a cópia. Como a variável é um tipo-referência, você terá duas referências ao mesmo objeto. Por exemplo: <code>Circle c = new Circle(42); Circle refc = c;</code>
Declarar uma variável que possa armazenar um tipo-valor ou o valor <code>null</code>	Declare a variável utilizando o modificador <code>? com o tipo</code> . Por exemplo: <code>int? i = null;</code>
Passar um argumento para um parâmetro <code>ref</code>	Prefixe o argumento com a palavra-chave <code>ref</code> . Isso torna o parâmetro um alias para o argumento real, em vez de uma cópia do argumento. O método pode mudar o valor do parâmetro e essa mudança será efetuada no argumento real e não em uma cópia local. Por exemplo: <code>static void Main() { int arg = 42; DoWork(ref arg); Console.WriteLine(arg); }</code>

Passar um argumento para um parâmetro out	Prefixe o argumento com a palavra-chave out. Isso torna o parâmetro um alias para o argumento real, em vez de uma cópia do argumento. O método deve atribuir um valor ao parâmetro e esse valor se torna argumento real. Por exemplo:
	<pre>static void Main() { int arg; DoWork(out arg); Console.WriteLine(arg); }</pre>
Fazer boxing em um valor	Inicialize ou atribua uma variável do tipo object com o valor. Por exemplo:
	<pre>object o = 42;</pre>
Fazer unboxing em um valor	Faça o casting da referência de objeto que referencia o valor boxed para o tipo da variável. Por exemplo:
	<pre>int i = (int)o;</pre>
Fazer casting seguro de um objeto	Utilize o operador is para testar se o casting é válido. Por exemplo:
	<pre>WrappedInt wi = new WrappedInt(); ... object o = wi; if (o is WrappedInt) { WrappedInt temp = (WrappedInt)o; ... }</pre> <p>Outra alternativa é usar o operador as para fazer o casting e testar se o resultado é null. Por exemplo:</p> <pre>WrappedInt wi = new WrappedInt(); ... object o = wi; WrappedInt temp = o as WrappedInt; if (temp != null) ... </pre>

CAPÍTULO 9

Como criar tipos-valor com enumerações e estruturas

Neste capítulo, você vai aprender a:

- Declarar um tipo enumerado.
- Criar e utilizar um tipo enumerado.
- Declarar um tipo-estrutura.
- Criar e utilizar um tipo-estrutura.
- Explicar as diferenças de comportamento entre uma estrutura e uma classe.

O Capítulo 8, "Valores e referências", abordou os dois tipos fundamentais do Microsoft Visual C#: os *tipos-valor* e os *tipos-referência*. Não esqueça que uma variável de tipo-valor armazena seu valor diretamente na pilha, ao passo que uma variável de tipo-referência armazena uma referência a um objeto no heap. O Capítulo 7, "Criação e gerenciamento de classes e objetos", apresentou a criação de seus próprios tipos-referência através da definição de classes. Neste capítulo, você poderá a criar seus próprios tipos-valor.

O C# suporta duas espécies de tipos-valor: *enumerações* e *estruturas*. Veremos uma de cada vez.

Enumerações

Suponha que você queira representar as estações do ano em um programa. Você poderia utilizar os valores inteiros 0, 1, 2 e 3 para descrever a primavera, o verão, o outono e o inverno, respectivamente. Esse sistema funcionaria, mas não é muito intuitivo. Se você usasse o valor inteiro 0 no código, não seria óbvio que um 0 representa primavera. Além disso, não seria uma solução muito sólida. Por exemplo, se você declarar uma variável *int* chamada *season*, não há como impedir que se atribua a ela um valor inteiro válido fora do conjunto 0, 1, 2 ou 3. O C# oferece uma solução melhor. Você pode criar uma enumeração (às vezes chamada de tipo *enum*), cujos valores estão limitados a um conjunto de nomes simbólicos.

Declare uma enumeração

Defina uma enumeração utilizando a palavra-chave *enum*, seguida por um conjunto de símbolos que identificam os valores válidos que o tipo pode ter, incluídos entre chaves. Veja como declarar um tipo enumerado chamado *Season*, cujos valores literais estão limitados aos nomes simbólicos *Spring*, *Summer*, *Fall* e *Winter*:

```
enum Season { Spring, Summer, Fall, Winter }
```

Utilize uma enumeração

Depois que você declarar um tipo enumerado, poderá utilizá-lo exatamente como qualquer outro tipo. Se o nome da enumeração for *Season*, você pode criar variáveis do tipo *Season*, campos do tipo *Season* e parâmetros do tipo *Season*, como mostrado neste exemplo:

```
enum Season { Spring, Summer, Fall, Winter }

class Example
{
    public void Method(Season parameter) // exemplo de parâmetro de método
    {
        Season localVariable; // exemplo de variável local
        ...

        private Season currentSeason; // exemplo de campo
    }
}
```

Para que o valor de uma variável de tipo enumerado possa ser lido, é necessário atribuir-lhe um valor. Você só pode atribuir um valor definido pela enumeração a uma variável do tipo enumerado, como ilustrado aqui:

```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // escreve 'Fall'
```



Nota Como ocorre com todos os tipos-valor, você pode criar uma versão nullable de uma variável de tipo enumerado utilizando o modificador *?*. Você então pode atribuir à variável o valor *null*, bem como os valores definidos pela enumeração:

```
Season? colorful = null;
```

Observe que você tem de escrever *Season.Fall* em vez de *Fall*. Todos os nomes literais de enumerações têm escopo definido pelo seu tipo enumerado. Isso é muito útil, porque torna possível que diferentes tipos enumerados coincidentemente contenham literais com o mesmo nome.

Além disso, observe que, quando você exibe uma variável de tipo enumerado utilizando *Console.WriteLine*, o compilador gera um código que escreve o nome do literal cujo valor corresponde ao valor da variável. Se necessário, você pode converter explicitamente uma variável de tipo enumerado em uma string que representa seu

valor atual, utilizando o método *ToString* predefinido que todos os tipos enumerados automaticamente contêm, como demonstrado no exemplo a seguir:

```
string name = colorful.ToString();
Console.WriteLine(name); // também escreve 'Fall'
```

Muitos dos operadores padrão utilizados em variáveis do tipo inteiro também podem ser empregados em variáveis de enumeração (exceto os operadores *bit a bit* e os operadores de *deslocamento*, que serão abordados no Capítulo 16, "Indexadores"). Por exemplo, você pode comparar a igualdade de duas variáveis de enumeração do mesmo tipo utilizando o operador de igualdade (`==`) e ainda efetuar cálculos aritméticos em uma variável de tipo enumerado (embora o resultado talvez nem sempre tenha um significado).

Escolha valores literais de enumeração

Internamente, uma enumeração associa um valor inteiro a cada elemento da enumeração. Por padrão, a numeração inicia em 0 para o primeiro elemento e sobe em incrementos de 1. É possível recuperar o valor inteiro subjacente de uma variável de tipo enumerado. Para isso, você deve fazer um casting para seu tipo subjacente. A discussão sobre unboxing no Capítulo 8 mostrou que o casting converte os dados de um tipo em outro, desde que a conversão seja válida e significativa. O fragmento de código a seguir escreve o valor 2 e não a palavra *Fall* (lembre-se de que na enumeração *Season*, *Spring* é 0, *Summer* 1, *Fall* 2 e *Winter* 3):

```
enum Season { Spring, Summer, Fall, Winter }
...
Season colorful = Season.Fall;
Console.WriteLine((int)colorful); // escreve '2'
```

Se preferir, você pode associar uma constante inteira específica (como 1) a um literal de enumeração (como *Spring*), como no exemplo a seguir:

```
enum Season { Spring = 1, Summer, Fall, Winter }
```



Importante O valor inteiro com o qual você inicializa um literal de enumeração deve ser um valor constante em tempo de compilação (como 1).

Se você não fornecer explicitamente um valor inteiro constante a um literal de enumeração, o compilador fornecerá um valor que é uma unidade maior do que o valor do literal de enumeração anterior, exceto para o primeiro literal de enumeração, ao qual o compilador fornece o valor padrão 0. No exemplo anterior, os valores subjacentes de *Spring*, *Summer*, *Fall* e *Winter* são atualmente 1, 2, 3 e 4.

Você pode fornecer mais de um literal de enumeração o mesmo valor subjacente. Por exemplo, no Reino Unido, o outono (*Fall*) é chamado de *Autumn*. Você pode agradar as duas culturas como mostrado a seguir:

```
enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }
```

Escolha o tipo subjacente de uma enumeração

Quando você declara uma enumeração, os literais de enumeração recebem valores do tipo *int*. Você também pode optar por basear sua enumeração em um tipo inteiro subjacente diferente. Por exemplo, para declarar que o tipo subjacente de *Season* é um *short* em vez de um *int*, você pode escrever o seguinte:

```
enum Season : short { Spring, Summer, Fall, Winter }
```

A principal razão para fazer isso é economizar memória; um *int* ocupa mais memória do que um *short* e, se você não precisa de todo o intervalo de valores disponíveis para um *int*, talvez faça sentido utilizar um tipo menor de dado.

Você pode basear uma enumeração em qualquer um dos oito tipos de inteiro: *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long* ou *ulong*. Os valores de todos os literais de enumeração devem estar dentro do intervalo do tipo base escolhido. Por exemplo, se basear uma enumeração no tipo de dado *byte*, você poderá ter no máximo 256 literais (começando em zero).

Agora que você sabe como declarar uma enumeração, a próxima etapa é utilizá-la. No exercício a seguir, você trabalhará com um aplicativo de console para declarar e utilizar uma classe de enumeração que representa os meses do ano.

Crie e utilize uma enumeração

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
 2. Abra o projeto *StructsAndEnums*, localizado na pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 9\Windows X\StructsAndEnums` na sua pasta Documentos.
 3. Na janela Code and Text Editor, exiba o arquivo *Month.cs*.
- O arquivo-fonte está vazio, a não ser pela declaração de um namespace chamado *StructsAndEnums* e um comentário `// TODO:`:
4. Exclua o comentário `// TODO:` e adicione uma enumeração chamada *Month* para modelar os meses do ano dentro do namespace *StructsAndEnums*, como mostrado em negrito no código a seguir. Os 12 literais de enumeração para *Month* são *January* a *December*.

```
namespace StructsAndEnums
{
    enum Month
    {
        January, February, March, April,
        May, June, July, August,
        September, October, November, December
    }
}
```

5. Exiba o arquivo *Program.cs* na janela Code and Text Editor.

Como nos exercícios dos capítulos anteriores, o método *Main* chama o método *doWork* e captura todas as exceções que ocorrerem.

6. Na janela Code and Text Editor, adicione uma instrução ao método *doWork* para declarar uma variável chamada *first* do tipo *Month* e inicialize-a como *Month.January*. Adicione outra instrução para escrever o valor da variável *first* no console.

O método *doWork* deve ser semelhante a este:

```
static void doWork()
{
    Month first = Month.January;
    Console.WriteLine(first);
}
```



Nota Quando você digita o ponto depois de *Month*, o Microsoft IntelliSense exibe automaticamente todos os valores na enumeração *Month*.

7. No menu Debug, clique em Start Without Debugging.

O Visual Studio 2013 compila e executa o programa. Confirme que a palavra *January* está escrita no console.

8. Pressione Enter para fechar o programa e retornar ao ambiente de programação do Visual Studio 2013.

9. Adicione mais duas instruções ao método *doWork* para incrementar a variável *first* e exibir seu novo valor no console, como mostrado aqui:

```
static void doWork()
{
    Month first = Month.January;
    Console.WriteLine(first);
    first++;
    Console.WriteLine(first);
}
```

10. No menu Debug, clique em Start Without Debugging.

O Visual Studio 2013 compila e executa o programa. Confirme que as palavras *January* e *February* estão escritas no console.

Observe que efetuar uma operação matemática (como a operação de incremento) em uma variável de tipo enumerado altera o valor inteiro interno da variável. Quando a variável é escrita no console, é exibido o valor de enumeração correspondente.

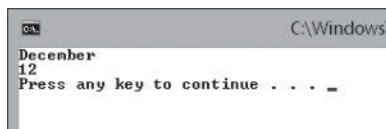
11. Pressione Enter para fechar o programa e retornar ao ambiente de programação do Visual Studio 2013.

12. Modifique a primeira instrução no método *doWork* para inicializar a variável *first* como *Month.December*, conforme mostrado em negrito:

```
static void doWork()
{
    Month first = Month.December;
    Console.WriteLine(first);
    first++;
    Console.WriteLine(first);
}
```

- 13.** No menu Debug, clique em Start Without Debugging.

O Visual Studio 2013 compila e executa o programa. Desta vez, a palavra *December* é escrita no console, seguida pelo número 12.



Embora você possa efetuar cálculos aritméticos em uma enumeração, se os resultados dessa operação estiverem fora do intervalo dos valores definidos para o enumerador, tudo o que o runtime pode fazer é interpretar o valor da variável como o valor inteiro correspondente.

- 14.** Pressione Enter para fechar o programa e retornar ao ambiente de programação do Visual Studio 2013.

Estruturas

O Capítulo 8 ilustrou que as classes definem tipos-referência, que são sempre criados no heap. Em alguns casos, a classe pode conter tão poucos dados que a sobrecarga de gerenciamento do heap se torna desproporcional. Nesses casos, é melhor definir o tipo como uma estrutura. Uma estrutura é um tipo-valor. Como as estruturas são armazenadas na pilha, desde que a estrutura seja razoavelmente pequena, a sobrecarga de gerenciamento da memória, em geral, é reduzida.

Como uma classe, uma estrutura pode ter campos, métodos e (com uma exceção importante, discutida mais adiante neste capítulo) construtores próprios.

Tipos-estrutura comuns

Talvez você não tenha percebido isso, mas já usou estruturas em exercícios anteriores neste livro. No C#, os tipos numéricos primitivos *int*, *long* e *float* são alias para as estruturas *System.Int32*, *System.Int64* e *System.Single*, respectivamente. Essas estruturas têm campos e métodos, e você pode chamar métodos nas variáveis e literais desses tipos. Por exemplo, todas essas estruturas fornecem um método *ToString* que pode converter um valor numérico na sua representação de string. Todas as instruções a seguir são válidas no C#:

```

int i = 55;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());

```

Você não vê com frequência esse uso do método *ToString*, porque o método *Console.WriteLine* o chama automaticamente quando ele é necessário. É mais comum utilizar alguns dos métodos estáticos expostos por essas estruturas. Por exemplo, nos capítulos anteriores você utilizou o método estático *int.Parse* para converter uma string no seu valor inteiro correspondente. Assim, você está chamando o método *Parse* da estrutura *Int32*:

```

string s = "42";
int i = int.Parse(s); // exatamente o mesmo que Int32.Parse

```

Essas estruturas também incluem alguns campos estáticos úteis. Por exemplo, *Int32.MaxValue* é o valor máximo que um *int* pode armazenar e *Int32.MinValue* é o menor valor que pode ser armazenado em um *int*.

A tabela a seguir mostra os tipos primitivos no C# e seus tipos equivalentes no Microsoft .NET Framework. Observe que os tipos *string* e *object* são classes (tipos-referência) em vez de estruturas.

Palavra-chave	Tipo equivalente	Classe ou estrutura
bool	System.Boolean	Estrutura
byte	System.Byte	Estrutura
decimal	System.Decimal	Estrutura
double	System.Double	Estrutura
float	System.Single	Estrutura
int	System.Int32	Estrutura
long	System.Int64	Estrutura
object	System.Object	Classe
sbyte	System.SByte	Estrutura
short	System.Int16	Estrutura
string	System.String	Classe
uint	System.UInt32	Estrutura
ulong	System.UInt64	Estrutura
ushort	System.UInt16	Estrutura

Declare uma estrutura

Para declarar seu tipo-estrutura, você utiliza a palavra-chave *struct* seguida pelo nome do tipo, seguido pelo corpo da estrutura entre chaves de abertura e fechamento. Sintaticamente, o processo é semelhante a declarar uma classe. Por exemplo, veja uma estrutura chamada *Time* que contém três campos *public int* chamados *hours*, *minutes* e *seconds*:

```
struct Time
{
    public int hours, minutes, seconds;
}
```

Assim como nas classes, na maioria dos casos não é recomendável tornar *public* os campos de uma estrutura; não há como controlar os valores armazenados nos campos *public*. Por exemplo, qualquer pessoa poderia configurar o valor de *minutes* ou *seconds* com um valor maior que 60. Uma ideia melhor é tornar os campos *private* e fornecer sua estrutura com construtores e métodos para inicializar e manipular esses campos, como mostrado neste exemplo:

```
struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm, int ss)
    {
        this.hours = hh % 24;
        this.minutes = mm % 60;
        this.seconds = ss % 60;
    }

    public int Hours()
    {
        return this.hours;
    }
}
```



Nota Por padrão, você não pode utilizar muitos dos operadores comuns nos seus próprios tipos-estrutura. Por exemplo, você não pode empregar operadores como o de igualdade (`==`) e o de desigualdade (`!=`) nas suas próprias variáveis de tipo-estrutura. No entanto, pode compará-las usando o método predefinido *Equals()* exposto por todas as estruturas e também pode declarar explicitamente e implementar operadores para seus próprios tipos-estrutura. A sintaxe para fazer isso será abordada no Capítulo 21, “Consulta a dados na memória usando expressões de consulta”.

Ao copiar uma variável do tipo-valor, você obtém duas cópias do valor. Por outro lado, ao copiar uma variável do tipo-referência, você obtém duas referências ao mesmo objeto. Em resumo, use estruturas para valores pequenos de dados para os quais elas sejam tão eficientes, ou quase tão eficientes, para copiar o valor quanto seriam para copiar um endereço. Utilize classes para dados mais complexos a fim de copiar com eficiência.



Dica Utilize estruturas para implementar conceitos simples cujas características principais são seus valores, em vez da funcionalidade que fornecem.

Entenda as diferenças entre estrutura e classe

Uma estrutura e uma classe são sintaticamente semelhantes, mas existem algumas diferenças importantes. Vamos examinar algumas dessas variações:

- Você não pode declarar um construtor padrão (um construtor sem parâmetros) para uma estrutura. O exemplo a seguir seria compilado se *Time* fosse uma classe, mas, como *Time* é uma estrutura, a compilação falha:

```
struct Time
{
    public Time() { ... } // erro de tempo de compilação
    ...
}
```

A razão pela qual você não pode declarar seu próprio construtor padrão em uma estrutura é que o compilador *sempre* gera um. Em uma classe, o compilador só gerará o construtor padrão se você não escrever seu próprio construtor. O construtor padrão gerado pelo compilador para uma estrutura sempre define os campos como *0*, *false* ou *null* – assim como para uma classe. Portanto, você deve garantir que um valor de estrutura criado pelo construtor padrão se comporte logicamente e faça sentido com esses valores padrão. Isso tem algumas ramificações que serão exploradas no próximo exercício.

Você pode inicializar os campos com valores diferentes, fornecendo um construtor não padrão. Contudo, quando faz isso, seu construtor não padrão deve inicializar explicitamente todos os campos de sua estrutura; a inicialização padrão não acontece mais. Se isso não for feito, ocorrerá um erro de tempo de compilação. Por exemplo, embora o exemplo seguinte fosse compilado e inicializasse silenciosamente *seconds* como *0* se *Time* fosse uma classe, como *Time* é uma estrutura, a compilação falha:

```
struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm)
    {
        this.hours = hh;
        this.minutes = mm;
    } // erro de tempo de compilação: seconds não inicializada
}
```

- Em uma classe, você pode inicializar os campos de instância no seu ponto de declaração. Em uma estrutura, isso não é possível. O exemplo a seguir compilaria se *Time* fosse uma classe, mas, como *Time* é uma estrutura, ele causa um erro de tempo de compilação:

```
struct Time
{
    private int hours = 0; // erro de tempo de compilação
```

```

private int minutes;
private int seconds;
...
}

```

A tabela abaixo resume as principais diferenças entre uma estrutura e uma classe.

Pergunta	Estrutura	Classe
Esse é um tipo-valor ou um tipo-referência?	Uma estrutura é um tipo-valor.	Uma classe é um tipo-referência.
As instâncias são colocadas na pilha ou no heap?	As instâncias de estrutura são chamadas valores e residem na pilha.	As instâncias de classe são chamadas objetos e são colocadas no heap.
Você pode declarar um construtor padrão?	Não.	Sim.
Se você declarar seu construtor, o compilador ainda gerará o construtor padrão?	Sim.	Não.
Se você não inicializar um campo no seu construtor, o compilador o inicializará automaticamente para você?	Não.	Sim.
Você pode inicializar campos de instância no seu ponto de declaração?	Não.	Sim.

Existem outras diferenças entre classes e estruturas no que se refere à herança. Essas diferenças serão abordadas no Capítulo 12 “Herança”.

Declare variáveis de estrutura

Após ter definido um tipo-estrutura, você pode utilizá-lo exatamente da mesma maneira que qualquer outro tipo. Por exemplo, se você definiu a estrutura *Time*, pode criar variáveis, campos e parâmetros do tipo *Time*, como mostrado neste exemplo:

```

struct Time
{
    private int hours, minutes, seconds;
    ...
}

class Example
{
    private Time currentTime;

    public void Method(Time parameter)
    {
        Time localVariable;
        ...
    }
}

```



Nota Assim como nas enumerações, você pode criar uma versão nullable de uma variável de estrutura utilizando o modificador ?. Você pode então atribuir o valor *null* à variável:

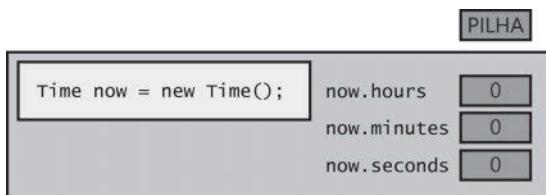
```
Time? currentTime = null;
```

Entenda a inicialização de estruturas

Anteriormente neste capítulo, vimos como os campos em uma estrutura podem ser inicializados com um construtor. Se você chamar um construtor, as várias regras descritas antes garantirão que todos os campos na estrutura sejam inicializados:

```
Time now = new Time();
```

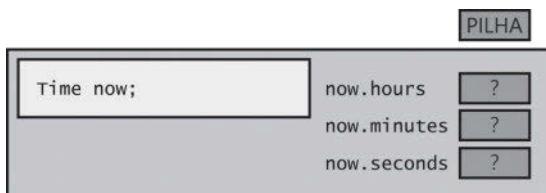
A figura a seguir ilustra os campos dessa estrutura:



Entretanto, como as estruturas são tipos-valor, você também pode criar variáveis de estrutura sem chamar um construtor, como no exemplo a seguir:

```
Time now;
```

Desta vez, a variável é criada, mas seus campos permanecem no estado não inicializado. A figura a seguir ilustra o estado dos campos na variável *now*. Qualquer tentativa de acessar os valores contidos nesses campos resultará em um erro de compilação:



Observe que, em ambos os casos, a variável *Time* é criada na pilha.

Se você escreveu seu próprio construtor de estrutura, pode também utilizá-lo para inicializar uma variável de estrutura. Conforme já explicado neste capítulo, um construtor de estrutura deve sempre inicializar explicitamente todos os seus campos. Por exemplo:

```
struct Time
{
    private int hours, minutes, seconds;
```

```

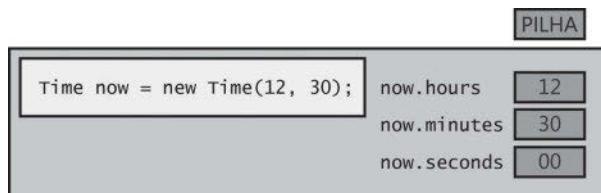
...
public Time(int hh, int mm)
{
    hours = hh;
    minutes = mm;
    seconds = 0;
}
}

```

O exemplo a seguir inicializa *now* ao chamar um construtor definido pelo usuário:

```
Time now = new Time(12, 30);
```

A ilustração a seguir mostra o efeito desse exemplo:



Está na hora de colocar esse conhecimento em prática. No exercício a seguir, você vai criar e utilizar uma estrutura para representar uma data.

Crie e utilize um tipo-estrutura

1. No projeto *StructsAndEnums*, exiba o arquivo *Date.cs* na janela Code and Text Editor.
2. Adicione uma estrutura chamada *Date* dentro do namespace *StructsAndEnums*.

Essa estrutura deve conter três campos privados: um *year* nomeado do tipo *int*, um *month* nomeado do tipo *Month* (utilizando a enumeração que você criou no exercício anterior) e um *day* nomeado do tipo *int*. A estrutura *Date* deve ser igual a esta:

```
struct Date
{
    private int year;
    private Month month;
    private int day;
}
```

Agora considere o construtor padrão que o compilador vai gerar para *Date*. Esse construtor define *year* como 0, *month* como 0 (o valor de January) e *day* como 0. O valor *year* 0 não é válido (porque não há ano 0) e o valor *day* 0 também não é válido (porque cada mês começa no dia 1). Uma maneira de corrigir esse problema é converter os valores *year* e *day* implementando a estrutura *Date* para que, quando o campo *year* contiver o valor *Y*, esse valor represente o ano *Y* + 1900 (ou você pode escolher um século diferente, se preferir) e, quando o campo *day* contiver o valor *D*, esse valor represente o dia *D* + 1. O construtor padrão vai então configurar os três campos com valores que representam a data de 1 de janeiro de 1900.

Se você pudesse substituir o construtor padrão e escrever o seu próprio, isso não seria problema, pois então poderia inicializar os campos *year* e *day* diretamente com valores válidos. Contudo, não é possível fazer isso e, assim, você precisa implementar a lógica em sua estrutura para converter os valores gerados pelo compilador em valores significativos para o domínio de seu problema.

Contudo, embora não seja possível substituir o construtor padrão, ainda é uma boa prática definir construtores não padrão para permitir que o usuário inicialize explicitamente os campos de uma estrutura com valores não padrão significativos.

3. Adicione um construtor *public* à estrutura *Date*. Esse construtor deve receber três parâmetros: um *int* chamado *ccyy* para *year*, um *Month* chamado *mm* para *month* e um *int* chamado *dd* para *day*. Utilize esses três parâmetros para inicializar os campos correspondentes. Um campo *year* com o valor *Y* representa o ano *Y* + 1900; portanto, você precisa inicializar o campo *year* com o valor *ccyy* – 1900. Um campo *day* com o valor *D* representa o dia *D* + 1; assim, você precisa inicializar o campo *day* com o valor *dd* – 1.

A estrutura *Date* agora deve se parecer com isto (o construtor é mostrado em negrito):

```
struct Date
{
    private int year;
    private Month month;
    private int day;

    public Date(int ccyy, Month mm, int dd)
    {
        this.year = ccyy - 1900;
        this.month = mm;
        this.day = dd - 1;
    }
}
```

4. Adicione um método *public* chamado *ToString* à estrutura *Date*, após o construtor. Esse método não recebe argumento algum e retorna uma representação da data na forma de string. Lembre-se de que o valor do campo *year* representa *year* + 1900 e o valor do campo *day* representa *day* + 1.



Nota O método *ToString* é um pouco diferente dos métodos que você viu até aqui. Cada tipo, incluindo estruturas e classes que você define, terá automaticamente um método *ToString*, queira você ou não. Seu comportamento padrão é converter os dados de uma variável em uma representação de string desses dados. Algumas vezes, o comportamento padrão é significativo, outras vezes, é menos que isso. Por exemplo, o comportamento padrão do método *ToString* gerado para a classe *Date* simplesmente gera a string "StructsAndEnums.Date". Para citar Zaphod Beeblebrox em *The Restaurant at the End of the Universe* (por Douglas Adams, Pan MacMillan, 1980), isso é "inteligente, mas estúpido". Você precisa definir uma nova versão desse método que substitua o comportamento padrão utilizando a palavra-chave *override*. A redefinição de métodos será discutida com mais detalhes no Capítulo 12.

O método *ToString* deve ser semelhante a este:

```
struct Date
{
    ...
    public override string ToString()
    {
        string data = String.Format("{0} {1} {2}", this.month, this.day + 1,
                                     this.year + 1900);
        return data;
    }
}
```

O método *Format* da classe *String* torna possível formatar dados. Ele opera de modo semelhante ao método *Console.WriteLine*, exceto pelo fato de que, em vez de exibir dados no console, ele retorna o resultado formatado como uma string. Neste exemplo, os parâmetros posicionais são substituídos pelas representações de texto dos valores do campo *month*, a expressão *this.day + 1* e a expressão *this.year + 1900*. O método *ToString* retorna como resultado a string formatada.

5. Exiba o arquivo Program.cs na janela Code and Text Editor.
6. No método *doWork*, transforme em comentário as quatro instruções existentes.
7. Adicione ao método *doWork* instruções que declarem uma variável local chamada *defaultDate* e a inicializem com um valor *Date* construído por meio do construtor *Date* padrão. Adicione outra instrução ao método *doWork* para escrever a variável *defaultDate* no console chamando *Console.WriteLine*.



Nota O método *Console.WriteLine* chama automaticamente o método *ToString* do seu argumento para formatar o argumento como uma string.

O método *doWork* deve agora ser semelhante a este:

```
static void doWork()
{
    ...
    Date defaultDate = new Date();
    Console.WriteLine(defaultDate);
}
```



Nota Quando você digita a palavra-chave *new*, o IntelliSense detecta automaticamente que existem dois construtores disponíveis para o tipo *Date*.

8. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo. Verifique se a data January 1 1900 foi escrita no console.

9. Pressione a tecla Enter para retornar ao ambiente de programação do Visual Studio 2013.
10. Na janela Code and Text Editor, retorne ao método *doWork* e adicione mais duas instruções. Na primeira instrução, declare uma variável local chamada *weddingAnniversary* e a inicialize como July 4 2013. (Eu realmente me casei no Dia da Independência dos EUA, embora tenha sido há muitos anos.) Na segunda instrução, escreva o valor de *weddingAnniversary* no console.

O método *doWork* deve agora ser semelhante a este:

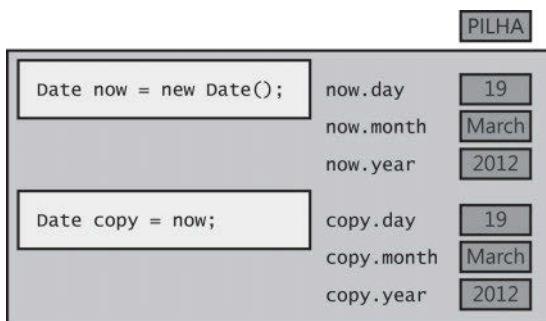
```
static void doWork()
{
    ...
    Date weddingAnniversary = new Date(2013, Month.July, 4);
    Console.WriteLine(weddingAnniversary);
}
```

11. No menu Debug, clique em Start Without Debugging e, então, confirme que July 4 2013 está escrito no console abaixo da informação anterior.
12. Pressione Enter para fechar o programa e retornar ao Visual Studio 2013.

Copie variáveis de estrutura

Você só pode inicializar ou atribuir uma variável de estrutura a outra variável de estrutura se a do lado direito estiver totalmente inicializada (ou seja, se todos os seus campos estiverem preenchidos com valores válidos e não com valores indefinidos). O exemplo a seguir é compilado porque *now* está completamente inicializada. A figura mostra os resultados de uma atribuição assim (essa imagem foi criada em terça-feira, 19 de março de 2013).

```
Date now = new Date();
Date copy = now;
```



A compilação do exemplo a seguir falha porque *now* não está inicializada:

```
Date now;
Date copy = now; // erro de tempo de compilação: now não foi atribuída
```

Quando você copia uma variável de estrutura, cada campo posicionado no lado esquerdo é definido diretamente a partir do campo correspondente no lado direito.

Essa cópia ocorre como uma operação simples e rápida, que copia o conteúdo da estrutura inteira e que nunca lança uma exceção. Compare esse comportamento com a ação equivalente caso *Time* fosse uma classe, em que as duas variáveis (*now* e *copy*) terminariam fazendo referência ao *mesmo* objeto no heap.



Nota Se você for programador de C++, deve observar que esse comportamento de copy não pode ser personalizado.

No último exercício deste capítulo, você vai comparar o comportamento da cópia de uma estrutura com o de uma classe.

Compare o comportamento de uma estrutura com o de uma classe

1. No projeto *StructsAndEnums*, exiba o arquivo *Date.cs* na janela Code and Text Editor.
2. Adicione o método a seguir à estrutura *Date*. Esse método adianta a data na estrutura em um mês. Se, após avançar a data, o valor do campo *month* ultrapassar o mês de dezembro, esse código redefinirá o mês com janeiro e incrementará o valor do campo *year* em 1.

```
struct Date
{
    ...
    public void AdvanceMonth()
    {
        this.month++;
        if (this.month == Month.December + 1)
        {
            this.month = Month.January;
            this.year++;
        }
    }
}
```

3. Exiba o arquivo *Program.cs* na janela Code and Text Editor.
4. No método *doWork*, transforme em comentário as duas primeiras instruções que não são comentários, que criam e exibem o valor da variável *defaultDate*.
5. Adicione o seguinte código, mostrado em negrito, ao final do método *doWork*. Esse código gera uma cópia da variável *weddingAnniversary*, chamada *weddingAnniversaryCopy*, e imprime o valor dessa nova variável.

```
static void doWork()
{
    ...
    Date weddingAnniversaryCopy = weddingAnniversary;
    Console.WriteLine("Value of copy is {0}", weddingAnniversaryCopy);
}
```

- 6.** Adicione as seguintes instruções mostradas em negrito ao final do método *doWork*. Essas instruções chamam o método *AdvanceMonth* da variável *weddingAnniversary* e, depois, exibem o valor das variáveis *weddingAnniversary* e *weddingAnniversaryCopy*:

```
static void doWork()
{
    ...
    weddingAnniversary.AdvanceMonth();
    Console.WriteLine("New value of weddingAnniversary is {0}", weddingAnniversary);
    Console.WriteLine("Value of copy is still {0}", weddingAnniversaryCopy);
}
```

- 7.** No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo. Verifique que a janela do console exibe as seguintes mensagens:

```
July 4 2013
Value of copy is July 4 2013
New value of weddingAnniversary is August 4 2013
Value of copy is still July 4 2013
```

A primeira mensagem mostra o valor inicial da variável *weddingAnniversary* (July 4 2013). A segunda mensagem exibe o valor da variável *weddingAnniversaryCopy*. Observe que ela contém a mesma data armazenada na variável *weddingAnniversary* (July 4 2013). A terceira mensagem exibe o valor da variável *weddingAnniversary* após mudar o mês da variável para agosto (August 4 2013). A última instrução exibe o valor da variável *weddingAnniversaryCopy*. Observe que seu valor original de July 4 2013 não mudou.

Se *Date* fosse uma classe, a criação de uma cópia referenciaria o mesmo objeto na memória que a instância original. Portanto, mudar o mês na instância original também mudaria a data referenciada por meio da cópia. Você vai conferir essa afirmação nos próximos passos.

- 8.** Pressione Enter para retornar ao Visual Studio 2013.
- 9.** Exiba o arquivo Date.cs na janela Code and Text Editor.
- 10.** Mude a estrutura *Date* para uma classe, como mostrado em negrito no exemplo de código a seguir:

```
class Date
{
    ...
}
```

- 11.** No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo novamente. Verifique que a janela do console exibe as seguintes mensagens:

```
July 4 2013
Value of copy is July 4 2013
New value of weddingAnniversary is August 4 2013
Value of copy is still August 4 2013
```

As três primeiras mensagens são as mesmas anteriores. Entretanto, a quarta mensagem mostra que o valor da variável *weddingAnniversaryCopy* mudou para August 4 2013.

- 12.** Pressione Enter para retornar ao Visual Studio 2013.

Estruturas e compatibilidade com o Windows Runtime no Windows 8 e no Windows 8.1

Todos os aplicativos C# são executados com o Common Language Runtime (CLR) do .NET Framework. O CLR é responsável por fornecer um ambiente seguro e protegido para o código de seu aplicativo, na forma de uma *máquina virtual* (se você conhece Java, esse conceito deve ser familiar). Quando um aplicativo C# é compilado, o compilador converte o código C# em um conjunto de instruções utilizando um código de pseudomáquina chamado Common Intermediate Language (CIL). São essas as instruções armazenadas em um assembly. Quando um aplicativo C# é executado, o CLR assume a responsabilidade por converter as instruções CIL em instruções de máquina reais que o processador de seu computador pode entender e executar. Esse ambiente todo é conhecido como ambiente de execução *gerenciada*, e os programas C# são muitas vezes referidos como *código gerenciado*. Também é possível escrever código gerenciado em outras linguagens suportadas pelo .NET Framework, como Visual Basic e F#.

No Windows 7 e anteriores, é possível ainda escrever aplicativos não gerenciados, também conhecidos como *código nativo*, baseados nas APIs Win32, as APIs que fazem interface diretamente com o sistema operacional Windows (o CLR também converte muitas das funções do .NET Framework em chamadas de API Win32, caso você esteja executando um aplicativo gerenciado, embora esse processo seja totalmente transparente para seu código). Para fazer isso, utilize uma linguagem como C++. O .NET Framework torna possível integrar código gerenciado em aplicativos não gerenciados e vice-versa, por meio de um conjunto de tecnologias de interoperabilidade. Os detalhes do funcionamento dessas tecnologias e como você as utiliza estão fora dos objetivos deste livro – basta dizer que isso nem sempre foi simples.

O Windows 8 e o Windows 8.1 oferecem uma estratégia alternativa, na forma do Windows Runtime, ou WinRT. O WinRT fornece uma camada sobre a API Win32 (e outras APIs selecionadas, nativas do Windows), otimizada para dispositivos e interfaces do usuário baseados em toque, como os encontrados nos tablets para Windows 8 e Windows 8.1. Ao compilar um aplicativo nativo no Windows 8 ou Windows 8.1, você utiliza as APIs expostas pelo WinRT, em vez do Win32. Da mesma forma, o CLR do Windows 8 e do Windows 8.1 também utiliza WinRT; todo código gerenciado escrito com C# ou qualquer outra linguagem gerenciada ainda é executado pelo CLR, mas, em tempo de execução, o CLR converte seu código em chamadas de API WinRT, em vez de Win32. Entre elas, o CLR e o WinRT são responsáveis por gerenciar e executar seu código com segurança.

Um propósito importante do WinRT é simplificar a interoperabilidade entre as linguagens, para que seja possível integrar com mais facilidade componentes desenvolvidos em diferentes linguagens de programação, em um único aplicativo sem emendas. Contudo, essa simplicidade tem um custo, e você precisa estar preparado para assumir alguns compromissos, com base nos diferentes conjuntos de recursos das várias linguagens disponíveis. Em especial, por motivos históricos, embora o C++ suporte estruturas, ele não reconhece funções membro. Em termos de C#, uma função membro é um método de instância. Assim, se você estiver compilando estruturas C# que deseja empacotar em uma biblioteca para disponibilizar para desenvolvedores que estejam programando em C++ (ou qualquer outra linguagem não gerenciada), essas estruturas não devem conter métodos de instância. A mesma restrição se aplica aos métodos estáticos em estruturas. Se quiser incluir métodos de instância ou estáticos, você deve transformar sua estrutura em uma classe. Além disso, as estruturas não podem conter campos privados e todos os campos públicos devem ser tipos primitivos do C#, de acordo com os tipos-valor ou strings.

O WinRT também impõe algumas outras restrições para classes e estruturas C#, caso você queira disponibilizá-las para aplicativos nativos. O Capítulo 12 fornece mais informações.

Resumo

Neste capítulo, vimos como criar e utilizar enumerações e estruturas. Você conheceu algumas semelhanças e diferenças entre uma estrutura e uma classe e viu como definir construtores para inicializar os campos em uma estrutura. Aprendeu também a representar uma estrutura como uma string, substituindo o método *ToString*.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 10, “Arrays”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar uma enumeração	<p>Escreva a palavra-chave enum, seguida pelo nome do tipo, seguido por um par de chaves contendo uma lista separada por vírgulas dos nomes literais da enumeração. Por exemplo:</p> <pre>enum Season { Spring, Summer, Fall, Winter }</pre>

Para	Faça isto
Declarar uma variável de tipo enumerado	Escreva o nome da enumeração à esquerda, seguido pelo nome da variável, seguido por ponto e vírgula. Por exemplo: <code>Season currentSeason;</code>
Atribuir um valor a uma variável do tipo enumerado	Escreva o nome do literal de enumeração em combinação com o nome da enumeração à qual ele pertence. Por exemplo: <code>currentSeason = Spring; // erro</code> <code>currentSeason = Season.Spring; // correto</code>
Declarar um tipo-estrutura	Escreva a palavra-chave struct, seguida pelo nome do tipo-estrutura, seguido pelo corpo da estrutura (os construtores, métodos e campos). Por exemplo: <code>struct Time</code> <code>{</code> <code> public Time(int hh, int mm, int ss)</code> <code> { ... }</code> <code> ...</code> <code> private int hours, minutes, seconds;</code> <code>}</code>
Declarar uma variável de estrutura	Escreva o nome do tipo-estrutura, seguido pelo nome da variável, seguido por um ponto e vírgula. Por exemplo: <code>Time now;</code>
Inicializar uma variável de estrutura com um valor	Inicialize a variável com um valor de estrutura criado pela chamada do construtor da estrutura. Por exemplo: <code>Time lunch = new Time(12, 30, 0);</code>

CAPÍTULO 10

Arrays

Neste capítulo, você vai aprender a:

- Declarar variáveis do tipo array.
- Preencher um array com um conjunto de itens de dados.
- Acessar os itens de dados armazenados em um array.
- Iterar pelos itens de dados de um array.

Você viu a criação e a utilização de tipos diferentes de variáveis. Porém, os exemplos de variáveis apresentados até aqui têm algo em comum – o armazenamento de informações sobre um único item (*int*, *float*, *Circle*, *Date* e assim por diante). Se for preciso manipular um conjunto de itens, o que acontece? A criação de uma variável para cada item do conjunto é uma solução, mas isso possibilita muitas outras questões: de quantas variáveis você precisa? Como você deve nomeá-las? Se fosse necessário executar a mesma operação em cada item do conjunto (como incrementar cada uma das variáveis em um conjunto de inteiros), como seria possível evitar a repetição excessiva de código? Essa solução pressupõe que, ao escrever o programa, você saiba de quantos itens precisará, mas com que frequência isso acontece? Por exemplo, se você estiver escrevendo um aplicativo que lê e processa os registros de um banco de dados, quantos registros estão no banco de dados e qual a probabilidade de esse número mudar?

Os arrays apresentam um mecanismo que auxilia na resolução desses problemas.

Declare e crie um array

Um *array* é uma sequência não ordenada de itens. Todos os itens em um array têm o mesmo tipo, ao contrário dos campos em uma estrutura ou classe, que têm tipos diferentes. Os itens em um array residem em um bloco contíguo da memória e são acessados por meio de um índice; ao contrário dos campos em uma estrutura ou classe, que são acessados pelo nome.

Declare variáveis de array

Você declara uma variável de array especificando o nome do tipo de elemento, seguido por um par de colchetes, seguido pelo nome da variável. Os colchetes significam que a variável é um array. Por exemplo, para declarar um array de variáveis *int* chamada *pins* (para armazenar um conjunto de números de identificação pessoal), você pode escrever o seguinte:

```
int[] pins; // Números de Identificação Pessoal
```



Nota Se você for programador em Microsoft Visual Basic, deve observar que na declaração são utilizados colchetes, não parênteses. Caso conheça C e C++, deve observar também que o tamanho do array não faz parte da declaração. Os programadores Java devem discernir que os colchetes devem ser colocados *antes* do nome da variável.

Os elementos do array não estão restritos aos tipos primitivos. Também é possível criar arrays de estruturas, enumerações e classes. Por exemplo, você pode desenvolver um array de estruturas *Date* assim:

```
Date[] datas;
```



Dica Muitas vezes, é útil dar nomes no plural para as variáveis de array, como *locais* (onde cada elemento é um *Local*), *pessoas* (onde cada elemento é uma *Pessoa*) ou *tempos* (onde cada elemento é um *Tempo*).

Crie uma instância de array

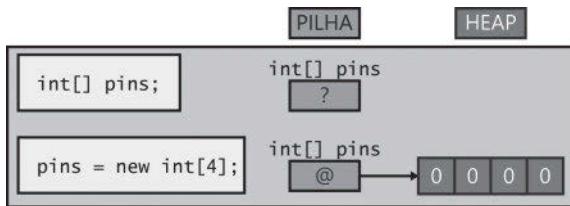
Os arrays são tipos-referência, independentemente do tipo dos seus elementos. Isso significa que uma variável de array *referencia* um bloco contíguo de memória armazenando elementos de array no heap, assim como uma variável de classe referencia um objeto no heap. (Para uma descrição de valores e referências, e das diferenças entre pilha e heap, consulte o Capítulo 8, “Valores e referências”.) Essa regra se aplica independentemente do tipo dos itens de dados do array. Mesmo que o array contenha um tipo-valor, como *int*, a memória ainda será alocada no heap; esse é o único caso em que os tipos-valor não alocam memória na pilha.

Lembre-se de que, ao declarar uma variável de classe, a memória não é alocada para o objeto até que você crie a instância utilizando *new*. Os arrays seguem o mesmo padrão: quando você declara uma variável de array, não declara seu tamanho e as memórias não são alocadas (somente a utilizada para armazenar a referência na pilha). O array aloca memória apenas quando a instância é criada, sendo esse também o ponto no qual você especifica o tamanho do array.

Para criar uma instância de array, você utiliza a palavra-chave *new*, seguida pelo tipo do elemento, seguido pelo tamanho (entre colchetes) do array que está sendo criado. Criar um array também inicializa seus elementos com os valores padrão agora familiares (*0*, *null* ou *false*, dependendo se o tipo é numérico, uma referência ou um booleano, respectivamente). Por exemplo, para criar e inicializar um novo array de quatro inteiros para a variável *pins* declarada antes, você escreve o seguinte:

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

A figura a seguir ilustra o que acontece quando você declara um array e depois, quando cria uma instância do array:



Como a memória para a instância do array é alocada dinamicamente, o tamanho do array não precisa ser uma constante; ele pode ser calculado em tempo de execução, como mostrado neste exemplo:

```
int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];
```

Você também pode criar um array cujo tamanho é 0. Isso talvez pareça estranho, mas é útil para situações nas quais o tamanho do array é determinado dinamicamente e pode até ser 0. Um array de tamanho 0 não é um array *null*; é um array contendo zero elementos.

Preencha e utilize um array

Quando você cria uma instância de array, todos os elementos do array são inicializados com um valor padrão que depende do seu tipo. Por exemplo, o padrão para todos os valores numéricos é 0, os objetos são inicializados com *null*, valores *DateTime* são definidos com a data e hora "01/01/0001 00:00:00" e strings são inicializadas com *null*. Se preferir, você pode modificar esse comportamento e inicializar os elementos de um array com valores específicos. Você consegue isso fornecendo uma lista de valores separados por vírgula e entre chaves. Por exemplo, para inicializar *pins* como um array de quatro variáveis *int* cujos valores são 9, 3, 7 e 2, escreva isto:

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

Os valores entre as chaves não precisam ser constantes; eles podem ser valores calculados em tempo de execução, como mostrado no exemplo a seguir, que preenche o array *pins* com quatro números aleatórios:

```
Random r = new Random();
int[] pins = new int[4]{ r.Next() % 10, r.Next() % 10,
r.Next() % 10, r.Next() % 10 };
```



Nota A classe *System.Random* é um gerador de números pseudoaleatórios. O método *Next* retorna um inteiro aleatório não negativo no intervalo de 0 a *Int32.MaxValue*, por padrão. O método *Next* é sobreescarregado e outras versões permitem especificar o valor mínimo e o valor máximo do intervalo. O construtor padrão para a classe *Random* semeia o gerador de números aleatórios com um valor de semente baseado na data/hora, o que reduz a possibilidade de a classe duplicar uma sequência de números aleatórios. Usando uma versão sobreescarregada do construtor, você pode fornecer seu próprio valor de semente. Assim, pode gerar uma sequência repetível de números aleatórios para propósitos de teste.

O número de valores entre as chaves deve corresponder exatamente ao tamanho da instância do array que está sendo criado:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // erro de tempo de compilação
int[] pins = new int[4]{ 9, 3, 7 };    // erro de tempo de compilação
int[] pins = new int[4]{ 9, 3, 7, 2 }; // OK
```

Ao inicializar uma variável de array dessa maneira, você pode omitir a expressão `new` e o tamanho do array. Nesse caso, o compilador calcula o tamanho a partir do número de inicializadores e gera o código para criar o array, como no exemplo a seguir:

```
int[] pins = { 9, 3, 7, 2 };
```

Se você criar um array de estruturas ou objetos, poderá inicializar cada estrutura do array chamando o construtor da estrutura ou da classe, como mostrado neste exemplo:

```
Time[] schedule = { new Time(12,30), new Time(5,30) };
```

Crie um array implicitamente tipado

Quando você declara um array, o tipo do elemento precisa corresponder ao tipo dos elementos que você quer armazenar no array. Por exemplo, se declarar `pins` como um array de `int`, como mostrado nos exemplos anteriores, você não poderá armazenar um `double`, uma `string`, uma `struct` ou qualquer coisa que não seja um `int` nesse array. Se especificar uma lista de inicializadores ao declarar um array, você poderá deixar o compilador C# inferir o tipo real dos elementos do array, assim:

```
var names = new[]{"John", "Diana", "James", "Francesca"};
```

Nesse exemplo, o compilador C# determina que a variável `names` é um array de strings. Vale indicar algumas peculiaridades sintáticas nessa declaração. Primeiro, você omite os colchetes do tipo; a variável `names` nesse exemplo é declarada simplesmente como `var` e não `var[]`. Segundo, você deve especificar o operador `new` e os colchetes antes da lista inicializadora.

Se utilizar essa sintaxe, você precisará assegurar que todos os inicializadores tenham o mesmo tipo. O próximo exemplo causa o erro de tempo de compilação “No best type found for implicitly typed array” (Não foi possível encontrar um tipo mais adequado para o array implicitamente tipado):

```
var bad = new[]{"John", "Diana", 99, 100};
```

Mas, em alguns casos, o compilador converterá elementos para um tipo diferente, se isso fizer sentido. No código a seguir, o array `numbers` é um array de valores `double`, porque as constantes 3.5 e 99.999 são ambas `double` e o compilador C# pode converter os valores inteiros 1 e 2 em valores `double`:

```
var numbers = new[]{1, 2, 3.5, 99.999};
```

Geralmente, é melhor evitar a mistura de tipos, esperando que o compilador os converta para você.

Arrays implicitamente tipados são mais úteis quando você está trabalhando com tipos anônimos, conforme descritos no Capítulo 7, “Criação e gerenciamento de clas-

ses e objetos". O código a seguir cria um array de objetos anônimos, cada um contendo dois campos que especificam o nome e a idade dos membros da minha família:

```
var names = new[] { new { Name = "John", Age = 47 },
                    new { Name = "Diana", Age = 46 },
                    new { Name = "James", Age = 20 },
                    new { Name = "Francesca", Age = 18 } };
```

Os campos dos tipos anônimos devem ser os mesmos para cada elemento do array.

Acesse um elemento individual de um array

Para acessar um elemento individual de um array, você deve fornecer um índice indicando o elemento desejado. Os índices dos arrays começam em zero; assim, o elemento inicial de um array reside no índice 0 e não no índice 1. Um valor de índice de 1 acessa o segundo elemento. Por exemplo, você pode ler o conteúdo do elemento 2 do array *pins* para armazená-lo em uma variável *int* utilizando o código a seguir:

```
int myPin;
myPin = pins[2];
```

Da mesma maneira, você pode alterar o conteúdo de um array atribuindo um valor a um elemento indexado:

```
myPin = 1645;
pins[2] = myPin;
```

Todo acesso aos elementos do array é verificado quanto aos limites. Se você especificar um índice menor que 0 ou maior ou igual ao tamanho do array, o compilador lançará uma exceção *IndexOutOfRangeException*, como neste exemplo:

```
try
{
    int[] pins = { 9, 3, 7, 2 };
    Console.WriteLine(pins[4]); // erro, o 4º é o último elemento está no índice 3
}
catch (IndexOutOfRangeException ex)
{
    ...
}
```

Itere por um array

Na verdade, todos os arrays são instâncias da classe *System.Array* do Microsoft .NET Framework, e essa classe define algumas propriedades e métodos úteis. Por exemplo, você pode consultar a propriedade *Length* para descobrir quantos elementos um array contém e iterar por todos os elementos de um array utilizando uma instrução *for*. O código de exemplo a seguir escreve os valores dos elementos do array *pins* no console:

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```



Nota *Length* é uma propriedade e não um método, razão pela qual não é necessário usar parênteses para chamá-la. Ela indica o comprimento de um array. Você pode aprender sobre as propriedades no Capítulo 15, “Implementação de propriedades para acessar campos”.

É comum que novos programadores se esqueçam de que arrays iniciam no elemento 0 e que o último elemento está na posição *Length* – 1. O C# fornece a instrução *foreach* com a qual é possível iterar pelos elementos de um array sem se preocupar com essas questões. Por exemplo, veja a instrução *for* anterior reescrita como uma instrução *foreach* equivalente:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

A instrução *foreach* declara uma variável de iteração (neste exemplo, *int pin*) que recebe automaticamente o valor de cada elemento do array. O tipo dessa variável deve corresponder ao tipo dos elementos do array. A instrução *foreach* é a maneira preferida de iterar por um array; ela expressa diretamente a intenção do código e toda a estrutura do loop *for* desaparece. Mas, em alguns casos, você verá que é melhor reverter para uma instrução *for*:

- Uma instrução *foreach* sempre itera por todo o array. Se quiser iterar apenas por uma parte conhecida de um array (por exemplo, a primeira metade) ou pular certos elementos (por exemplo, a cada três elementos), é mais fácil utilizar uma instrução *for*.
- Uma instrução *foreach* sempre itera do índice 0 ao índice *Length* – 1. Se quiser iterar de trás para frente ou em alguma outra sequência, é mais fácil utilizar uma instrução *for*.
- Se o corpo do loop precisa saber o índice do elemento em vez do valor do elemento, você terá de utilizar uma instrução *for*.
- Se precisar modificar os elementos do array, você terá de utilizar uma instrução *for*. Isso acontece porque a variável de iteração da instrução *foreach* é uma cópia somente-leitura de cada elemento do array.



Dica É perfeitamente seguro tentar iterar por um array de comprimento zero utilizando uma instrução *foreach*.

Você pode declarar a variável de iteração como *var* e deixar o compilador C# deduzir o tipo da variável a partir do tipo dos elementos no array. Isso é especialmente útil se você não conhecer o tipo dos elementos no array, por exemplo, quando o array contém objetos anônimos. O exemplo a seguir demonstra como é possível iterar pelo array dos membros da família mostrado anteriormente:

```
var names = new[] { new { Name = "John", Age = 47 },
    new { Name = "Diana", Age = 46 },
```

```

        new { Name = "James", Age = 20 },
        new { Name = "Francesca", Age = 18 } };
foreach (var familyMember in names)
{
    Console.WriteLine("Name: {0}, Age: {1}", familyMember.Name, familyMember.Age);
}

```

Passe arrays como parâmetros e valores de retorno para um método

Você pode definir métodos que recebem arrays como parâmetros ou os passam de volta como valores de retorno.

A sintaxe para passar um array como parâmetro é a mesma da declaração de um array. Por exemplo, o código a seguir define um método chamado *ProcessData* que recebe como parâmetro um array de valores inteiros. O corpo do método itera pelo array e executa algum processamento não especificado em cada elemento:

```

public void ProcessData(int[] data)
{
    foreach (int i in data)
    {
        ...
    }
}

```

É importante lembrar que os arrays são objetos de referência; portanto, se você modificar o conteúdo de um array passado como parâmetro dentro de um método, como *ProcessData*, a modificação será visível por todas as referências ao array, incluindo o argumento original passado como parâmetro.

Para retornar um array de um método, especifique o tipo do array como o tipo de retorno. No método, você cria e preenche o array. O exemplo a seguir solicita do usuário o tamanho de um array, seguido dos dados de cada elemento. O array criado pelo método é passado de volta como o valor de retorno:

```

public int[] ReadData()
{
    Console.WriteLine("How many elements?");
    string reply = Console.ReadLine();
    int numElements = int.Parse(reply);

    int[] data = new int[numElements];
    for (int i = 0; i < numElements; i++)
    {
        Console.WriteLine("Enter data for element {0}", i);
        reply = Console.ReadLine();
        int elementData = int.Parse(reply);
        data[i] = elementData;
    }
    return data;
}

```

Você pode chamar o método *ReadData* como segue:

```
int[] data = ReadData();
```

Parâmetros de array e o método Main

Talvez você tenha notado que o método *Main* de um aplicativo recebe como parâmetro um array de strings:

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

Lembre-se de que o método *Main* é chamado quando seu programa começa a ser executado; ele é o ponto de entrada de seu aplicativo. Se você inicia o aplicativo a partir da linha de comando, pode especificar argumentos adicionais. O sistema operacional Microsoft Windows passa esses argumentos para o Common Language Runtime (CLR), o qual, por sua vez, os passa como argumentos para o método *Main*. Esse mecanismo proporciona uma maneira simples de permitir que o usuário forneça informações quando um aplicativo começa a ser executado, em vez de solicitá-las interativamente, o que é útil se você deseja construir utilitários que podem ser executados a partir de scripts automatizados.

O exemplo a seguir foi extraído de um aplicativo utilitário hipotético chamado *MyFileUtil* que processa arquivos. Ele espera um conjunto de nomes de arquivo na linha de comando e chama o método *ProcessFile* (não mostrado) para tratar de cada arquivo especificado:

```
static void Main(string[] args)
{
    foreach (string filename in args)
    {
        ProcessFile(filename);
    }
}
```

O usuário poderia executar o aplicativo *MyFileUtil* a partir da linha de comando, desta forma:

```
MyFileUtil C:\Temp\ TestData.dat C:\Users\John\Documents\MyDoc.txt
```

Cada argumento de linha de comando é separado por um espaço. Fica por conta do aplicativo *MyFileUtil* verificar se esses argumentos são válidos.

Copie arrays

Os arrays são tipos-referência. (Lembre-se de que um array é uma instância da classe *System.Array*.) Uma variável de array contém uma referência a uma instância do array. Isso significa que, ao copiar uma variável de array, você realmente acaba ficando com duas referências à mesma instância do array, como mostrado no exemplo a seguir:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias e pins referenciam a mesma instância de array
```

Nesse exemplo, se você modificar o valor em `pins[1]`, a modificação também será visível na leitura de `alias[1]`.

Se quiser fazer uma cópia da instância do array (os dados no heap) à qual uma variável de array se refere, você terá de fazer duas coisas. Primeiramente, você cria uma nova instância de array do mesmo tipo e com o mesmo comprimento do array que está copiando. Segundo, copia os dados, elemento por elemento, do array original para o novo array, como neste exemplo:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
    copy[i] = pins[i];
```

Observe que esse código utiliza a propriedade `Length` do array original para especificar o tamanho do array.

Copiar um array é, na verdade, um requisito comum de muitos aplicativos – tão comum que a classe `System.Array` fornece alguns métodos úteis que você pode empregar para copiar um array, em vez de escrever seu próprio código. Por exemplo, o método `CopyTo`, que copia o conteúdo de um array para outro, partindo de determinado índice inicial. O exemplo a seguir copia todos os elementos do array `pins` para o array `copy`, começando no elemento zero:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

Outra maneira de copiar os valores é utilizar o método estático da classe `System.Array` chamado `Copy`. Como ocorre com `CopyTo`, você deve inicializar o array de destino antes de chamar `Copy`:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```



Nota Certifique-se de especificar um valor válido para o parâmetro `length` do método `Array.Copy`. Se você fornecer um valor negativo, o método lançará uma exceção `ArgumentOutOfRangeException`. Se especificar um valor maior que o número de elementos no array de origem, o método lançará uma exceção `ArgumentException`.

Ainda há outra alternativa: utilizar o método de instância `System.Array` chamado `Clone`. Você pode chamar esse método para criar um array completo e copiá-lo em uma única ação:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```



Nota Os métodos *Clone* foram descritos no Capítulo 8. O método *Clone* da classe *Array* retorna um tipo *object*, em vez de um *Array*, razão pela qual você deve fazer o casting do array para o tipo apropriado ao utilizá-lo. Além disso, os métodos *Clone*, *CopyTo* e *Copy* criam uma cópia rasa de um array (as cópias rasas e profundas também foram descritas no Capítulo 8). Se os elementos do array que estão sendo copiados contêm referências, o método *Clone* simplesmente copia as referências, em vez dos objetos que estão sendo referidos. Depois da cópia, ambos os arrays irão referenciar o mesmo conjunto de objetos. Se precisar criar uma cópia profunda desse array, você deverá utilizar código apropriado em um loop *for*.

Arrays multidimensionais

Os arrays apresentados até agora continham uma única dimensão e podem ser considerados listas simples de valores. É possível criar arrays com mais de uma dimensão. Por exemplo, para criar um array bidimensional, especifique um array que exija dois índices de inteiros. O código a seguir gera um array bidimensional de 24 inteiros, chamado *items*. Se ajudar, você pode considerar um array bidimensional como uma tabela, com a primeira dimensão especificando um número de linhas e a segunda especificando um número de colunas.

```
int[,] items = new int[4, 6];
```

Para acessar um elemento do array, forneça dois valores de índice para especificar a “célula” que armazena o elemento. (Uma célula é a interseção entre uma linha e uma coluna.) O código a seguir mostra alguns exemplos que utilizam o array *items*:

```
items[2, 3] = 99;           // define o elemento em cell(2,3) como 99
items[2, 4] = items[2,3];   // copia o elemento em cell(2, 3) para cell(2, 4)
items[2, 4]++;             // incrementa o valor inteiro em cell(2, 4)
```

Não há limite para o número de dimensões que podem ser especificadas para um array. O próximo exemplo de código cria e utiliza um array chamado *cube*, que contém três dimensões. Observe que é necessário especificar três índices para acessar cada elemento do array.

```
int[, ,] cube = new int[5, 5, 5];
cube[1, 2, 1] = 101;
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

Neste ponto, recomendamos cuidado ao criar arrays com mais de três dimensões. Especificamente, os arrays podem consumir muita memória. O array *cube* contém 125 elementos ($5 * 5 * 5$). Um array quadridimensional, no qual cada dimensão tem um tamanho de 5, contém 625 elementos. Se você começar a criar arrays com três ou mais dimensões, logo poderá esgotar a memória. Portanto, você deve sempre estar preparado para capturar e tratar exceções *OutOfMemoryException*, ao utilizar arrays multidimensionais.

Crie arrays irregulares

No C#, os arrays multidimensionais normais às vezes são chamados de arrays *retangulares*. Cada dimensão tem um formato regular. Por exemplo, no array tabular bidimensional *items* a seguir, cada linha tem uma coluna contendo 40 elementos e no total existem 160 elementos:

```
int[,] items = new int[4, 40];
```

Como mencionado na seção anterior, os arrays multidimensionais podem consumir muita memória. Se o aplicativo utiliza apenas parte dos dados de cada coluna, alocar memória para elementos não utilizados é um desperdício. Nesse cenário, você pode utilizar um array *irregular* (*jagged array*), no qual cada coluna tem um comprimento diferente, como este:

```
int[][] items = new int[4][];  
int[] columnForRow0 = new int[3];  
int[] columnForRow1 = new int[10];  
int[] columnForRow2 = new int[40];  
int[] columnForRow3 = new int[25];  
items[0] = columnForRow0;  
items[1] = columnForRow1;  
items[2] = columnForRow2;  
items[3] = columnForRow3;  
...
```

Nesse exemplo, o aplicativo exige apenas 3 elementos na primeira coluna, 10 elementos na segunda, 40 na terceira e 25 elementos na última coluna. Esse código ilustra um array de arrays — em vez de *items* ser um array bidimensional, ele tem apenas uma dimensão, mas nessa dimensão os próprios elementos são arrays. Além disso, o tamanho total do array *items* é de 78 elementos, em vez de 160; não há espaço alocado para elementos que o aplicativo não vai utilizar.

Vale destacar um pouco da sintaxe nesse exemplo. A declaração a seguir especifica que *items* é um array de arrays de *int*.

```
int[][] items;
```

A instrução seguinte inicializa *items* para armazenar quatro elementos, cada um dos quais sendo um array de comprimento indeterminado:

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

Os arrays *columnForRow0* a *columnForRow3* são todos arrays *int* unidimensionais, inicializados para armazenar o volume de dados exigido para cada coluna. Por fim, cada array de coluna recebe os elementos apropriados do array *items*, como segue:

```
items[0] = columnForRow0;
```

Lembre-se de que os arrays são objetos de referência; portanto, essa instrução simplesmente adiciona uma referência para *columnForRow0* ao primeiro elemento do array *items* – ela não copia dados realmente. Os dados dessa coluna podem ser preenchidos atribuindo-se um valor a um elemento indexado em *columnForRow0* ou fazendo-se referência a ele por meio do array *items*. As instruções a seguir são equivalentes:

```
columnForRow0[1] = 99;  
items[0][1] = 99;
```

Você pode ampliar ainda mais essa ideia, se quiser criar arrays de arrays de arrays, em vez de arrays retangulares tridimensionais e assim por diante.



Nota Se você já escreveu código com a linguagem de programação Java, deve conhecer esse conceito. O Java não tem arrays multidimensionais; em vez disso, é possível criar arrays de arrays exatamente como acabamos de descrever.

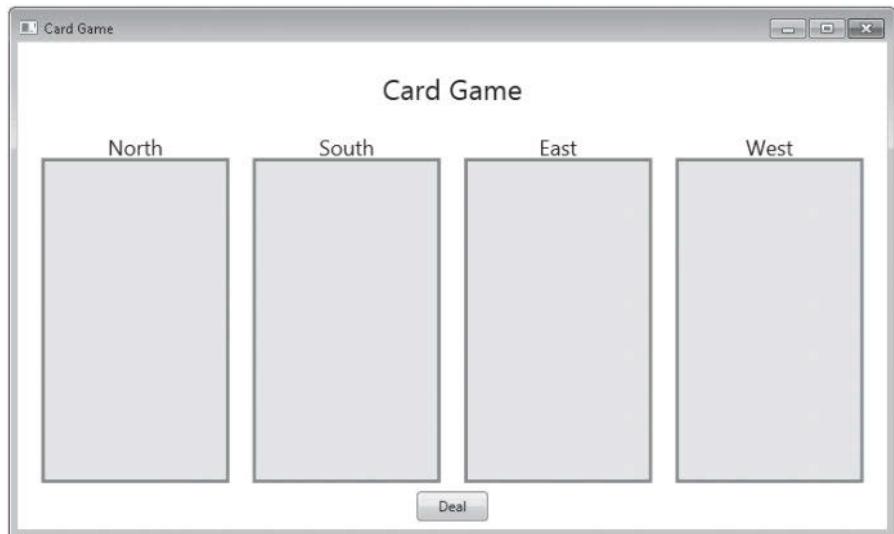
No exercício a seguir, você utilizará arrays para implementar um aplicativo que distribui as cartas como parte de um jogo de baralho. O aplicativo exibe um formulário com quatro mãos de cartas dadas aleatoriamente a partir de um baralho comum (52 cartas). Você concluirá o código que dá as cartas de cada mão.

Use arrays para implementar um jogo de cartas

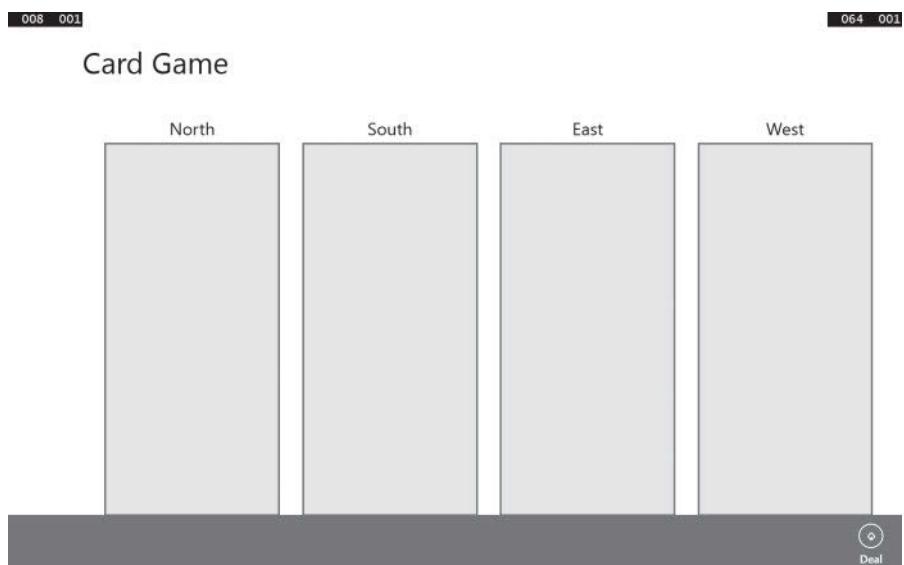
1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto Cards, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 10\Windows X\Cards na sua pasta Documentos.
3. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.

Será exibido um formulário com a legenda Card Game, quatro caixas de texto (intituladas North, South, East e West) e um botão com a legenda Deal (distribuir).

Se você está usando Windows 7, o formulário aparece deste modo:



Se você está usando o Windows 8.1, o botão Deal está na barra de aplicativos, e não no formulário principal. O aplicativo aparece deste modo:





Nota Esse é o mecanismo preferido para posicionar botões de comando em aplicativos Windows Store e, daqui em diante, todos os aplicativos Windows Store apresentados neste livro seguirão esse estilo.

4. Clique em Deal.

Nada acontece. Você ainda não implementou o código que dá as cartas – é o que vai fazer neste exercício.

5. Retorne ao Visual Studio 2013. No menu Debug, clique em Stop Debugging.

6. No Solution Explorer, localize o arquivo Value.cs. Abra esse arquivo na janela Code and Text Editor.

Esse arquivo contém uma enumeração chamada *Value*, que representa os diversos valores que uma carta pode ter, em ordem crescente:

```
enum Value { Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack,
Queen, King, Ace }
```

7. Abra o arquivo Suit.cs na janela Code and Text Editor.

Esse arquivo contém uma enumeração chamada *Suit*, que representa os naipes das cartas contidas em um baralho comum:

```
enum Suit { Clubs, Diamonds, Hearts, Spades }
```

8. Exiba o arquivo PlayingCard.cs na janela Code and Text Editor.

Esse arquivo contém a classe *PlayingCard*. Essa classe modela uma única carta do baralho.

```
class PlayingCard
{
    private readonly Suit suit;
    private readonly Value value;

    public PlayingCard(Suit s, Value v)
    {
        this.suit = s;
        this.value = v;
    }

    public override string ToString()
    {
        string result = string.Format("{0} of {1}", this.value, this.suit);
        return result;
    }

    public Suit CardSuit()
    {
        return this.suit;
    }

    public Value CardValue()
    {
        return this.value;
    }
}
```

Essa classe tem dois campos *readonly* que representam o valor e o naipe da carta. O construtor inicializa esses campos.



Nota Um campo *readonly* é útil para modelar dados que não devem ser alterados após a sua inicialização. Para atribuir um valor a um campo *readonly*, utilize um inicializador quando você o declarar, ou um construtor, mas, a partir de então, você não poderá alterá-lo.

A classe contém dois métodos, chamados *CardValue* e *CardSuit*, que retornam essas informações, e ela sobrescreve o método *ToString* para retornar uma representação de string da carta.



Nota Na realidade, os métodos *CardValue* e *CardSuit* são implementados de modo mais eficiente como propriedades, o que você vai aprender a fazer no Capítulo 15.

9. Abra o arquivo Pack.cs na janela Code and Text Editor.

Esse arquivo contém a classe *Pack*, que modela um baralho. No início da classe *Pack* existem dois campos públicos *const int*, chamados *NumSuits* e *CardsPerSuit*. Esses dois campos especificam o número de naipes contidos em um baralho e o número cartas de cada naipe. A variável privada *CardPack* é um array bidimensional de objetos *PlayingCard*. Você usará a primeira dimensão para especificar o naipe e a segunda para especificar o valor da carta no naipe. A variável *randomCardSelector* é um número aleatório gerado com base na classe *Random*. Você utilizará a variável *randomCardSelector* para ajudar a embaralhar as cartas, antes de serem distribuídas em cada rodada.

```
class Pack
{
    public const int NumSuits = 4;
    public const int CardsPerSuit = 13;
    private PlayingCard[,] cardPack;
    private Random randomCardSelector = new Random();
    ...
}
```

10. Localize o construtor padrão da classe *Pack*. Atualmente, esse construtor está vazio, exceto por um comentário *// TODO:*. Exclua o comentário e adicione a instrução mostrada em negrito a seguir, para instanciar o array *cardPack* com os valores apropriados para cada dimensão:

```
public Pack()
{
    this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
}
```

- 11.** Adicione ao construtor de *Pack* o código a seguir mostrado em negrito. Essas instruções preenchem o array *cardPack* com um baralho ordenado completo.

```
public Pack()
{
    this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
    for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
    {
        for (Value value = Value.Two; value <= Value.Ace; value++)
        {
            this.cardPack[(int)suit, (int)value] = new PlayingCard(suit, value);
        }
    }
}
```

O loop *for* externo itera sobre a lista de valores da enumeração *Suit* e o loop interno itera sobre os valores que cada carta pode ter em cada naipe. O loop interno gera um novo objeto *PlayingCard* do naipe e do valor especificados e o adiciona ao elemento pertinente no array *cardPack*.



Nota Você deve utilizar um dos tipos inteiros como índices em um array. *Suit* e *value* são variáveis do tipo enumerado. Entretanto, as enumerações se baseiam em tipos inteiros, de modo que é seguro convertê-los em *int*, como demonstra o código.

- 12.** Procure o método *DealCardFromPack* na classe *Pack*. O objetivo desse método é escolher uma carta aleatória no baralho, removê-la do baralho para impedir que seja novamente selecionada e, então, passá-la de volta como valor de retorno do método.

A primeira tarefa desse método é escolher um naipe qualquer. Exclua o comentário e a instrução que lança a exceção *NotImplementedException* desse método, e substitua-os pela seguinte instrução mostrada em negrito:

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
}
```

Essa instrução utiliza o método *Next* do objeto gerador de números aleatórios, *randomCardSelector*, para retornar um número aleatório correspondente a um naipe. O parâmetro para o método *Next* especifica o limite máximo exclusivo do intervalo a ser utilizado; o valor selecionado está entre 0 e esse valor menos 1. Observe que o valor retornado é um *int*, de modo que é necessário convertê-lo para depois atribuí-lo a uma variável *Suit*.

Há sempre a possibilidade de que não existam mais cartas do naipe selecionado. Você precisa resolver essa situação e escolher outro naipe, se necessário.

- 13.** Depois do código que seleciona um naipe aleatoriamente, adicione o loop *while* a seguir (mostrado em negrito).

Esse loop chama o método *IsSuitEmpty* para detectar se no baralho ainda existem cartas do naipe especificado (você vai implementar a lógica desse método em breve). Se não existirem, ele selecionará outro naipe aleatoriamente (ele pode até escolher o mesmo naipe outra vez) e verificará outra vez. O loop repetirá esse processo até encontrar um naipe com pelo menos uma carta existente.

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
    while (this.IsSuitEmpty(suit))
    {
        suit = (Suit)randomCardSelector.Next(NumSuits);
    }
}
```

14. Você acabou de selecionar um naipe aleatoriamente, com pelo menos uma carta ainda existente. A próxima tarefa é escolher uma carta aleatória desse naipe. Você pode utilizar o gerador de números aleatórios para selecionar um valor de carta, mas, como antes, não é possível assegurar que a carta com o valor escondido ainda não tenha sido distribuída. Entretanto, você pode utilizar a mesma solução anterior: chamar o método *IsCardAlreadyDealt* (que você examinará e completará posteriormente) para detectar se a carta já foi distribuída e, nesse caso, escolher outra carta aleatória e tentar de novo, repetindo o processo até que uma carta seja encontrada. Para isso, adicione as seguintes instruções mostradas em negrito ao método *DealCardFromPack*, depois do código existente:

```
public PlayingCard DealCardFromPack()
{
    ...
    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
    while (this.IsCardAlreadyDealt(suit, value))
    {
        value = (Value)randomCardSelector.Next(CardsPerSuit);
    }
}
```

15. Você acabou de selecionar uma carta aleatória que ainda não foi distribuída. Adicione o código a seguir no final do método *DealCardFromDeck*, para retornar essa carta e definir com *null* o elemento correspondente no array *cardPack*:

```
public PlayingCard DealCardFromPack()
{
    ...
    PlayingCard card = this.cardPack[(int)suit, (int)value];
    this.cardPack[(int)suit, (int)value] = null;
    return card;
}
```

16. Localize o método *IsSuitEmpty*. Lembre-se de que o objetivo desse método é aceitar um parâmetro *Suit* e retornar um valor booleano indicando se ainda existem outras cartas desse naipe no baralho. Exclua o comentário e a instrução que lança a exceção *NotImplementedException* a partir desse método, e adicione o seguinte código mostrado em negrito:

```

private bool IsSuitEmpty(Suit suit)
{
    bool result = true;
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        if (!IsCardAlreadyDealt(suit, value))
        {
            result = false;
            break;
        }
    }

    return result;
}

```

Esse código itera sobre os possíveis valores das cartas e determina se ainda existe no array *cardPack* alguma carta com o naipe e o valor especificados, por meio do método *IsCardAlreadyDealt*, o qual você completará no próximo passo. Se o loop encontrar uma carta, o valor da variável *result* será definido como *false* e a instrução *break* encerrará o loop. Se o loop terminar sem encontrar uma carta, a variável *result* permanecerá definida com seu valor inicial, *true*. O valor da variável *result* é passado de volta como o valor de retorno do método.

- 17.** Procure o método *IsCardAlreadyDealt*. O objetivo desse método é determinar se a carta com o naipe e o valor especificados já foi distribuída e retirada do baralho. Você verá mais adiante que, ao distribuir uma carta, o método *DealFromPack* a retira do array *cardPack* e define o elemento correspondente como *null*. Substitua o comentário e a instrução que lança a exceção *NotImplementedException* nesse método pelo código apresentado em negrito:

```

private bool IsCardAlreadyDealt(Suit suit, Value value)
{
    return (this.cardPack[(int)suit, (int)value] == null);
}

```

Essa instrução retorna *true* se o elemento do array *cardPack* correspondente ao naipe e ao valor for *null* e, caso contrário, retorna *false*.

- 18.** O próximo passo é adicionar a carta selecionada a uma das mãos. Abra o arquivo *Hand.cs* e o exiba na janela Code and Text Editor. Esse arquivo contém a classe *Hand*, que implementa a mão de cartas (ou seja, todas as cartas distribuídas para um único jogador).

Esse arquivo contém um campo *public const int*, chamado *HandSize*, definido com o tamanho de uma das mãos de cartas (13). Também contém um array de objetos *PlayingCard*, inicializado por meio da constante *HandSize*. O campo *playingCardCount* será utilizado por seu código para rastrear a quantidade de cartas contidas atualmente na mão, à medida que é preenchida.

```

class Hand
{
    public const int HandSize = 13;
    private PlayingCard[] cards = new PlayingCard[HandSize];
    private int playingCardCount = 0;
    ...
}

```

O método *ToString* gera uma representação de string das cartas da mão. Ele utiliza o loop *foreach* para iterar sobre os itens do array de cartas e chama o método *ToString* em cada objeto *PlayingCard* encontrado. Essas strings são concatenadas com um caractere de nova linha (o caractere *\n*) para fins de formatação.

```
public override string ToString()
{
    string result = "";
    foreach (PlayingCard card in this.cards)
    {
        result += card.ToString() + "\n";
    }

    return result;
}
```

- 19.** Localize o método *AddCardToHand* na classe *Hand*. O objetivo desse método é adicionar à mão a carta especificada como parâmetro. Adicione a esse método as instruções mostradas em negrito a seguir:

```
public void AddCardToHand(PlayingCard cardDealt)
{
    if (this.playingCardCount >= HandSize)
    {
        throw new ArgumentException("Too many cards");
    }
    this.cards[this.playingCardCount] = cardDealt;
    this.playingCardCount++;
}
```

Esse código verifica primeiramente se a mão ainda não está cheia. Se a mão estiver cheia, ele lança uma exceção *ArgumentException* (isso nunca deve ocorrer, mas é uma boa prática de segurança). Caso contrário, a carta é adicionada ao array *cards* no índice especificado pela variável *playingCardCount*, e essa variável é, então, incrementada.

- 20.** No Solution Explorer, expanda o nó *MainWindow.xaml* e abra o arquivo *MainWindow.xaml.cs* na janela Code and Text Editor.

Esse é o código para a janela Card Game. Localize o método *dealClick*. Esse método é executado quando o usuário clica no botão Deal. Atualmente, ele contém um bloco *try* vazio e uma rotina de tratamento de exceções que exibe uma mensagem se ocorrer uma exceção.

- 21.** Adicione ao bloco *try* à instrução mostrada em negrito a seguir:

```
private void dealClick(object sender, RoutedEventArgs e)
{
    try
    {
        pack = new Pack();
    }
    catch (Exception ex)
    {
        ...
    }
}
```

Essa instrução simplesmente cria um novo baralho. Antes, vimos que essa classe contém um array bidimensional que armazena as cartas do baralho, e o construtor preenche esse array com os detalhes de cada carta. Agora você precisa criar quatro mãos de cartas a partir desse baralho.

- 22.** Adicione ao bloco *try* as instruções mostradas em negrito a seguir:

```
try
{
    pack = new Pack();

    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        hands[handNum] = new Hand();
    }
}
catch (Exception ex)
{
    ...
}
```

Esse loop *for* gera quatro mãos do baralho e as armazena em um array chamado *hands*. Inicialmente, cada mão está vazia; portanto, você precisa distribuir as cartas do baralho a cada uma das mãos.

- 23.** Adicione ao loop *for* o seguinte código mostrado em negrito:

```
try
{
    ...
    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        hands[handNum] = new Hand();
        for (int numCards = 0; numCards < Hand.HandSize; numCards++)
        {
            PlayingCard cardDealt = pack.DealCardFromPack();
            hands[handNum].AddCardToHand(cardDealt);
        }
    }
}
catch (Exception ex)
{
    ...
}
```

O loop *for* interno preenche cada mão por meio do método *DealCardFromPack*, para recuperar uma carta aleatória do baralho, e, por meio do método *AddCardToHand*, para adicionar essa carta à mão.

- 24.** Adicione, após o loop *for* externo, o seguinte código mostrado em negrito:

```
try
{
    ...
    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        ...
    }

    north.Text = hands[0].ToString();
    south.Text = hands[1].ToString();
}
```

```

        east.Text = hands[2].ToString();
        west.Text = hands[3].ToString();
    }
catch (Exception ex)
{
    ...
}

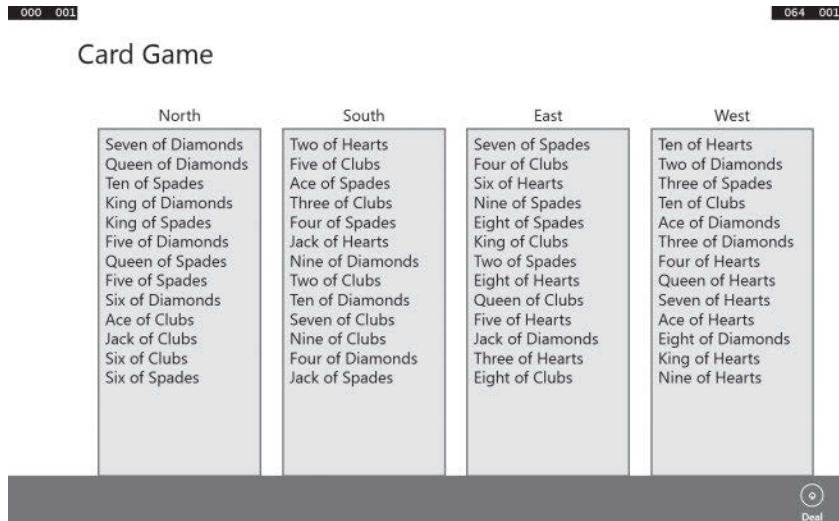
```

Quando todas as cartas estiverem distribuídas, esse código exibirá cada mão nas caixas de texto do formulário. Essas caixas de texto são chamadas de *north*, *south*, *east* e *west*. O código utiliza o método *ToString* de cada mão para formatar a saída.

Se ocorrer uma exceção nesse ponto, a rotina de tratamento *catch* exibirá uma caixa de mensagem com a mensagem de erro da exceção.

25. No menu Debug, clique em Start Debugging. Quando a janela Card Game for exibida, clique em Deal.

As cartas do baralho devem ser distribuídas aleatoriamente a cada mão, e as cartas de cada mão devem ser exibidas no formulário, como mostra a imagem a seguir:



26. Clique em Deal novamente. Verifique que um novo conjunto de mãos é distribuído e que as cartas de cada mão mudam.
27. Retorne ao Visual Studio e interrompa a depuração.

Resumo

Neste capítulo, você aprendeu a criar e utilizar arrays para manipular conjuntos de dados. Viu como declarar e inicializar arrays, acessar os dados armazenados em arrays, passar arrays como parâmetros para métodos e retornar arrays a partir de métodos. Aprendeu também a criar arrays multidimensionais e a utilizar arrays de arrays.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 11.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar uma variável de array	Escreva o nome do tipo de elemento, seguido por colchetes, seguidos pelo nome da variável, seguido por um ponto e vírgula. Por exemplo: bool[] flags;
Criar uma instância de um array	Escreva a palavra-chave new, seguida pelo nome do tipo de elemento, seguido pelo tamanho do array entre colchetes. Por exemplo: bool[] flags = new bool[10];
Inicializar os elementos de um array com valores específicos	Para um array, escreva os valores específicos em uma lista separada por vírgulas incluindo-os entre chaves. Por exemplo: bool[] flags = { true, false, true, false };
Descobrir o número de elementos de um array	Utilize a propriedade Length. Por exemplo: int [] flags = ...; ... int noOfElements = flags.Length;
Acessar um único elemento de um array	Escreva o nome da variável do array, seguida pelo índice inteiro do elemento entre colchetes. Lembre-se de que a indexação de um array começa em 0, não em 1. Por exemplo: bool initialElement = flags[0];
Iterar pelos elementos de um array	Utilize uma instrução for ou uma instrução foreach. Por exemplo: bool[] flags = { true, false, true, false }; for (int i = 0; i < flags.Length; i++) { Console.WriteLine(flags[i]); } foreach (bool flag in flags) { Console.WriteLine(flag); }

Para	Faça isto
Declarar uma variável de array multidimensional	<p>Escreva o nome do tipo de elemento, seguido por um conjunto de colchetes com uma vírgula separadora indicando o número de dimensões, seguidos pelo nome da variável, seguido por um ponto e vírgula. Por exemplo, utilize o seguinte para criar um array bidimensional chamado table e inicializá-lo para armazenar 4 linhas de 6 colunas:</p> <pre>int[,] table; table = new int[4,6];</pre>
Declarar uma variável de array irregular (<i>jagged array</i>)	<p>Declare a variável como um array de arrays filhos. Você pode inicializar cada array filho com um comprimento diferente. Por exemplo, utilize o seguinte para criar um array irregular chamado items e inicializar cada array filho:</p> <pre>int[][] items; items = new int[4][]; items[0] = new int[3]; items[1] = new int[10]; items[2] = new int[40]; items[3] = new int[25];</pre>

CAPÍTULO 11

Arrays de parâmetros

Neste capítulo, você vai aprender a:

- Escrever um método que pode aceitar qualquer número de argumentos utilizando a palavra-chave *params*.
- Escrever um método que pode aceitar qualquer número de argumentos de qualquer tipo utilizando a palavra-chave *params* em combinação com o tipo *object*.
- Explicar as diferenças entre os métodos que aceitam arrays de parâmetros e os que admitem parâmetros opcionais.

Caso você queira escrever métodos que podem receber como parâmetros qualquer número de argumentos, possivelmente de tipos diferentes, os arrays de parâmetro serão úteis. Se você já domina os conceitos de orientação a objetos, é possível que tenha ficado frustrado com a frase anterior. Afinal de contas, para a solução desse problema, a estratégia orientada a objetos indica que é preciso a definição de métodos sobre-carregados. Sobrecarregar, no entanto, não é sempre o mais adequado, especialmente se você precisa criar um método que possa aceitar um número realmente diverso de parâmetros, cada um dos quais podendo variar no tipo, quando o método é chamado. Neste capítulo, você vai aprender sobre a utilização de arrays de parâmetros para lidar com situações desse tipo.

Sobrecarga – uma recapitulação

Sobrecarregar é o termo técnico utilizado para declarar dois ou mais métodos com o mesmo nome no mesmo escopo. A capacidade de sobrecarregar um método é muito útil para os casos nos quais você quer realizar a mesma ação sobre argumentos de tipos diferentes. O exemplo clássico de sobrecarga no Microsoft Visual C# é o método *Console.WriteLine*. Esse método é sobre-carregado várias vezes para permitir a passagem de qualquer argumento de tipo primitivo. O exemplo de código a seguir ilustra algumas das maneiras pelas quais o método *WriteLine* é definido na classe *Console*:

```
class Console
{
    public static void WriteLine(Int32 value)
    public static void WriteLine(Double value)
    public static void WriteLine(Decimal value)
    public static void WriteLine(Boolean value)
    public static void WriteLine(String value)
    ...
}
```



Nota A documentação do método `WriteLine` utiliza para seus parâmetros os tipos-estrutura definidos no namespace `System`, em vez dos alias do C# para esses tipos. Por exemplo, a sobrecarga que imprime o valor de um `int` recebe, na verdade, um `Int32` como parâmetro. Consulte o Capítulo 9, “Como criar tipos-valor com enumerações e estruturas”, para ver uma lista dos tipos-estrutura e seus mapeamentos em alias do C# para esses tipos.

Embora muito útil, a sobrecarga não cobre todos os casos. Em particular, a sobrecarga não trata facilmente uma situação em que o tipo dos parâmetros não varia, mas o número de parâmetros, sim. Mas, e se você quisesse escrever muitos valores no console, por exemplo? Teria de fornecer versões de `Console.WriteLine` que pudessem receber dois parâmetros de várias combinações, outras versões que pudessem receber três parâmetros e assim por diante? Isso logo se tornaria tedioso. E a duplicação maciça de todos esses métodos sobrecarregados não o preocuparia? Deveria. Felizmente, há uma maneira de escrever um método que recebe um número variável de argumentos: você pode utilizar um array de parâmetros (um parâmetro declarado com a palavra-chave `params`).

Para entender como os arrays `params` resolvem esse problema, é útil compreender primeiro as utilizações e deficiências dos arrays comuns.

Argumentos de arrays

Suponha que você queira escrever um método para determinar o valor mínimo em um conjunto de valores passados como parâmetros. Uma maneira seria utilizar um array. Por exemplo, para descobrir o menor valor em diversos valores `int`, você poderia escrever um método chamado `Min` com um único parâmetro representando um array de valores `int`:

```
class Util
{
    public static int Min(int[] paramList)
    {
        // Verifica se o chamador forneceu pelo menos um parâmetro.
        // Se não, lança uma exceção ArgumentException – não é possível
        // encontrar o menor valor em uma lista vazia.
        if (paramList == null || paramList.Length == 0)
        {
            throw new ArgumentException("Util.Min: not enough arguments");
        }

        // Define o valor mínimo atual encontrado na lista de parâmetros como o primeiro item
        int currentMin = paramList[0];

        // Itera pela lista de parâmetros, verificando se algum deles
        // é menor que o valor armazenado em currentMin
        foreach (int i in paramList)
        {
            // Se o loop encontrar um item menor que o valor armazenado em
            // currentMin, configura currentMin com esse valor
            if (i < currentMin)
            {
```

```

        currentMin = i;
    }
}

// No final do loop, currentMin armazena o valor do menor
// item da lista de parâmetros; portanto, retorna esse valor.
return currentMin;
}
}

```



Nota A classe *ArgumentException* é especificamente projetada para ser lançada por um método caso o argumento fornecido não atenda aos requisitos do método.

Para utilizar o método *Min* a fim de descobrir o mínimo de duas variáveis *int* chamadas *first* e *second*, escreva o seguinte:

```

int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array);

```

E para utilizar o método *Min* a fim de descobrir o mínimo de três variáveis *int* (chamadas *first*, *second* e *third*), escreva o seguinte:

```

int[] array = new int[3];
array[0] = first;
array[1] = second;
array[2] = third;
int min = Util.Min(array);

```

Você pode ver que essa solução evita a necessidade de um grande número de sobrecargas, mas ela tem um preço: é preciso escrever um código adicional para preencher o array que você passa. Evidentemente, se preferir, você pode utilizar um array anônimo, como este:

```
int min = Util.Min(new int[] {first, second, third});
```

Contudo, a questão é que ainda é necessário criar e preencher um array, e a sintaxe pode ficar um pouco confusa. A solução é fazer o compilador escrever um pouco desse código para você, utilizando um array *params* como parâmetro para o método *Min*.

Declare um array *params*

Com um array *params* é possível passar um número variável de argumentos para um método. Você indica um array *params* utilizando a palavra-chave *params* como modificador de parâmetro do array, ao definir os parâmetros do método. Por exemplo, aqui está o método *Min* novamente, desta vez com seu parâmetro de array declarado como um array *params*:

```

class Util
{
    public static int Min(params int[] paramList)

```

```
{
    // código exatamente como antes
}
}
```

O efeito da palavra-chave *params* no método *Min* é que ela torna possível chamar-lo utilizando qualquer número de argumentos inteiros, sem se preocupar com a criação de um array. Por exemplo, para descobrir o mínimo de dois valores inteiros, escreva simplesmente:

```
int min = Util.Min(first, second);
```

O compilador traduz essa chamada em um código semelhante a este:

```
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array);
```

Para descobrir o mínimo de três valores inteiros, você poderia escrever o código mostrado aqui, que também é convertido pelo compilador no código correspondente que utiliza um array:

```
int min = Util.Min(first, second, third);
```

Ambas as chamadas a *Min* (uma chamada com dois argumentos e outra com três argumentos) determinam o mesmo método *Min* com a palavra-chave *params*. E como você provavelmente pode imaginar, é possível chamar esse método *Min* com qualquer número de argumentos *int*. O compilador apenas conta o número de argumentos *int*, cria um array *int* desse tamanho, preenche o array com os argumentos e então chama o método passando o único parâmetro de array.



Nota Se você é programador de C ou C++, talvez reconheça *params* como um equivalente seguro das macros *varargs* do arquivo de cabeçalho *stdarg.h*. O Java também tem um recurso *varargs* que funciona de maneira semelhante à palavra-chave *params* no C#.

Há vários pontos sobre os arrays *params* que valem a pena mencionar:

- Você não pode utilizar a palavra-chave *params* com arrays multidimensionais. O código no exemplo a seguir não compilará:

```
// erro de tempo de compilação
public static int Min(params int[,] table)
{ ... }
```

- Você não pode sobrecarregar um método baseado apenas na palavra-chave *params*. A palavra-chave *params* não faz parte da assinatura do método, como mostrado neste exemplo:

```
// erro de tempo de compilação: declaração duplicada
public static int Min(int[] paramList)
{ ... }
public static int Min(params int[] paramList)
{ ... }
```

- Você não pode especificar o modificador *ref* ou *out* com arrays *params*, como mostrado neste exemplo:

```
// erros de tempo de compilação
public static int Min(ref params int[] paramList)
...
public static int Min(out params int[] paramList)
...
```

- Um array *params* deve ser o último parâmetro. (Isso significa que você só pode ter um array *params* por método.) Considere este exemplo:

```
// erro de tempo de compilação
public static int Min(params int[] paramList, int i)
...
```

- Um método não *params* sempre tem prioridade sobre um método *params*. Isso significa que, se quiser, você ainda pode criar uma versão sobrecarregada de um método para os casos comuns, como no exemplo a seguir:

```
public static int Min(int leftHandSide, int rightHandSide)
...
public static int Min(params int[] paramList)
...
```

A primeira versão do método *Min* é empregada quando chamada por meio da utilização de dois argumentos *int*. A segunda versão é usada se algum outro número de argumentos *int* for fornecido. Isso inclui o caso no qual o método é chamado sem argumentos. A adição do método de array sem *params* pode ser uma técnica de otimização útil, porque o compilador não precisará criar e preencher muitos arrays.

Utilize *params object[]*

Um array de parâmetros de tipo *int* é muito útil. Com ele é possível passar qualquer número de argumentos *int* em uma chamada de método. Mas, e se não variar só o número de argumentos, mas também o tipo de argumento? O C# tem uma maneira de resolver esse problema também. A técnica é baseada no fato de que *object* é a raiz de todas as classes e que o compilador pode gerar código que converte tipos-valor (coisas que não são classes) em objetos utilizando boxing, conforme descrito no Capítulo 8, "Valores e referências". Você pode utilizar um array de parâmetros do tipo *object* para declarar um método que aceita qualquer número de argumentos *object*, permitindo que os argumentos passados sejam de qualquer tipo. Veja este exemplo:

```
class Black
{
    public static void Hole(params object [] paramList)
    ...
}
```

Dei a esse método o nome *Black.Hole* ("buraco negro") porque os argumentos não podem escapar dele:

- Você pode passar o método sem argumento, caso em que o compilador passará um array de objetos cujo comprimento é 0:

```
Black.Hole();
// convertido em Black.Hole(new object[0]);
```

- Você pode chamar o método *Black.Hole* passando *null* como o argumento. Um array é um tipo-referência; portanto, você pode iniciá-lo com *null*:

```
Black.Hole(null);
```

- Você pode passar o método *Black.Hole* para um array real. Em outras palavras, pode criar manualmente o array que costuma ser gerado pelo compilador:

```
object[] array = new object[2];
array[0] = "forty two";
array[1] = 42;
Black.Hole(array);
```

- Você pode passar para o método *Black.Hole* argumentos de tipos diferentes, e esses argumentos serão automaticamente encapsulados dentro de um array *object*:

```
Black.Hole("forty two", 42);
//convertido em Black.Hole(new object[]{"forty two", 42});
```

O método *Console.WriteLine*

A classe *Console* contém muitas sobrecargas para o método *WriteLine*. Uma delas é semelhante a esta:

```
public static void WriteLine(string format, params Object[] arg);
```

Essa sobrecarga permite que o método *WriteLine* suporte um argumento de string de formato que contém espaços reservados, cada um dos quais pode ser substituído em tempo de execução por uma variável de qualquer tipo. Aqui está um exemplo de chamada a esse método (as variáveis *fname* e *lname* são *strings*, *mi* é um *char* e *age* é um *int*):

```
Console.WriteLine("Forename:{0}, Middle Initial:{1}, Last name:{2}, Age:{3}",
    fname, mi, lname, age);
```

O compilador resolve essa chamada no seguinte:

```
Console.WriteLine("Forename:{0}, Middle Initial:{1}, Last name:{2}, Age:{3}",
    new object[4]{fname, mi, lname, age});
```

Utilize um array *params*

No exercício a seguir, você vai implementar e testar um método *static* chamado *Sum*. O objetivo desse método é calcular a soma de um número variável de argumentos *int* passados para ele, retornando o resultado como um *int*. Você fará isso escrevendo *Sum* para aceitar um parâmetro *params int[]*. Você implementará duas verificações no parâmetro *params* a fim de assegurar que o método *Sum* seja completamente robusto. Então, chamará o método *Sum* com diversos argumentos diferentes para testá-lo.

Escreva um método de array *params*

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto ParamsArray, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 11\Windows X\ParamsArray na sua pasta Documentos.

O projeto ParamsArray contém a classe *Program* no arquivo Programs.cs, incluindo a estrutura do método *doWork* que você viu em capítulos anteriores. Você vai implementar o método *Sum* como um método estático de outra classe, chamada *Util* (abreviação de “utilitário”), a qual vai adicionar ao projeto.

3. No Solution Explorer, clique com o botão direito do mouse no projeto ParamsArray na solução ParamsArray, aponte para Add e então clique em Class.
4. Na caixa de diálogo Add New Item – ParamsArray, no painel central, clique no template Class. Na caixa Name, digite **Util.cs** e clique em Add.

O arquivo Util.cs é criado e adicionado ao projeto. Ele contém uma classe vazia chamada *Util* no namespace *ParamsArray*.

5. Adicione à classe *Util* um método estático público chamado *Sum*. Esse método deve retornar um *int* e aceitar um array *params* de valores *int* chamado *paramList*. Ele deve se parecer com isto:

```
public static int Sum(params int[] paramList)
{
}
```

O primeiro passo na implementação do método *Sum* é verificar o parâmetro *paramList*. Além de conter um conjunto válido de números inteiros, ele pode também ser *null* ou um array de comprimento zero. Em ambos os casos, é difícil calcular a soma; portanto, a melhor opção é lançar uma exceção *ArgumentException*. (Você poderá argumentar que a soma dos inteiros em um array de comprimento zero é 0, mas, neste exemplo, tratamos essa situação como uma exceção.)

6. Adicione a *Sum* o código a seguir mostrado em negrito. Esse código lança uma exceção *ArgumentException* se *paramList* é *null*.

O método *Sum* agora deve se parecer com isto:

```
public static int Sum(params int[] paramList)
{
    if (paramList == null)
    {
        throw new ArgumentException("Util.Sum: null parameter list");
    }
}
```

7. Adicione código ao método *Sum* para lançar uma exceção *ArgumentException* se o comprimento do array de lista de parâmetros for 0, como mostrado em negrito aqui:

```
public static int Sum(params int[] paramList)
{
```

```

    if (paramList == null)
    {
        throw new ArgumentException("Util.Sum: null parameter list");
    }

    if (paramList.Length == 0)
    {
        throw new ArgumentException("Util.Sum: empty parameter list");
    }
}

```

Se o array passar nesses dois testes, a próxima etapa será adicionar todos os elementos do array. Você pode utilizar uma instrução *foreach* para adicionar todos os elementos. Você precisará de uma variável local para armazenar o total.

8. Declare uma variável do tipo inteiro chamada *sumTotal* e a inicialize como 0, imediatamente após o código do passo anterior.

```

public static int Sum(params int[] paramList)
{
    ...
    if (paramList.Length == 0)
    {
        throw new ArgumentException("Util.Sum: empty parameter list");
    }

    int sumTotal = 0;
}

```

9. Adicione uma instrução *foreach* ao método *Sum* para iterar pelo array *paramList*. O corpo desse loop *foreach* deve adicionar cada elemento do array em *sumTotal*. No final do método, retorne o valor de *sumTotal* utilizando uma instrução *return*, como mostrado em negrito a seguir:

```

public static int Sum(params int[] paramList)
{
    ...
    int sumTotal = 0;
    foreach (int i in paramList)
    {
        sumTotal += i;
    }
    return sumTotal;
}

```

10. No menu Build, clique em Build Solution e confirme que sua solução compila sem erros.

Teste o método *Util.Sum*

1. Exiba o arquivo Program.cs na janela Code and Text Editor.

2. Na janela Code and Text Editor, exclua o comentário `// TODO:` e adicione a seguinte instrução ao método `doWork`:

```
Console.WriteLine(Util.Sum(null));
```

3. No menu Debug, clique em Start Without Debugging.

O programa compila e executa, escrevendo a seguinte mensagem no console:

```
Exception: Util.Sum: null parameter list
```

Isso confirma que a primeira verificação no método funciona.

4. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2013.

5. Na janela Code and Text Editor, altere a chamada a `Console.WriteLine` em `doWork`, como mostrado aqui:

```
Console.WriteLine(Util.Sum());
```

Dessa vez, o método está sendo chamado sem argumento. O compilador converterá a lista de argumentos vazia em um array vazio.

6. No menu Debug, clique em Start Without Debugging.

O programa compila e executa, escrevendo a seguinte mensagem no console:

```
Exception: Util.Sum: empty parameter list
```

Isso confirma que a segunda verificação no método funciona.

7. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2013.

8. Altere a chamada a `Console.WriteLine` em `doWork` como a seguir:

```
Console.WriteLine(Util.Sum(10, 9, 8, 7, 6, 5, 4, 3, 2, 1));
```

9. No menu Debug, clique em Start Without Debugging.

Verifique que o programa compila, executa e escreve o valor 55 no console.

10. Pressione Enter para fechar o aplicativo e retornar ao Visual Studio 2013.

Compare arrays de parâmetros e parâmetros opcionais

O Capítulo 3, “Como escrever métodos e aplicar escopo”, ilustrou como é possível definir métodos que aceitam parâmetros opcionais. Em princípio, parece existir um nível de sobreposição entre os métodos que usam arrays de parâmetros e aqueles que aceitam parâmetros opcionais. Entretanto, há diferenças básicas entre eles:

- Um método que aceita parâmetros opcionais também tem uma lista de parâmetros fixos, e você não pode passar uma lista arbitrária de argumentos. O compilador gera um código que insere os valores padrão na pilha para os argumentos ausentes, antes da execução do método, e o método não reconhece quais dos argumentos são fornecidos pelo chamador, nem os padrões gerados pelo compilador.
- Um método que utiliza um array de parâmetros de fato possui uma lista totalmente arbitrária de parâmetros, e eles não têm um valor padrão. Além disso, o método pode determinar com exatidão quantos argumentos o chamador forneceu.

Em geral, os arrays de parâmetros são utilizados nos métodos que aceitam qualquer número de parâmetros (inclusive nenhum), enquanto os parâmetros opcionais são utilizados somente quando não é conveniente instruir um chamador a fornecer um argumento para cada parâmetro.

Existe ainda uma última questão a considerar. Se você definir um método que aceita uma lista de parâmetros e fornecer uma sobrecarga que aceita parâmetros opcionais, nem sempre se torna evidente qual versão do método será chamada, se a lista de argumentos na instrução de chamada corresponder às assinaturas dos dois métodos. Você examinará essa situação no último exercício deste capítulo.

Compare um array *params* e parâmetros opcionais

1. Retorne à solução ParamsArray no Visual Studio 2013 e exiba o arquivo Util.cs na janela Code and Text Editor.
2. Adicione a instrução a seguir *Console.WriteLine*, mostrada em negrito, ao início do método *Sum*, na classe *Util*:

```
public static int Sum(params int[] paramList)
{
    Console.WriteLine("Using parameter list");
    ...
}
```

3. Adicione outra implementação do método *Sum* à classe *Util*. Essa versão deve aceitar quatro parâmetros *int* opcionais, cada um com um valor padrão 0. No corpo do método, apresente na saída a mensagem “Using optional parameters” e depois calcule e retorne a soma dos quatro parâmetros. O método completo deve se parecer com este código em negrito:

```
class Util
{
    ...
    public static int Sum(int param1 = 0, int param2 = 0, int param3 = 0, int
param4 = 0)
    {
        Console.WriteLine("Using optional parameters");
        int sumTotal = param1 + param2 + param3 + param4;
        return sumTotal;
    }
}
```

4. Exiba o arquivo Program.cs na janela Code and Text Editor.

5. No método *doWork*, transforme em comentário o código existente e adicione a seguinte instrução:

```
Console.WriteLine(Util.Sum(2, 4, 6, 8));
```

Essa instrução chama o método *Sum* e passa quatro parâmetros *int*. Essa chamada combina com ambas as sobrecargas do método *Sum*.

6. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

Quando o aplicativo for executado, ele exibirá as seguintes mensagens:

```
Using optional parameters  
20
```

Nesse caso, o compilador gerou um código que chamou o método que aceita quatro parâmetros opcionais. Essa é a versão do método que mais corresponde à chamada do método.

7. Pressione Enter e retorne ao Visual Studio.

8. No método *doWork*, altere a instrução que chama o método *Sum* e remova o último argumento (8), como mostrado a seguir:

```
Console.WriteLine(Util.Sum(2, 4, 6));
```

9. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

Quando o aplicativo for executado, ele exibirá estas mensagens:

```
Using optional parameters  
12
```

O compilador ainda gerou um código que chamou o método que aceita parâmetros opcionais, embora a assinatura do método não correspondesse exatamente à chamada. Diante da escolha entre utilizar um método que aceita parâmetros opcionais e um método que aceita uma lista de parâmetros, o compilador usará o método que aceita parâmetros opcionais.

10. Pressione Enter e retorne ao Visual Studio.

11. No método *doWork*, altere novamente a instrução que chama o método *Sum* e adicione mais dois argumentos:

```
Console.WriteLine(Util.Sum(2, 4, 6, 8, 10));
```

12. No menu Debug, clique em Start Without Debugging para compilar e executar o aplicativo.

Quando o aplicativo for executado, ele exibirá estas mensagens:

```
Using parameter list  
30
```

Dessa vez, há mais argumentos do que o método que aceita parâmetros opcionais especifica, de modo que o compilador gerou um código que chama o método que aceita um array de parâmetros.

13. Pressione Enter e retorne ao Visual Studio.

Resumo

Neste capítulo, você aprendeu a utilizar um array *params* para definir um método que pode receber quantidades variáveis de argumentos. Viu também como utilizar um array *params* de tipos *object* para criar um método que aceita qualquer número de argumentos de qualquer tipo. Além disso, percebeu como o compilador resolve chamadas de método quando pode escolher entre chamar um método que aceita um array de parâmetros e chamar um método que aceita parâmetros opcionais.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 12, “Herança”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Escrever um método que aceita qualquer número de argumentos de determinado tipo	<p>Escreva um método cujo parâmetro seja um array <i>params</i> do tipo dado. Por exemplo, um método que aceita qualquer número de argumentos <i>bool</i> é declarado desta maneira:</p> <pre>someType Method(params bool[] flags) { ... }</pre>
Escrever um método que aceita qualquer número de argumentos de qualquer tipo	<p>Escreva um método cujo parâmetro seja um array <i>params</i> de elementos do tipo <i>object</i>. Por exemplo:</p> <pre>someType Method(params object[] paramList) { ... }</pre>

CAPÍTULO 12

Herança

Neste capítulo, você vai aprender a:

- Criar uma classe derivada que herda recursos de uma classe base.
- Controlar a ocultação e sobrecarga de métodos utilizando as palavras-chave *new*, *virtual* e *override*.
- Limitar a acessibilidade dentro de uma hierarquia de heranças utilizando a palavra-chave *protected*.
- Definir métodos de extensão como um mecanismo alternativo ao uso de herança.

No contexto da programação orientada a objetos, herança é um conceito-chave. Ela pode ser uma ferramenta para evitar a repetição quando se define classes diferentes com muitas características em comum e que estão claramente relacionadas entre si. É possível que sejam classes diferentes do mesmo tipo, cada uma com sua própria característica – por exemplo, *gerentes*, *operários* e *todos os empregados* de uma fábrica. Caso você precise escrever um aplicativo para a simulação da fábrica, como especificar que gerentes e operários têm várias características comuns, mas, ao mesmo tempo, são diferentes? Por exemplo, todos têm um número de referência de funcionário, mas as responsabilidades e as tarefas executadas pelos gerentes são diferentes das que os operários têm.

É neste momento que a herança mostra sua utilidade.

O que é herança?

Se você perguntar a vários programadores experientes o significado do termo *herança*, normalmente receberá respostas diferentes e conflitantes. Parte da confusão resulta do fato de que a própria palavra herança tem vários significados com diferenças sutis entre eles. Se alguém deixa algo para você em um testamento, dizemos que você herdou. Da mesma maneira, você herda metade dos seus genes da sua mãe e a outra metade do seu pai. Esses dois usos da palavra têm pouco a ver com a herança em programação.

Herança em programação é, essencialmente, classificação – é uma relação entre classes. Por exemplo, quando estava no colégio, é provável que você tenha aprendido sobre mamíferos e que cavalos e baleias são exemplos de mamíferos. Cada um tem todos os atributos que um mamífero tem (respira ar, amamenta seus filhotes, tem sangue quente e assim por diante), mas cada um também tem suas próprias características (um cavalo tem cascos, mas uma baleia tem barbatanas e uma cauda).

Como você poderia modelar um cavalo e uma baleia em um programa? Uma maneira seria criar duas classes distintas, chamadas *Horse* e *Whale*. Cada classe poderia implementar os comportamentos que são únicos a esse tipo de mamífero, como *Trot* (trotar, para um cavalo) ou *Swim* (nadar, para uma baleia), de uma maneira específica. Mas como você trataria os comportamentos que são comuns a um cavalo e a uma baleia, como *Breathe* (respirar) ou *SuckleYoung* (amamentar)? Você poderia adicionar métodos duplicados com esses nomes às duas classes, mas essa situação torna-se um pesadelo de manutenção, especialmente se também decidir começar a modelar outros tipos de mamíferos, por exemplo, *Human* ou *Aardvark* (tamanduá).

No C#, você pode utilizar a herança de classe para resolver essas questões. Um cavalo, uma baleia, um humano e um tamanduá são tipos de mamíferos; portanto, você pode criar uma classe chamada *Mammal* que forneça a funcionalidade comum exibida por esses tipos. Você pode então declarar que todas as classes *Horse*, *Whale*, *Human* e *Aardvarks* herdam de *Mammal*. Essas classes incluiriam automaticamente a funcionalidade da classe *Mammal* (*Breathe*, *SuckleYoung* e assim por diante), mas você também poderia ampliar cada classe com a funcionalidade única de um tipo de mamífero particular à classe correspondente – o método *Trot* para a classe *Horse* e o método *Swim* para a classe *Whale*. Se for necessário modificar a maneira com que um método comum como *Breathe* funciona, você precisará alterá-lo apenas em um único lugar, a classe *Mammal*.

Herança

Você declara que uma classe herda de outra classe utilizando a seguinte sintaxe:

```
class DerivedClass : BaseClass
{
    ...
}
```

A classe derivada herda da classe base e os métodos na classe base se tornam parte da classe derivada. No C#, uma classe pode ser derivada de, no máximo, uma classe base; uma classe *não pode* ser derivada de duas ou mais classes. Mas, a menos que *DerivedClass* seja declarada como *sealed*, você pode criar outras classes derivadas que herdam de *DerivedClass* utilizando a mesma sintaxe. (Discutiremos as classes seladas no Capítulo 13, “Como criar interfaces e definir classes abstratas”.)

```
class DerivedSubClass : DerivedClass
{
    ...
}
```

No exemplo descrito anteriormente, você poderia declarar a classe *Mammal* como mostrado no exemplo a seguir. Os métodos *Breathe* e *SuckleYoung* são comuns a todos os mamíferos.

```
class Mammal
{
    public void Breathe()
    {
        ...
    }
}
```

```
public void SuckleYoung()
{
    ...
}
...
}
```

Então você poderia definir classes para cada tipo diferente de mamífero, acrescentando outros métodos, conforme necessário, como no exemplo a seguir:

```
class Horse : Mammal
{
    ...
    public void Trot()
    {
        ...
    }
}

class Whale : Mammal
{
    ...
    public void Swim()
    {
        ...
    }
}
```



Nota Se você é programador de C++, deve notar que não pode explicitar se a herança é pública, privada ou protegida. A herança no C# é sempre implicitamente pública. Se você conhece Java, note o uso do sinal de dois pontos e que não há palavra-chave *extends*.

Se você criar um objeto *Horse* em seu aplicativo, poderá chamar os métodos *Trot*, *Breathe* e *SuckleYoung*:

```
Horse myHorse = new Horse();
myHorse.Trot();
myHorse.Breathe();
myHorse.SuckleYoung();
```

Da mesma forma, você pode criar um objeto *Whale*, mas desta vez poderá chamar os métodos *Swim*, *Breathe* e *SuckleYoung*; *Trot* não está disponível, pois só é definido na classe *Horse*.



Importante A herança só se aplica às classes, não às estruturas. Não é possível definir uma hierarquia de herança própria com estruturas e não é possível definir uma estrutura derivada de uma classe ou de outra estrutura.

Na verdade, todas as estruturas herdam de uma classe abstrata chamada *System.ValueType*. (O Capítulo 13 explorará as classes abstratas.) Esse é apenas um detalhe de implementação do modo como o .NET Framework define o comportamento comum de tipos-valor baseados em pilha; é improvável que você utilize *ValueType* diretamente em seus aplicativos.

A classe *System.Object* revisitada

A classe *System.Object* é a classe raiz de todas as classes. Todas as classes derivam implicitamente de *System.Object*. Por consequência, o compilador C# reescreve silenciosamente a classe *Mammal* como o código a seguir (podendo ser escrita explicitamente, se você quiser):

```
class Mammal : System.Object
{
    ...
}
```

Qualquer método da classe *System.Object* é automaticamente repassado para baixo na cadeia de herança das classes que derivam de *Mammal*, como *Horse* e *Whale*. Em termos práticos, isso significa que todas as classes que você define herdam automaticamente as características da classe *System.Object*. Isso inclui métodos, como *ToString* (discutido no Capítulo 2, “Variáveis, operadores e expressões”), que é utilizado para converter um objeto em uma string, em geral para propósitos de exibição.

Chame construtores da classe base

Além dos métodos por ela herdados, uma classe derivada contém automaticamente todos os campos da classe base. Esses campos vão precisar de inicialização quando um objeto for criado. Esse tipo de inicialização costuma ser realizado em um construtor. Lembre-se de que todas as classes têm pelo menos um construtor. (Se você não fornecer um, o compilador gerará um construtor padrão.)

Uma boa prática é o construtor em uma classe derivada chamar o construtor de sua classe base como parte da inicialização, o que permite ao construtor da classe base realizar qualquer inicialização adicional exigida. Você pode especificar a palavra-chave *base* para chamar um construtor de classe base ao definir um construtor para uma classe herdeira, como mostrado neste exemplo:

```
class Mammal // classe base
{
    public Mammal(string name) // construtor para a classe base
    {
        ...
    }
    ...
}
```

```
class Horse : Mammal // classe derivada
{
    public Horse(string name)
        : base(name) // chama Mammal(name)
    {
        ...
    }
    ...
}
```

Se você não chamar explicitamente um construtor da classe base em um construtor da classe derivada, o compilador tentará inserir silenciosamente uma chamada ao construtor padrão da classe base antes de executar o código no construtor da classe derivada. Considerando o exemplo anterior, o compilador reescreve este código:

```
class Horse : Mammal
{
    public Horse(string name)
    {
        ...
    }
    ...
}
```

assim:

```
class Horse : Mammal
{
    public Horse(string name)
        : base()
    {
        ...
    }
    ...
}
```

Isso funcionará se *Mammal* tiver um construtor padrão público. Mas nem todas as classes têm um construtor padrão público (por exemplo, lembre-se de que o compilador gera apenas um construtor padrão, caso você não escreva construtores não padrão), caso no qual esquecer-se de chamar o construtor correto da classe base resulta em um erro de compilação.

Atribua classes

Exemplos anteriores mostraram como declarar uma variável utilizando um tipo classe e como utilizar a palavra-chave *new* para criar um objeto. Também há exemplos de como as regras de verificação de tipo do C# impedem que você atribua um objeto de um tipo a uma variável declarada com um tipo diferente. Por exemplo, dadas as definições das classes *Mammal*, *Horse* e *Whales* mostradas aqui, o código depois dessas definições é inválido:

```
class Mammal
{
    ...
}
class Horse : Mammal
{
```

```

    ...
}

class Whale : Mammal
{
    ...
}

Horse myHorse = new Horse(...);
Whale myWhale = myHorse;           // erro - tipos diferentes

```

Mas é possível referenciar um objeto a partir de uma variável de um tipo diferente, desde que o tipo utilizado seja uma classe que esteja acima na hierarquia de heranças. Portanto, as instruções a seguir são válidas:

```

Horse myHorse = new Horse(...);
Mammal myMammal = myHorse; // válido, Mammal é a classe base de Horse

```

Se você pensar em termos lógicos, todos os *Horses* são *Mammals*; portanto, é possível atribuir de uma maneira segura um objeto do tipo *Horse* a uma variável do tipo *Mammal*. A hierarquia de herança significa que você pode imaginar um *Horse* como um tipo especial de *Mammal*; ele tem tudo que um *Mammal* tem, com algumas características a mais, definidas por todos os métodos e campos que você adicionou à classe *Horse*. Você também pode fazer uma variável *Mammal* referenciar um objeto *Whale*. Mas há uma limitação significativa: ao referenciar um objeto *Horse* ou *Whale* utilizando uma variável *Mammal*, você só pode acessar métodos e campos definidos pela classe *Mammal*. Qualquer método adicional definido pela classe *Horse* ou *Whale* não é visível pela classe *Mammal*:

```

Horse myHorse = new Horse(...);
Mammal myMammal = myHorse;
myMammal.Breathe();           // OK - Breathe faz parte da classe Mammal
myMammal.Trot();             // erro - Trot não faz parte da classe Mammal

```



Nota A discussão anterior explica por que você pode atribuir quase tudo a uma variável *object*. Lembre-se de que *object* é um alias de *System.Object* e de que todas as classes herdam de *System.Object*, direta ou indiretamente.

Preste atenção, porque a situação inversa não é verdadeira. Você não pode atribuir um objeto *Mammal* irrestritamente a uma variável *Horse*:

```

Mammal myMammal = new Mammal(...);
Horse myHorse = myMammal; // erro

```

Isso parece uma restrição estranha, mas lembre-se de que nem todos os objetos *Mammals* são *Horses* – alguns podem ser *Whales*. É possível atribuir um objeto *Mammal* a uma variável *Horse*, contanto que você verifique primeiro se *Mammal* é realmente um *Horse*, utilizando o operador *as* ou *is* ou utilizando um casting (o Capítulo 7, “Criação e gerenciamento de classes e objetos”, discutiu os operadores *is* e *as* e o casting). O exemplo de código a seguir emprega o operador *as* para verificar

se *myMammal* referencia um *Horse* e, em caso afirmativo, a atribuição a *myHorseAgain* resulta em *myHorseAgain* referenciando o mesmo objeto *Horse*. Se *myMammal* referenciar algum outro tipo de *Mammal*, o operador *as* retornará, em vez disso, *null*.

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse;           // myMammal referencia um Horse
...
Horse myHorseAgain = myMammal as Horse; // OK - myMammal era um Horse
...
Whale myWhale = new Whale(...);
myMammal = myWhale;
...
myHorseAgain = myMammal as Horse;      // retorna null - myMammal era uma Whale
```

Declare métodos *new*

Uma das tarefas mais difíceis no campo da programação é inventar nomes exclusivos e significativos para os identificadores. Se você está definindo um método para uma classe e essa classe faz parte de uma hierarquia de herança, mais cedo ou mais tarde tentará reutilizar um nome que já está em uso por uma das classes mais acima na hierarquia. Se acontecer de uma classe base e uma classe derivada declararem dois métodos que têm a mesma assinatura, você receberá um aviso ao compilar o aplicativo.



Nota A assinatura do método é o nome do método e o número e tipos dos seus parâmetros, mas não seus tipos de retorno. Dois métodos que possuem o mesmo nome e a mesma lista de parâmetros têm a mesma assinatura, mesmo que retornem tipos diferentes.

Um método em uma classe derivada mascara (ou oculta) um método em uma classe base que tem a mesma assinatura. Por exemplo, se você compilar o código a seguir, o compilador gerará uma mensagem de aviso informado que *Horse.Talk* oculta o método herdado *Mammal.Talk*:

```
class Mammal
{
    ...
    public void Talk() // pressupõe que todos os mamíferos podem falar
    {
        ...
    }
}

class Horse : Mammal
{
    ...
    public void Talk() // cavalos falam diferentemente dos outros mamíferos!
    {
        ...
    }
}
```

Embora seu código seja compilado e executado, você deve considerar seriamente esse aviso. Se outra classe derivar de *Horse* e chamar o método *Talk*, ela talvez esteja esperando que o método implementado na classe *Mammal* seja chamado. Mas o método *Talk* na classe *Horse* oculta o método *Talk* na classe *Mammal* e, em vez disso, o método *Horse.Talk* será chamado. Na maioria das vezes, essa coincidência é um engano e você deve pensar em renomear os métodos para evitar conflitos. Mas se tiver certeza de que quer que os dois métodos tenham a mesma assinatura, ocultando assim o método *Mammal.Talk*, você pode silenciar o aviso utilizando a palavra-chave *new* como a seguir:

```
class Mammal
{
    ...
    public void Talk()
    {
        ...
    }
}

class Horse : Mammal
{
    ...
    new public void Talk()
    {
        ...
    }
}
```

Dessa maneira, o uso da palavra-chave *new* não altera o fato de que os dois métodos não têm relação alguma e de que a ocultação continua ocorrendo. Ela simplesmente desativa o aviso. "Eu sei o que estou fazendo; portanto, pare de me mostrar esses avisos".

Declare métodos virtuais

Às vezes, você quer ocultar a maneira como um método é implementado em uma classe base. Como exemplo, considere o método *ToString* em *System.Object*. A finalidade de *ToString* é converter um objeto na sua representação de string. Como esse método é muito útil, ele é um membro da classe *System.Object*; portanto, fornece automaticamente um método *ToString* a todas as classes. Mas como a versão de *ToString* implementada por *System.Object* sabe como converter uma instância de uma classe derivada em uma string? Uma classe derivada poderá conter qualquer número de campos com valores interessantes que deverão fazer parte da string. A resposta é que a implementação de *ToString* em *System.Object* é, na verdade, um pouco simplista. Tudo o que ele pode fazer é converter um objeto em uma string que contém o nome do seu tipo, como "Mammal" ou "Horse". No final das contas, não é muito útil. Então, por que fornecer um método tão inútil? A resposta para essa segunda pergunta exige que você pense um pouco mais detalhadamente.

Obviamente, *ToString* é uma boa ideia como conceito, e todas as classes devem fornecer um método que possa ser utilizado para converter objetos em strings para propósitos de exibição ou depuração. É só a implementação que exige atenção. De fato, você não precisa chamar o método *ToString* definido por *System.Object*; ele é simplesmente um espaço reservado. Em vez disso, talvez você ache mais útil fornecer

sua própria versão do método *ToString* em cada classe que definir, desconsiderando a implementação padrão em *System.Object*. A versão em *System.Object* só está lá como uma rede de segurança, no caso de uma classe não implementar ou exigir sua própria versão específica do método *ToString*.

Um método escrito para ser redefinido é chamado método *virtual*. Você precisa saber a diferença entre *redefinir* um método e *ocultar* um método. Redefinir um método é um mecanismo para fornecer diferentes implementações do mesmo método – os métodos estão todos relacionados porque se destinam a executar a mesma tarefa, mas de uma maneira específica à classe. Ocultar um método é um meio de substituir um método por outro – os métodos em geral não estão relacionados e podem executar tarefas totalmente diferentes. Sobrescrever um método é um conceito útil de programação; ocultar um método é, muitas vezes, um erro.

Você pode marcar um método como *virtual* utilizando a palavra-chave *virtual*. Por exemplo, o método *ToString* na classe *System.Object* é definido assim:

```
namespace System
{
    class Object
    {
        public virtual string ToString()
        {
            ...
        }
        ...
    }
    ...
}
```



Nota Se você tem experiência em desenvolvimento com Java, deve notar que os métodos C# não são virtuais por padrão.

Declare métodos *override*

Se uma classe base declara que um método é *virtual*, uma classe derivada pode utilizar a palavra-chave *override* para declarar outra implementação desse método, como demonstrado aqui:

```
class Horse : Mammal
{
    ...
    public override string ToString()
    {
        ...
    }
}
```

A nova implementação do método na classe derivada pode chamar a implementação original do método na classe base utilizando a palavra-chave *base*, assim:

```
public override string ToString()
{
```

```
base.ToString();
...
}
```

Há algumas regras importantes que você precisa seguir ao declarar métodos polimórficos (conforme discutido no quadro, "Métodos virtuais e polimorfismo") utilizando as palavras-chave *virtual* e *override*:

- Um método *virtual* não pode ser privado; seu objetivo é ser exposto para outras classes por meio de herança. Da mesma forma, os métodos *override* não podem ser privados, pois uma classe não pode alterar o nível de proteção de um método que herda. Contudo, os métodos *override* podem ter uma forma especial de privacidade, conhecida como *acesso protegido*, conforme você vai descobrir na próxima seção.
- As assinaturas dos métodos *virtual* e *override* devem ser idênticas; elas devem ter o mesmo nome, número e tipos de parâmetros. Além disso, o dois métodos devem retornar o mesmo tipo.
- Você só pode redefinir um método *virtual*. Se o método da classe base não for *virtual* e você tentar redefini-lo, isso resultará em um erro de tempo de compilação. Esse tema é delicado; cabe à classe base decidir se seus métodos podem ser redefinidos.
- Se a classe derivada não declarar o método utilizando a palavra-chave *override*, ela não redefinirá o método da classe base – ela ocultará o método. Em outras palavras, torna-se uma implementação de um método completamente diferente que, por acaso, tem o mesmo nome. Como antes, isso acarretará um aviso de ocultação em tempo de compilação, que você pode silenciar utilizando a palavra-chave *new*, como já foi descrito.
- Um método *override* é implicitamente *virtual* e pode ser redefinido em uma classe derivada posterior. Mas você não pode declarar explicitamente que um método *override* é *virtual* utilizando a palavra-chave *virtual*.



Métodos virtuais e polimorfismo

Os métodos virtuais tornam possível chamar diferentes versões do mesmo método com base no tipo de objeto determinado dinamicamente em tempo de execução. Considere as classes do exemplo a seguir, que definem uma variação na hierarquia *Mammal* descrita antes:

```
class Mammal
{
    ...
    public virtual string GetTypeName()
    {
        return "This is a mammal";
    }
}

class Horse : Mammal
{
```

```

...
public override string GetTypeName()
{
    return "This is a horse";
}
}

class Whale : Mammal
{
...
public override string GetTypeName ()
{
    return "This is a whale";
}
}

class Aardvark : Mammal
{
...
}

```

Note duas coisas: em primeiro lugar, a palavra-chave *override* usada pelo método *GetTypeName* nas classes *Horse* e *Whale*; em segundo lugar, o fato de que a classe *Aardvark* não tem um método *GetTypeName*.

Agora, examine o bloco de código a seguir:

```

Mammal myMammal;
Horse myHorse = new Horse(...);
Whale myWhale = new Whale(...);
Aardvark myAardvark = new Aardvark(...);

myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName()); // Horse
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName()); // Whale
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName()); // Aardvark

```

Qual será a saída das três diferentes instruções *Console.WriteLine*? À primeira vista, você poderia esperar que todas elas imprimissem "This is a mammal" ("Este é um mamífero"), porque cada instrução chama o método *GetTypeName* na variável *myMammal*, que é um *Mammal*. Mas, no primeiro caso, você pode ver que *myMammal* na verdade é uma referência a um *Horse*. (Lembre-se de que você pode atribuir um *Horse* a uma variável *Mammal* porque a classe *Horse* herda da classe *Mammal*.) Visto que o método *GetTypeName* é definido como *virtual*, o runtime determina que deve chamar o método *Horse.GetTypeName*; portanto, a instrução na verdade imprime a mensagem "This is a horse" ("Este é um cavalo"). A mesma lógica se aplica à segunda instrução *Console.WriteLine*, que emite a mensagem "This is a whale" ("Esta é uma baleia"). A terceira instrução chama *Console.WriteLine* em um objeto *Aardvark*. Mas a classe *Aardvark* não tem um método *GetTypeName*; então, o método padrão na classe *Mammal* é chamado, retornando a string "This is a mammal".

Esse fenômeno da mesma instrução chamando um método diferente, dependendo de seu contexto, é chamado de *polimorfismo*, que literalmente significa "muitas formas".

Entenda o acesso *protected*

As palavras-chave de acesso *public* e *private* geram dois extremos de acessibilidade: os campos e métodos públicos de uma classe são acessíveis a todos, enquanto os campos e métodos privados de uma classe são acessíveis apenas à própria classe.

Esses dois extremos são suficientes considerando-se as classes isoladamente. Mas como todos os programadores experientes em orientação a objetos sabem, classes isoladas não podem resolver problemas complexos. A herança é uma maneira muito poderosa de conectar as classes, e há claramente uma relação especial e próxima entre uma classe derivada e sua classe base. Em geral, é útil para uma classe base permitir que as classes derivadas acessem alguns de seus membros, enquanto oculta esses mesmos membros das classes que não fazem parte da hierarquia de herança. Nessa situação, você pode marcar os membros com a palavra-chave *protected*: Isso funciona assim:

- Se uma classe A deriva de outra classe B, ela pode acessar os membros de classe protegidos da classe B. Em outras palavras, dentro da classe A derivada, um membro protegido da classe B é público.
- Se uma classe A não deriva de outra classe B, ela não pode acessar membro algum protegido da classe B. Assim, dentro da classe A, um membro protegido da classe B é privado.

O C# dá aos programadores completa liberdade para declarar métodos e campos como protegidos. Mas a maioria das diretrizes de programação orientada a objetos recomenda manter seus campos estritamente privados sempre que possível, e só afrouxar essas restrições quando for muito necessário. Os campos públicos violam o encapsulamento porque todos os usuários da classe têm acesso direto e irrestrito aos campos. Os campos protegidos mantêm o encapsulamento para os usuários de uma classe, para quem são inacessíveis. Contudo, os campos protegidos ainda permitem que o encapsulamento seja violado por outras classes que herdam da classe base.



Nota Você pode acessar um membro protegido de uma classe base não só em uma classe derivada, mas também em classes derivadas da classe derivada. Um membro protegido de uma classe base mantém sua acessibilidade protegida em uma classe derivada e é acessível às outras classes derivadas.

No exercício a seguir, você vai definir uma hierarquia simples de classes para modelar diferentes tipos de veículos. Serão definidas uma classe base chamada *Vehicle* e classes derivadas chamadas *Airplane* e *Car*. Você vai definir métodos comuns chamados *StartEngine* e *StopEngine* na classe *Vehicle* e vai adicionar alguns métodos às duas classes derivadas que são específicos para essas classes. Por fim, vai adicionar um método virtual chamado *Drive* à classe *Vehicle* e vai redefinir a implementação padrão desse método nas duas classes derivadas.

Crie uma hierarquia de classes

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto Vehicles, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 12\Windows X\Vehicles na sua pasta Documentos.

O projeto Vehicles contém o arquivo Program.cs, que define a classe *Program* com os métodos *Main* e *doWork* que vimos nos exercícios anteriores.

3. No Solution Explorer, clique com o botão direito do mouse no projeto Vehicles, aponte para Add e então clique em Class.

A caixa de diálogo Add New Item – Vehicles se abre.

4. Na caixa de diálogo Add New Item – Vehicles, verifique se o template Class está destacado no painel central, digite **Vehicle.cs** na caixa Name e clique em Add.

O arquivo Vehicle.cs é criado e adicionado ao projeto e aparece na janela Code and Text Editor. O arquivo contém a definição de uma classe vazia chamada *Vehicle*.

5. Adicione os métodos *StartEngine* e *StopEngine* à classe *Vehicle* como mostrado em negrito a seguir:

```
class Vehicle
{
    public void StartEngine(string noiseToMakeWhenStarting)
    {
        Console.WriteLine("Starting engine: {0}", noiseToMakeWhenStarting);
    }

    public void StopEngine(string noiseToMakeWhenStopping)
    {
        Console.WriteLine("Stopping engine: {0}", noiseToMakeWhenStopping);
    }
}
```

Todas as classes que derivam da classe *Vehicle* herdarão esses métodos. Os valores dos parâmetros *noiseToMakeWhenStarting* e *noiseToMakeWhenStopping* serão diferentes para cada tipo diferente de veículo, e isso o ajudará posteriormente a identificar qual veículo está sendo iniciado e parado.

6. No menu Project, clique em Add Class.

A caixa de diálogo Add New Item – Vehicles se abre mais uma vez.

7. Na caixa Name, digite **Airplane.cs** e clique em Add.

Um novo arquivo contendo uma classe chamada *Airplane* é adicionado ao projeto e aparece na janela Code and Text Editor.

8. Na janela Code and Text Editor, modifique a definição da classe *Airplane* de modo que ela herde da classe *Vehicle*, como mostrado em negrito:

```
class Airplane : Vehicle
{
}
```

9. Adicione os métodos *TakeOff* e *Land* à classe *Airplane*, como mostrado em negrito:

```
class Airplane : Vehicle
{
    public void TakeOff()
    {
        Console.WriteLine("Taking off");
    }

    public void Land()
    {
        Console.WriteLine("Landing");
    }
}
```

10. No menu Project, clique em Add Class.

A caixa de diálogo Add New Item – Vehicles se abre mais uma vez.

11. Na caixa de texto Name, digite **Car.cs** e clique em Add.

Um novo arquivo contendo uma classe chamada *Car* é adicionado ao projeto e aparece na janela Code and Text Editor.

12. Na janela Code and Text Editor, modifique a definição da classe *Car* de modo que ela derive da classe *Vehicle*, como mostrado em negrito:

```
class Car : Vehicle
{
}
```

13. Adicione os métodos *Accelerate* e *Brake* à classe *Car*, como mostrado em negrito:

```
class Car : Vehicle
{
    public void Accelerate()
    {
        Console.WriteLine("Accelerating");
    }

    public void Brake()
    {
        Console.WriteLine("Braking");
    }
}
```

14. Exiba o arquivo *Vehicle.cs* na janela Code and Text Editor.

15. Adicione a implementação padrão do método virtual *Drive* à classe *Vehicle*, como apresentado aqui em negrito:

```
class Vehicle
{
    ...
    public virtual void Drive()
    {
        Console.WriteLine("Default implementation of the Drive method");
    }
}
```

16. Exiba o arquivo Program.cs na janela Code and Text Editor.
17. No método *doWork*, exclua o comentário *// TODO*: e adicione código para criar uma instância da classe *Airplane* e testar os métodos simulando uma viagem rápida de avião, como a seguir:

```
static void doWork()
{
    Console.WriteLine("Journey by airplane:");
    Airplane myPlane = new Airplane();
    myPlane.StartEngine("Contact");
    myPlane.TakeOff();
    myPlane.Drive();
    myPlane.Land();
    myPlane.StopEngine("Whirr");
}
```

18. Adicione as seguintes instruções (mostradas em negrito) ao método *doWork*, depois do código que você acabou de escrever. Essas instruções criam uma instância da classe *Car* e testam seus métodos.

```
static void doWork()
{
    ...
    Console.WriteLine("\nJourney by car:");
    Car myCar = new Car();
    myCar.StartEngine("Brm brm");
    myCar.Accelerate();
    myCar.Drive();
    myCar.Brake();
    myCar.StopEngine("Phut phut");
}
```

19. No menu Debug, clique em Start Without Debugging.

Na janela de console, observe que o programa emite mensagens simulando as diferentes etapas de uma viagem de avião e de carro, como mostrado na imagem a seguir:

```
C:\Windows\system32
Journey by airplane:
Starting engine: Contact
Taking off
Default implementation of the Drive method
Landing
Stopping engine: Whirr

Journey by car:
Starting engine: Brm brm
Accelerating
Default implementation of the Drive method
Braking
Stopping engine: Phut phut
Press any key to continue . . .
```

Observe que os dois meios de transporte chamam a implementação padrão do método virtual *Drive* porque nenhuma classe atualmente redefine esse método.

20. Pressione Enter para fechar o aplicativo e retornar ao Visual Studio 2013.

- 21.** Exiba a classe *Airplane* na janela Code and Text Editor. Redefina o método *Drive* na classe *Airplane*, como segue em negrito:

```
class Airplane : Vehicle
{
    ...
    public override void Drive()
    {
        Console.WriteLine("Flying");
    }
}
```



Nota O IntelliSense exibe uma lista dos métodos virtuais disponíveis. Se você selecionar o método *Drive* na lista IntelliSense, o Visual Studio inserirá automaticamente no seu código uma instrução que chama o método *base.Drive*. Se isso acontecer, exclua a instrução, pois ela não é necessária neste exercício.

- 22.** Exiba a classe *Car* na janela Code and Text Editor. Redefina o método *Drive* na classe *Car*, como segue em negrito:

```
class Car : Vehicle
{
    ...
    public override void Drive()
    {
        Console.WriteLine("Motoring");
    }
}
```

- 23.** No menu Debug, clique em Start Without Debugging.

Na janela de console, observe que o objeto *Airplane* agora exibe a mensagem *Flying* quando o aplicativo chama o método *Drive*, e o objeto *Car* apresenta a mensagem *Motoring*.

```
C:\Windows\system32
Journey by airplane:
Starting engine: Contact
Taking off
Flying ←
Landing
Stopping engine: Whirr

Journey by car:
Starting engine: Brm brm
Accelerating
Motoring ←
Braking
Stopping engine: Phut phut
Press any key to continue . . .
```

- 24.** Pressione Enter para fechar o aplicativo e retornar ao Visual Studio 2013.
25. Exiba o arquivo Program.cs na janela Code and Text Editor.

- 26.** Adicione as instruções mostradas em negrito ao final do método *doWork*:

```
static void doWork()
{
    ...
    Console.WriteLine("\nTesting polymorphism");
    Vehicle v = myCar;
    v.Drive();
    v = myPlane;
    v.Drive();
}
```

Esse código testa o polimorfismo do método virtual *Drive*. O código cria uma referência ao objeto *Car* utilizando uma variável *Vehicle* (o que é seguro, pois todos os objetos *Car* são *Vehicle*) e então chama o método *Drive* empregando essa variável *Vehicle*. As duas instruções finais referenciam a variável *Vehicle* no objeto *Airplane* e chamam o que parece ser o mesmo método *Drive* mais uma vez.

- 27.** No menu Debug, clique em Start Without Debugging.

Na janela de console, verifique que as mesmas mensagens aparecem, como anteriormente, seguidas por este texto:

```
Testing polymorphism
Motoring
Flying
```

```
C:\Windows\system32
Journey by airplane:
Starting engine: Contact
Taking off
Flying
Landing
Stopping engine: Whirr
Journey by car:
Starting engine: Brm brm
Accelerating
Motoring
Braking
Stopping engine: Phut phut
Testing polymorphism
Motoring
Flying
Press any key to continue . . .
```

O método *Drive* é virtual; portanto, o runtime (não o compilador) determina qual versão do método *Drive* chamar ao ativá-lo por meio de uma variável *Vehicle*, com base no tipo real do objeto referenciado por essa variável. No primeiro caso, o objeto *Vehicle* referencia um *Car*; assim, o aplicativo chama o método *Car:Drive*. No segundo caso, o objeto *Vehicle* referencia um *Airplane*; portanto, o aplicativo chama o método *Airplane:Drive*.

- 28.** Pressione Enter para fechar o aplicativo e retornar ao Visual Studio 2013.

Métodos de extensão

A herança é um recurso poderoso que torna possível ampliar a funcionalidade de uma classe criando uma nova classe derivada dela. Mas, às vezes, o uso da herança não é o mecanismo mais apropriado para adicionar novos comportamentos, em especial se você precisa estender rapidamente um tipo sem afetar o código existente.

Por exemplo, suponha que você queira adicionar um novo recurso ao tipo *int*, como um método chamado *Negate* que retorna o valor negativo equivalente que um inteiro atualmente contém. (Eu sei que você poderia simplesmente utilizar o operador unário de menos [-] para realizar a mesma tarefa, mas seja paciente comigo.) Uma maneira de conseguir isso é definir um novo tipo chamado *NegInt32* que herda de *System.Int32* (*int* é um alias para *System.Int32*) e que adiciona o método *Negate*:

```
class NegInt32 : System.Int32 // Não tente isso!
{
    public int Negate()
    {
        ...
    }
}
```

Teoricamente, *NegInt32* herdará toda a funcionalidade associada ao tipo *System.Int32*, além do método *Negate*. Há duas razões para você não querer seguir essa estratégia:

- Esse método só será aplicado ao tipo *NegInt32* e, se você quiser utilizá-lo com as variáveis *int* existentes no seu código, terá de alterar a definição de cada variável *int* para o tipo *NegInt32*.
- O tipo *System.Int32* é na verdade uma estrutura, não uma classe, e você não pode utilizar herança com estruturas.

É aí que os métodos de extensão tornam-se muito úteis.

Utilizando um método de extensão é possível estender um tipo existente (uma classe ou uma estrutura) com métodos estáticos adicionais. Esses métodos estáticos tornam-se imediatamente disponíveis para seu código em qualquer instrução que referencie dados do tipo estendido.

Você define um método de extensão em uma classe *estática* e especifica o tipo que o método aplica a ela como o primeiro parâmetro para o método, juntamente com a palavra-chave *this*. A seguir, um exemplo que mostra como você pode implementar o método de extensão *Negate* para o tipo *int*:

```
static class Util
{
    public static int Negate(this int i)
    {
        return -i;
    }
}
```

A sintaxe parece um pouco estranha, mas é a palavra-chave *this* prefixando o parâmetro para *Negate* que o identifica como um método de extensão, e o fato de que o parâmetro que *this* prefixa é um *int* significa que você está estendendo o tipo *int*.

Para utilizar o método de extensão, coloque a classe *Util* no escopo. (Se necessário, adicione uma instrução *using* especificando o namespace à qual a classe *Util* pertence.) Então, você poderá simplesmente utilizar a notação de ponto (.) para referenciar o método, desta maneira:

```
int x = 591;
Console.WriteLine("x.Negate {0}", x.Negate());
```

Observe que não é preciso referenciar a classe *Util* em nenhum lugar na instrução que chama o método *Negate*. O compilador C# detecta automaticamente todos os métodos de extensão para determinado tipo a partir de todas as classes estáticas que estão em escopo. Você também pode chamar o método *Util.Negate* passando um *int* como parâmetro, utilizando a sintaxe comum que já vimos, embora esse uso torne óbvio o propósito da definição do método como um método de extensão:

```
int x = 591;
Console.WriteLine("x.Negate {0}", Util.Negate(x));
```

No exercício a seguir, você adicionará um método de extensão ao tipo *int*. Com esse método de extensão é possível converter o valor de uma variável *int* da base 10 para uma representação desse valor em uma base numérica diferente.

Crie um método de extensão

1. No Visual Studio 2013, abra o projeto *ExtensionMethod*, localizado na pasta *\Microsoft Press\ Visual CSharp Step by Step\Chapter 12\Windows X\ExtensionMethod* na sua pasta Documentos.
2. Exiba o arquivo *Util.cs* na janela Code and Text Editor.

Esse arquivo contém uma classe estática chamada *Util* em um namespace chamado *Extensions*. Lembre-se de que você deve definir métodos de extensão dentro de uma classe estática. A classe está vazia, a não ser pelo comentário *// TODO:*

3. Exclua o comentário e declare um método estático público à classe *Util*, chamado *ConvertToBase*. O método deve receber dois parâmetros: um parâmetro *int* chamado *i*, prefixado com a palavra-chave *this* para indicar que se trata de um método de extensão para o tipo *int*, e outro parâmetro *int* normal, chamado *baseToConvertTo*.

O método converterá o valor em *i* para a base indicada por *baseToConvertTo*. O método deve retornar um *int* contendo o valor convertido.

O método *ConvertToBase* deve ser semelhante a este:

```
static class Util
{
    public static int ConvertToBase(this int i, int baseToConvertTo)
    {
    }
}
```

- Adicione uma instrução *if* ao método *ConvertToBase* que verifica se o valor do parâmetro *baseToConvertTo* está entre 2 e 10.

O algoritmo utilizado por este exercício não funciona de maneira confiável fora desse intervalo de valores. Lance uma exceção *ArgumentException* com uma mensagem adequada se o valor de *baseToConvertTo* estiver fora desse intervalo.

O método *ConvertToBase* deve ser semelhante a este:

```
public static int ConvertToBase(this int i, int baseToConvertTo)
{
    if (baseToConvertTo < 2 || baseToConvertTo > 10)
    {
        throw new ArgumentException("Value cannot be converted to base " +
            baseToConvertTo.ToString());
    }
}
```

- Adicione as seguintes instruções mostradas em negrito ao método *ConvertToBase*, após a instrução que lança a exceção *ArgumentException*.

Esse código implementa um algoritmo bem conhecido que converte um número de base 10 para uma base numérica diferente. (O Capítulo 5, “Atribuição composta e instruções de iteração”, apresentou uma versão desse algoritmo para converter um número decimal em octal.)

```
public static int ConvertToBase(this int i, int baseToConvertTo)
{
    ...
    int result = 0;
    int iterations = 0;
    do
    {
        int nextDigit = i % baseToConvertTo;
        i /= baseToConvertTo;
        result += nextDigit * (int)Math.Pow(10, iterations);
        iterations++;
    }
    while (i != 0);

    return result;
}
```

- Exiba o arquivo Program.cs na janela Code and Text Editor.

7. Adicione a seguinte diretiva *using* após a diretiva *using System*; na parte superior do arquivo:

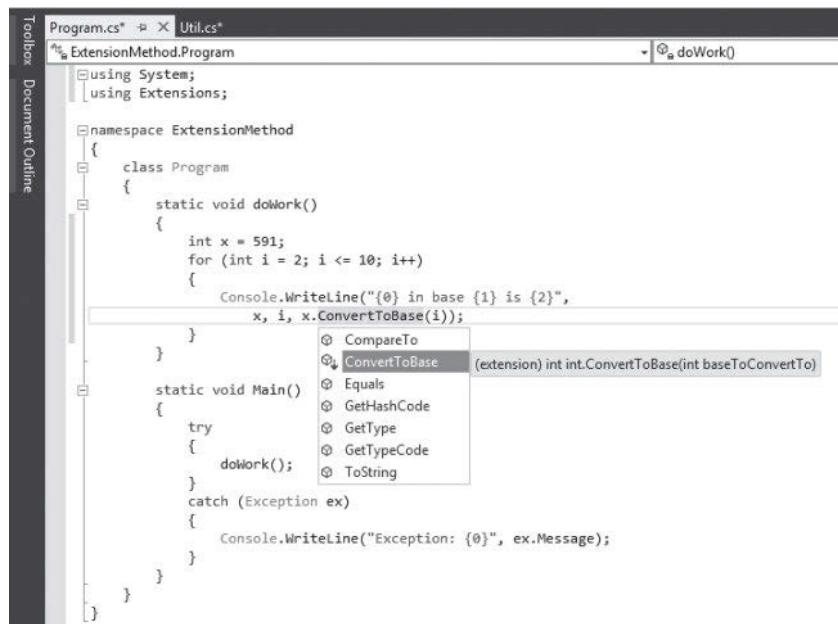
```
using Extensions;
```

Essa instrução dá escopo ao namespace que contém a classe *Util*. O método de extensão *ConvertToBase* não será visível no arquivo *Program.cs* se você não fizer isso.

8. Adicione as seguintes instruções, mostradas em negrito, ao final do método *doWork* da classe *Program*, substituindo o comentário *// TODO*:

```
static void doWork()
{
    int x = 591;
    for (int i = 2; i <= 10; i++)
    {
        Console.WriteLine("{0} in base {1} is {2}",
            x, i, x.ConvertToBase(i));
    }
}
```

Esse código cria um *int* chamado *x* e o configura com o valor 591. (Você pode escolher o valor inteiro que desejar.) O código então utiliza um loop para imprimir o valor 591 em todas as bases numéricas entre 2 e 10. Observe que *ConvertToBase* aparece como um método de extensão no IntelliSense quando você digita o ponto (.) após o *x* na instrução *Console.WriteLine*.



9. No menu Debug, clique em Start Without Debugging. Confirme que o programa exibe no console as mensagens que mostram o valor 591 nas diferentes bases numéricas, assim:

```

C:\Windows\system32
591 in base 2 is 1001001111
591 in base 3 is 210220
591 in base 4 is 21033
591 in base 5 is 4331
591 in base 6 is 2423
591 in base 7 is 1503
591 in base 8 is 1117
591 in base 9 is 726
591 in base 10 is 591
Press any key to continue . . .

```

10. Pressione Enter para fechar o programa e retornar ao Visual Studio 2013.

Resumo

Neste capítulo, você aprendeu a utilizar a herança para definir uma hierarquia de classes e agora deve entender como redefinir métodos herdados e implementar métodos virtuais. Também aprendeu a adicionar um método de extensão a um tipo existente.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 13.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Criar uma classe derivada a partir de uma classe base	Declare o novo nome da classe, seguido por dois-pontos e pelo nome da classe base. Por exemplo: class DerivedClass : BaseClass { ... }
Chamar um construtor de classe base como parte do construtor para uma classe que o herda	Sufixe a definição do construtor com uma chamada para a base, antes do corpo do construtor da classe derivada, e forneça os parâmetros necessários para o construtor base. Por exemplo: class DerivedClass : BaseClass { ... public DerivedClass(int x) : base(x) { ... } ... }

Para	Faça isto
Declarar um método virtual	Use a palavra-chave <code>virtual</code> ao declarar o método. Por exemplo:
Implementar um método em uma classe derivada que redefine um método virtual herdado	Use a palavra-chave <code>override</code> ao declarar o método na classe derivada. Por exemplo:
Definir um método de extensão para um tipo	Adicione um método público estático a uma classe estática. O primeiro parâmetro deve ser do tipo estendido, precedido pela palavra-chave <code>this</code> . Por exemplo:

```

class Mammal
{
    public virtual void Breathe()
    {
        ...
    }
}

class Whale : Mammal
{
    public override void Breathe()
    {
        ...
    }
}

static class Util
{
    public static int Negate(this int i)
    {
        return -i;
    }
}

```

CAPÍTULO 13

Como criar interfaces e definir classes abstratas

Neste capítulo, você vai aprender a:

- Definir uma interface, especificando as assinaturas e os tipos de retorno de métodos.
- Implementar uma interface em uma estrutura ou classe.
- Fazer referência a uma classe por meio de uma interface.
- Capturar detalhes de implementação comuns em uma classe abstrata.
- Implementar classes seladas que não podem ser utilizadas para derivar novas classes.

O verdadeiro poder da herança de uma classe vem da herança de uma interface. Uma interface não contém qualquer código ou dado; ela especifica os métodos e as propriedades que devem ser fornecidos por uma classe que herda da interface. Utilizar uma interface possibilita a separação completa dos nomes e das assinaturas dos métodos de uma classe de um lado e a implementação do método de outro.

Classes abstratas e interfaces se parecem, exceto pelo fato de que podem conter código e dados. É possível, entretanto, especificar que determinados métodos de uma classe abstrata sejam virtuais, de maneira que uma classe que herde da classe abstrata disponha de sua própria implementação desses métodos. Muitas vezes, você faz uso de classes abstratas com interfaces, e elas fornecem conjuntamente uma técnica fundamental que possibilita construir estruturas de programação extensíveis, como está descrito neste capítulo.

Interfaces

Suponha que você queira definir uma nova classe na qual possa armazenar coleções de objetos, quase como um array. Mas, em vez de utilizar um array, você quer fornecer um método chamado *RetrieveInOrder* para permitir que os aplicativos recuperem objetos em uma sequência que dependa do tipo de objeto que a coleção contém (com um array normal é possível iterar pelo seu conteúdo e, por padrão, os itens são recuperados de acordo com seus índices). Por exemplo, se a coleção armazena objetos alfanuméricos, como strings, ela deve permitir que um aplicativo recupere essas strings em sequência, de acordo com a intercalação (collation) do computador; se a coleção armazena objetos numéricos, como inteiros, ela deve permitir que o aplicativo recupere os objetos em ordem numérica.

Ao definir a classe de coleção, você não quer restringir os tipos de objetos que ela pode armazenar (os objetos podem ser até mesmo tipos de classe ou estruturas) e, consequentemente, não sabe como ordenar esses objetos. Portanto, a pergunta é, ao escrever a classe de coleção, como fornecer um método nessa classe que ordene os objetos cujos tipos você não conhece? À primeira vista, esse problema parece semelhante ao problema *ToString* descrito no Capítulo 12, “Herança”, que poderia ser resolvido declarando um método virtual que as subclasses da sua classe de coleção podem redefinir. Mas esse não é o caso. Não há relacionamento de herança entre a classe de coleção e os objetos que ela armazena; portanto, um método virtual não seria muito útil. Se você pensar um pouco, o problema é que a maneira como os objetos na coleção devem ser ordenados depende do tipo dos objetos nela existentes, não da coleção. A solução, então, é exigir que todos os objetos fornecam um método, como o método *CompareTo* mostrado no próximo exemplo, que o método *RetrieveInOrder* da coleção pode chamar, permitindo que a coleção compare esses objetos entre si.

```
int CompareTo(object obj)
{
    // retorna 0 se essa instância é igual a obj
    // retorna < 0 se essa instância é menor que obj
    // retorna > 0 se essa instância é maior que obj
    ...
}
```

Você pode definir uma interface para objetos colecionáveis que inclua o método *CompareTo* e especificar que a classe de coleção só pode conter as classes que implementam essa interface. Uma interface é, assim, semelhante a um contrato. Se uma classe implementar uma interface, esta garante que a classe conterá todos os métodos especificados na interface. Esse mecanismo assegura que você será capaz de chamar o método *CompareTo* em todos os objetos da coleção e ordená-los.

Utilizando interfaces é possível realmente separar “o que” do “como”. A interface fornece apenas o nome, o tipo de retorno e os parâmetros do método. A forma como o método é implementado não é uma preocupação da interface. A interface descreve a funcionalidade que uma classe deve fornecer, mas não como essa funcionalidade é implementada.

Defina uma interface

A definição de uma interface é sintaticamente semelhante à definição de uma classe, exceto que é utilizada a palavra-chave *interface* em lugar de *class*. Dentro da interface, os métodos são declarados exatamente como em uma classe ou em uma estrutura, exceto pelo fato de você nunca especificar um modificador de acesso (*public*, *private* ou *protected*). Além disso, em uma interface, os métodos não têm implementação; eles são simplesmente declarações, e todos os tipos que implementam a interface devem fornecer suas próprias implementações. Assim, você deve substituir o corpo do método por um ponto e vírgula. Veja um exemplo:

```
interface IComparable
{
    int CompareTo(object obj);
}
```



Dica A documentação do Microsoft .NET Framework recomenda começar o nome de interfaces com a letra *I* maiúscula. Essa convenção é o último vestígio da notação húngara no C#. Casualmente, o namespace *System* define a interface *IComparable* que acabamos de mostrar.

Uma interface não pode conter dados; você não pode adicionar campos (nem mesmo privados) a uma interface.

Implemente uma interface

Para implementar uma interface, você declara uma classe ou estrutura que herda da interface e implementa *todos* os métodos especificados por ela. Não se trata de herança propriamente dita, embora a sintaxe seja a mesma e parte da semântica (que veremos mais adiante neste capítulo) tenha muitas das características da herança. Você deve notar que, ao contrário da herança de classe, uma estrutura pode implementar uma interface.

Por exemplo, suponha que você esteja definindo a hierarquia *Mammal* descrita no Capítulo 12, mas precise especificar que mamíferos terrestres fornecem um método chamado *NumberOfLegs* que retorna como um *int* o número de patas que um mamífero tem. (Mamíferos marinhos não implementam essa interface.) Você poderia definir a interface dos mamíferos terrestres *ILandBound* que contém esse método, assim:

```
interface ILandBound
{
    int NumberOfLegs();
}
```

Você poderia então implementar essa interface na classe *Horse*. Você herda da interface e fornece uma implementação de cada método definido pela interface (neste caso, existe apenas um método: *NumberOfLegs*).

```
class Horse : ILandBound
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

Ao implementar uma interface, você deve garantir que cada método corresponda exatamente ao método da interface correspondente, de acordo com as regras a seguir:

- O nome e o tipo de retorno dos métodos devem se corresponder exatamente.
- Qualquer parâmetro (incluindo os modificadores de palavra-chave *ref* e *out*) devem se corresponder exatamente.
- Todos os métodos que implementam uma interface devem ser publicamente acessíveis. Mas se você estiver utilizando uma implementação explícita de interface, o método não deve ter um qualificador de acesso.

Se houver alguma diferença entre a definição da interface e sua implementação declarada, a classe não compilará.



Dica O ambiente de desenvolvimento integrado (IDE) do Microsoft Visual Studio pode ajudar a reduzir erros de codificação causados pela não implementação dos métodos em uma interface. O Implement Interface Wizard pode gerar stubs para cada item em uma interface implementada por uma classe. Então, você preenche esses stubs com o código apropriado. Veremos como utilizar esse assistente nos exercícios posteriores deste capítulo.

Uma classe pode herdar de outra classe e implementar uma interface ao mesmo tempo. Nesse caso, o C# não faz distinção entre a classe base e a interface utilizando palavras-chave específicas, como o Java faz. Em vez disso, o C# emprega uma notação posicional. A classe base é sempre nomeada primeiramente, seguida por uma vírgula, seguida pela interface. O exemplo a seguir define *Horse* como uma classe que é um *Mammal*, mas que também implementa a interface *ILandBound*:

```
interface ILandBound
{
    ...
}

class Mammal
{
    ...
}

class Horse : Mammal , ILandBound
{
    ...
}
```



Nota Uma interface, *InterfaceA*, pode herdar de outra interface, *InterfaceB*. Tecnicamente, isso é conhecido como *extensão de interface*, em vez de herança. Nesse caso, qualquer classe ou estrutura que implemente *InterfaceA* deve fornecer implementações de todos os métodos de *InterfaceB* e de *InterfaceA*.

Referencie uma classe por meio de sua interface

Da mesma maneira que você pode referenciar um objeto utilizando uma variável definida como uma classe mais elevada na hierarquia, também pode referenciar um objeto utilizando uma variável definida como uma interface que sua classe implementa. Considerando o exemplo anterior, você pode referenciar um objeto *Horse* utilizando uma variável *ILandBound*, como mostrado a seguir:

```
Horse myHorse = new Horse(...);
ILandBound iMyHorse = myHorse; // válido
```

Isso funciona porque todos os cavalos são mamíferos terrestres (*land bound*), embora o contrário não seja verdadeiro – você não pode atribuir um objeto *ILandBound* a uma variável *Horse* sem antes fazer um casting nela, para verificar se realmente ela faz referência a um objeto *Horse* e não a alguma outra classe que implementa a interface *ILandBound*.

A técnica de referenciar um objeto por meio de uma interface é útil, pois você pode utilizá-la para definir métodos que podem receber diferentes tipos como parâmetros, contanto que os tipos implementem uma interface especificada. Por exemplo, o método *FindLandSpeed* mostrado abaixo pode receber qualquer parâmetro que implemente a interface *ILandBound*:

```
int FindLandSpeed(ILandBound landBoundMammal)
{
    ...
}
```

Você pode verificar se um objeto é uma instância de uma classe que implementa uma interface específica utilizando o operador *is*, o qual foi demonstrado no Capítulo 8, "Valores e referências". O operador *is* é utilizado para determinar se um objeto tem um tipo especificado, funcionando em interfaces e também em classes e estruturas. Por exemplo, o bloco de código a seguir verifica se a variável *myHorse* realmente implementa a interface *ILandBound*, antes de tentar atribuí-la a uma variável *ILandBound*:

```
if (myHorse is ILandBound)
{
    ILandBound iLandBoundAnimal = myHorse;
}
```

Observe que, ao referenciar um objeto por meio de uma interface, você só pode chamar os métodos que são visíveis pela interface.

Trabalhe com várias interfaces

Uma classe pode ter no máximo uma classe base, mas pode implementar um número ilimitado de interfaces. Uma classe deve implementar todos os métodos declarados por essas interfaces.

Se uma estrutura ou classe implementa mais de uma interface, você especifica as interfaces como uma lista separada por vírgulas. Se uma classe também tem uma classe base, as interfaces são listadas *após* a classe base. Por exemplo, suponha que você defina outra interface chamada *IGrazable* que contém o método *ChewGrass* para todos os animais de pasto. Você pode definir a classe *Horse* assim:

```
class Horse : Mammal, ILandBound, IGrazable
{
    ...
}
```

Implemente uma interface explicitamente

Até agora, os exemplos mostraram classes que implementam implicitamente uma interface. Se você examinar outra vez a interface *ILandBound* e a classe *Horse* (mostrada a seguir), perceberá que, embora a classe *Horse* implemente a interface *ILandBound*, não há nada na implementação do método *NumberOfLegs* na classe *Horse* indicando que ela faz parte da interface *ILandBound*.

```
interface ILandBound
{
    int NumberOfLegs();
}

class Horse : ILandBound
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

Isso não seria problemático em um cenário simples, mas vamos supor que a classe *Horse* implementasse várias interfaces. Nada existe para impedir que diversas interfaces especifiquem um método com o mesmo nome, embora possa ter semântica distinta. Por exemplo, vamos supor que você quisesse implementar um sistema de transportes baseado em carroças puxadas a cavalos. Uma jornada longa poderia ser dividida em vários estágios ou “pernas”. Para saber por quantas pernas cada cavalo puxou a carroça, você poderia definir a seguinte interface:

```
interface IJourney
{
    int NumberOfLegs();
}
```

Se você implementar essa interface na classe *Horse*, enfrentará um problema interessante:

```
class Horse : ILandBound, IJourney
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

Este é um código válido, mas o cavalo tem quatro patas ou ele puxou a carroça por quatro pernas do percurso? Pela perspectiva do C#, a resposta é: as duas opções! Por padrão, o C# não distingue qual interface o método está implementando, de modo que o mesmo método implementa as duas interfaces.

Para solucionar esse problema e discernir qual método faz parte de qual implementação de interface, você pode implementar as interfaces explicitamente. Para isso, especifique a qual interface um método pertence, quando você a implementar, como a seguir:

```

class Horse : ILandBound, IJourney
{
    ...
    int ILandBound.NumberOfLegs()
    {
        return 4;
    }

    int IJourney.NumberOfLegs()
    {
        return 3;
    }
}

```

Agora é possível discernir que o cavalo tem quatro patas e puxou a carroça por três pernas da jornada.

Além de prefixar o nome do método com o nome da interface, existe outra diferença sutil nessa sintaxe: os métodos não são marcados como *public*. Não é possível especificar a proteção para os métodos que fazem parte de uma implementação explícita de interface. Isso leva a outro fenômeno interessante. Se você criar a variável *Horse* no código, não poderá chamar qualquer dos dois métodos *NumberOfLegs*, porque não são visíveis. No que diz respeito à classe *Horse*, ambos são privados. Na verdade, isso faz sentido. Se os métodos fossem visíveis por meio da classe *Horse*, qual método o código a seguir chamaría, o da interface *ILandBound* ou o da interface *IJourney*?

```

Horse horse = new Horse();
...
int legs = horse.NumberOfLegs();

```

Como é possível acessar esses métodos? A resposta é: fazendo referência ao objeto *Horse* pela interface adequada, como a seguir:

```

Horse horse = new Horse();
...
I Journey journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs();
ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs();

```

É recomendável implementar interfaces explicitamente, sempre que possível.

Restrições das interfaces

É importante lembrar que uma interface nunca contém qualquer implementação. As restrições a seguir são consequências naturais disso:

- Você não tem permissão para definir campos em uma interface, nem mesmo campos estáticos. Um campo é um detalhe de implementação de uma classe ou estrutura.

- Você não tem permissão para definir um construtor em uma interface. Um construtor também é considerado um detalhe de implementação de uma classe ou estrutura.
- Você não tem permissão para definir destrutor em uma interface. Um destrutor contém as instruções utilizadas para destruir uma instância de objeto. (Os destrutores estão descritos no Capítulo 14, "Coleta de lixo e gerenciamento de recursos".)
- Você não pode especificar um modificador de acesso para qualquer método. Todos os métodos de uma interface são implicitamente públicos.
- Você não pode aninhar tipo algum (como enumerações, estruturas, classes ou interfaces) dentro de uma interface.
- Uma interface não pode ser herdada de uma estrutura nem de uma classe, embora uma interface possa herdar de outra interface. As estruturas e classes contêm implementações; se uma interface tivesse permissão para herdar de qualquer uma das duas, estaria herdando alguma implementação.

Defina e utilize interfaces

Nos exercícios a seguir, você definirá e implementará interfaces que fazem parte de um pacote de desenho gráfico simples. Você definirá duas interfaces chamadas *IDraw* e *IColor*, e então definirá as classes que as implementam. Cada classe determinará uma forma que pode ser desenhada sobre um canvas, em um formulário. (Um *canvas* é um controle que pode ser usado para desenhar linhas, texto e formas na tela.)

A interface *IDraw* define os seguintes métodos:

- *SetLocation* Com esse método é possível especificar a posição como coordenadas x e y da forma sobre o canvas.
- *Draw* Esse método desenha a forma sobre o canvas, no local especificado pelo método *SetLocation*.

A interface *IColor* define o seguinte método:

- *SetColor* Esse método é utilizado para especificar a cor da forma. Quando desenhada sobre o canvas, a forma será exibida com essa cor.

Defina as interfaces *IDraw* e *IColor*

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto Drawing, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 13\Windows X\Drawing na sua pasta Documentos.

O projeto Drawing é um aplicativo gráfico. Ele contém um formulário chamado *DrawingPad*. Esse formulário dispõe de um controle canvas, chamado *drawingCanvas*. Você utilizará esse formulário e o canvas para testar seu código.

3. No Solution Explorer, clique no projeto Drawing. No menu Project, clique em Add New Item.

A caixa de diálogo Add New Item – Drawing se abre.

4. No painel esquerdo da caixa de diálogo Add New Item – Drawing, clique em Visual C# e depois em Code. No painel central, clique no template Interface. Na caixa Name, digite **IDraw.cs** e clique em Add.

O Visual Studio cria o arquivo IDraw.cs e o adiciona a seu projeto. O arquivo IDraw.cs aparece na janela Code and Text Editor e deve ser como segue:

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

namespace Drawing
{
    interface IDraw
    {
    }
}
```

5. No arquivo IDraw.cs, se você estiver usando o Windows 8.1, adicione a seguinte diretiva *using* à lista localizada no início do arquivo:

```
using Windows.UI.Xaml.Controls;
```

Se estiver usando o Windows 7 ou o Windows 8, adicione em vez disso esta diretiva *using*:

```
using System.Windows.Controls;
```

Você fará uma referência à classe *Canvas* nessa interface. A classe *Canvas* está localizada no namespace *Windows.UI.Xaml.Controls* para aplicativos Windows Store e no namespace *System.Windows.Controls* para aplicativos Windows Presentation Foundation (WPF).

6. Adicione os métodos mostrados aqui em negrito à interface *IDraw*:

```
interface IDraw
{
    void SetLocation(int xCoord, int yCoord);
    void Draw(Canvas canvas);
}
```

7. No menu Project, clique em Add New Item novamente.

8. Na caixa de diálogo Add New Item – Drawing, no painel central, clique no template Interface. Na caixa Name, digite **IColor.cs** e clique em Add.

O Visual Studio gera o arquivo IColor.cs e o adiciona a seu projeto. O arquivo IColor.cs aparece na janela Code and Text Editor.

- 9.** No arquivo `IColor.cs`, se você estiver usando o Windows 8.1, adicione a seguinte diretiva *using* à lista no início do arquivo:

```
using Windows.UI;
```

Se estiver usando o Windows 7 ou o Windows 8, adicione esta diretiva *using*:

```
using System.Windows.Media;
```

Nessa interface, você fará referência à classe *Color*, a qual está localizada no namespace *Windows.UI* para aplicativos Windows Store e no namespace *System.Windows.Media* para aplicativos WPF.

- 10.** Adicione o seguinte método mostrado em negrito à definição da interface *IColor*:

```
interface IColor
{
    void SetColor(Color color);
}
```

Agora você definiu as interfaces *IDraw* e *IColor*. A próxima etapa é criar algumas classes que as implementam. No exercício a seguir, você criará duas novas classes de formas, chamadas *Square* e *Circle*. Essas classes implementarão as duas interfaces.

Crie as classes *Square* e *Circle* e implemente as interfaces

- 1.** No menu Project, clique em Add Class.
 - 2.** Na caixa de diálogo Add New Item – Drawing, no painel central, verifique se o template Class está selecionado. Na caixa Name, digite **Square.cs** e clique em Add.
- O Visual Studio gera o arquivo `Square.cs` e o exibe na janela Code and Text Editor.
- 3.** Se você estiver usando o Windows 8.1, adicione as seguintes diretivas *using* à lista no início do arquivo `Square.cs`:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

Se estiver usando o Windows 7 ou o Windows 8, adicione estas diretivas *using* no início do arquivo `Square.cs`:

```
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Controls;
```

4. Modifique a definição da classe *Square* de modo que ela implemente as interfaces *IDraw* e *IColor*, como mostrado aqui em negrito:

```
class Square : IDraw, IColor
{
}
```

5. Adicione as seguintes variáveis privadas, apresentadas em negrito, à classe *Square*.

```
class Square : IDraw, IColor
{
    private int sideLength;
    private int locX = 0, locY = 0;
    private Rectangle rect = null;
}
```

Essas variáveis armazenarão a posição e o tamanho do objeto *Square* sobre o canvas. A classe *Rectangle* está localizada no namespace *Windows.UI.Xaml.Shapes* para aplicativos Windows Store e no namespace *System.Windows.Shapes* para aplicativos WPF. Você utilizará essa classe para desenhar o quadrado:

6. Adicione à classe *Square* o construtor mostrado em negrito a seguir:

```
class Square : IDraw, IColor
{
    ...
    public Square(int sideLength)
    {
        this.sideLength = sideLength;
    }
}
```

Esse construtor inicializa o campo *sideLength* e especifica o comprimento de cada lado do quadrado.

7. Na definição da classe *Square*, clique com o botão direito do mouse na interface *IDraw*. No menu de atalho que aparece, aponte para Implement Interface e clique em Implement Interface Explicitly, como ilustra a imagem a seguir:



Esse recurso instrui o Visual Studio a gerar implementações padrão dos métodos na interface *IDraw*. Se preferir, você também pode adicionar manualmente os métodos à classe *Square*. O exemplo a seguir mostra o código gerado pelo Visual Studio:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    throw new NotImplementedException();
}

void IDraw.Draw(Canvas canvas)
{
    throw new NotImplementedException();
}
```

Cada um desses métodos lança atualmente uma exceção *NotImplementedException*. Você deve substituir o corpo desses métodos pelo seu código.

- 8.** No método *IDraw.SetLocation*, substitua o código existente, que lança uma exceção *NotImplementedException*, pelas instruções mostradas em negrito a seguir:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    this.locX = xCoord;
    this.locY = yCoord;
}
```

Esse código armazena os valores passados pelos parâmetros nos campos *locX* e *locY*, no objeto *Square*.

- 9.** Substitua o código gerado no método *IDraw.Draw* pelas instruções mostradas aqui em negrito:

```
void IDraw.Draw(Canvas canvas)
{
    if (this.rect != null)
    {
        canvas.Children.Remove(this.rect);
    }
    else
    {
        this.rect = new Rectangle();
    }

    this.rect.Height = this.sideLength;
    this.rect.Width = this.sideLength;
    Canvas.SetTop(this.rect, this.locY);
    Canvas.SetLeft(this.rect, this.locX);
    canvas.Children.Add(this.rect);
}
```

Esse método processa o objeto *Square*, desenhando uma forma *Rectangle* no canvas. (Um quadrado é tão somente um retângulo com o mesmo tamanho para os quatro lados.) Se o *Rectangle* já foi desenhado (possivelmente, em outro local e com outra cor), ele será removido do canvas. A altura e largura de *Rectangle* são definidas pelo valor do campo *sideLength*. A posição do *Rectangle*

no canvas é definida por meio dos métodos estáticos *SetTop* e *SetLeft* da classe *Canvas* e, em seguida, o *Rectangle* é adicionado ao canvas. (Isso causa a sua exibição.)

10. Adicione o método *SetColor* da interface *IColor* à classe *Square*, como mostrado a seguir:

```
void IColor.SetColor(Color color)
{
    if (this.rect != null)
    {
        SolidColorBrush brush = new SolidColorBrush(color);
        this.rect.Fill = brush;
    }
}
```

Esse método verifica se o objeto *Square* foi realmente exibido. (O campo *rect* será *null* se ainda não tiver sido processado.) O código define a propriedade *Fill* do campo *rect* com a cor especificada, através do objeto *SolidColorBrush*. (Os detalhes do funcionamento da classe *SolidBrushClass* estão fora dos objetivos desta discussão.)

11. No menu Project, clique em Add Class. Na caixa de diálogo Add New Item – Drawing, na caixa Name, digite **Circle.cs** e clique em Add.

O Visual Studio gera o arquivo Circle.cs e o exibe na janela Code and Text Editor.

12. Se você estiver usando o Windows 8.1, adicione as seguintes diretivas *using* à lista no início do arquivo Circle.cs:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

Se estiver usando o Windows 7 ou o Windows 8, adicione estas diretivas *using* no início do arquivo Circle.cs:

```
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Controls;
```

13. Modifique a definição da classe *Circle* de modo que ela implemente as interfaces *IDraw* e *IColor*, como mostrado aqui em negrito:

```
class Circle : IDraw, IColor
{
}
```

14. Adicione as seguintes variáveis privadas, mostradas em negrito, à classe *Circle*.

```
class Circle : IDraw, IColor
{
    private int diameter;
    private int locX = 0, locY = 0;
    private Ellipse circle = null;
}
```

Essas variáveis armazenarão a posição e o tamanho do objeto *Circle* sobre o canvas. A classe *Ellipse* fornece a funcionalidade que você vai utilizar para desenhar o círculo.

- 15.** Adicione o construtor, mostrado aqui em negrito, à classe *Circle*.

```
class Circle : IDraw, IColor
{
    ...
    public Circle(int diameter)
    {
        this.diameter = diameter;
    }
}
```

Esse construtor inicializa o campo *diameter*.

- 16.** Adicione o seguinte método *SetLocation* à classe *Circle*:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    this.locX = xCoord;
    this.locY = yCoord;
}
```

Esse método implementa parte da interface *IDraw*, e o código é exatamente o mesmo da classe *Square*.

- 17.** Adicione o método *Draw*, mostrado aqui, à classe *Circle*.

```
void IDraw.Draw(Canvas canvas)
{
    if (this.circle != null)
    {
        canvas.Children.Remove(this.circle);
    }
    else
    {

        this.circle = new Ellipse();
    }

    this.circle.Height = this.diameter;
    this.circle.Width = this.diameter;
    Canvas.SetTop(this.circle, this.locY);
    Canvas.SetLeft(this.circle, this.locX);
    canvas.Children.Add(this.circle);
}
```

Esse método também faz parte da interface *IDraw*. Ele é semelhante ao método *Draw* da classe *Square*, exceto pelo fato de que processa o objeto *Circle* ao desenhar uma forma *Ellipse* sobre o canvas. (Um círculo é uma elipse em que a largura e a altura são idênticas.)

- 18.** Adicione o seguinte método *SetColor* à classe *Circle*:

```
void IColor.SetColor(Color color)
{
    if (this.circle != null)
    {
        SolidColorBrush brush = new SolidColorBrush(color);
        this.circle.Fill = brush;
    }
}
```

Esse método faz parte da interface *IColor*. Como antes, esse método é parecido com o da classe *Square*.

Você concluiu as classes *Square* e *Circle*. Agora pode utilizar o formulário para testá-las.

Teste as classes *Square* e *Circle*

1. Exiba o arquivo DrawingPad.xaml na janela Design View.
2. No meio do formulário, clique na área sombreada.
A área sombreada do formulário é o objeto *Canvas*, e essa ação define o foco nesse objeto.
3. Na janela Properties, clique no botão Events Handlers. (Esse botão possui um ícone parecido com um relâmpago.)
4. Se estiver usando o Windows 8.1, na lista de eventos, localize o evento *Tapped* e clique duas vezes nele. Se estiver usando o Windows 7 ou o Windows 8, localize o evento *MouseLeftButtonDown* e clique duas vezes nele.

O Visual Studio gera um método chamado *drawingCanvas_Tapped* (para aplicativos Windows Store) ou *drawingCanvas_MouseLeftButtonDown* (WPF) para a classe *DrawingPadWindow*, e o exibe na janela Code and Text Editor. Essa é uma rotina de tratamento de eventos executada quando o usuário toca no canvas com um dedo (aplicativos Windows Store) ou clica com o botão esquerdo do mouse sobre o canvas (WPF). Você aprenderá mais sobre rotinas de tratamento de evento no Capítulo 20, “Separação da lógica do aplicativo e tratamento de eventos”.



Nota Se estiver usando um mouse no Windows 8.1, você também pode clicar com o botão esquerdo, o qual lança o mesmo evento do gesto de toque.

5. Se você estiver usando o Windows 8.1, adicione a seguinte diretiva *using* à lista no início do arquivo DrawingPad.xaml.cs:

```
using Windows.UI;
```

O namespace *Windows.UI* contém a definição da classe *Colors*, a qual você usará ao definir a cor de uma forma, quando for desenhada. No WPF, essa classe é definida no namespace *System.Windows.Media*, o qual já é referenciado pelo arquivo DrawingPad.xaml.

- 6.** Adicione o seguinte código, mostrado em negrito, ao método *drawingCanvas_Tapped* ou *drawingCanvas_MouseLeftButtonDown*:

```
private void drawingCanvas_Tapped(object sender, TappedRoutedEventArgs e)
// Se você está usando WPF, o método é declarado como:
// private void drawingCanvas_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Point mouseLocation = e.GetPosition(this.drawingCanvas);
    Square mySquare = new Square(100);

    if (mySquare is IDraw)
    {
        IDraw drawSquare = mySquare;
        drawSquare.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
        drawSquare.Draw(drawingCanvas);
    }
}
```

O parâmetro de *TappedRoutedEventArgs* (aplicativos Windows Store) ou *MouseButtonEventArgs* (WPF) para esse método fornece informações úteis sobre a posição do mouse. Mais especificamente, o método *GetPosition* retorna uma estrutura *Point* contendo as coordenadas x e y do mouse. O código que você adicionou gera um novo objeto *Square*. Então, ele verifica se esse objeto implementa a interface *IDraw* (essa é uma boa prática e ajuda a garantir que seu código não falhe em tempo de execução, caso você tente referenciar um objeto por meio de uma interface que ele não implementa) e cria uma referência para o objeto utilizando essa interface. Convém lembrar que, quando você implementa explicitamente uma interface, os métodos definidos por essa interface só estarão disponíveis ao se criar uma referência a essa interface. (Os métodos *SetLocation* e *Draw* são privados para a classe *Square* e estão disponíveis apenas pela interface *IDraw*.) Em seguida, o código define a localização do *Square* com a posição do dedo do usuário ou do mouse. Observe que as coordenadas x e y na estrutura *Point* são valores *double*, de modo que esse código os converte em *ints*. Em seguida, o código chama o método *Draw* para exibir o objeto *Square*.

- 7.** No final do método *drawingCanvas_Tapped* ou *drawingCanvas_MouseLeftButtonDown*, adicione o seguinte código mostrado em negrito:

```
private void drawingCanvas_Tapped(object sender, TappedRoutedEventArgs e)
// Se você está usando WPF, o método é declarado como:
// private void drawingCanvas_MouseLeftButtonDown(object sender,
MouseButtonEventArgs e)
{
    ...
    if (mySquare is IColor)
    {
        IColor colorSquare = mySquare;
        colorSquareSetColor(Colors.BlueViolet);
    }
}
```

Esse código testa a classe *Square* para verificar se ela implementa a interface *IColor*; em caso afirmativo, ele gera uma referência à classe *Square* por meio dessa interface e chama o método *SetColor* para definir a cor do objeto *Square* com *Colors.BlueViolet*. (A classe *Colors* é fornecida como parte do .NET Framework.)



Importante Você deve chamar *Draw* antes de chamar *SetColor*. Isso porque o método *SetColor* só definirá a cor do objeto *Square* se ele já tiver sido desenhado. Se você chamar *SetColor* antes de *Draw*, a cor não será definida e o objeto *Square* não será exibido.

8. Volte para o arquivo DrawingPad.xaml na janela Design View. No meio do formulário, clique no objeto *Canvas*.
9. Se estiver usando o Windows 8.1, na lista de eventos, localize o evento *RightTapped* e clique duas vezes nele. Se estiver usando o Windows 7 ou o Windows 8, localize o evento *MouseRightButtonDown* e clique duas vezes nele.

Esses eventos ocorrem quando o usuário toca no canvas, continua tocando e depois tira o dedo (aplicativos Windows Store) ou clica com o botão direito do mouse sobre o canvas (WPF).



Nota Se estiver usando o Windows 8.1 com um mouse, você pode clicar com o botão direito ou tocar com o dedo, continuar tocando e soltar — ambos os gestos lançam o evento *RightTapped*.

10. Adicione o código a seguir, mostrado em negrito, ao método *drawingCanvas_RightTapped* (aplicativos Windows Store) ou *drawingCanvas_MouseRightButtonDown* (WPF):

```
private void drawingCanvas_RightTapped(object sender, HoldingRoutedEventArgs e)
// Se você está usando WPF, o método é declarado como:
// private void drawingCanvas_MouseRightButtonDown(object sender,
MouseButtonEventArgs e)
{
    Point mouseLocation = e.GetPosition(this.drawingCanvas);
    Circle myCircle = new Circle(100);

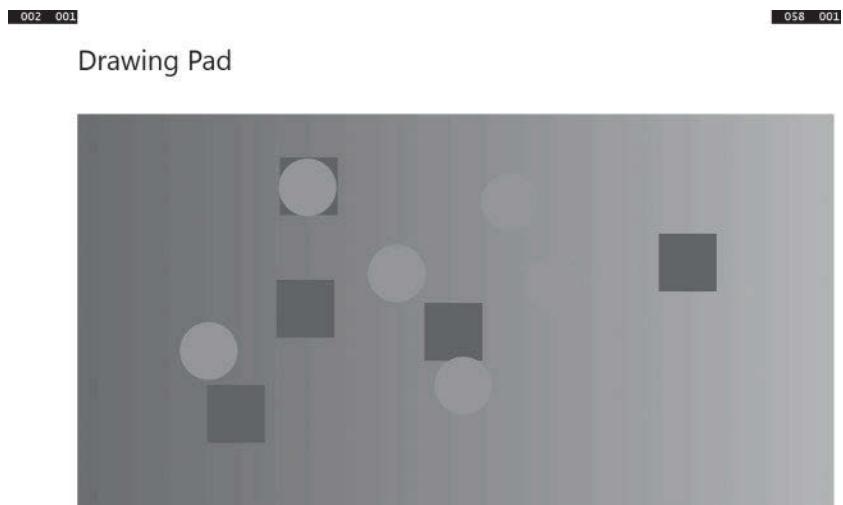
    if (myCircle is IDraw)
    {
        IDraw drawCircle = myCircle;
        drawCircle.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
        drawCircle.Draw(drawingCanvas);
    }

    if (myCircle is IColor)
    {
        IColor colorCircle = myCircle;
        colorCircle.SetColor(Colors.HotPink);
    }
}
```

A lógica nesse código é semelhante à do método que manipula o botão esquerdo do mouse, exceto pelo fato de que exibe um objeto *Circle* em *HotPink*. Ela também é muito parecida com a dos métodos *drawingCanvas_Tapped* e *dra-*

wingCanvas_MouseRightButtonDown, exceto que desenha e preenche um círculo no canvas.

11. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
12. Quando a janela Drawing Pad abrir, toque ou clique em qualquer lugar no canvas exibido nessa janela. Deve aparecer um quadrado violeta.
13. Toque com o dedo, continue tocando e solte ou clique com o botão direito do mouse em qualquer lugar no canvas. Deve ser exibido um círculo rosa. Você pode clicar os botões direito e esquerdo do mouse quantas vezes desejar; cada clique desenhárá um quadrado ou um círculo na posição do mouse. A imagem a seguir mostra o aplicativo em execução no Windows 8.1. A versão para Windows 7 do aplicativo é semelhante:



14. Retorne ao Visual Studio e interrompa a depuração.

Classes abstratas

Você pode implementar as interfaces *ILandBound* e *IGrazable*, discutidas antes do conjunto de exercícios anterior, em várias classes diferentes, dependendo de quantos tipos de mamíferos deseja modelar em seu aplicativo C#. Em situações dessa natureza, é muito comum que as partes das classes derivadas compartilhem implementações em comum. Por exemplo, a duplicação nas duas classes seguintes é óbvia:

```
class Horse : Mammal, ILandBound, IGrazable
{
    ...
    void IGrazable.ChewGrass()
```

```

    {
        Console.WriteLine("Chewing grass");
        // código para pasto
    }
}

class Sheep : Mammal, ILandBound, IGrazable
{
    ...
    void IGrazable.ChewGrass()
    {
        Console.WriteLine("Chewing grass");
        // o mesmo código de cavalo para pasto
    }
}

```

A duplicação no código é um sinal de aviso. Se possível, você deve refatorar o código para evitar essa duplicação e reduzir custos de manutenção associados. Para refatorar, coloque a implementação comum em uma nova classe, criada especificamente para essa finalidade. Na realidade, você pode inserir uma nova classe na hierarquia de classes, como mostrado pelo exemplo de código a seguir:

```

class GrazingMammal : Mammal, IGrazable
{
    ...
    void IGrazable.ChewGrass()
    {
        // código comum para pasto
        Console.WriteLine("Chewing grass");
    }
}

class Horse : GrazingMammal, ILandBound
{
    ...
}

class Sheep : GrazingMammal, ILandBound
{
    ...
}

```

Essa é uma boa solução, mas há algo que ainda não está muito certo: você pode criar instâncias da classe *GrazingMammal* (e também da classe *Mammal*). Isso realmente não faz sentido. A classe *GrazingMammal* existe para fornecer uma implementação padrão comum. Seu único objetivo é ser herdada. Essa classe é uma abstração da funcionalidade comum, em vez de uma entidade por si própria.

Para declarar que a criação de instâncias de uma classe não é permitida, você pode explicitar que a classe é abstrata, utilizando a palavra-chave *abstract*, como no exemplo a seguir:

```

abstract class GrazingMammal : Mammal, IGrazable
{
    ...
}

```

Se agora você tentar instanciar um objeto *GrazingMammal*, o código não compilará:

```
GrazingMammal myGrazingMammal = new GrazingMammal(...); // inválido
```

Métodos abstratos

Uma classe abstrata pode conter métodos abstratos. Um método abstrato é semelhante em princípio a um método virtual (o qual foi abordado no Capítulo 12), exceto por não conter um corpo de método. Uma classe derivada *precisa* redefinir esse método. O exemplo a seguir define o método *DigestGrass* na classe *GrazingMammal* como um método abstrato; mamíferos de pasto poderiam utilizar o mesmo código para pastar, mas eles devem fornecer uma implementação própria do método *DigestGrass*. Um método abstrato é útil se não fizer sentido fornecer uma implementação padrão na classe abstrata, mas se você quiser assegurar que uma classe que herda forneça uma implementação própria desse método.

```
abstract class GrazingMammal : Mammal, IGrazable
{
    abstract void DigestGrass();
    ...
}
```

Classes seladas

Utilizar herança nem sempre é fácil e exige prudência. Se criar uma interface ou uma classe abstrata, você estará intencionalmente escrevendo algo que será herdado no futuro. O problema é que prever o futuro é difícil. Com prática e experiência, você pode desenvolver habilidades para produzir uma hierarquia fácil de usar e flexível em termos de interfaces, classes abstratas e classes, mas isso exige esforço e também é necessário um entendimento sólido do problema que está sendo modelado. Ou seja, a menos que você projete conscientemente uma classe com a intenção de utilizá-la como uma classe base, é muito pouco provável que ela funcione bem como uma classe base. No C#, se quiser, você pode utilizar a palavra-chave *sealed* para impedir que uma classe seja utilizada como uma classe base. Por exemplo:

```
sealed class Horse : GrazingMammal, ILandBound
{
    ...
}
```

Se alguma classe tentar utilizar *Horse* como sua classe base, um erro de tempo de compilação será gerado. Observe que uma classe selada não pode declarar método virtual algum e que uma classe abstrata não pode ser selada.

Métodos selados

Você também pode utilizar a palavra-chave *sealed* para declarar que um método individual em uma classe não selada está selado. Isso significa que uma classe derivada não poderá redefinir esse método. Você só pode selar um método *override*, e esse método é declarado como *sealed override*. Considere as palavras-chave *interface*, *virtual*, *override* e *sealed*, como descrito a seguir:

- Uma interface introduz o nome de um método.
- Um método virtual é a primeira implementação de um método.
- Um método override é outra implementação de um método.
- Um método sealed é a última implementação de um método.

Implemente e utilize uma classe abstrata

Os exercícios a seguir utilizam uma classe abstrata para racionalizar uma parte do código que você desenvolveu no exercício anterior. As classes *Square* e *Circle* contêm uma alta proporção de código duplicado. Compensa fatorar esse código em uma classe abstrata, chamada *DrawingShape*, porque isso ajudará a facilitar a manutenção das classes *Square* e *Circle* mais adiante.

Crie a classe abstrata *DrawingShape*

1. Retorne ao projeto Drawing no Visual Studio.



Nota Uma cópia operacional finalizada do exercício anterior está disponível no projeto Drawing, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 13\Windows X\Drawing Using Interfaces em sua pasta Documentos.

2. No Solution Explorer, clique no projeto Drawing na solução Drawing. No menu Project, clique em Add Class.

A caixa de diálogo Add New Item – Drawing se abre.

3. Na caixa Name, digite **DrawingShape.cs** e clique em Add.

O Visual Studio gera a classe e a exibe na janela Code and Text Editor.

4. No arquivo DrawingShape.cs, se você estiver usando o Windows 8.1, adicione as seguintes diretivas *using* à lista no início do arquivo:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

Se estiver usando o Windows 7 ou o Windows 8, adicione estas diretivas *using*:

```
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Controls;
```

O objetivo dessa classe é conter o código comum às classes *Circle* e *Square*. Um programa não poderá instanciar diretamente um objeto *DrawingShape*.

- 5.** Modifique a definição da classe *DrawingShape* e declare-a como *abstract*, como mostrado aqui em negrito:

```
abstract class DrawingShape
{
}
```

- 6.** Adicione as seguintes variáveis privadas, mostradas em negrito, à classe *DrawingShape*.

```
abstract class DrawingShape
{
    protected int size;
    protected int locX = 0, locY = 0;
    protected Shape shape = null;
}
```

As classes *Square* e *Circle* utilizam os campos *locX* e *locY* para especificar a localização do objeto sobre o canvas, para que você possa mover esses campos para a classe abstrata. De modo semelhante, as classes *Square* e *Circle* usam um campo para indicar o tamanho do objeto quando foi processado; embora tenha outro nome em cada classe (*sideLength* e *diameter*), semanticamente o campo executa a mesma tarefa nas duas classes. O nome *size* é uma abstração eficiente do objetivo desse campo.

Internamente, a classe *Square* usa um objeto *Rectangle* para se desenhar sobre o canvas, e a classe *Circle* utiliza um objeto *Ellipse*. Essas duas classes fazem parte de uma hierarquia baseada na classe abstrata *Shape* do .NET Framework. A classe *DrawingShape* emprega um campo *Shape* para representar esses dois tipos.

- 7.** Adicione o seguinte construtor à classe *DrawingShape*:

```
abstract class DrawingShape
{
    ...
    public DrawingShape(int size)
    {
        this.size = size;
    }
}
```

Esse código inicializa o campo *size* do objeto *DrawingShape*.

- 8.** Adicione os métodos *SetLocation* e *SetColor* à classe *DrawingShape*, como mostrado em negrito no código a seguir. Esses métodos fornecem as implementações herdadas por todas as classes derivadas da classe *DrawingShape*. Observe que eles não estão marcados como *virtual*, e não se espera que uma classe derivada os substitua. Além disso, a classe *DrawingShape* não está declarada como se implementasse as interfaces *IDraw* ou *IColor* (implementação de interfaces é um recurso das classes *Square* e *Circle* e não dessa classe abstrata), de modo que esses métodos são apenas declarados como *public*.

```
abstract class DrawingShape
{
    ...
    public void SetLocation(int xCoord, int yCoord)
}
```

```

{
    this.locX = xCoord;
    this.locY = yCoord;
}

public void SetColor(Color color)
{
    if (this.shape != null)
    {
        SolidColorBrush brush = new SolidColorBrush(color);
        this.shape.Fill = brush;
    }
}
}

```

9. Adicione o método *Draw* à classe *DrawingShape*. Diferentemente dos métodos anteriores, esse método é declarado como *virtual*, e espera-se que as classes derivadas o anulem para estender a funcionalidade. O código contido nesse método verifica se o campo *shape* não é nulo e depois o desenha no canvas. As classes que herdam esse método devem fornecer um código próprio para instanciar o objeto *shape*. (Lembre-se de que a classe *Square* cria um objeto *Rectangle* e a classe *Circle* gera um objeto *Ellipse*.)

```

abstract class DrawingShape
{
    ...
    public virtual void Draw(Canvas canvas)
    {
        if (this.shape == null)
        {
            throw new InvalidOperationException("Shape is null");
        }

        this.shape.Height = this.size;
        this.shape.Width = this.size;
        Canvas.SetTop(this.shape, this.locY);
        Canvas.SetLeft(this.shape, this.locX);
        canvas.Children.Add(this.shape);
    }
}

```

Você finalizou a classe abstrata *DrawingShape*. O próximo passo é mudar as classes *Square* e *Circle* para que herdem dessa classe e remover o código duplicado das classes *Square* e *Circle*.

Modifique as classes *Square* e *Circle* para herdar da classe *DrawingShape*

1. Exiba o código da classe *Square* na janela Code and Text Editor.
2. Modifique a definição da classe *Square* para que ela herde da classe *DrawingShape*, além de implementar as interfaces *IDraw* e *IColor*.

```

class Square : DrawingShape, IDraw, IColor
{
    ...
}

```

Observe que você deve especificar a classe da qual a classe *Square* herda antes de qualquer interface que ela implementa.

3. Na classe *Square*, remova as definições dos campos *sideLength*, *rect*, *locX* e *locY*. Esses campos não são mais necessários, pois agora são fornecidos pela classe *DrawingShape*.
4. Substitua o construtor existente pelo código a seguir, que chama o construtor da classe base:

```
class Square : DrawingShape, IDraw, IColor
{
    public Square(int sideLength)
        : base(sideLength)
    {
    }
    ...
}
```

Observe que o corpo desse construtor está vazio porque o construtor da classe base se encarrega de toda a inicialização necessária.

5. Remova os métodos *IDraw.SetLocation* e *IColor.SetColor* da classe *Square*. A classe *DrawingShape* fornece a implementação desses métodos.
6. Modifique a definição do método *Draw*. Declare-o como *public override* e remova a referência à interface *IDraw*. Mais uma vez, a classe *DrawingShape* já disponibiliza a funcionalidade básica para esse método, mas você a estenderá com um código específico, necessário à classe *Square*.

```
public override void Draw(Canvas canvas)
{
    ...
}
```

7. Substitua o corpo do método *Draw* pelo código mostrado em negrito:

```
public override void Draw(Canvas canvas)
{
    if (this.shape != null)
    {
        canvas.Children.Remove(this.shape);
    }
    else
    {
        this.shape = new Rectangle();
    }

    base.Draw(canvas);
}
```

Essas instruções instanciam o campo *shape* herdado da classe *DrawingShape* como uma nova instância da classe *Rectangle*, se ela ainda não foi instanciada, e depois chamam o método *Draw* da classe *DrawingShape*.

8. Repita os passos 2 a 6 para a classe *Circle*, exceto pelo fato de que o construtor deve ser chamado de *Circle* com um parâmetro chamado *diameter*, e no método *Draw* você deve instanciar o campo *shape* como um novo objeto *Ellipse*. O código completo para a classe *Circle* deve ficar parecido com o seguinte:

```
class Circle : DrawingShape, IDraw, IColor
{
    public Circle(int diameter)
        : base(diameter)
    {
    }

    public override void Draw(Canvas canvas)
    {
        if (this.shape != null)
        {
            canvas.Children.Remove(this.shape);
        }
        else
        {
            this.shape = new Ellipse();
        }

        base.Draw(canvas);
    }
}
```

9. No menu Debug, clique em Start Debugging. Quando a janela Drawing Pad abrir, verifique que os objetos *Square* aparecem quando você clica com o botão esquerdo do mouse na janela, e os objetos *Circle* são exibidos quando clica com o botão direito do mouse na janela. O aplicativo deve se comportar exatamente como antes.
10. Retorne ao Visual Studio e interrompa a depuração.

Compatibilidade com o Windows Runtime no Windows 8 e no Windows 8.1 revisitada

O Capítulo 9, “Como criar tipos-valor com enumerações e estruturas”, descreveu como o Windows 8 e o Windows 8.1 implementam o Windows Runtime (WinRT), como uma camada sobre as APIs nativas do Windows, fornecendo uma interface de programação simplificada para os desenvolvedores compilarem aplicativos não gerenciados (um aplicativo não gerenciado não é executado com o .NET Framework; você os compila utilizando uma linguagem como C++, em vez de C#). Os aplicativos gerenciados utilizam o Common Language Runtime (CLR) para executar aplicativos .NET Framework. O .NET Framework fornece um amplo conjunto de bibliotecas e recursos. No Windows 7 e anteriores, o CLR implementa esses recursos utilizando as APIs nativas do Windows. Caso você esteja compilando aplicativos e serviços de desktop ou empresariais no Windows 8 e no Windows 8.1, esse mesmo conjunto de recursos ainda está disponível (embora o .NET Framework tenha sido atualizado para a versão 4.5), e qualquer aplicativo C# que funcione no Windows 7 deve ser executado sem alteração no Windows 8 e no Windows 8.1.

No Windows 8 e no Windows 8.1, os aplicativos Windows Store sempre são executados com o WinRT. Isso significa que, se você está compilando aplicativos Windows Store com uma linguagem gerenciada, como o C#, o CLR chama na verdade o WinRT, em vez das APIs nativas do Windows. A Microsoft forneceu uma camada de mapeamento entre o CLR e o WinRT que pode traduzir de forma transparente os pedidos feitos para o .NET Framework, a fim de criar objetos e chamar métodos nos pedidos de método e chamadas de método equivalentes no WinRT. Por exemplo, quando você cria um valor *Int32* no .NET Framework (um *int* no C#), esse código é traduzido de forma a criar um valor utilizando o tipo de dado WinRT equivalente. Contudo, embora o CLR e o WinRT tenham uma grande quantidade de funcionalidade coincidente, nem todos os recursos do .NET Framework 4.5 têm recursos correspondentes no WinRT. Consequentemente, os aplicativos Windows Store têm acesso apenas a um subconjunto reduzido dos tipos e métodos fornecidos pelo .NET Framework 4.5 (o IntelliSense no Visual Studio 2013 mostra automaticamente o modo de exibição restrito dos recursos disponíveis, quando você está compilando aplicativos Windows Store com C#, omitindo os tipos e métodos não disponíveis através do WinRT).

Por outro lado, o WinRT fornece um expressivo conjunto de recursos e tipos que não têm equivalente direto no .NET Framework ou que funcionam de maneira significativamente diferente dos recursos correspondentes no .NET Framework e, portanto, não podem ser traduzidos facilmente. O WinRT disponibiliza esses recursos para o CLR por meio de uma camada de mapeamento que os faz ser parecidos com os tipos e métodos do .NET Framework, e você pode chamá-los diretamente a partir de código gerenciado. Para aplicativos Windows Store, a principal área de preocupação é a maneira pela qual a interface do usuário é implementada, e é por isso que alguns dos exercícios deste livro pedem para que você faça referência a namespaces diferentes para aplicativos de desktop com janelas em execução no Windows 7 e no Windows 8, e para aplicativos Windows Store em execução no Windows 8.1 – *System.Windows* e seus subnamespaces para aplicativos de área de trabalho *versus* *Windows.UI* e seus subnamespaces para aplicativos Windows Store. Esses namespaces contêm tipos que são implementados por diferentes assemblies; os tipos do namespace *System.Windows* residem nos assemblies *WindowsBase*, *PresentationCore* e *PresentationFramework* para o .NET Framework 4.5, enquanto os tipos do namespace *Windows.UI* estão localizados no assembly *Windows* para WinRT.



Nota Rigorosamente falando, o WinRT não utiliza assemblies, mas tem sua estrutura própria para armazenar bibliotecas de código executável. Mas as bibliotecas WinRT expõem metadados armazenados no mesmo formato dos assemblies .NET Framework; portanto, podem ser lidos pelo CLR. Para o CLR, a aparência e o comportamento das bibliotecas WinRT são iguais aos dos assemblies .NET Framework. O CLR pode criar objetos definidos nessas bibliotecas e chamar seus métodos. A diferença é que o WinRT cria e gerencia esses objetos, mas essa diferença não é visível para seu código de aplicativo executado com o CLR.

Assim, a integração implementada pelo CLR e pelo WinRT permite que o CLR utilize tipos WinRT de forma transparente, mas também suporta interoperabilidade na direção inversa: você pode definir tipos utilizando código gerenciado e torná-los disponíveis para aplicativos não gerenciados, desde que esses tipos estejam de acordo com as expectativas do WinRT. O Capítulo 9 destaca os requisitos das estruturas a esse respeito (métodos de instância e estáticos em estruturas não estão disponíveis por meio do WinRT, e campos privados não são suportados). Se você estiver compilando classes com a intenção de que sejam utilizadas por aplicativos não gerenciados por meio de WinRT, suas classes devem seguir estas regras:

- Os campos públicos e os parâmetros e valores de retorno de todo método público devem ser tipos WinRT ou tipos .NET Framework que possam ser traduzidos de forma transparente para tipos WinRT pelo WinRT. Exemplos de tipos .NET Framework suportados incluem adequar tipos-valor (como estruturas e enumerações) e aqueles correspondentes às primitivas do C# (*int*, *long*, *float*, *double*, *string* e assim por diante). Campos privados são suportados em classes e podem ser de qualquer tipo disponível no .NET Framework; eles não precisam se adequar ao WinRT.
- Fora *ToString*, as classes não podem redefinir métodos de *System.Object* e não podem declarar construtores protegidos.
- O namespace no qual uma classe é definida deve ter o mesmo nome do assembly que implementa a classe. Além disso, o nome do namespace (e, portanto, o nome do assembly) não deve começar com "Windows".
- Você não pode herdar de tipos gerenciados em aplicativos não gerenciados por meio do WinRT. Portanto, todas as classes públicas devem ser seladas. Caso precise implementar polimorfismo, você pode criar uma interface pública e implementar essa interface nas classes que precisam ser polimórficas.
- Você pode lançar qualquer tipo de exceção que esteja incluída no subconjunto do .NET Framework disponível para aplicativos Windows Store; você não pode criar suas próprias classes de exceção personalizadas. Se seu código lançar uma exceção não tratada quando for chamado a partir de um aplicativo não gerenciado, o WinRT lançará uma exceção equivalente no código não gerenciado.

O WinRT tem outros requisitos a respeito dos recursos de código C#, abordados posteriormente neste livro. Esses requisitos serão destacados à medida que cada recurso for descrito.

Resumo

Neste capítulo, vimos como definir e implementar interfaces e classes abstratas. A tabela a seguir resume as diversas combinações de palavras-chave válidas (sim), inválidas (não) e obrigatórias (exigida) ao se definir métodos para interfaces, classes e estruturas.

Palavra-chave	Interface	Classe abstrata	Classe	Classe selada	Estrutura
abstract	Não	Sim	Não	Não	Não
new	Sim ¹	Sim	Sim	Sim	Não ²
override	Não	Sim	Sim	Sim	Não ³
private	Não	Sim	Sim	Sim	Sim
protected	Não	Sim	Sim	Sim	Não ⁴
public	Não	Sim	Sim	Sim	Sim
sealed	Não	Sim	Sim	Exigida	Não
virtual	Não	Sim	Sim	Não	Não

¹ Uma interface pode estender outra interface e introduzir um novo método com a mesma assinatura.

² As estruturas não aceitam herança, de modo que não podem ocultar métodos.

³ As estruturas não aceitam herança, de modo que não podem redefinir métodos.

⁴ As estruturas não aceitam herança; uma estrutura é implicitamente selada e não pode ser derivada.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 14.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar uma interface	Utilize a palavra-chave interface. Por exemplo: <pre>interface IDemo { string GetName(); string GetDescription(); }</pre>
Implementar uma interface	Declare uma classe utilizando a mesma sintaxe da herança de classes e então implemente todas as funções-membro da interface. Por exemplo: <pre>class Test : IDemo { public string IDemo.GetName() { ... } public string IDemo.GetDescription() { ... } }</pre>

Para	Faça isto
Criar uma classe abstrata que só possa ser utilizada como uma classe base, contendo métodos abstratos	Declare a classe utilizando a palavra-chave abstract. Para cada método abstrato, declare o método com a palavra-chave abstract e sem um corpo de método. Por exemplo: <pre>abstract class GrazingMammal { abstract void DigestGrass(); ... }</pre>
Criar uma classe selada que não possa ser utilizada como uma classe base	Declare a classe utilizando a palavra-chave sealed. Por exemplo: <pre>sealed class Horse { ... }</pre>

CAPÍTULO 14

Coleta de lixo e gerenciamento de recursos

Neste capítulo, você vai aprender a:

- Gerenciar os recursos do sistema utilizando a coleta de lixo.
- Escrever código que executa quando um objeto é finalizado usando um destrutor.
- Liberar um recurso em determinado momento e de modo seguro quanto a exceções escrevendo uma instrução *try/finally*.
- Liberar um recurso em determinado momento e de modo seguro quanto a exceções escrevendo uma instrução *using*.
- Implementar a interface *IDisposable* para dar suporte ao descarte seguro quanto a exceções em uma classe.

Nos capítulos anteriores, você aprendeu a criar variáveis e objetos, e agora provavelmente já entende como a memória é alocada quando variáveis e objetos são criados. (Caso você tenha esquecido, os tipos-valor são criados na pilha e os tipos-referência são memória alocada a partir do heap.) Como os computadores não possuem memória infinita, é preciso que ela seja recuperada quando uma variável ou objeto não necessitar mais dela. Os tipos-valor são destruídos e sua memória é reivindicada quando eles saem de escopo. Essa é a parte fácil. Mas e os tipos-referência? Você cria um objeto utilizando a palavra-chave *new*, mas como e quando se dá a destruição de um objeto? Falaremos disso neste capítulo.

O tempo de vida de um objeto

Em primeiro lugar, vamos recapitular o que acontece quando você cria um objeto.

Um objeto é criado com o operador *new*. O exemplo a seguir cria uma nova instância da classe *Square* que foi discutida no Capítulo 13, “Como criar interfaces e definir classes abstratas”.

```
Square mySquare = new Square(); // Square is a reference type
```

Do seu ponto de vista, a operação *new* é apenas uma, mas, por baixo do pano, a criação do objeto é, na verdade, um processo de duas fases:

1. A operação *new* aloca uma parte da memória *bruta* a partir do heap. Você não tem controle algum sobre essa fase da criação de um objeto.

2. A operação *new* converte a parte da memória bruta em um objeto; ela tem de inicializar o objeto. Você pode controlar essa fase utilizando um construtor.



Nota Se você é programador de C++, deve notar que no C# não é possível sobrepor a operação *new* para controlar a alocação.

Depois de criar um objeto, você pode acessar seus membros utilizando o operador `(.)`. Por exemplo, a classe *Square* inclui um método chamado *Draw* que pode ser chamado:

```
mySquare.Draw();
```



Nota Esse código é baseado na versão da classe *Square* que herda da classe abstrata *DrawingShape* e que não implementa explicitamente a interface *IDraw*. Para mais informações, consulte o Capítulo 13.

Quando a variável *mySquare* sai de escopo, o objeto *Square* não está mais sendo referenciado ativamente; o objeto pode ser destruído e a memória que ele está utilizando pode ser reivindicada (contudo, talvez isso não aconteça imediatamente, conforme você vai ver mais adiante). Assim como a criação de objetos, sua destruição é um processo de duas fases. As duas fases da destruição espelham exatamente as duas fases de criação:

1. O Common Language Runtime (CLR) precisa organizar as coisas. Você pode controlar isso escrevendo um *destrutor*.
2. O CLR precisa retornar para o heap a memória que anteriormente pertencia ao objeto; a memória em que o objeto residia precisa ser desalocada. Você não tem controle algum sobre essa fase.

O processo de destruição de um objeto e devolução da memória para o heap é conhecido como *coleta de lixo*.



Nota Se você é programador de C++, deve lembrar que o C# não tem um operador *delete*. O CLR controla quando um objeto é destruído.

Escreva destrutores

Você pode utilizar um destrutor para executar qualquer limpeza necessária quando um objeto vai para a coleta de lixo. O CLR limpará automaticamente os recursos gerenciados utilizados por um objeto e, em muitos desses casos, é desnecessário escrever um destrutor. Contudo, se um recurso gerenciado é grande (como um array multidimensional), talvez faça sentido torná-lo disponível para descarte imediato, definindo como *null* qualquer referência que o objeto tenha a ele. Além disso, se um objeto faz referência, direta ou indiretamente, a um recurso não gerenciado, um destrutor pode se mostrar útil.



Nota Recursos não gerenciados indiretos são razoavelmente comuns. Exemplos incluem fluxos de arquivo, conexões de rede, conexões de banco de dados e outros recursos gerenciados pelo sistema operacional Microsoft Windows. Assim, se você abrir um arquivo em um método, talvez queira adicionar um destrutor que o feche quando o objeto for destruído. Mas pode haver uma maneira melhor e mais oportuna de fechar o arquivo, dependendo da estrutura do código presente em sua classe (consulte a discussão sobre a instrução *using*, posteriormente neste capítulo).

Um destrutor é um método especial, parecido com um construtor, exceto pelo fato de o CLR o chamar depois da referência a um objeto desaparecer. A sintaxe para escrever um destrutor é um til (~), seguido pelo nome da classe. Por exemplo, aqui está uma classe simples que abre um arquivo para leitura em seu construtor e o fecha em seu destrutor (observe que se trata simplesmente de um exemplo e não é recomendado seguir sempre esse padrão para abrir e fechar arquivos):

```
class FileProcessor
{
    FileStream file = null;

    public FileProcessor(string fileName)
    {
        this.file = File.OpenRead(fileName); // abre o arquivo para leitura
    }

    ~FileProcessor()
    {
        this.file.Close(); // fecha o arquivo
    }
}
```

Há algumas restrições importantes que dizem respeito aos destrutores:

- Os destrutores se aplicam somente a tipos-referência; você não pode declarar um destrutor em um tipo-valor, como um *struct*.

```
struct MyStruct
{
    ~ MyStruct() { ... } // erro de tempo de compilação
}
```

- Você não pode especificar um modificador de acesso (como *public*) para um destrutor. Você nunca chama o destrutor no seu próprio código; uma parte do CLR, chamada *coletor de lixo*, faz isso para você.

```
public ~ FileProcessor() { ... } // erro de tempo de compilação
```

- Um destrutor não pode aceitar qualquer parâmetro. Novamente, isso ocorre porque você nunca chama o destrutor por conta própria.

```
~ FileProcessor(int parameter) { ... } // erro de tempo de compilação
```

Internamente, o compilador C# converte automaticamente um destrutor em uma redefinição do método *Object.Finalize*. O compilador converte este destrutor:

```
class FileProcessor
{
    ~FileProcessor() { // Seu código entra aqui }
}
```

nisto:

```
class FileProcessor
{
    protected override void Finalize()
    {
        try { // Seu código entra aqui }
        finally { base.Finalize(); }
    }
}
```

O método *Finalize* gerado pelo compilador contém o corpo do destrutor dentro de um bloco *try*, seguido por um bloco *finally* que chama o método *Finalize* da classe base. (As palavras-chave *try* e *finally* foram descritas no Capítulo 6, “Gerenciamento de erros e exceções”.) Isso garante que um destrutor sempre chamará o destrutor da sua classe base, mesmo que ocorra uma exceção durante seu código de destrutor.

É importante entender que apenas o compilador pode fazer essa conversão. Você não pode escrever seu próprio método para substituir *Finalize*, nem pode chamar *Finalize* por conta própria.

Por que utilizar o coletor de lixo?

Você nunca pode destruir um objeto utilizando código C#. Não há uma sintaxe que faça isso. É o CLR que faz isso para você, em um momento de escolha própria. Além disso, lembre-se de que você também pode fazer mais de uma variável de referência referenciar o mesmo objeto. No exemplo de código a seguir, as variáveis *myFp* e *referenceToMyFp* apontam para o mesmo objeto *FileProcessor*:

```
FileProcessor myFp = new FileProcessor();
FileProcessor referenceToMyFp = myFp;
```

Quantas referências a um objeto você pode criar? Quantas quiser! Isso tem um impacto sobre o tempo de vida de um objeto. O CLR tem de manter o controle de todas essas referências. Se a variável *myFp* desaparecer (saindo do escopo), outras variáveis (como *referenceToMyFp*) podem ainda existir e os recursos utilizados pelo objeto *FileProcessor* não poderão ser reivindicados (o arquivo não deve ser fechado). Assim, o tempo de vida de um objeto não pode estar vinculado a determinada variável de referência. Um objeto só poderá ser destruído e sua memória se tornar disponível para reutilização quando *todas* as referências a ele desaparecerem.

Dá para ver que o gerenciamento da vida dos objetos é complexo, sendo esse o motivo pelo qual os projetistas do C# optaram por impedir que seu código assuma essa responsabilidade. Se fosse sua responsabilidade destruir os objetos, mais cedo ou mais tarde uma das seguintes situações poderia ocorrer:

- Você poderia esquecer-se de destruir o objeto. Isso significa que o destrutor do objeto (se houver) não seria executado, a limpeza não ocorreria e a memória não seria retornada para o heap. Você poderia esgotar a memória.
- Você poderia tentar destruir um objeto ativo e correria o risco de ter uma ou mais variáveis contendo uma referência a um objeto destruído, o que é conhecido como *referência oscilante*. Uma referência oscilante referencia uma memória não utilizada ou talvez um objeto completamente diferente que agora ocupa a mesma parte da memória. De uma maneira ou de outra, o resultado do uso de uma referência variável seria indefinido ou, pior, seria um risco de segurança. Tudo seria possível.
- Você poderia tentar destruir o mesmo objeto mais de uma vez. Isso poderia ou não ser desastroso, dependendo do código do destrutor.

Esses problemas são inaceitáveis em uma linguagem como o C#, que coloca a robustez e a segurança no alto de sua lista de objetivos de projeto. Em vez disso, o coletor de lixo destrói os objetos para você. Ele garante o seguinte:

- Todo objeto será destruído e seu destrutor será executado. Quando um programa terminar, todos os objetos existentes serão destruídos.
- Cada objeto será destruído apenas uma vez.
- Cada objeto será destruído somente quando se tornar inacessível – isto é, quando não houver qualquer referência ao objeto no processo de execução de seu aplicativo.

Essas garantias são muito úteis e liberam o programador das enfadonhas tarefas de limpeza, que são passíveis de erro. Elas permitem que você se concentre na lógica do programa e seja mais produtivo.

Quando ocorre a coleta de lixo? Essa pergunta pode parecer estranha. Afinal de contas, a coleta de lixo ocorre quando um objeto não é mais necessário. É isso mesmo, mas não necessariamente de imediato. A coleta de lixo pode ser um processo caro; portanto, o CLR só coleta o lixo quando há necessidade (quando a memória disponível está diminuindo ou o tamanho do heap ultrapassa o limite definido pelo sistema, por exemplo) e, então, coleta o máximo possível. É melhor fazer algumas limpezas grandes do que muitas limpezas pequenas.



Nota Você pode ativar o coletor de lixo em um programa chamando o método estático *Collect* da classe *GC*, localizada no namespace *System*. Mas, exceto em alguns casos, isso não é recomendável. O método *GC.Collect* inicia o coletor de lixo, mas o processo executa de modo assíncrono – o método *GC.Collect* não aguarda o término da coleta de lixo antes de seu retorno, de modo que você ainda não sabe se seus objetos foram destruídos. Deixe o CLR decidir qual o melhor momento para coletar o lixo.

Outra característica do coletor de lixo é que você não sabe a ordem em que os objetos serão destruídos. A questão final a discutir talvez seja a mais importante: os destrutores não são executados até que os objetos sofram coleta de lixo. Se você escrever um destrutor, sabe que ele será executado, mas simplesmente não sabe quan-

do. Assim, você nunca deve escrever um código que dependa dos destrutores em execução em uma sequência específica ou em um ponto específico em seu aplicativo.

Como funciona o coletor de lixo?

O coletor de lixo executa em sua própria thread e só pode executar em certas ocasiões – em geral, quando o aplicativo chega ao final de um método. Enquanto ele executa, outras threads em execução no seu aplicativo são temporariamente suspensas. Isso acontece porque o coletor de lixo talvez precise mover os objetos e atualizar as referências de objeto, e ele não pode fazer isso enquanto os objetos estão em uso.



Nota Uma *thread* é um caminho de execução separado em um aplicativo. O Windows utiliza threads para facilitar a execução de várias operações simultâneas por um aplicativo.

O coletor de lixo é um software complexo e autoajustável, e implementa diversas otimizações para tentar equilibrar a necessidade de manter memória disponível perante o requisito de manter o desempenho do aplicativo. Os detalhes dos algoritmos e estruturas internas utilizados pelo coletor de lixo estão fora dos objetivos deste livro (e a Microsoft refina continuamente o modo como o coletor de lixo realiza seu trabalho), mas, em alto nível, o que ele faz é o seguinte:

1. Constrói um mapa de todos os objetos acessíveis. Ele faz isso seguindo repetidamente os campos de referência dentro dos objetos. Esse mapa é construído cuidadosamente, certificando-se de que referências circulares não causem uma recursão infinita. Qualquer objeto que *não* esteja nesse mapa é considerado inacessível.
2. Verifica se algum dos objetos inacessíveis tem um destrutor que precisa ser executado (um processo chamado *finalização*). Todo objeto inacessível que exija finalização é colocado em uma fila especial chamada *fila freachable* (pronuncia-se efe-rítchból).
3. Desaloca os objetos inacessíveis restantes (aqueles que não exigem finalização), movendo os objetos *acessíveis* para a parte inferior do heap, desfragmentando assim o heap e liberando a memória em sua parte superior. Quando o coletor de lixo move um objeto acessível, ele também atualiza todas as referências ao objeto.
4. Nesse ponto, ele permite que outras threads se reiniciem.
5. Finaliza os objetos inacessíveis que exigem finalização (agora na fila *freachable*), executando os métodos *Finalize* em sua própria thread.

Recomendações

Escrever classes que contenham destrutores aumenta a complexidade do seu código e do processo de coleta de lixo, e faz seu programa executar com mais lentidão. Se o programa não contiver destrutor algum, o coletor de lixo não precisará posicionar objetos inacessíveis na fila *freachable* e finalizá-los. Evidentemente, não fazer coisa alguma é mais rápido do que fazer. Portanto, tente evitar o uso de destrutores, exce-

to quando você realmente precisar deles; utilize-os apenas para reivindicar recursos não gerenciados. Por exemplo, considere a possibilidade de雇用 uma instrução *using*, conforme será descrito mais adiante neste capítulo.

Você precisa ter bastante cuidado ao escrever um destrutor. Em particular, esteja ciente de que, se seu destrutor chamar outros objetos, é possível que o coletor de lixo já tenha chamado os destrutores desses outros objetos. Lembre-se de que a ordem da finalização não é garantida. Portanto, certifique-se de que os destrutores não dependam um do outro nem se sobreponham – evite que dois destrutores tentem liberar o mesmo recurso, por exemplo.

Gerenciamento de recursos

Às vezes, é desaconselhável liberar um recurso em um destrutor; alguns recursos são simplesmente muito valiosos para que permaneçam ociosos por um período de tempo arbitrário até que o coletor de lixo realmente os libere. Os recursos escassos, como memória, conexões de banco de dados ou handles de arquivo, precisam ser liberados, e isso precisa ser feito o mais cedo possível. Nessas situações, a única opção é você mesmo liberar o recurso. Você pode fazer isso criando um *método de descarte*. Um método de descarte descarta um recurso. Se uma classe tem um método de descarte, você pode chamá-lo e controlar quando o recurso será liberado.



Nota O termo *método de descarte* refere-se ao propósito do método e não ao seu nome. Um método de descarte pode ser nomeado utilizando qualquer identificador C# válido.

Métodos de descarte

Um exemplo de classe que implementa um método de descarte é a *TextReader* do namespace *System.IO*. Essa classe fornece um mecanismo para ler caracteres em um fluxo de entrada sequencial. A classe *TextReader* contém um método virtual chamado *Close*, o qual fecha o fluxo. A classe *StreamReader* (que lê os caracteres de um fluxo, como um arquivo aberto) e a classe *StringReader* (que lê os caracteres de uma string) derivam da classe *TextReader*, e ambas redefinem o método *Close*. Aqui está um exemplo que lê as linhas de texto de um arquivo utilizando a classe *StreamReader* e então as exibe na tela:

```
TextReader reader = new StreamReader(filename);
string line;
while ((line = reader.ReadLine()) != null)
{
    Console.WriteLine(line);
}
reader.Close();
```

O método *ReadLine* lê a próxima linha de texto do fluxo e a armazena em uma string. O método *ReadLine* retorna *null* se não restar nada no fluxo. É importante chamar *Close* quando você tiver terminado com *reader*, para liberar o handle de arquivo e recursos associados. Mas há um problema nesse exemplo: não é seguro quanto a

exceções. Se a chamada para *ReadLine* ou *WriteLine* gerar uma exceção, a chamada para *Close* não acontecerá – será pulada. Se isso acontecer com muita frequência, você ficará sem handles de arquivos e não será capaz de abrir mais arquivo algum.

Descarte seguro quanto a exceções

Uma maneira de garantir que um método de descarte (como *Close*) seja sempre chamado, independentemente de haver ou não uma exceção, é chamá-lo dentro de um bloco *finally*. Veja o exemplo anterior codificado utilizando essa técnica:

```
TextReader reader = new StreamReader(filename);
try
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
finally
{
    reader.Close();
}
```

Utilizar um bloco *finally* como esse funciona, mas tem várias desvantagens que o tornam uma solução longe da ideal:

- O código se tornará rapidamente difícil de manejá-lo se você remover mais de um recurso. (Você acaba tendo blocos *try* e *finally* aninhados.)
- Em alguns casos, talvez seja preciso modificar o código para fazê-lo se ajustar a esse idioma. (Por exemplo, talvez você precise reordenar a declaração da referência de recurso, lembrar-se de inicializar a referência como *null* e verificar se a referência não é *null* no bloco *finally*.)
- Ele falha ao criar uma abstração da solução. Isso significa que a solução é difícil de entender, e você precisará repetir o código onde essa funcionalidade for necessária.
- A referência ao recurso permanece no escopo após o bloco *finally*. Isso significa que você pode accidentalmente tentar utilizar o recurso após ele ter sido liberado.

A instrução *using* é projetada para solucionar todos esses problemas.

A instrução *using* e a interface *IDisposable*

A instrução *using* fornece um mecanismo limpo para controlar os tempos de vida dos recursos. Você pode criar um objeto e esse ser destruído quando o bloco da instrução *using* terminar.



Importante Não confunda a instrução *using* mostrada nesta seção com a diretiva *using* que coloca um namespace no escopo. Infelizmente, essa mesma palavra-chave tem dois significados diferentes.

A sintaxe para uma instrução *using* é:

```
using ( type variable = initialization )
{
    StatementBlock
}
```

Aqui está a melhor maneira de garantir que seu código sempre chamará o método *Close* de um *TextReader*:

```
using (TextReader reader = new StreamReader(filename))
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
```

Essa instrução *using* é precisamente equivalente à seguinte transformação:

```
{
    TextReader reader = new StreamReader(filename);
    try
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }
    finally
    {
        if (reader != null)
        {
            ((IDisposable)reader).Dispose();
        }
    }
}
```



Nota A instrução *using* introduz um bloco próprio para propósitos de definição de escopo. Esse arranjo significa que a variável declarada em uma instrução *using* sai automaticamente de escopo no final da instrução embutida e não há como você accidentalmente tentar acessar um recurso removido*.

A variável declarada em uma instrução *using* deve ser do tipo que implementa a interface *IDisposable*. A interface *IDisposable* reside no namespace *System* e contém apenas um método, chamado *Dispose*:

```
namespace System
{
    interface IDisposable
    {
```

* N. de R.T.: O autor se refere à tentativa de acesso a um recurso após a execução do método *Dispose* sobre esse recurso.

```

        void Dispose();
    }
}

```

O objetivo do método *Dispose* é liberar os recursos utilizados por um objeto. Ocorre que a classe *StreamReader* implementa a interface *IDisposable* e seu método *Dispose* chama *Close* para fechar o fluxo. Você pode empregar uma instrução *using* como uma maneira adequada, segura e robusta quanto a exceções para garantir que um recurso seja sempre liberado. Isso resolve todos os problemas que existiam na solução manual *try/finaly*. Você agora tem uma solução que:

- É facilmente escalonável se for necessário descartar múltiplos recursos.
- Não altera a lógica do código do programa.
- Abstrai o problema e evita a repetição.
- É robusta. Você não pode referenciar accidentalmente a variável que foi declarada dentro da instrução *using* (nesse caso, *reader*) após o bloco da instrução ter terminado, pois ela não está mais no escopo – você obterá um erro de tempo de compilação.

Chame o método *Dispose* a partir de um destrutor

Ao escrever suas próprias classes, você deve escrever um destrutor ou implementar a interface *IDisposable* para que as instâncias de sua classe possam ser gerenciadas por uma instrução *using*? Uma chamada para um destrutor *acontecerá*, mas você só não sabe quando. Por outro lado, você sabe exatamente quando uma chamada para o método *Dispose* acontece, mas só não pode ter certeza de que ela realmente acontecerá, pois ela precisa que o programador que está utilizando suas classes se lembre de escrever uma instrução *using*. Mas é possível garantir que o método *Dispose* sempre execute, chamando-o a partir de um destrutor. Isso funciona como uma alternativa útil. Você poderá se esquecer de chamar o método *Dispose*, mas pelo menos pode ter certeza de que ele será chamado, mesmo que seja somente quando o programa terminar. Vamos investigar esse recurso em detalhes nos exercícios do final do capítulo, mas aqui está um exemplo de como você poderia implementar a interface *IDisposable*:

```

class Example : IDisposable
{
    private Resource scarce;          // recurso escasso a gerenciar e descartar
    private bool disposed = false;     // sinalizador para indicar se o recurso
                                      // já foi descartado

    ...
    ~Example()
    {
        this.Dispose(false);
    }

    public virtual void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {

```

```
if (!this.disposed)
{
    if (disposing)
    {
        // libera recursos grandes e gerenciados aqui
        ...
    }
    // libera recursos não gerenciados aqui
    ...
    this.disposed = true;
}
}

public void SomeBehavior() // método de exemplo
{
    checkIfDisposed();
    ...
}

...

private void checkIfDisposed()
{
    if (this.disposed)
    {
        throw new ObjectDisposedException("Example: object has been disposed of");
    }
}
```

Observe as seguintes características da classe *Example*:

- A classe implementa a interface *IDisposable*.
- O método *Dispose* público pode ser chamado a qualquer momento pelo código de seu aplicativo.
- O método *Dispose* público chama a versão protegida e sobrecarregada do método *Dispose* que recebe um parâmetro booleano, passando o valor *true* como argumento. Esse método é que faz o descarte de recursos.
- O destrutor chama a versão protegida e sobrecarregada do método *Dispose* que recebe um parâmetro booleano, passando o valor *false* como argumento. O destrutor só é chamado pelo coletor de lixo quando seu objeto estiver sendo finalizado.
- É seguro chamar o método *Dispose* protegido várias vezes. A variável *disposed* indica se o método já foi executado e é um recurso de segurança para evitar que o método tente descartar os recursos várias vezes, se for chamado simultaneamente. (Seu aplicativo poderia chamar *Dispose*, mas antes que o método terminasse, seu objeto poderia ir para a coleta de lixo e o método *Dispose* seria executado novamente pelo CLR do destrutor.) Os recursos são liberados somente na primeira vez que o método executa.
- O método *Dispose* protegido suporta o descarte de recursos gerenciados (como um array grande) e não gerenciados (como um handle de arquivo). Se o parâmetro *disposing* for *true*, esse método provavelmente foi chamado a partir do método *Dispose* público. Nesse caso, todos os recursos gerenciados e não

gerenciados são liberados. Se o parâmetro `disposing` for `false`, esse método provavelmente foi chamado a partir do destrutor, e o coletor de lixo está finalizando o objeto. Nesse caso, não é necessário (nem seguro quanto a exceções) liberar os recursos gerenciados, pois eles serão (ou já podem ter sido) manipulados pelo coletor de lixo, de modo que somente os recursos não gerenciados são liberados.

- O método `Dispose` público chama o método estático `GC.SuppressFinalize`. Esse método faz o coletor de lixo parar de chamar o destrutor nesse objeto, pois o objeto já foi finalizado.
- Todos os métodos comuns da classe (como `SomeBehavior`) verificam se o objeto já foi descartado. Se afirmativo, eles geram uma exceção.

Implemente o descarte seguro quanto a exceções

No conjunto de exercícios a seguir, você vai examinar como a instrução `using` ajuda a garantir que os recursos utilizados por objetos em seus aplicativos possam ser liberados de maneira oportuna, mesmo que ocorra uma exceção no código do aplicativo. Inicialmente, você implementará uma classe simples que implementa um destrutor e examinará quando esse destrutor é chamado pelo coletor de lixo.



Nota O objetivo da classe `Calculator` criada nesses exercícios é mostrar os princípios básicos da coleta de lixo apenas para propósitos ilustrativos. A classe não consome realmente quaisquer recursos gerenciados ou não gerenciados significativos. Normalmente, você criaria um destrutor ou implementaria a interface `IDisposable` para uma classe simples como essa.

Crie uma classe simples que utilize um destrutor

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. No menu File, aponte para New e então clique em Project.
A caixa de diálogo New Project se abre.
3. Na caixa de diálogo New Project, no painel à esquerda sob Templates, clique em Visual C#. No painel central, selecione o template Console Application. Na caixa Name próxima à parte inferior da caixa de diálogo, digite **GarbageCollection-Demo**. No campo Location, especifique `\Microsoft Press\Visual CSharp Step By Step\Chapter 14` na sua pasta Documentos e então clique em OK.



Dica Você pode usar o botão `Browse` adjacente ao campo Location para navegar até a pasta `Microsoft Press\Visual CSharp Step By Step\Chapter 14`, em vez de digitar o caminho manualmente.

O Visual Studio cria um novo aplicativo de console e exibe o arquivo Program.cs na janela Code and Text Editor.

- 4.** No menu Project, clique em Add Class.

A caixa de diálogo Add New Item – GarbageCollectionDemo se abre.

- 5.** Na caixa de diálogo Add New Item – GarbageCollectionDemo, verifique se o template Class está selecionado. Na caixa Name, digite **Calculator.cs** e clique em Add.

A classe *Calculator* é criada e exibida na janela Code and Text Editor.

- 6.** Adicione à classe *Calculator* o seguinte método público *Divide* mostrado em negrito:

```
class Calculator
{
    public int Divide(int first, int second)
    {
        return first / second;
    }
}
```

Esse é um método muito simples que divide o primeiro parâmetro pelo segundo e retorna o resultado. Ele é fornecido apenas para adicionar um pouco de funcionalidade que possa ser chamada por um aplicativo.

- 7.** Acima do método *Divide*, adicione o construtor público mostrado em negrito no código a seguir, no início da classe *Calculator*:

```
class Calculator
{
    public Calculator()
    {
        Console.WriteLine("Calculator being created");
    }
    ...
}
```

O objetivo desse construtor é permitir que você verifique se um objeto *Calculator* foi criado com êxito.

- 8.** Adicione à classe *Calculator* o destrutor mostrado em negrito no código a seguir, após o construtor:

```
class Calculator
{
    ...
    ~Calculator()
    {
        Console.WriteLine("Calculator being finalized");
    }
    ...
}
```

Esse destrutor simplesmente exibe uma mensagem para que você possa ver quando o coletor de lixo é executado e finaliza instâncias dessa classe. Ao escrever classes para aplicativos reais, normalmente você não produziria saída de texto em um destrutor.

9. Exiba o arquivo Program.cs na janela Code and Text Editor.
10. Na classe *Program*, adicione ao método *Main* as seguintes instruções que aparecem em negrito:

```
static void Main(string[] args)
{
    Calculator calculator = new Calculator();
    Console.WriteLine("{0} / {1} = {2}", 120, 15, calculator.Divide(120, 15));
    Console.WriteLine("Program finishing");
}
```

Esse código cria um objeto *Calculator*, chama o método *Divide* desse objeto (e exibe o resultado) e, então, gera uma mensagem na saída quando o programa termina.

11. No menu Debug, clique em Start Without Debugging. Verifique que o programa exibe a seguinte série de mensagens:

```
Calculator being created
120 / 15 = 8
Program finishing
Calculator being finalized
```

Observe que o finalizador do objeto *Calculator* só é executado quando o aplicativo está para terminar, após a conclusão do método *Main*.

12. Na janela de console, pressione a tecla Enter e retorne ao Visual Studio 2013.

O CLR garante que todos os objetos criados por seus aplicativos se sujeitarão à coleta de lixo, mas nem sempre você pode garantir quando isso acontecerá. No exercício, o programa teve vida muito curta e o objeto *Calculator* foi finalizado quando o CLR fez a limpeza no final do programa. Mas você também pode verificar que esse é o caso em aplicativos mais importantes, com classes que consomem recursos escassos – e, a menos que dê os passos necessários para fornecer um meio de descartá-los, os objetos criados por seus aplicativos poderão reter seus recursos até que esses aplicativos terminem. Se o recurso fosse um arquivo, isso poderia impedir o acesso de outros usuários a esse arquivo; se o recurso fosse uma conexão de banco de dados, seu aplicativo poderia impedir a conexão de outros usuários no mesmo banco de dados. Em condições ideais, você quer liberar os recursos assim que acabar de utilizá-los, em vez de esperar que o aplicativo termine.

No próximo exercício, você vai implementar a interface *IDisposable* na classe *Calculator* e permitir que o programa finalize objetos *Calculator* no momento que escolher.

Implemente a interface *IDisposable*

1. Exiba o arquivo Calculator.cs na janela Code and Text Editor.

- 2.** Modifique a definição da classe *Calculator* de modo que ela implemente a interface *IDisposable*, como mostrado aqui em negrito:

```
class Calculator : IDisposable
{
    ...
}
```

- 3.** Adicione o método a seguir, chamado *Dispose*, no final da classe *Calculator*. Esse método é definido pela interface *IDisposable*:

```
class Calculator : IDisposable
{
    ...
    public void Dispose()
    {
        Console.WriteLine("Calculator being disposed");
    }
}
```

Normalmente, você adicionaria ao método *Dispose* um código para liberar os recursos mantidos pelo objeto. Neste caso não há nenhum, e o objetivo da instrução *Console.WriteLine* nesse método é apenas para que você possa ver quando o método *Dispose* é executado. Contudo, você pode ver que, em um aplicativo real, é provável que houvesse alguma duplicação de código entre o destrutor e o método *Dispose*. Para eliminar essa duplicação, normalmente você colocaria esse código em um lugar e o chamaria em outro. Sabendo que não é possível chamar um destrutor explicitamente a partir do método *Dispose*, faz sentido, em vez disso, chamar o método *Dispose* a partir do destrutor e colocar a lógica que libera os recursos nesse método.

- 4.** Modifique o destrutor de modo que ele chame o método *Dispose*, como mostrado em negrito no código a seguir (deixe a instrução que mostra a mensagem no lugar, no finalizador, para que você possa ver quando ele está sendo executado pelo coletor de lixo):

```
~Calculator()
{
    Console.WriteLine("Calculator being finalized");
    this.Dispose();
}
```

Quando você quer destruir um objeto *Calculator* em um aplicativo, o método *Dispose* não é executado automaticamente; seu código precisa chamá-lo explicitamente (com uma instrução como *calculator.Dispose()*) ou criar o objeto *Calculator* dentro de uma instrução *using*. Em seu programa, você vai adotar esta última estratégia.

- 5.** Exiba o arquivo Program.cs na janela Code and Text Editor. Modifique as instruções no método *Main* que criam o objeto *Calculator* e chamam o método *Divide*, como mostrado aqui em negrito:

```
static void Main(string[] args)
{
    using (Calculator calculator = new Calculator())
    {
```

```

        Console.WriteLine("{0} / {1} = {2}", 120, 15, calculator.Divide(120, 15));
    }

    Console.WriteLine("Program finishing");
}

```

- 6.** No menu Debug, clique em Start Without Debugging. Verifique que agora o programa exibe a seguinte série de mensagens:

```

Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
Calculator being finalized
Calculator being disposed

```

A instrução *using* faz com que o método *Dispose* seja executado antes da instrução que exibe a mensagem "Program finishing". Contudo, você pode ver que o destrutor do objeto *Calculator* ainda é executado quando o aplicativo termina, e ele chama o método *Dispose* outra vez. Isso é claramente um desperdício de processamento.

- 7.** Na janela de console, pressione a tecla Enter e retorne ao Visual Studio 2013.

Descartar mais de uma vez os recursos mantidos por um objeto pode ser desastroso ou não, mas definitivamente não é uma boa prática. A estratégia recomendada para resolver esse problema é adicionar um campo booleano privado à classe para indicar se o método *Dispose* já foi chamado e, então, examinar esse campo no método *Dispose*.

Impeça que um objeto seja descartado mais de uma vez

- Exiba o arquivo Calculator.cs na janela Code and Text Editor.
- Adicione à classe *Calculator* um campo booleano privado, chamado *disposed*, e inicialize o valor desse campo com *false*, como mostrado em negrito a seguir:

```

class Calculator : IDisposable
{
    private bool disposed = false;
    ...
}

```

O objetivo desse campo é monitorar o estado desse objeto e indicar se o método *Dispose* foi chamado.

- Modifique o código do método *Dispose* para exibir a mensagem somente se o campo *disposed* for *false*. Após exibir a mensagem, defina o campo *disposed* com *true*, como demonstrado aqui:

```

public void Dispose()
{
    if (!this.disposed)

```

```

{
    Console.WriteLine("Calculator being disposed");
}

this.disposed = true;
}

```

- 4.** No menu Debug, clique em Start Without Debugging. Observe que o programa exibe a seguinte série de mensagens:

```

Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
Calculator being finalized

```

Agora o objeto *Calculator* está sendo descartado apenas uma vez, mas o destrutor ainda está executando. Novamente, isso é um desperdício; não faz sentido executar um destrutor para um objeto que já liberou seus recursos.

- 5.** Na janela de console, pressione a tecla Enter e retorno ao Visual Studio 2013.
6. Na classe *Calculator*, adicione ao método *Dispose* a seguinte instrução que aparece em negrito:

```

public void Dispose()
{
    if (!disposed)
    {
        Console.WriteLine("Calculator being disposed");
    }
    this.disposed = true;
GC.SuppressFinalize(this);
}

```

A classe *GC* dá acesso ao coletor de lixo e implementa vários métodos estáticos com os quais é possível controlar algumas das ações que ele executa. Com o método *SuppressFinalize*, você pode indicar se o coletor de lixo não deve realizar a finalização no objeto especificado, e isso impede a execução do destrutor.



Importante A classe *GC* expõe vários métodos com os quais é possível configurar o coletor de lixo. Contudo, em geral, é melhor deixar o próprio CLR gerenciar o coletor de lixo, pois você pode prejudicar seriamente o desempenho de seu aplicativo se chamar esses métodos de forma imprudente. Você deve tratar o método *SuppressFinalize* com extrema cautela, pois, se deixar de descartar um objeto, correrá o risco de perder dados (se deixar de fechar um arquivo corretamente, por exemplo, quaisquer dados colocados em buffer na memória, mas ainda não gravados no disco, podem ser perdidos). Só chame esse método em situações como as mostradas neste exercício, quando você sabe que um objeto já foi descartado.

7. No menu Debug, clique em Start Without Debugging. Observe que o programa exibe a seguinte série de mensagens:

```
Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
```

Pode-se ver que o destrutor não está mais em execução, pois o objeto *Calculator* já foi descartado, antes do término do programa.

8. Na janela de console, pressione a tecla Enter e retorne ao Visual Studio 2013.

Segurança de thread e o método *Dispose*

O exemplo de uso do campo *disposed* para impedir que um objeto seja descartado várias vezes funciona bem na maioria dos casos, mas lembre-se de que você não tem controle sobre quando o finalizador é executado. Nos exercícios deste capítulo, ele sempre foi executado no término do programa, mas nem sempre esse é o caso – ele pode ser executado a qualquer momento, após a última referência a um objeto ter desaparecido. Assim, é possível que o finalizador possa ser chamado pelo coletor de lixo em sua própria thread, enquanto o método *Dispose* está sendo executado, especialmente se o método *Dispose* tiver um volume de trabalho significativo. Você poderia reduzir a possibilidade da liberação múltipla de recursos movendo a instrução que define o campo *disposed* com true para mais perto do início do método *Dispose*. Porém, nesse caso, correria o risco de não liberar realmente os recursos, caso ocorra uma exceção depois de ter definido essa variável, mas antes de os ter liberado.

Para eliminar por completo as chances de duas threads concorrentes descartarem os mesmos recursos no mesmo objeto simultaneamente, você pode escrever seu código de maneira segura quanto a thread, incorporando-o em uma instrução *lock* do C#, como segue:

```
public void Dispose()
{
    lock(this)
    {
        if (!disposed)
        {
            Console.WriteLine("Calculator being disposed");
        }
        this.disposed = true;
        GC.SuppressFinalize(this);
    }
}
```

O objetivo da instrução *lock* é impedir que o mesmo bloco de código seja executado ao mesmo tempo em threads diferentes. O argumento dessa instrução (*this* no exemplo anterior) deve ser uma referência para um objeto. O código entre as chaves define o escopo da instrução *lock*. Quando a execução atinge a instrução *lock*, se o objeto especificado estiver travado nesse momento, a thread que solicitou a trava é bloqueada e o código é suspenso nesse ponto. Quando a thread que atualmente contém a trava atinge a chave de fechamento da instrução *lock*, a trava é liberada, permitindo que a thread bloqueada adquira a trava e continue. Contudo, quando isso acontecer, o campo *disposed* estará definido com true, de modo que a segunda thread não tentará executar o código do bloco *if (!disposed)*.

É seguro utilizar travas dessa maneira, mas pode prejudicar o desempenho. Uma alternativa é utilizar a estratégia já descrita neste capítulo, por meio da qual apenas o descarte repetido de recursos gerenciados é suprimido (não é seguro quanto à exceção descartar recursos gerenciados mais de uma vez; você não comprometerá a segurança de seu computador, mas poderá afetar a integridade lógica de seu aplicativo se tentar descartar um objeto gerenciado que não existe mais). Essa estratégia implementa versões sobrecarregadas do método *Dispose*; a instrução *using* chama *Dispose()*, o qual, por sua vez, executa a insrtução *Dispose(true)*, enquanto o destrutor chama *Dispose(false)*. Os recursos gerenciados só serão liberados se o parâmetro da versão sobrecarregada do método *Dispose* for true. Para obter mais informações, consulte o exemplo da seção “Chame o método *Dispose* a partir de um destrutor”.

O objetivo da instrução *using* é garantir que um objeto sempre seja descartado, mesmo que ocorra uma exceção enquanto ele está sendo utilizado. No último exercício deste capítulo, você vai verificar se isso acontece, gerando uma exceção no meio de um bloco *using*.

Verifique se um objeto é descartado após uma exceção

1. Exiba o arquivo Program.cs na janela Code and Text Editor.
2. Modifique a instrução que chama o método *Divide* do objeto *Calculator*, como mostrado em negrito:

```
static void Main(string[] args)
{
    using (Calculator calculator = new Calculator())
    {
        Console.WriteLine("{0} / {1} = {2}", 120, 0, calculator.Divide(120, 0));
    }
    Console.WriteLine("Program finishing");
}
```

A instrução corrigida tenta dividir 120 por 0.

3. No menu Debug, clique em Start Without Debugging.

Conforme você pode ter antecipado, o aplicativo lança uma exceção *DivideByZeroException* não tratada.

4. Na caixa de mensagem de GarbageCollectionDemo, clique em Cancel (você precisa ser rápido, antes que os botões Debug e Close Program apareçam).

Verifique que a mensagem "Calculator being disposed" aparece após a exceção não tratada na janela de console.

```
Calculator being created
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
  at GarbageCollectionDemo.Calculator.Divide(Int32 first, Int32 second) in d:\Visual CSharp Step By Step\Chapter 14\Windows 8.1\GarbageCollectionDemo\GarbageCollectionDemo\Calculator.cs:line 26
  at GarbageCollectionDemo.Program.Main(String[] args) in d:\Visual CSharp Step By Step\Chapter 14\Windows 8.1\GarbageCollectionDemo\GarbageCollectionDemo\Program.cs:line 15
Calculator being disposed←
Press any key to continue . . .
```



Nota Se você foi lento demais e os botões Debug e Close Program já apareceram, clique em Close Program e execute o aplicativo novamente, sem depuração.

5. Na janela de console, pressione a tecla Enter e retorne ao Visual Studio 2013.

Resumo

Neste capítulo, vimos como o coletor de lixo funciona e como o .NET Framework o utiliza para descartar objetos e resgatar memória. Você aprendeu a escrever um destrutor para limpar os recursos utilizados por um objeto quando a memória é reciclada pelo coletor de lixo. Viu também como utilizar a instrução *using* para implementar descarte de recursos seguro quanto a exceções e como implementar a interface *IDisposable* para dar suporte a essa forma de descarte de objetos.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 15, "Implementação de propriedades para acessar campos".
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Escrever um destrutor	Escreva um método cujo nome seja igual ao nome da classe e seja iniciado com um til (~). O método não deve ter um modificador de acesso (como public) e não pode ter parâmetro algum nem retornar um valor. Por exemplo:
	<pre>class Example { ~Example() { ... } }</pre>
Chamar um destrutor	Você não pode chamar um destrutor. Somente o coletor de lixo pode chamá-lo.
Forçar uma coleta de lixo (não recomendável)	Chame GC.Collect.
Liberar um recurso em determinado momento (mas com o risco de vazamentos de recurso se uma exceção interromper a execução)	Escreva um método de descarte (um método que descarta um recurso) e chame-o explicitamente no programa. Por exemplo:
	<pre>class TextReader { ... public virtual void Close() { ... } } class Example { void Use() { TextReader reader = ...; // usa reader reader.Close(); } }</pre>
Suportar descarte seguro quanto a exceções em uma classe	Implemente a interface IDisposable. Por exemplo:
	<pre>class SafeResource : IDisposable { ... public void Dispose() { // Descarta os recursos aqui } }</pre>
Implementar descarte seguro quanto a exceções para um objeto que implementa a interface IDisposable	Crie o objeto em uma instrução using. Por exemplo:
	<pre>using (SafeResource resource = new SafeResource()) { // Usa SafeResource aqui ... }</pre>

Esta página foi deixada em branco intencionalmente.

PARTE III

Definição de tipos extensíveis em C#

CAPÍTULO 15	Implementação de propriedades para acessar campos	337
CAPÍTULO 16	Indexadores	359
CAPÍTULO 17	Genéricos	376
CAPÍTULO 18	Coleções	406
CAPÍTULO 19	Enumeração sobre coleções	428
CAPÍTULO 20	Separação da lógica do aplicativo e tratamento de eventos	443
CAPÍTULO 21	Consulta a dados na memória usando expressões de consulta	477
CAPÍTULO 22	Sobrecarga de operadores	502

Esta página foi deixada em branco intencionalmente.

CAPÍTULO 15

Implementação de propriedades para acessar campos

Neste capítulo, você vai aprender a:

- Encapsular campos lógicos utilizando propriedades.
- Controlar o acesso de leitura às propriedades declarando métodos de acesso *get*.
- Controlar o acesso de gravação às propriedades declarando métodos de acesso *set*.
- Criar interfaces que declaram propriedades.
- Implementar interfaces que contêm propriedades utilizando estruturas e classes.
- Gerar propriedades automaticamente com base em definições de campo.
- Utilizar propriedades para inicializar objetos.

Este capítulo discute a definição e a utilização das propriedades para encapsular campos e dados em uma classe. Até aqui, os capítulos deram ênfase ao fato de que você deve tornar privados os campos em uma classe e fornecer métodos para armazenar e recuperar valores. Com essa estratégia, o acesso seguro e controlado aos campos está garantido, e passa a ser viável encapsular a lógica e as regras adicionais relacionadas aos valores permitidos. Mas não é natural que a sintaxe acesse um campo dessa forma. Quando quer ler ou gravar uma variável, você, em geral, vai utilizar uma instrução de atribuição; é por isso que pode soar como algo grosseiro chamar um método para alcançar o mesmo efeito em um campo (que é, afinal de contas, apenas uma variável). As propriedades são projetadas para resolver esse problema.

Implemente encapsulamento com métodos

Em primeiro lugar, vamos recapitular a motivação original para utilizar os métodos a fim de ocultar os campos.

Considere a seguinte estrutura, que representa uma posição na tela de um computador como um par de coordenadas x e y. Suponha que o intervalo de valores válidos para a coordenada x resida entre 0 e 1280 e o intervalo de valores válidos para a coordenada y resida entre 0 e 1024.

```
struct ScreenPosition
{
    public int X;
    public int Y;
```

```

public ScreenPosition(int x, int y)
{
    this.X = rangeCheckedX(x);
    this.Y = rangeCheckedY(y);
}

private static int rangeCheckedX(int x)
{
    if (x < 0 || x > 1280)
    {
        throw new ArgumentOutOfRangeException("X");
    }
    return x;
}

private static int rangeCheckedY(int y)
{
    if (y < 0 || y > 1024)
    {
        throw new ArgumentOutOfRangeException("Y");
    }
    return y;
}

```

Um problema dessa estrutura é que ela não segue a regra de ouro do encapsulamento – isto é, ela não mantém seus dados privados. Geralmente, os dados públicos são uma má ideia porque a classe não pode controlar os valores que um aplicativo específica. Por exemplo, o construtor *ScreenPosition* verifica a faixa de seus parâmetros para garantir que estejam em um intervalo especificado, mas nenhuma verificação desse tipo pode ser feita no acesso “bruto” aos campos públicos. Mais cedo ou mais tarde (provavelmente mais cedo), um erro ou um mau entendimento por parte de um desenvolvedor que utilize essa classe em um aplicativo poderá fazer *X* ou *Y* sair desse intervalo:

```

ScreenPosition origin = new ScreenPosition(0, 0);
...
int xpos = origin.X;
origin.Y = -100; // opa

```

A maneira de resolver esse problema é criar campos privados e adicionar um método de acesso e um método modificador para ler e gravar respectivamente o valor de cada campo privado. Os métodos modificadores podem então verificar o intervalo dos novos valores de campo. Por exemplo, o código a seguir contém um método de acesso (*GetX*) e um modificador (*SetX*) para o campo *X*. Observe que *SetX* verifica seu valor de parâmetro:

```

struct ScreenPosition
{
    ...
    public int GetX()
    {
        return this.X;
    }

    public void SetX(int newX)
    {

```

```

        this.x = rangeCheckedX(newX);
    }
    ...
private static int rangeCheckedX(int x) { ... }
private static int rangeCheckedY(int y) { ... }
private int x, y;
}

```

O código agora impõe com êxito as restrições de intervalo de valores, o que é bom. Mas há um preço a ser pago por essa valiosa garantia – *ScreenPosition* não tem mais uma sintaxe natural do tipo campo; em vez disso, utiliza a complicada sintaxe baseada em método. O exemplo a seguir aumenta o valor de *X* por 10. Para fazer isso, ele precisa ler o valor de *X* utilizando o método de acesso *GetX* e então gravar o valor de *X* utilizando o método modificador *SetX*.

```

int xpos = origin.GetX();
origin.SetX(xpos + 10);

```

Compare esse código com o código equivalente caso o campo *X* fosse público:

```
origin.X += 10;
```

Não há dúvida de que, neste caso, utilizar campos públicos é sintaticamente mais claro, conciso e fácil. Infelizmente, o uso de campos públicos quebra o encapsulamento. Utilizando propriedades, você pode combinar o melhor dos dois mundos (campos e métodos) para manter o encapsulamento, permitindo a sintaxe do tipo campo.

O que são propriedades?

Uma *propriedade* é um cruzamento entre um campo e um método – ela parece um campo, mas atua como um método. Você acessa uma propriedade utilizando exatamente a mesma sintaxe empregada para acessar um campo. O compilador, porém, converte automaticamente essa sintaxe do tipo campo em chamadas a métodos de acesso (às vezes referidos como métodos de *obtenção de propriedade* e de *definição de propriedade*).

A sintaxe de uma declaração de propriedade se parece com esta:

```

AccessModifier TypePropertyName
{
    get
    {
        // código de acesso de leitura
    }

    set
    {
        // código de acesso de gravação
    }
}

```

Uma propriedade pode conter dois blocos de código, começando com as palavras-chave *get* e *set*. O bloco *get* contém instruções que são executadas quando a

propriedade é lida e o bloco *set* engloba instruções que são executadas quando a propriedade é gravada. O tipo de propriedade especifica o tipo de dado lido e gravado pelos métodos de acesso *get* e *set*.

O próximo exemplo de código mostra a estrutura *ScreenPosition* reescrita utilizando propriedades. Ao examinar esse código, observe o seguinte:

- *_x* e *_y* minúsculos são campos *private*.
- *X* e *Y* maiúsculos são propriedades *public*.
- A todos os métodos de acesso *set* são passados os dados a serem gravados, utilizando um parâmetro oculto e predefinido chamado *value*.

```
struct ScreenPosition
{
    private int _x, _y;

    public ScreenPosition(int X, int Y)
    {
        this._x = rangeCheckedX(X);
        this._y = rangeCheckedY(Y);
    }

    public int X
    {
        get { return this._x; }
        set { this._x = rangeCheckedX(value); }
    }

    public int Y
    {
        get { return this._y; }
        set { this._y = rangeCheckedY(value); }
    }

    private static int rangeCheckedX(int x) { ... }
    private static int rangeCheckedY(int y) { ... }
}
```

Nesse exemplo, um campo privado implementa diretamente cada propriedade, mas isso é apenas uma das maneiras de implementar uma propriedade. Tudo o que é necessário é que um método de acesso *get* retorne um valor do tipo especificado. Esse valor poderia ser calculado de forma fácil e dinâmica, em vez de simplesmente ser recuperado dos dados armazenados; nesse caso não há necessidade de um campo físico.



Nota Embora os exemplos deste capítulo mostrem como definir propriedades para uma estrutura, eles são igualmente aplicáveis às classes; a sintaxe é a mesma.

Nomes de propriedades e campos: um alerta

A seção “Nomeando variáveis” no Capítulo 2, “Variáveis, operadores e expressões”, descreve algumas recomendações para dar nomes às variáveis. Em especial, ela diz que você não deve iniciar um identificador com um sublinhado. Contudo, você pode ver que a estrutura *ScreenPosition* não segue essa orientação completamente; ela contém dois campos chamados *_x* e *_y*. Há um bom motivo para essa anomalia. O quadro “Convenção de nomes e acessibilidade” no Capítulo 7, “Criação e gerenciamento de classes e objetos”, descreve como é comum utilizar identificadores que começam com uma letra maiúscula para métodos e campos publicamente acessíveis, e utilizar identificadores que começam com uma letra minúscula para métodos e campos privados. Consideradas em conjunto, essas duas práticas podem fazer com que você dê a propriedades e campos privados o mesmo nome, diferindo apenas pela letra inicial, e muitas organizações fazem exatamente isso.

Se sua organização adota essa estratégia, você deve estar ciente de um importante inconveniente. Examine o código a seguir, que implementa uma classe chamada *Employee*. O campo *employeeID* é privado, mas a propriedade *EmployeeID* fornece acesso público a ele.

```
class Employee
{
    private int employeeID;

    public int EmployeeID
    {
        get { return this.EmployeeID; }
        set { this.EmployeeID = value; }
    }
}
```

Esse código compilará perfeitamente bem, mas resultará em um programa que lança uma exceção *StackOverflowException* sempre que a propriedade *EmployeeID* for acessada. Isso ocorre porque os métodos de acesso *get* e *set* referenciam a propriedade (com a letra maiúscula *E*) em vez do campo privado (com a letra minúscula *e*), o que causa um loop recursivo infinito que acaba fazendo o processo esgotar a memória disponível. Esse tipo de erro é muito difícil de descobrir! Por isso, os exemplos deste livro nomeiam os campos privados utilizados para fornecer os dados para propriedades com um sublinhado inicial; isso os torna muito mais fáceis de distinguir dos nomes de propriedades. Todos os outros campos privados continuarão a utilizar identificadores camelCase, sem um sublinhado inicial.

Utilize propriedades

Ao utilizar uma propriedade em uma expressão, você pode empregá-la em um contexto de leitura (quando estiver recuperando seu valor) ou em um contexto de gravação (quando estiver modificando seu valor). O exemplo a seguir mostra como ler os valores das propriedades *X* e *Y* da estrutura *ScreenPosition*:

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X;      // chama origin.X.get
int ypos = origin.Y;      // chama origin.Y.get
```

Observe as propriedades e os campos são acessados usando uma sintaxe idêntica. Quando você utiliza uma propriedade em um contexto de leitura, o compilador automaticamente traduz seu código do tipo campo em uma chamada ao método de acesso *get* dessa propriedade. Da mesma maneira, se você utilizar uma propriedade em um contexto de gravação, o compilador automaticamente traduzirá seu código do tipo campo em uma chamada para o método de acesso *set* dessa propriedade.

```
origin.X = 40;      // chama origin.X.set, com o valor configurado como 40
origin.Y = 100;     // chama origin.Y.Set, com o valor configurado como 100
```

Os valores atribuídos são passados para os métodos de acesso *set* utilizando a variável *value*, conforme descrito na seção anterior. O runtime faz isso automaticamente.

Além disso, é possível usar uma propriedade em um contexto de leitura/gravação. Nesse caso, tanto o método de acesso *get* quanto o método de acesso *set* são empregados. Por exemplo, o compilador traduz automaticamente as instruções em chamadas aos métodos de acesso *get* e *set* como a seguinte:

```
origin.X += 10;
```



Dica Você pode declarar propriedades *static* assim como campos e métodos *static*. As propriedades estáticas são acessadas por meio do nome da classe ou estrutura, em vez de uma instância da classe ou estrutura.

Propriedades somente-leitura

Você pode declarar uma propriedade que contenha apenas um método de acesso *get*. Nesse caso, só poderá utilizar a propriedade em um contexto de leitura. Por exemplo, veja a propriedade *X* da estrutura *ScreenPosition* declarada como uma propriedade somente-leitura:

```
struct ScreenPosition
{
    private int _x;
    ...
    public int X
    {
        get { return this._x; }
    }
}
```

A propriedade *X* não contém um método de acesso *set*; portanto, qualquer tentativa de utilizar *X* em um contexto de gravação falhará, conforme demonstrado no exemplo a seguir:

```
origin.X = 140; // erro de tempo de compilação
```

Propriedades somente-gravação

Da mesma forma, você pode declarar uma propriedade que contém apenas um método de acesso *set*. Nesse caso, só poderá utilizar a propriedade no contexto de gravação. Por exemplo, veja a propriedade *X* da estrutura *ScreenPosition* declarada como uma propriedade de gravação:

```
struct ScreenPosition
{
    private int _x;
    ...
    public int X
    {
        set { this._x = rangeCheckedX(value); }
    }
}
```

A propriedade *X* não contém um método de acesso *get*; qualquer tentativa de utilizar *X* em um contexto de leitura falhará, como ilustrado aqui:

```
Console.WriteLine(origin.X);      // erro de tempo de compilação
origin.X = 200;                   // compila normalmente
origin.X += 10;                  // erro de tempo de compilação
```



Nota As propriedades somente-gravação são úteis para dados que devem ter segurança absoluta, como senhas. Teoricamente, um aplicativo que implementa segurança permitirá que você defina sua senha, mas nunca que você a leia. Ao tentar se conectar, o usuário poderá informar a senha. O método de logon pode comparar essa senha com a senha armazenada e retornar apenas uma indicação de uma possível combinação.

Acessibilidade de propriedades

Você pode especificar a acessibilidade de uma propriedade (*public*, *private* ou *protected*) ao declará-la. Mas também é possível, dentro da declaração da propriedade, redefinir sua acessibilidade para os métodos de acesso *get* e *set*. Por exemplo, a versão da estrutura *ScreenPosition* mostrada no código a seguir define o método de acesso *set* das propriedades *X* e *Y* como *private*. (Os métodos de acesso *get* são *public*, porque as propriedades são *public*.)

```
struct ScreenPosition
{
    private int _x, _y;
    ...
    public int X
    {
        get { return this._x; }
        private set { this._x = rangeCheckedX(value); }
    }

    public int Y
    {
        get { return this._y; }
        private set { this._y = rangeCheckedY(value); }
    }
    ...
}
```

Você deve observar algumas regras ao definir métodos de acesso com acessibilidades diferentes entre si:

- É possível alterar a acessibilidade de apenas um dos métodos de acesso quando definidos. Não faria muito sentido definir uma propriedade como *public* apenas para alterar a acessibilidade de ambos os métodos de acesso para *private*.
- O modificador não deve especificar uma acessibilidade que seja menos restritiva do que a da propriedade. Por exemplo, se a propriedade é declarada como *private*, você não pode especificar o método de acesso de leitura como *public*. (Em vez disso, tornaria a propriedade *public* e o método de acesso de escrita *private*.)

Restrições de uma propriedade

As propriedades se parecem, agem e têm comportamento igual aos dos campos, quando você lê ou escreve dados com elas. Mas elas não são campos verdadeiros, e determinadas restrições se aplicam:

- Você pode atribuir um valor por meio de uma propriedade de uma estrutura ou classe somente depois que a estrutura ou a classe foi inicializada. O exemplo de código a seguir não é válido porque a variável *location* não foi inicializada (utilizando *new*):

```
ScreenPosition location;
location.X = 40; // erro de tempo de compilação, location não foi atribuída
```



Nota Isso pode parecer trivial, mas se *X* fosse um campo em vez de uma propriedade, o código seria válido. Por isso, você deve definir estruturas e classes utilizando propriedades desde o início, em vez de usar campos que mais tarde migrarão para propriedades. O código que emprega suas classes e estruturas poderá não funcionar mais, se você transformar os campos em propriedades. Retornaremos a essa questão na seção “Como gerar propriedades automáticas”, mais adiante neste capítulo.

- Você não pode utilizar uma propriedade como um argumento *ref* ou *out* para um método (embora seja possível usar um campo gravável como um argumento *ref* ou *out*). Isso faz sentido, porque a propriedade na realidade não aponta para uma posição da memória; em vez disso, aponta para um método de acesso, como no exemplo a seguir:

```
MyMethod(ref location.X); // erro de tempo de compilação
```

- Uma propriedade pode conter no máximo um método de acesso *get* e um método de acesso *set*. Uma propriedade não pode conter outros métodos, campos ou outras propriedades.
- Os métodos de acesso *get* e *set* não podem receber parâmetro algum. Os dados que estão sendo atribuídos são passados automaticamente para o método de acesso *set*, utilizando a variável *value*.
- Você não pode declarar propriedades *const*, como demonstrado aqui:

```
const int X { get { ... } set { ... } } // erro de tempo de compilação
```

Uso adequado das propriedades

As propriedades são um recurso poderoso, e quando utilizadas do modo correto, podem contribuir para facilitar o entendimento e a manutenção do código. Mas elas não substituem um cuidadoso projeto orientado a objetos que focaliza o comportamento dos objetos em vez de suas propriedades. O acesso aos campos privados por meio de métodos ou propriedades comuns não torna, por si só, seu código bem projetado. Por exemplo, uma conta de banco armazena um saldo, indicando que os fundos estão disponíveis. Você poderia ficar tentado a criar uma propriedade *Balance* em uma classe *BankAccount*, como esta:

```
class BankAccount
{
    private decimal _balance;
    ...
    public decimal Balance
    {
        get { return this._balance; }
        set { this._balance = value; }
    }
}
```

Esse é um projeto ruim, pois não representa a funcionalidade exigida para sacar ou depositar dinheiro em uma conta. (Se conhecer um banco em que você pode mudar o saldo da sua conta diretamente sem efetuar depósito, me avise!) Ao programar, tente expressar o problema em questão na própria solução, e não em um labirinto de sintaxe de baixo nível: Conforme o exemplo a seguir ilustra, forneça métodos *Deposit* e *Withdraw* para a classe *BankAccount*, em vez de um método set de propriedade:

```
class BankAccount
{
    private decimal _balance;
    ...
    public decimal Balance { get { return this._balance; } }
    public void Deposit(money amount) { ... }
    public bool Withdraw(money amount) { ... }
}
```

Declare propriedades de interface

Já discutimos interfaces no Capítulo 13, “Como criar interfaces e definir classes abstratas”. As interfaces podem definir tanto propriedades quanto métodos. Para fazer isso, você declara a palavra-chave *get* ou *set* (ou ambas), mas substitui o corpo do método de acesso *get* ou *set* por um ponto e vírgula, como mostrado aqui:

```
interface IScreenPosition
{
    int X { get; set; }
    int Y { get; set; }
}
```

Qualquer classe ou estrutura que implemente essa interface deve implementar as propriedades *X* e *Y* com os métodos de acesso *get* e *set*.

```
struct ScreenPosition : IScreenPosition
{
    ...
    public int X
    {
        get { ... }
        set { ... }
    }

    public int Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Se você implementar as propriedades de interface em uma classe, poderá declarar as implementações da propriedade como *virtual*, o que permite às classes derivadas redefinirem as implementações.

```
class ScreenPosition : IScreenPosition
{
    ...
    public virtual int X
    {
        get { ... }
        set { ... }
    }
    public virtual int Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```



Nota Esse exemplo mostra uma *classe*. Lembre-se de que a palavra-chave *virtual* não é válida ao se criar uma *estrutura*, porque estruturas não aceitam herança.

Você também pode optar por implementar uma propriedade utilizando a sintaxe explícita de implementação de interface abordada no Capítulo 13. Uma implementação explícita de uma propriedade é não pública e não virtual (e não pode ser redefinida).

```
struct ScreenPosition : IScreenPosition
{
    ...
    int IScreenPosition.X
    {
        get { ... }
        set { ... }
    }

    int IScreenPosition.Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Substitua métodos por propriedades

O Capítulo 13 lhe ensinou a criar um aplicativo de desenho com o qual o usuário pode colocar círculos e quadrados no canvas de uma janela. Nos exercícios daquele capítulo, você fatorou a funcionalidade comum das classes *Circle* e *Square* em uma classe abstrata chamada *DrawingShape*. A classe *DrawingShape* fornece os métodos *SetLocation* e *SetColor* que tornam possível ao aplicativo especificar a posição e a cor de uma forma na tela. No exercício a seguir, você vai modificar a classe *DrawingShape* para expor a localização e a cor de uma forma como propriedades.

Utilize propriedades

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra o projeto Drawing, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 15\Windows X\Drawing Using Properties na sua pasta Documentos.
3. Exiba o arquivo DrawingShape.cs na janela Code and Text Editor.

Esse arquivo contém a mesma classe *DrawingShape* que está no Capítulo 13, exceto que, seguindo as recomendações descritas anteriormente neste capítulo, o campo *size* foi renomeado como *_size* e os campos *locX* e *locY* foram renomeados como *_x* e *_y*.

```
abstract class DrawingShape
{
    protected int _size;
    protected int _x = 0, _y = 0;
    ...
}
```

4. Abra o arquivo IDraw.cs do projeto Drawing na janela Code and Text Editor.

Essa interface especifica o método *SetLocation*, como segue:

```
interface IDraw
{
    void SetLocation(int xCoord, int yCoord);
    ...
}
```

O objetivo desse método é definir os campos *_x* e *_y* do objeto *DrawingShape* com os valores passados. Esse método pode ser substituído por duas propriedades.

5. Exclua esse método e substitua-o pela definição de duas propriedades chamadas *X* e *Y*, como mostrado aqui em negrito:

```
interface IDraw
{
    int X { get; set; }
    int Y { get; set; }
    ...
}
```

6. Na classe *DrawingShape*, exclua o método *SetLocation* e substitua-o pelas seguintes implementações das propriedades *X* e *Y*:

```
public int X
{
    get { return this._x; }
    set { this._x = value; }
}

public int Y
{
    get { return this._y; }
    set { this._y = value; }
}
```

7. Exiba o arquivo DrawingPad.xaml.cs na janela Code and Text Editor. Se estiver usando o Windows 8.1, localize o método *drawingCanvas_Tapped*. Se estiver usando o Windows 7 ou o Windows 8, localize o método *drawingCanvas_MouseLeftButtonDown*.

Esses métodos são executados quando o usuário toca na tela ou clica com o botão esquerdo do mouse, e desenham um quadrado na tela, no ponto onde o usuário tocou ou clicou.

8. Localize a instrução que chama o método *SetLocation* para definir a posição do quadrado na tela. Ela está localizada no seguinte bloco da instrução *if*, como realçado a seguir:

```
if (mySquare is IDraw)
{
    IDraw drawSquare = mySquare;
    drawSquare.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
    drawSquare.Draw(drawingCanvas);
}
```

- 9.** Substitua essa instrução pelo código que define as propriedades *X* e *Y* do objeto *Square*, como mostrado em negrito no código a seguir:

```
if (mySquare is IDraw)
{
    IDraw drawSquare = mySquare;
    drawSquare.X = (int)mouseLocation.X;
    drawSquare.Y = (int)mouseLocation.Y;
    drawSquare.Draw(drawingCanvas);
}
```

- 10.** Se estiver usando o Windows 8.1, localize o método *drawingCanvas_RightTapped*. Se estiver usando o Windows 7 ou o Windows 8, localize o método *drawingCanvas_MouseRightButtonDown*.

Esses métodos são executados quando o usuário toca e continua tocando na tela ou clica com o botão direito do mouse, e desenham um círculo na tela.

- 11.** Nesse método, substitua a instrução que chama o método *SetLocation* do objeto *Circle* e, em vez disso, defina as propriedades *X* e *Y* como mostrado em negrito no exemplo a seguir:

```
if (myCircle is IDraw)
{
    IDraw drawCircle = myCircle;
    drawCircle.X = (int)mouseLocation.X;
    drawCircle.Y = (int)mouseLocation.Y;
    drawCircle.Draw(drawingCanvas);
}
```

- 12.** Abra o arquivo *IColor.cs* do projeto *Drawing* na janela Code and Text Editor. Essa interface especifica o método *SetColor*, como segue:

```
interface IColor
{
    void SetColor(Color color);
}
```

- 13.** Exclua esse método e substitua-o pela definição de uma propriedade chamada *Color*, conforme apresentado aqui:

```
interface IColor
{
    Color Color { set; }
}
```

Essa é uma propriedade somente-gravação, já que fornece um método de acesso *set*, mas nenhum método de acesso *get*. Isso porque a cor não é armazenada na classe *DrawingShape*, sendo especificada apenas quando cada forma é desenhada; você não pode consultar uma forma para saber qual é sua cor.



Nota É uma prática comum uma propriedade compartilhar o mesmo nome com um tipo (*Color* nesse exemplo).

- 14.** Retorne à classe *DrawingShape* na janela Code and Text Editor. Substitua o método *SetColor* nessa classe pela propriedade *Color* mostrada aqui:

```
public Color Color
{
    set
    {
        if (this.shape != null)
        {
            SolidColorBrush brush = new SolidColorBrush(value);
            this.shape.Fill = brush;
        }
    }
}
```



Dica O código do método de acesso *set* é quase idêntico ao do método *SetColor* original, exceto que a instrução que cria o objeto *SolidColorBrush* recebe o parâmetro *value*.

- 15.** Retorne ao arquivo *DrawingPad.xaml.cs* na janela Code and Text Editor. No método *drawingCanvas_Tapped* (Windows 8.1) ou no método *drawingCanvas_MouseLeftButtonDown* (Windows 7 ou Windows 8), modifique a instrução que define a cor do objeto *Square*, de acordo com o código em negrito a seguir:

```
if (mySquare is IColor)
{
    IColor colorSquare = mySquare;
    colorSquare.Color = Colors.BlueViolet;
}
```

- 16.** Da mesma forma, no método *drawingCanvas_RightTapped* (Windows 8.1) ou no método *drawingCanvas_MouseRightButtonDown* (Windows 7 ou Windows 8), modifique a instrução que define a cor do objeto *Circle*.

```
if (myCircle is IColor)
{
    IColor colorCircle = myCircle;
    colorCircle.Color = Colors.HotPink;
}
```

- 17.** No menu Debug, clique em Start Debugging para compilar e executar o projeto.

- 18.** Verifique que o aplicativo funciona da mesma maneira que antes. Se você tocar na tela ou clicar com o botão esquerdo do mouse no canvas, o aplicativo deverá desenhar um quadrado, e se tocar e continuar tocando ou clicar com o botão direito do mouse, o aplicativo deverá desenhar um círculo. A imagem a seguir mostra o aplicativo em execução no Windows 8.1.

014 001

038 001

Drawing Pad



- 19.** Retorne ao ambiente de programação do Visual Studio 2013 e interrompa a depuração.

Como gerar propriedades automáticas

Conforme já mencionado neste capítulo, o principal propósito das propriedades é ocultar a implementação dos campos. Isso é bom se suas propriedades realizam algum trabalho útil, mas se os métodos de acesso *get* e *set* simplesmente envolvem operações que apenas leem ou atribuem um valor a um campo, você poderia questionar o valor dessa estratégia. Contudo, há pelo menos duas boas razões para você definir propriedades em vez de expor dados como campos públicos, mesmo nessas situações:

■ Compatibilidade com aplicativos Os campos e as propriedades se expõem utilizando diferentes metadados nos assemblies. Se você desenvolver uma classe e decidir utilizar os campos públicos, qualquer aplicativo que usar essa classe referenciará esses itens como campos. Embora seja possível empregar a mesma sintaxe C# para ler e gravar um campo utilizado para ler e gravar uma propriedade, o código compilado é bem diferente – o compilador C# só oculta as diferenças. Se você mais tarde decidir que precisa transformar esses campos em propriedades (talvez os requisitos do negócio tenham mudado e você precise executar uma lógica adicional ao atribuir valores), os aplicativos existentes não serão capazes de utilizar a versão atualizada da classe sem uma recompilação. Isso é complicado se você instalou o aplicativo em um grande número de desktops dos usuários por toda uma organização. Há maneiras de contornar isso, mas, em geral, é melhor evitar essa situação desde o início.

■ **Compatibilidade com interfaces** Se estiver implementando uma interface e ela definir um item como uma propriedade, você deverá escrever uma propriedade que corresponda à especificação na interface, mesmo se a propriedade apenas ler e gravar os dados em um campo privado. Você não pode implementar uma propriedade simplesmente expondo um campo público com o mesmo nome.

Os projetistas da linguagem C# perceberam que os programadores são pessoas atarefadas que não devem desperdiçar tempo escrevendo mais código do que o necessário. Por isso, o compilador C# pode gerar o código para propriedades automaticamente, desta maneira:

```
class Circle
{
    public int Radius{ get; set; }
    ...
}
```

Nesse exemplo, a classe *Circle* contém uma propriedade chamada *Radius*. Além do tipo, você não especificou como essa propriedade funciona – os métodos de acesso *get* e *set* estão vazios. O compilador C# converte essa definição em um campo privado e em uma implementação padrão que se parece com esta:

```
class Circle
{
    private int _radius;
    public int Radius{
        get
        {
            return this._radius;
        }
        set
        {
            this._radius = value;
        }
    ...
}
```

Portanto, com o mínimo de esforço, você pode implementar uma propriedade simples utilizando um código gerado automaticamente; se precisar incluir uma lógica adicional posteriormente, você poderá fazer isso sem estragar aplicativos existentes. Você deve observar, porém, que com uma propriedade automaticamente gerada é necessário especificar um método de acesso *get* e um método de acesso *set* – uma propriedade automática não pode ser somente-leitura ou somente-gravação.



Nota A sintaxe para definir uma propriedade automática é quase idêntica à sintaxe para definir uma propriedade em uma interface. A exceção é que uma propriedade automática pode especificar um modificador de acesso, como *private*, *public* ou *protected*.

Com essa estratégia é possível simular propriedades somente-leitura e somente-gravação automáticas, marcando o método de acesso *get* ou *set* como privado. No entanto, essa estratégia é considerada uma péssima prática de programação, pois pode introduzir erros sutis no seu código. Portanto, optei por não mostrar um exemplo!

Como inicializar objetos com propriedades

No Capítulo 7, você aprendeu a definir construtores para inicializar um objeto. Um objeto pode ter diversos construtores e você pode definir construtores com parâmetros variados para inicializar diferentes elementos em um objeto. Por exemplo, você poderia definir uma classe que modela um triângulo desta maneira:

```
public class Triangle
{
    private int side1Length;
    private int side2Length;
    private int side3Length;

    // construtor padrão - valores padrão para todos os lados
    public Triangle()
    {
        this.side1Length = this.side2Length = this.side3Length = 10;
    }
    // especifica o comprimento para side1Length, valores padrão para os outros
    public Triangle(int length1)
    {
        this.side1Length = length1;
        this.side2Length = this.side3Length = 10;
    }

    // especifica o comprimento para side1Length e side2Length,
    // valor padrão para side3Length
    public Triangle(int length1, int length2)
    {
        this.side1Length = length1;
        this.side2Length = length2;
        this.side3Length = 10;
    }

    // especifica o comprimento para todos os lados
    public Triangle(int length1, int length2, int length3)
    {
        this.side1Length = length1;
        this.side2Length = length2;
        this.side3Length = length3;
    }
}
```

Dependendo de quantos campos uma classe contém e das várias combinações que você quer permitir para inicializar os campos, talvez seja necessário escrever vários construtores. Também há possíveis problemas se muitos dos campos tiverem o mesmo tipo: talvez você não seja capaz de escrever um construtor único para todas as combinações dos campos. Por exemplo, na classe *Triangle* anterior, você não poderia adicionar facilmente um construtor que só inicializasse os campos *side1Length* e *side3Length*, porque ele não teria uma assinatura única; receberia dois parâmetros *int*, e o construtor que inicializa *side1Length* e *side2Length* já tem essa assinatura. Uma possível solução é definir um construtor que aceite parâmetros opcionais e especificar valores para os parâmetros como argumentos nomeados quando você criar um objeto *Triangle*. Entretanto, uma solução mais eficiente e transparente é inicializar os campos privados com um conjunto de valores padrão e expô-los como propriedades, como demonstrado a seguir:

```

public class Triangle
{
    private int side1Length = 10;
    private int side2Length = 10;
    private int side3Length = 10;

    public int Side1Length
    {
        set { this.side1Length = value; }
    }

    public int Side2Length
    {
        set { this.side2Length = value; }
    }

    public int Side3Length
    {
        set { this.side3Length = value; }
    }
}

```

Ao criar uma instância de uma classe, você pode inicializá-la especificando os nomes e os valores para qualquer propriedade pública que tenha métodos de acesso *set*. Por exemplo, é possível criar objetos *Triangle* e inicializar qualquer combinação dos três lados, desta maneira:

```

Triangle tri1 = new Triangle { Side3Length = 15 };
Triangle tri2 = new Triangle { Side1Length = 15, Side3Length = 20 };
Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };
Triangle tri4 = new Triangle { Side1Length = 9, Side2Length = 12,
                             Side3Length = 15 };

```

Essa sintaxe é conhecida como *inicializador de objeto*. Quando você chama um inicializador de objeto utilizando essa sintaxe, o compilador C# gera o código que chama o construtor padrão e então chama o método de acesso *set* de cada propriedade identificada para inicializá-la com o valor especificado. Você também pode especificar inicializadores de objeto em combinação com construtores não padrão. Por exemplo, se a classe *Triangle* também fornecer um construtor que recebeu um único parâmetro string descrevendo o tipo de triângulo, você poderá chamar esse construtor e inicializar as outras propriedades desta maneira:

```

Triangle tri5 = new Triangle("Equilateral triangle") { Side1Length = 3,
                                                       Side2Length = 3,
                                                       Side3Length = 3 };

```

O mais importante a lembrar é que o construtor executa primeiro, e as propriedades são configuradas subsequentemente. Entender essa sequência é importante se o construtor configurar os campos em um objeto com valores específicos e se as propriedades que você especifica mudarem esses valores.

Você também pode utilizar inicializadores de objeto com propriedades automáticas, como veremos no próximo exercício. Neste exercício, você definirá uma classe para modelar polígonos regulares que contêm propriedades automáticas, para fornecer acesso a informações sobre o número de lados do polígono e o comprimento desses lados.

Defina propriedades automáticas e utilize inicializadores de objeto

- No Visual Studio 2013, abra o projeto AutomaticProperties, localizado na pasta \Microsoft Press\ Visual CSharp Step by Step\Chapter 15\Windows X\Extension-Method na sua pasta Documentos.

O projeto AutomaticProperties contém o arquivo de Program.cs que define a classe *Program* com os métodos *Main* e *doWork* que vimos nos exercícios anteriores.

- No Solution Explorer, clique com o botão direito do mouse no projeto AutomaticProperties, aponte para Add e clique em Class para abrir a caixa de diálogo Add New Item – AutomaticProperties. Na caixa Name, digite **Polygon.cs** e clique em Add.

O arquivo Polygon.cs, contendo a classe *Polygon*, é criado e adicionado ao projeto e aparece na janela Code and Text Editor.

- Adicione à classe *Polygon* as propriedades automáticas *NumSides* e *SideLength*, mostradas em negrito:

```
class Polygon
{
    public int NumSides { get; set; }
    public double SideLength { get; set; }
}
```

- Adicione o seguinte construtor padrão, mostrado em negrito, à classe *Polygon*:

```
class Polygon
{
    ...
    public Polygon()
    {
        this.NumSides = 4;
        this.SideLength = 10.0;
    }
}
```

Esse construtor inicializa os campos *NumSides* e *SideLength* com valores padrão. Neste exercício, o polígono padrão é um quadrado com lados de comprimento de 10 unidades.

- Exiba o arquivo Program.cs na janela Code and Text Editor.
- Adicione ao método *doWork* as instruções mostradas aqui em negrito, substituindo o comentário *// TODO:*:

```
static void doWork()
{
    Polygon square = new Polygon();
    Polygon triangle = new Polygon { NumSides = 3 };
    Polygon pentagon = new Polygon { SideLength = 15.5, NumSides = 5 };
}
```

Essas instruções criam objetos *Polygon*. A variável *square* é inicializada pelo construtor padrão. As variáveis *triangle* e *pentagon* também são inicializadas através do construtor padrão e esse código muda o valor das propriedades expostas pela classe *Polygon*. No caso da variável *triangle*, a propriedade *NumSides* é configurada como 3, mas a propriedade *SideLength* permanece no valor padrão 10.0. Para a variável *pentagon*, o código altera os valores das propriedades *SideLength* e *NumSides*.

7. Adicione o seguinte código, mostrado em negrito, ao final do método *doWork*:

```
static void doWork()
{
    ...
    Console.WriteLine("Square: number of sides is {0}, length of each side is {1}",
        square.NumSides, square.SideLength);
    Console.WriteLine("Triangle: number of sides is {0}, length of each side is {1}",
        triangle.NumSides, triangle.SideLength);
    Console.WriteLine("Pentagon: number of sides is {0}, length of each side is {1}",
        pentagon.NumSides, pentagon.SideLength);
}
```

Essas instruções exibem os valores das propriedades *NumSides* e *SideLength* para cada objeto *Polygon*.

8. No menu Debug, clique em Start Without Debugging.

Verifique que o programa compila e executa, escrevendo na janela de console as mensagens mostradas aqui:

```
C:\Windows\system32\cmd.exe
Square: number of sides is 4, length of each side is 10
Triangle: number of sides is 3, length of each side is 10
Pentagon: number of sides is 5, length of each side is 15.5
Press any key to continue . . .
```

9. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2013.

Resumo

Neste capítulo, vimos como criar e utilizar propriedades para fornecer acesso controlado aos dados em um objeto. Examinamos também a criação de propriedades automáticas e o uso de propriedades ao inicializar objetos.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 16, “Indexadores”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar uma propriedade leitura/gravação para uma estrutura ou classe	Declare o tipo da propriedade, seu nome, um método de acesso get e um método de acesso set. Por exemplo:
	<pre>struct ScreenPosition { ... public int X { get { ... } set { ... } } ... }</pre>
Declarar uma propriedade somente-leitura para uma estrutura ou classe	Declare uma propriedade apenas com um método de acesso get. Por exemplo:
	<pre>struct ScreenPosition { ... public int X { get { ... } } ... }</pre>
Declarar uma propriedade somente-gravação para uma estrutura ou classe	Declare uma propriedade apenas com um método de acesso set. Por exemplo:
	<pre>struct ScreenPosition { ... public int X { set { ... } } ... }</pre>
Declarar uma propriedade em uma interface	Declare uma propriedade apenas com a palavra-chave get ou set, ou ambas. Por exemplo:
	<pre>interface IScreenPosition { int X { get; set; } // nenhum corpo int Y { get; set; } // nenhum corpo }</pre>

Implementar uma propriedade de interface em uma estrutura ou em uma classe

Na classe ou estrutura que implementa a interface, declare a propriedade e implemente os métodos de acesso. Por exemplo:

```
struct ScreenPosition : IScreenPosition
{
    public int X
    {
        get { ... }
        set { ... }
    }

    public int Y
    {
        get { ... }
        set { ... }
    }
}
```

Criar uma propriedade automática

Na classe ou estrutura que contém a propriedade, defina a propriedade com métodos de acesso get e set vazios. Por exemplo:

```
class Polygon
{
    public int NumSides { get; set; }
}
```

Usar propriedades para inicializar um objeto

Ao construir o objeto, especifique as propriedades e seus valores como uma lista dentro de chaves. Por exemplo:

```
Triangle tri3 =
    new Triangle { Side2Length = 12,
    Side3Length = 17 };
```

CAPÍTULO 16

Indexadores

Neste capítulo, você vai aprender a:

- Encapsular o acesso a um objeto com lógica de arrays utilizando indexadores.
- Controlar o acesso de leitura a indexadores declarando métodos de acesso *get*.
- Controlar o acesso de gravação a indexadores declarando métodos de acesso *set*.
- Criar interfaces que declaram indexadores.
- Implementar indexadores em estruturas e classes que herdam de interfaces.

O capítulo 15, “Implementação de propriedades para acessar campos”, descreve a implementação e o uso das propriedades como um meio de fornecer acesso controlado aos campos em uma classe. As propriedades são úteis para espelhar campos com um valor único. Já os indexadores são inestimáveis caso você queira fornecer acesso aos itens com múltiplos valores utilizando uma sintaxe natural e familiar.

O que é um indexador?

Considere um *indexador* um array inteligente quase como você considera uma propriedade um campo inteligente. Enquanto uma propriedade encapsula um único valor em uma classe, um indexador encapsula um conjunto de valores. A sintaxe utilizada para um indexador é a mesma utilizada para um array.

A melhor maneira de entender indexadores é trabalhar com um exemplo. Primeiro, consideraremos um problema e examinaremos uma solução que não utiliza indexadores. Em seguida, trabalharemos no mesmo problema e examinaremos uma solução melhor que utiliza indexadores. O problema diz respeito aos inteiros, ou, mais precisamente, ao tipo *int*.

Um exemplo que não utiliza indexadores

Normalmente, você utiliza um tipo *int* para armazenar um valor inteiro. Internamente, um *int* armazena o valor como uma sequência de 32 bits, em que cada bit pode ser 0 ou 1. Na maioria das vezes, você não se preocupa com essa representação binária interna; simplesmente utiliza um tipo *int* como um contêiner que armazena um valor inteiro. Às vezes, porém, os programadores empregam o tipo *int* para outros propósitos – alguns programas usam um *int* como um conjunto de flags binários e manipulam os bits individuais dentro de um *int*. Se você for um hacker em C, antigo como eu, o que vem a seguir deve ser bastante familiar!



Nota Alguns programas mais antigos podem usar tipos *int* para tentar economizar memória. Esses programas em geral remontam à época em que o tamanho da memória do computador era medido em kilobytes, em vez dos gigabytes disponíveis hoje, e a memória era extremamente escassa. Um único *int* armazena 32 bits, cada um dos quais pode ser 1 ou 0. Em alguns casos, os programadores atribuíam 1 para indicar o valor *true* e 0 para indicar *false*, e então empregavam um *int* como um conjunto de valores booleanos.

O C# dispõe de um conjunto de operadores para acessar e manipular os bits individuais em um *int*. Esses operadores são os seguintes:

- **O operador NOT (~)** É um operador unário que executa um complemento de bit a bit. Por exemplo, se pegar o valor de 8 bits *11001100* (204 em decimal) e aplicar o operador *~* a ele, o resultado será *00110011* (51 em decimal) – todos os 1s no valor original tornam-se 0s, e todos os 0s tornam-se 1s.



Nota Os exemplos mostrados aqui são puramente ilustrativos e sua precisão é apenas para 8 bits. No C#, o tipo *int* tem 32 bits; portanto, se você experimentar qualquer um desses exemplos em um aplicativo C#, obterá um resultado de 32 bits que pode ser diferente dos mostrados nesta lista. Por exemplo, em 32 bits, 204 é *00000000000000000000000011001100*, de modo que, no C#, *~204* é *111111111111111111111100110011* (que é a representação *int* de -205 no C#).

- **O operador de deslocamento para a esquerda (<<)** É um operador binário que realiza um deslocamento para a esquerda. A expressão *204 << 2* retorna o valor 48. (Em binário, o valor decimal 204 é *11001100* e, deslocando-o duas casas para a esquerda, o resultado é *00110000* ou 48 em decimal.) Os bits mais à esquerda são descartados e zeros são introduzidos à direita. Há um operador de deslocamento para a direita correspondente, *>>*.
- **O operador OR (|)** É um operador binário que realiza uma operação OR de bit a bit, retornando um valor que contém um 1 em cada posição em que um dos operandos tem um 1. Por exemplo, a expressão *204 | 24* tem o valor 220 (204 é *11001100*, 24 é *00011000* e 220 é *11011100*).

■ **O operador AND (&)** Executa uma operação AND de bit a bit. AND é semelhante ao operador OR de bit a bit, exceto por retornar um valor contendo um 1 em cada posição onde os dois operandos têm um 1. Portanto, $204 \& 24$ é 8 (204 é 11001100 , 24 é 00011000 e 8 é 00001000).

■ **O operador XOR (^)** Executa uma operação OR exclusiva de bit a bit, retornando um número 1 em cada bit onde há um número 1 em um ou outro operando, mas não em ambos. (Dois 1s resultam um 0 – essa é a parte “exclusiva” do operador.) Portanto, $204 ^ 24$ é 212 ($11001100 ^ 00011000$ é 11010100).

Você pode utilizar esses operadores em conjunto para calcular os valores dos bits individuais em um *int*. Por exemplo, a expressão a seguir emprega os operadores de deslocamento para a esquerda ($<<$) e o operador AND ($\&$) de bit a bit para determinar se o sexto bit a partir da direita da variável *byte* chamada *bits* está definido com 0 ou 1:

```
(bits & (1 << 5)) != 0
```



Nota Os operadores de bit a bit contam as posições dos bits da direita para a esquerda, e os bits são numerados a partir de 0. Assim, o bit 0 é o bit posicionado mais à direita, e o bit na posição 5 é o bit posicionado seis casas a partir da direita.

Suponha que a variável *bits* contém o valor decimal 42. Em binário, esse valor é 00101010. O valor decimal 1 é 00000001 em binário, e a expressão $1 << 5$ tem o valor 00100000; o sexto bit é 1. Em notação binária, a expressão *bits* $\&$ $(1 << 5)$ é 00101010 $\&$ 00100000, e o valor dessa expressão é o binário 00100000, que é não zero. Se a variável *bits* contiver o valor 65, ou 01000001 em binário, o valor da expressão será 01000001 $\&$ 00100000, que dá o resultado binário de 00000000 ou zero.

Esse é um exemplo relativamente complexo, mas simples, quando comparado com a seguinte expressão, que utiliza o operador de atribuição composta $\&=$ para definir o bit na posição 6 como 0:

```
bits &= ~(1 << 5)
```

De modo semelhante, para definir o bit na posição 6 como 1, você pode utilizar um operador OR ($|$) de bit a bit. A seguinte expressão complexa se baseia no operador de atribuição composta $|=$:

```
bits |= (1 << 5)
```

O problema em relação a esses exemplos é o fato de que, embora funcionem, eles são extremamente difíceis de serem entendidos. São complicados e a solução é de baixo nível: ela não cria uma abstração do problema que soluciona e, consequentemente, é muito difícil manter código que efetua esses tipos de operações.

O mesmo exemplo utilizando indexadores

Vamos deixar de lado a solução fraca por um momento para lembrar qual é o problema. Gostaríamos de utilizar um *int* não como um *int* (inteiro), mas como um array de bits. Portanto, a melhor maneira de resolver esse problema é utilizar um *int* como se fosse um array de bits; em outras palavras, o que gostaríamos de escrever para acessar o bit 6 casas a partir da direita na variável *bits* é uma expressão como a seguinte (lembre-se de que os arrays começam no índice 0):

```
bits[5]
```

E para definir o bit 4 casas a partir da direita como *true*, gostaríamos de escrever:

```
bits[3] = true
```



Nota Para os desenvolvedores experientes em C, o valor booleano *true* é sí-nônimo do valor binário 1 e o valor booleano *false* equivale ao valor binário 0. Consequentemente, a expressão *bits[3] = true* significa “Definir o bit 4 casas a partir da direita da variável *bits* como 1”.

Infelizmente, você não pode utilizar a notação de colchetes em um *int*; ela só funciona em um array ou em um tipo que se comporta como tal. Portanto, a solução para o problema é criar um novo tipo que funcione e seja utilizado como um array de variáveis *bool*, mas que seja implementado por meio de um *int*. Você pode realizar essa façanha definindo um indexador. Vamos chamar esse novo tipo de *IntBits*. *IntBits* conterá um valor *int* (initializado no seu construtor), mas a ideia é que utilizaremos *IntBits* como um array de variáveis *bool*.



Dica O tipo *IntBits* é menor e mais leve; portanto, faz sentido criá-lo como uma estrutura em vez de como uma classe.

```
struct IntBits
{
    private int bits;

    public IntBits(int initialValue)
    {
        bits = initialValue;
    }

    // indexador a ser escrito aqui
}
```

Para definir o indexador, você utiliza uma notação que é um cruzamento entre uma propriedade e um array. Introduza o indexador com a palavra-chave *this*, especifique o tipo do valor retornado pelo indexador e o tipo do valor a ser utilizado como o índice para o indexador, entre colchetes. O indexador da estrutura *IntBits* emprega um inteiro como tipo do argumento *index* e retorna um booleano. Ele se parece com este:

```

struct IntBits
{
    ...
    public bool this [ int index ]
    {
        get
        {
            return (bits & (1 << index)) != 0;
        }

        set
        {
            if (value) // ativa o bit se value for verdadeiro; caso contrário, o desativa
                bits |= (1 << index);
            else
                bits &= ~(1 << index);
        }
    }
}

```

Observe as seguintes considerações:

- Um indexador não é um método; não há parênteses contendo um parâmetro, mas há colchetes que especificam um índice. Esse índice é utilizado para especificar que elemento está sendo acessado.
- Todos os indexadores utilizam a palavra-chave *this*. Uma classe ou uma estrutura pode definir no máximo um indexador (embora seja possível sobrecarregá-lo e ter várias implementações), e seu nome é sempre *this*.
- Os indexadores contêm métodos de acesso *get* e *set* exatamente como as propriedades. Nesse exemplo, os métodos de acesso *get* e *set* contêm as expressões bit a bit complexas já discutidas.
- O índice especificado na declaração do indexador é preenchido com o valor do índice especificado quando o indexador é chamado. Os métodos de acesso *get* e *set* podem ler esse argumento para determinar qual elemento será acessado.



Nota Você deve fazer uma verificação de intervalo no valor do índice para evitar que exceções inesperadas ocorram no código do indexador.

Depois de declarar o indexador, você pode utilizar uma variável do tipo *IntBits* em vez de um *int* e aplicar a notação de colchetes, como mostrado no próximo exemplo:

```

int adapted = 126;      // 126 tem a representação binária 01111110
IntBits bits = new IntBits(adapted);
bool peek = bits[6];   // recupera bool no índice 6; deve ser true (1)
bits[0] = true;       // ativa o bit no índice 0 como true (1)
bits[3] = false;      // ativa o bit no índice 3 como false (0)
                        // o valor em bits agora é 01110111, ou 119 em decimal

```

Com certeza, essa sintaxe é muito mais fácil de entender. Ela captura direta e sucintamente a essência do problema.

Entenda os métodos de acesso do indexador

Quando você lê um indexador, o compilador traduz automaticamente seu código com lógica de arrays em uma chamada para o método de acesso *get* desse indexador. Considere o seguinte exemplo:

```
bool peek = bits[6];
```

Essa instrução é convertida em uma chamada ao método de acesso *get* para *bits*, e o argumento *index* é definido como 6.

Da mesma maneira, se você gravar em um indexador, o compilador traduzirá automaticamente seu código com lógica de array para uma chamada ao método de acesso *set* desse indexador, definindo o argumento *index* com o valor especificado entre colchetes, como ilustrado aqui:

```
bits[3] = true;
```

Essa instrução é convertida em uma chamada ao método de acesso *set* para *bits* onde *index* é 3. Assim como com propriedades normais, os dados gravados no indexador (nesse caso, *true*) tornam-se disponíveis dentro do método de acesso *set* utilizando-se a palavra-chave *value*. O tipo de *value* é o mesmo do próprio indexador (nesse caso, *bool*).

Também é possível utilizar um indexador em um contexto de leitura/gravação combinado. Nesse caso são utilizados os métodos de acesso *get* e *set*. Examine a próxima instrução, que utiliza o operador XOR (^) para inverter o valor do bit no índice 6 na variável *bits*:

```
bits[6] ^= true;
```

Esse código é automaticamente traduzido para:

```
bits[6] = bits[6] ^ true;
```

Esse código funciona porque o indexador declara ambos os métodos de acesso, *get* e *set*.



Nota Você pode declarar um indexador que contém apenas um método de acesso *get* (um indexador somente-leitura) ou apenas um método de acesso *set* (um método de acesso somente-gravação).

Compare indexadores e arrays

Quando você utiliza um indexador, a sintaxe é deliberadamente parecida com a de um array. Mas existem algumas diferenças importantes entre os indexadores e os arrays:

- Os indexadores podem utilizar subscriptos não numéricos, como uma string, conforme mostrado no exemplo a seguir, enquanto os arrays podem utilizar somente subscriptos inteiros.

```
public int this [ string name ] { ... } // OK
```

- Os indexadores podem ser sobre carregados (assim como os métodos), enquanto os arrays não podem.

```
public Name this [ PhoneNumber number ] { ... }
public PhoneNumber this [ Name name ] { ... }
```

- Os indexadores não podem ser utilizados como parâmetros *ref* ou *out*, enquanto os elementos do array podem.

```
IntBits bits;           // bits contém um indexador
Method(ref bits[1]); // erro de tempo de compilação
```

Propriedades, arrays e indexadores

Uma propriedade pode retornar um array, mas lembre-se de que os arrays são tipos-referência; portanto, expor um array como uma propriedade permite sobrescrever acidentalmente muitos dados. Examine a estrutura a seguir, que expõe uma propriedade array chamada *Data*:

```
struct Wrapper
{
    private int[] data;
    ...
    public int[] Data
    {
        get { return this.data; }
        set { this.data = value; }
    }
}
```

Agora, considere o código a seguir que utiliza essa propriedade:

```
Wrapper wrap = new Wrapper();
...
int[] myData = wrap.Data;
myData[0]++;
myData[1]++;
```

Isso parece bastante inócuo. Entretanto, como os arrays são tipos-referência, a variável *myData* referencia o mesmo objeto da variável privada *data* na estrutura *Wrapper*. Todas as alterações que você fizer nos elementos em *myData* serão feitas no array *data*; a instrução *myData[0]++* tem exatamente o mesmo efeito de *data[0]++*. Se não for essa a intenção, você deve utilizar o método *Clone* nos métodos de acesso *get* e *set* da propriedade *Data* para retornar uma cópia do array de dados ou criar uma cópia do valor que está sendo configurado, como mostrado no código a seguir. (O Capítulo 8, “Valores e referências”, apresentou o método *Clone* para copiar arrays.) Observe que o método *Clone* retorna um objeto, ao qual você deve aplicar um casting para um array de inteiros.

```
struct Wrapper
{
    private int[] data;
    ...
    public int[] Data
    {
        get { return this.data.Clone() as int[]; }
        set { this.data = value.Clone() as int[]; }
    }
}
```

Entretanto, essa estratégia pode tornar-se bem complicada e cara em termos de uso de memória. Os indexadores fornecem uma solução natural para esse problema – não expor o array inteiro como uma propriedade; simplesmente disponibilizar seus elementos individuais por meio de um indexador:

```
struct Wrapper
{
    private int[] data;
    ...
    public int this [int i]
    {
        get { return this.data[i]; }
        set { this.data[i] = value; }
    }
}
```

O código a seguir utiliza o indexador de maneira semelhante à propriedade mostrada anteriormente:

```
Wrapper wrap = new Wrapper();
...
int[] myData = new int[2];
myData[0] = wrap[0];
myData[1] = wrap[1];
myData[0]++;
myData[1]++;
```

Desta vez, incrementar os valores no array *MyData* não tem qualquer efeito sobre o array original no objeto *Wrapper*. Se quiser realmente modificar os dados no objeto *Wrapper*, você deve escrever instruções como esta:

```
wrap[0]++;

```

É muito mais claro e seguro!

Indexadores em interfaces

Você pode declarar indexadores em uma interface. Para fazer isso, especifique a palavra-chave *get*, a palavra-chave *set*, ou ambas, mas substitua o corpo do método *get* ou *set* por um ponto e vírgula. Toda classe ou estrutura que implementa a interface deve implementar os métodos de acesso *indexadores* declarados na interface, como demonstrado aqui:

```
interface IRawInt
{
    bool this [ int index ] { get; set; }
}
```

```
struct RawInt : IRawInt
{
    ...
    public bool this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Se você implementar o indexador da interface em uma classe, poderá declarar as implementações do indexador como *virtuais*. Isso permite que outras classes derivadas redefinam os métodos de acesso *get* e *set*, como no seguinte:

```
class RawInt : IRawInt
{
    ...
    public virtual bool this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Você também pode optar por implementar um indexador utilizando a sintaxe de implementação explícita abordada no Capítulo 13, “Como criar interfaces e definir classes abstratas”. Uma implementação explícita de um indexador é não pública e não virtual (e, portanto, não pode ser redefinida), como mostrado neste exemplo:

```
struct RawInt : IRawInt
{
    ...
    bool IRawInt.this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Indexadores em um aplicativo Windows

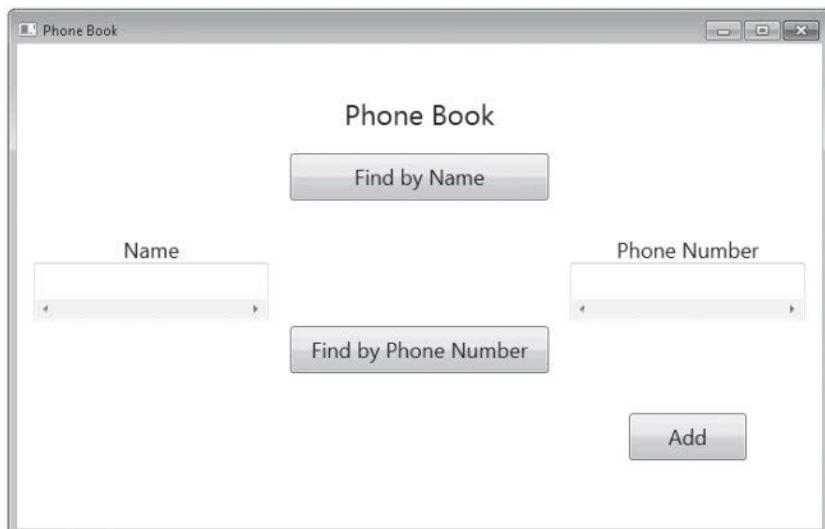
No exercício a seguir, você examinará um aplicativo de catálogo telefônico simples e completará sua implementação. Você escreverá dois indexadores na classe *PhoneBook*: um que aceita um parâmetro *Name* e retorna um *PhoneNumber* e outro que aceita um parâmetro *PhoneNumber* e retorna um *Name*. (As estruturas *Name* e *PhoneNumber* já foram escritas.) Você também precisará chamar esses indexadores nos locais corretos do programa.

Conheça o aplicativo

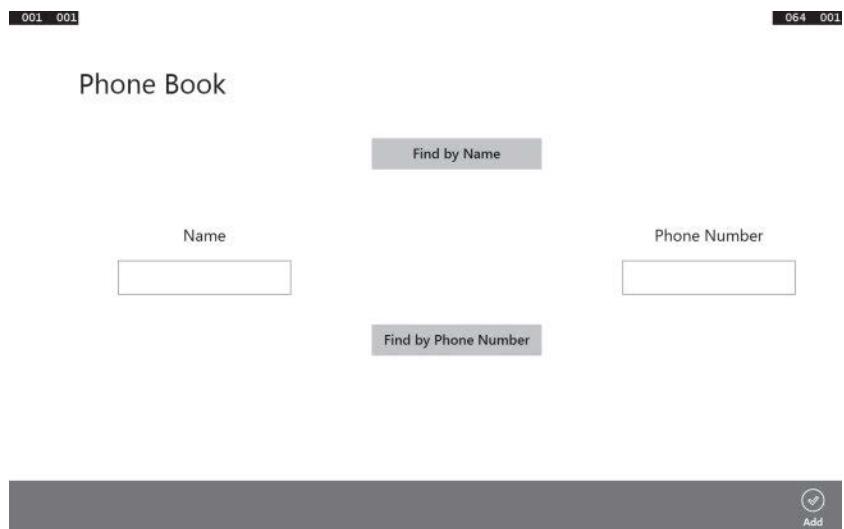
1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto Indexers, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 16\Windows X\Indexers na sua pasta Documentos.
Nesse aplicativo gráfico, o usuário pode procurar o número de telefone e também localizar o nome de um contato que corresponde a um número de telefone dado.
3. No menu Debug, clique em Start Debugging.

O projeto é compilado e executado. Aparece um formulário, exibindo duas caixas de texto vazias intituladas Name e Phone Number. O formulário também contém três botões: um para adicionar um par nome/número de telefone a uma lista de nomes e números de telefone armazenados pelo aplicativo; um para localizar um número de telefone quando um nome é dado; e um para localizar um nome quando é dado um número de telefone. Atualmente, esses botões não fazem coisa alguma.

Se você está usando o Windows 7 ou o Windows 8, o aplicativo aparece deste modo:



Se você está usando o Windows 8.1, o botão Add está localizado na barra de aplicativos, em vez de no formulário principal. Lembre-se de que, em um aplicativo Windows Store, você pode mostrar a barra de aplicativos clicando com o botão direito do mouse no formulário que exibe o aplicativo.



Sua tarefa é concluir o aplicativo para que os botões funcionem.

4. Retorne ao Visual Studio 2013 e interrompa a depuração.
5. Exiba o arquivo `Name.cs` do projeto `Indexers` na janela `Code and Text Editor`. Examine a estrutura `Name`. Seu propósito é atuar como um armazenador de nomes. O nome é fornecido como uma string para o construtor. O nome pode ser recuperado com a propriedade de string somente-leitura chamada `Text`. (Os métodos `Equals` e `GetHashCode` são utilizados para comparar `Names` durante a pesquisa em um array de valores `Name` – você pode ignorá-los por enquanto.)
6. Exiba o arquivo `PhoneNumber.cs` na janela `Code and Text Editor` e examine a estrutura `PhoneNumber`. Ela é semelhante à estrutura `Name`.
7. Exiba o arquivo `PhoneBook.cs` na janela `Code and Text Editor` e examine a classe `PhoneBook`.

Essa classe contém dois arrays privados: um array de valores `Name` chamado `names` e um array de valores `PhoneNumber` chamado `phoneNumbers`. A classe `PhoneBook` também contém um método `Add` que adiciona um número de telefone e um nome à agenda telefônica (phone book). Esse método é chamado quando o usuário clica no botão `Add` do formulário. O método `enlargeIfFull` é chamado por `Add` para verificar se os arrays estão cheios quando o usuário adiciona outra entrada. Esse método cria dois novos arrays maiores, copia o conteúdo dos arrays existentes para eles e então descarta os arrays antigos.

O método *Add* foi deliberadamente mantido simples e não verifica se um nome ou número de telefone já foi adicionado à agenda.

Atualmente, a classe *PhoneBook* não fornece uma funcionalidade com a qual o usuário possa encontrar um nome ou número de telefone; você vai adicionar dois indexadores para fornecer esse recurso no próximo exercício.

Escreva os indexadores

1. No arquivo *PhoneBook.cs*, exclua o comentário *// TODO: write 1st indexer here* e substitua-o por um indexador *public* somente-leitura para a classe *PhoneBook*, como mostrado em negrito no código a seguir. O indexador deve retornar *Name* e obter um item *PhoneNumber* como índice. Deixe o corpo do método de acesso *get* em branco.

O indexador deve ser semelhante a este:

```
sealed class PhoneBook
{
    ...
    public Name this[PhoneNumber number]
    {
        get
        {
        }
    }
    ...
}
```

2. Implemente o método de acesso *get* como mostrado em negrito no código a seguir.

A finalidade do método de acesso é localizar o nome que corresponde ao número de telefone especificado. Para fazer isso, chame o método estático *IndexOf* da classe *Array*. O método *IndexOf* executa uma pesquisa em um array, retornando o índice do primeiro item no array correspondente ao valor especificado. O primeiro argumento para *IndexOf* é o array a ser pesquisado (*phoneNumbers*). O segundo argumento para *IndexOf* é o item que você está pesquisando. *IndexOf* retorna o índice inteiro do elemento, se o encontrar; caso contrário, *IndexOf* retorna *-1*. Se o indexador encontrar o número de telefone, deve retornar o nome correspondente; caso contrário, deve retornar um valor *Name* vazio. (Observe que *Name* é uma estrutura; portanto, o construtor padrão define seu campo *name* privado como *null*.)

```
sealed class PhoneBook
{
    ...
    public Name this [PhoneNumber number]
    {
        get
        {
            int i = Array.IndexOf(this.phoneNumbers, number);
            if (i != -1)
                return new Name(phoneNumbers[i].name);
            else
                return new Name();
        }
    }
}
```

```

    {
        return this.names[i];
    }
    else
    {
        return new Name();
    }
}
...
}

```

- 3.** Remova o comentário `// TODO: write 2nd indexer here` e substitua-o por um segundo indexador `public` somente-leitura para a classe `PhoneBook` que retorna um `PhoneNumber` e aceita um único parâmetro `Name`. Implemente esse indexador da mesma maneira que o primeiro. (Observe mais uma vez que `PhoneNumber` é uma estrutura e, portanto, sempre tem um construtor padrão.)

O segundo indexador deve ser semelhante a este:

```

sealed class PhoneBook
{
    ...
    public PhoneNumber this [Name name]
    {
        get
        {
            int i = Array.IndexOf(this.names, name);
            if (i != -1)
            {
                return this.phoneNumbers[i];
            }
            else
            {
                return new PhoneNumber();
            }
        }
    }
    ...
}

```

Observe também que esses indexadores sobrecarregados podem coexistir porque os valores que indexam são de tipos diferentes, ou seja, suas assinaturas são diferentes. Se as estruturas `Name` e `PhoneNumber` fossem substituídas por strings simples (que elas encapsulam), as sobrecargas teriam a mesma assinatura, e a classe não compilaria.

- 4.** No menu Build, clique em Build Solution, corrija qualquer erro de sintaxe e recompile, se necessário.

Chame os indexadores

1. Exiba o arquivo MainWindow.xaml.cs na janela Code and Text Editor e então localize o método *findByNameClick*.

Esse método é chamado quando o botão Find By Name é clicado. Esse método estará vazio. Substitua o comentário *//TODO:* pelo código mostrado em negrito no exemplo a seguir. Esse código executa as seguintes tarefas:

- a. Lê o valor da propriedade *Text* na caixa de texto *name* do formulário. Isso é uma string contendo o nome de contato digitado pelo usuário.
- b. Se a string não estiver vazia, procura o número de telefone correspondente a esse nome no *PhoneBook*, utilizando o indexador. (Observe que a classe *MainWindow* contém um campo privado *PhoneBook* chamado *phoneBook*.) Constrói um objeto *Name* a partir da string e passa-o como o parâmetro para o indexador *PhoneBook*.
- c. Se a propriedade *Text* da estrutura *PhoneNumber* retornada pelo indexador não for *null* ou vazia, escreve o valor dessa propriedade na caixa de texto *phoneNumber* do formulário; caso contrário, exibe o texto "Not Found".

O método *findByNameClick* deve ser semelhante a este:

```
private void findByNameClick(object sender, RoutedEventArgs e)
{
    string text = name.Text;
    if (!String.IsNullOrEmpty(text))
    {
        Name personsName = new Name(text);
        PhoneNumber personsPhoneNumber = this.phoneBook[personsName];
        phoneNumber.Text = String.IsNullOrEmpty(personsPhoneNumber.Text) ?
            "Not Found" : personsPhoneNumber.Text;
    }
}
```

Além da instrução que acessa o indexador, há mais dois pontos de interesse nesse código:

- d. O método estático *String.IsNullOrEmpty* é utilizado para determinar se uma string está vazia ou contém um valor *null*. Esse é o método preferido para testar se uma string contém um valor. Ele retorna *true* se a string tiver um valor nulo ou for a string vazia; caso contrário, retorna *false*.
- e. O operador *:* utilizado pela instrução que preenche a propriedade *Text* da caixa de texto *phoneNumber* no formulário atua como uma instrução *if...else* em linha para uma expressão. Ele é um operador ternário que recebe os três operandos a seguir: uma expressão booleana, uma expressão para avaliar e retornar se a expressão booleana for verdadeira e outra expressão para avaliar e retornar se a expressão booleana for falsa. No código anterior, se a expressão *String.IsNullOrEmpty(personsPhoneNumber.Text)* for verdadeira,

não foi encontrada uma entrada correspondente na agenda e o texto "Not Found" aparecerá no formulário; caso contrário, será exibido o valor armazenado na propriedade *Text* da variável *personsPhoneNumber*.

A forma geral do operador ?: é a seguinte:

```
Resultado = <Expressão booleana> ? <Avalia se for verdadeira> : <Avalia se for falsa>
```

- 2.** Localize o método *findByPhoneNumberClick* no arquivo *MainWindow.xaml.cs*. Ele está abaixo do método *findByNameClick*.

O método *findByPhoneNumberClick* é chamado quando o botão Find By Phone Number é clicado. Atualmente, esse método está vazio, exceto por um comentário // TODO:. Você precisa implementá-lo como segue (o código completo está mostrado em negrito no exemplo a seguir):

- a.** Leia o valor da propriedade *Text* a partir da caixa *phoneNumber* do formulário. Isso é uma string contendo o número de telefone digitado pelo usuário.
- b.** Se a string não estiver vazia, procure o nome correspondente a esse número de telefone no *PhoneBook*, utilizando o indexador.
- c.** Grave a propriedade *Text* da estrutura *Name* retornada pelo indexador na caixa *name* do formulário.

O método completo deve ser parecido com este:

```
private void findByPhoneNumberClick(object sender, RoutedEventArgs e)
{
    string text = phoneNumber.Text;
    if (!String.IsNullOrEmpty(text))
    {
        PhoneNumber personsPhoneNumber = new PhoneNumber(text);
        Name personsName = this.phoneBook[personsPhoneNumber];
        name.Text = String.IsNullOrEmpty(personsName.Text) ?
                    "Not Found" : personsName.Text;
    }
}
```

- 3.** No menu Build, clique em Build Solution e corrija qualquer erro que ocorra.

Teste o aplicativo

- 1.** No menu Debug, clique em Start Debugging.
- 2.** Digite seu nome e o número de telefone nas caixas apropriadas e então clique em Add. (Se estiver usando o Windows 8.1, lembre-se de que o botão Add está na barra de aplicativo.)

Quando você clica no botão Add, o método *Add* armazena as informações no catálogo de telefone e limpa as caixas de texto para que elas estejam prontas para executar uma pesquisa.

3. Repita o passo 2 várias vezes com alguns nomes e números de telefone diferentes, de modo que a agenda telefônica contenha uma seleção de entradas. O aplicativo não realiza verificação de nomes e números de telefone digitados e você pode inserir o mesmo nome e número de telefone mais de uma vez. Para os propósitos desta demonstração, a fim de evitar confusão, certifique-se de fornecer nomes e números de telefone diferentes.
4. Na caixa Name, digite um nome que você utilizou no passo 3 e então clique em Find By Name.

O número de telefone que você adicionou para esse contato no passo 3 é recuperado do catálogo telefônico e exibido na caixa de texto Phone Number.

5. Digite um número de telefone para um diferente contato na caixa Phone Number e então clique em Find By Phone Number.
6. Digite um nome que você não inseriu no catálogo telefônico na caixa Name e então clique em Find By Name.

Desta vez, a caixa Phone Number exibe a mensagem "Not Found".

7. Feche o formulário e retorne ao Visual Studio 2013.

Resumo

Neste capítulo, vimos como utilizar indexadores para fornecer acesso do tipo array aos dados em uma classe. Você aprendeu a criar indexadores que podem aceitar um índice e retornar o respectivo valor utilizando uma lógica definida pelo método de acesso *get*, e viu como utiliza o método de acesso *set* com um índice para preencher um valor em um indexador.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 17, "Genéricos".
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Criar um indexador para uma classe ou estrutura	<p>Declare o tipo do indexador, seguido pela palavra-chave <i>this</i>, e então os argumentos do indexador entre colchetes. O corpo do indexador pode conter um método de acesso <i>get</i> e/ou <i>set</i>. Por exemplo:</p> <pre>struct RawInt { ... public bool this [int index] { get { ... } set { ... } } ... }</pre>
Definir um indexador em uma interface	<p>Defina um indexador com as palavras-chave <i>get</i> e/ou <i>set</i>. Por exemplo:</p> <pre>interface IRawInt { bool this [int index] { get; set; } }</pre>
Implementar um indexador de uma interface em uma classe ou estrutura	<p>Na classe ou estrutura que implementa a interface, defina o indexador e implemente os métodos de acesso. Por exemplo:</p> <pre>struct RawInt : IRawInt { ... public bool this [int index] { get { ... } set { ... } } ... }</pre>
Implementar um indexador definido por uma interface utilizando implementação de interface explícita em uma classe ou estrutura	<p>Na classe ou estrutura que implementa a interface, especifique a interface, mas não especifique a acessibilidade do indexador. Por exemplo:</p> <pre>struct RawInt : IRawInt { ... bool IRawInt.this [int index] { get { ... } set { ... } } ... }</pre>

CAPÍTULO 17

Genéricos

Neste capítulo, você vai aprender a:

- Explicar o objetivo dos genéricos.
- Definir uma classe segura (type-safe) utilizando genéricos.
- Criar instâncias de uma classe genérica com base nos tipos especificados como parâmetros de tipo.
- Implementar uma interface genérica.
- Definir um método genérico que implementa um algoritmo independentemente do tipo de dados em que opera.

O Capítulo 8, “Valores e referências”, mostrou como empregar o tipo *object* para referenciar uma instância de qualquer classe. É possível usar o tipo *object* para armazenar um valor de qualquer tipo assim como também se pode definir parâmetros através do tipo *object*, quando for necessário passar valores de qualquer tipo para um método. Um método também pode retornar valores de qualquer tipo, especificando *object* como o tipo de retorno. Mesmo que essa prática seja muito flexível, deixa o programador responsável por lembrar quais tipos de dados estão sendo de fato utilizados. Isso pode levar a erros de tempo de execução, caso o programador cometa um engano. Neste capítulo, você vai aprender sobre os genéricos, recurso projetado para ajudá-lo a evitar esse tipo de erro.

O problema do tipo *object*

Para entender os genéricos, vale a pena examinar detalhadamente o problema para o qual foram projetados para resolver.

Suponha que você precisasse modelar uma estrutura do tipo primeiro a entrar, primeiro a sair (first-in, first-out), como uma fila. Você poderia criar uma classe como a seguinte:

```
class Queue
{
    private const int DEFAULTQUEUESIZE = 100;
    private int[] data;
    private int head = 0, tail = 0;
    private int numElements = 0;

    public Queue()
    {
        this.data = new int[DEFAULTQUEUESIZE];
    }
}
```

```

public Queue(int size)
{
    if (size > 0)
    {
        this.data = new int[size];
    }
    else
    {
        throw new ArgumentOutOfRangeException("size", "Must be greater than zero");
    }
}

public void Enqueue(int item)
{
    if (this.numElements == this.data.Length)
    {
        throw new Exception("Queue full");
    }

    this.data[this.head] = item;
    this.head++;
    this.head %= this.data.Length;
    this.numElements++;
}

public int Dequeue()
{
    if (this.numElements == 0)
    {
        throw new Exception("Queue empty");
    }

    int queueItem = this.data[this.tail];
    this.tail++;
    this.tail %= this.data.Length;
    this.numElements--;
    return queueItem;
}
}

```

Essa classe usa um array a fim de fornecer um buffer circular para armazenar os dados. O tamanho desse array é especificado pelo construtor. Um aplicativo utiliza o método *Enqueue* para adicionar um item à fila e o método *Dequeue* para extrair um item dela. Os campos privados *head* e *tail* monitoram onde um item vai ser inserido no array e de onde um item vai ser recuperado do array. O campo *numElements* indica quantos itens existem no array. Os métodos *Enqueue* e *Dequeue* utilizam esses campos para determinar onde armazenar ou de onde recuperar um item e realizam alguma verificação de erro rudimentar. Um aplicativo pode criar um objeto *Queue* e chamar esses métodos, como mostrado no exemplo de código a seguir. Observe que os itens são retirados da fila na mesma ordem em que são enfileirados:

```

Queue queue = new Queue(); // Cria um novo Queue
queue.Enqueue(100);
queue.Enqueue(-25);
queue.Enqueue(33);

```

```
Console.WriteLine("{0}", queue.Dequeue()); // Exibe 100
Console.WriteLine("{0}", queue.Dequeue()); // Exibe -25
Console.WriteLine("{0}", queue.Dequeue()); // Exibe 33
```

Agora, a classe *Queue* funciona bem para filas de *ints*, mas e se você quiser criar filas de strings ou floats ou mesmo filas de tipos mais complexos, como *Circle* (consulte o Capítulo 7, “Criação e gerenciamento classes e objetos”), *Horse* ou *Whale* (consulte o Capítulo 12, “Herança”)? O problema é que o modo como a classe *Queue* está implementada a restringe a itens de tipo *int*, e se você tentar enfileirar um *Horse*, obterá um erro de tempo de compilação.

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Erro de tempo de compilação: não pode converter de Horse para int
```

Uma maneira de contornar essa restrição é especificar que o array na classe *Queue* contém itens de tipo *object*, atualizar os construtores e modificar os métodos *Enqueue* e *Dequeue* para que recebam um parâmetro *object* e retornem um *object*, como no seguinte:

```
class Queue
{
    ...
    private object[] data;
    ...
    public Queue()
    {
        this.data = new object[DEFAULTQUEUESIZE];
    }

    public Queue(int size)
    {
        ...
        this.data = new object[size];
        ...
    }
    public void Enqueue(object item)
    {
        ...
    }

    public object Dequeue()
    {
        ...
        object queueItem = this.data[this.tail];
        ...
        return queueItem;
    }
}
```

Lembre-se de que é possível utilizar o tipo *object* para referenciar um valor ou uma variável de qualquer tipo. Todos os tipos-referência herdam automaticamente (direta ou indiretamente) da classe *System.Object* no Microsoft .NET Framework (no C#, *object* é um alias para *System.Object*). Agora, como os métodos *Enqueue* e *Dequeue* manipulam *objects*, você pode operar em filas de *Circles*, *Horses*, *Whales* ou

qualquer outra classe vista nos exercícios anteriores deste livro. Mas é importante observar que você tem de fazer o casting do valor retornado pelo método *Dequeue*, a fim de realizar a conversão para o tipo apropriado, porque o compilador não executará automaticamente a conversão para o tipo *object*.

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Agora válido - Horse é um object

...
Horse dequeuedHorse =(Horse)queue.Dequeue(); // Precisa fazer o casting do object de volta para Horse
```

Se não fizer o casting do valor retornado, você receberá o erro de compilador “*Cannot implicitly convert type ‘object’ to ‘Horse’.*” Esse requisito de fazer um casting explícito deteriora grande parte da flexibilidade propiciada pelo tipo *object*. Além disso, é muito fácil escrever um código como este:

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse);
...
Circle myCircle = (Circle)queue.Dequeue(); // erro de tempo de execução
```

Embora esse código seja compilado, ele não é válido e lança uma exceção *System.InvalidCastException* em tempo de execução. O erro é causado pela tentativa de armazenar uma referência a um *Horse* em uma variável *Circle* ao ser retirado da fila, e os dois tipos não são compatíveis. Esse erro só é identificado em tempo de execução porque o compilador não tem informações suficientes para realizar essa verificação em tempo de compilação. O tipo real do objeto sendo desenfileirado somente torna-se aparente quando o código executa.

Outra desvantagem de utilizar a estratégia *object* para criar classes e métodos generalizados é que ela pode consumir memória e tempo adicionais do processador, se o runtime precisar converter um *object* em um tipo-valor e vice-versa. Considere o seguinte fragmento de código, que manipula uma fila de valores *int*:

```
Queue queue = new Queue();
int myInt = 99;
queue.Enqueue(myInt);           // faz boxing do int para object
...
myInt = (int)queue.Dequeue(); // faz unboxing de object para int
```

O tipo de dado *Queue* espera que os itens que armazena sejam objetos, e *object* é um tipo-referência. Enfileirar um tipo-valor, como um *int*, requer que ele sofra boxing para convertê-lo em um tipo-referência. Da mesma maneira, remover da fila um *int* requer que o item sofra unboxing para convertê-lo novamente em um tipo-valor. Consulte as seções “Boxing” e “Unboxing”, no Capítulo 8, para obter mais detalhes. Embora os procedimentos de boxing e unboxing ocorram de forma transparente, eles adicionam uma sobrecarga ao desempenho porque envolvem alocações dinâmicas de memória. Essa sobrecarga é pequena para cada item, mas aumenta quando um programa cria filas de numerosos tipos-valor.

A solução dos genéricos

O C# fornece genéricos para eliminar a necessidade de casting, melhorar a segurança, reduzir a quantidade de boxing necessária e facilitar a criação de classes e métodos generalizados. Classes e métodos genéricos aceitam *parâmetros de tipo*, que especificam os tipos dos objetos em que operam. No C#, você indica que uma classe é genérica fornecendo um parâmetro de tipo entre sinais de maior e menor, deste modo:

```
class Queue<T>
{
    ...
}
```

O *T* nesse exemplo atua como um espaço reservado para um tipo real em tempo de compilação. Ao escrever código para instanciar uma *Queue* genérica, você fornece o tipo que deve ser substituído por *T* (*Circle*, *Horse*, *int* e assim por diante). Ao definir os campos e métodos na classe, você utiliza esse mesmo espaço reservado para indicar o tipo desses itens, como segue:

```
class Queue<T>
{
    ...
    private T[] data; // array é de tipo 'T', onde 'T' é o parâmetro de tipo
    ...
    public Queue()
    {
        this.data = new T[DEFAULTQUEUESIZE]; // usa 'T' como o tipo de dado
    }

    public Queue(int size)
    {
        ...
        this.data = new T[size];
        ...
    }
    public void Enqueue(T item) // usa 'T' como tipo do parâmetro do método
    {
        ...
    }

    public T Dequeue() // usa 'T' como tipo do valor de retorno
    {
        ...
        T queueItem = this.data[this.tail]; // o dado no array é de tipo 'T'
        ...
        return queueItem;
    }
}
```

O parâmetro de tipo, *T*, pode ser qualquer identificador válido do C#, embora o caractere *T* sozinho seja normalmente utilizado. Ele é substituído pelo tipo que você especifica ao criar um objeto *Queue*. Os exemplos a seguir criam um *Queue* de *ints* e um *Queue* de *Horses*:

```
Queue<int> intQueue = new Queue<int>();
Queue<Horse> horseQueue = new Queue<Horse>();
```

Além disso, o compilador agora tem informações suficientes para fazer uma verificação de tipo estrita quando você compilar o aplicativo. Não é mais necessário fazer o casting dos dados ao chamar o método *Dequeue*, e o compilador capturará qualquer erro de descasamento de tipo antecipadamente:

```
intQueue.Enqueue(99);
int myInt = intQueue.Dequeue();      // nenhum casting necessário
Horse myHorse = intQueue.Dequeue(); // erro do compilador:
                                    // não pode converter o tipo 'int' para 'Horse'
                                    // implicitamente
```

Você deve estar ciente de que essa substituição de *T* por um tipo especificado não é simplesmente um mecanismo de substituição textual. Em vez disso, o compilador realiza uma substituição semântica completa para que você possa especificar qualquer tipo válido para *T*. Veja mais exemplos:

```
struct Person
{
    ...
}
...
Queue<int> intQueue = new Queue<int>();
Queue<Person> personQueue = new Queue<Person>();
```

O primeiro exemplo cria uma fila de inteiros, enquanto o segundo cria uma fila de valores *Person*. O compilador também gera as versões dos métodos *Enqueue* e *Dequeue* para cada fila. Para a fila *intQueue*, esses métodos são semelhantes a isto:

```
public void Enqueue(int item);
public int Dequeue();
```

Para a fila *personQueue*, esses métodos são semelhantes a isto:

```
public void Enqueue(Person item);
public Person Dequeue();
```

Compare essas definições com aquelas da versão baseada em objeto da classe *Queue* mostrada na seção anterior. Nos métodos derivados da classe genérica, o parâmetro *item* para *Enqueue* é passado como um tipo-valor que não exige boxing. Da mesma forma, o valor retornado por *Dequeue* também é um tipo-valor que não precisa sofrer unboxing. Um conjunto de métodos semelhante é gerado para as outras duas filas.



Nota O namespace *System.Collections.Generics* da biblioteca de classes do .NET Framework fornece uma implementação para a classe *Queue* que funciona de modo semelhante à classe que acabamos de descrever. Esse namespace também contém várias outras classes de coleção, e elas serão descritas com mais detalhes no Capítulo 18, “Coleções”.

O parâmetro de tipo não precisa ser uma classe simples ou um tipo-valor. Por exemplo, você pode criar uma fila de filas de inteiros (se alguma vez achar necessário), assim:

```
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

Uma classe genérica pode ter múltiplos parâmetros de tipo. Por exemplo, a classe genérica *Dictionary* definida no namespace *System.Collections.Generic* da biblioteca de classes do .NET Framework espera dois parâmetros de tipo: um tipo para chaves e outro para os valores (essa classe será descrita com mais detalhes no Capítulo 18).



Nota Você também pode definir estruturas e interfaces genéricas utilizando a mesma sintaxe de parâmetro de tipo das classes genéricas.

Classes genéricas *versus* generalizadas

É importante estar ciente de que uma classe genérica que utiliza parâmetros de tipo é diferente de uma classe *generalizada* projetada para receber parâmetros que podem ser convertidos em tipos diferentes via casting. Por exemplo, a versão baseada em objeto da classe *Queue* mostrada anteriormente é uma classe generalizada. Há uma *única* implementação dessa classe e seus métodos recebem parâmetros *object* e retornam tipos *object*. Você pode utilizar essa classe com tipos *int*, *string* e muitos outros, mas em cada caso está utilizando instâncias da mesma classe e precisa fazer um casting dos dados utilizados para o tipo *object*.

Compare isso com a classe *Queue<T>*. Sempre que utiliza essa classe com um parâmetro de tipo (como *Queue<int>* ou *Queue<Horse>*), você faz o compilador gerar uma classe totalmente nova, com funcionalidade definida pela classe genérica. Isso significa que *Queue<int>* é um tipo totalmente diferente de um *Queue<Horse>*, mas ambos têm o mesmo comportamento. Você pode pensar numa classe genérica como uma classe que define um template que é, então, utilizado pelo compilador para gerar novas classes de tipo específico sob demanda. As versões de tipo específico de uma classe genérica (*Queue<int>*, *Queue<Horse>*, etc.) são conhecidas como *tipos construídos*, e você deve tratá-las como tipos bem diferentes (apesar daquelas que têm um conjunto semelhante de métodos e propriedades).

Genéricos e restrições

Eventualmente, é recomendável assegurar que o parâmetro de tipo utilizado por uma classe genérica identifique um tipo que fornece certos métodos. Por exemplo, se estiver definindo uma classe *PrintableCollection*, talvez você queira garantir que todos os objetos armazenados na classe tenham um método *Print*. É possível especificar essa condição utilizando uma *restrição*.

Com uma restrição, você pode limitar os parâmetros de tipo de uma classe genérica àqueles que implementam um conjunto específico de interfaces e, portanto, fornecer os métodos definidos por essas interfaces. Por exemplo, se a interface *IPrintable* definisse o método *Print*, você poderia criar a classe *PrintableCollection* assim:

```
public class PrintableCollection<T> where T : IPrintable
```

Quando você criar essa classe com um parâmetro de tipo, o compilador fará uma verificação para garantir que o tipo utilizado por *T* de fato implemente a interface *IPrintable*; caso isso não aconteça, terminará com um erro de compilação.

Crie uma classe genérica

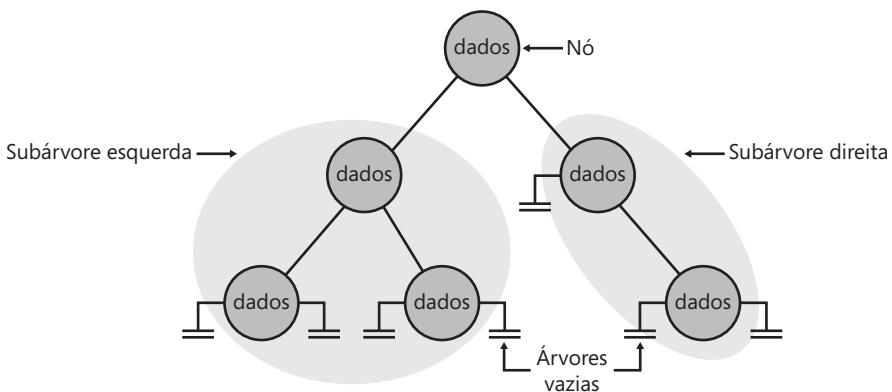
O namespace `System.Collections.Generic` da biblioteca de classes do .NET Framework contém várias classes genéricas prontamente disponíveis para você. É possível definir suas próprias classes genéricas, que é o que você fará nesta seção. Antes disso, vamos ver um pouco de teoria básica.

A teoria das árvores binárias

Nos exercícios a seguir, você definirá e utilizará uma classe que representa uma árvore binária.

Árvore binária é uma estrutura de dados que pode ser utilizada em várias operações, incluindo classificar e pesquisar dados de forma muito rápida. Livros inteiros foram escritos sobre as minúcias das árvores binárias, mas não é a finalidade desta obra abordá-las em detalhe. Em vez disso, vamos examinar apenas os fatos pertinentes. Se estiver interessado em aprender mais, consulte um livro como *The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition*, de Donald E. Knuth (Addison-Wesley Professional, 1998). Apesar de sua idade, esse é o trabalho seminal reconhecido sobre algoritmos de classificação e busca.

Uma árvore binária é uma estrutura de dados recursiva (de autorreferenciação) que pode estar vazia ou conter três elementos: um dado, que é conhecido como o *nó*, e duas subárvores, que são árvores binárias. As duas subárvores são chamadas convencionalmente de *subárvore esquerda* e *subárvore direita* porque, em geral, são representadas à esquerda e à direita do nó, respectivamente. Cada subárvore esquerda ou direita está vazia ou contém um nó e outras subárvores. Teoricamente, a estrutura inteira pode continuar *ad infinitum*. A seguinte imagem mostra a estrutura de uma pequena árvore binária.



O verdadeiro poder das árvores binárias torna-se evidente quando você as utiliza para ordenar dados. Se iniciar com uma sequência não ordenada de objetos do mesmo tipo, você poderá construir uma árvore binária ordenada e então percorrê-la para visitar cada nó em uma sequência ordenada. O algoritmo para inserir um item *B* em uma árvore binária ordenada *A* é mostrado a seguir:

```

Se a árvore  $B$  está vazia
Então
    Construa uma nova árvore  $B$ , com o novo item  $I$  como o nó, e subárvore
    esquerda e direita vazias
Senão
    Examine o valor do nó atual,  $N$ , da árvore  $B$ 
    Se o valor de  $N$  for maior do que o do novo item  $I$ 
        Então
            Se a subárvore esquerda de  $B$  está vazia
            Então
                Construa uma nova subárvore à esquerda de  $B$ , com o novo item  $I$  como o nó, e
                subárvore esquerda e direita vazias
            Senão
                Insira  $I$  na subárvore esquerda de  $B$ 
            Fim_Se
        Senão
            Se a subárvore direita de  $B$  está vazia
            Então
                Construa uma nova subárvore à direita de  $B$ , com o novo item  $I$  como o nó, e
                subárvore esquerda e direita vazias
            Senão
                Insira  $I$  na subárvore direita de  $B$ 
            Fim_Se
        Fim_Se
    Fim_Se

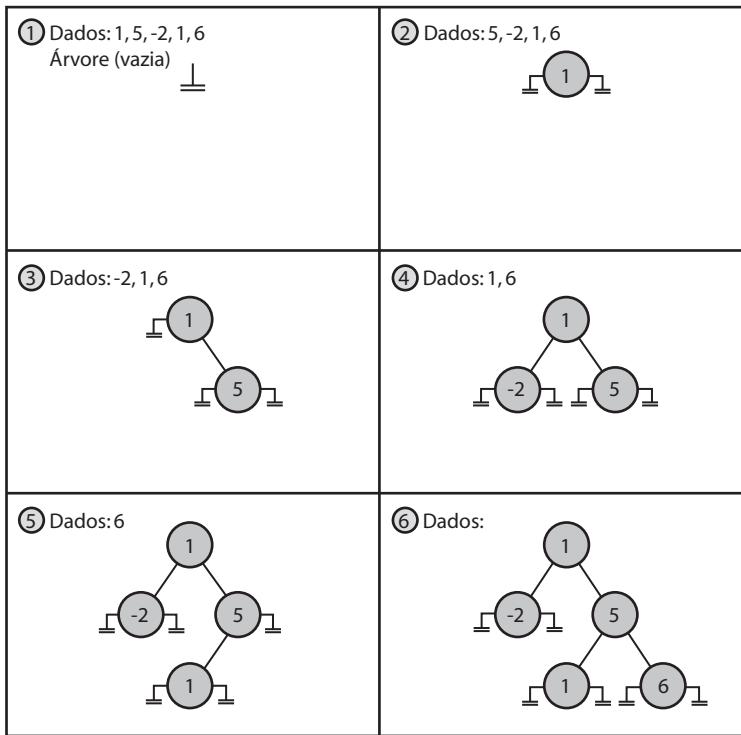
```

Observe que esse algoritmo é recursivo, chamando a si próprio para inserir o item na subárvore da esquerda ou da direita, dependendo de como o valor do item é comparado com o nó atual da árvore.



Nota A definição da expressão *maior que* depende dos tipos de dados no item e no nó. Para dados numéricos, maior que pode ser uma comparação aritmética simples, e para dados de texto, pode ser uma comparação de string; mas você deve dar às outras formas de dados suas próprias maneiras de comparar valores. Você vai aprender mais sobre isso quando implementar uma árvore binária na próxima seção, “Construa uma classe de árvore binária com genéricos”.

Se começar com uma árvore binária vazia e uma sequência não ordenada de objetos, você poderá iterar por essa sequência inserindo cada objeto na árvore binária com esse algoritmo, o que resultará em uma árvore ordenada. A imagem a seguir mostra os passos do processo para a construção de uma árvore a partir de um conjunto de cinco inteiros.

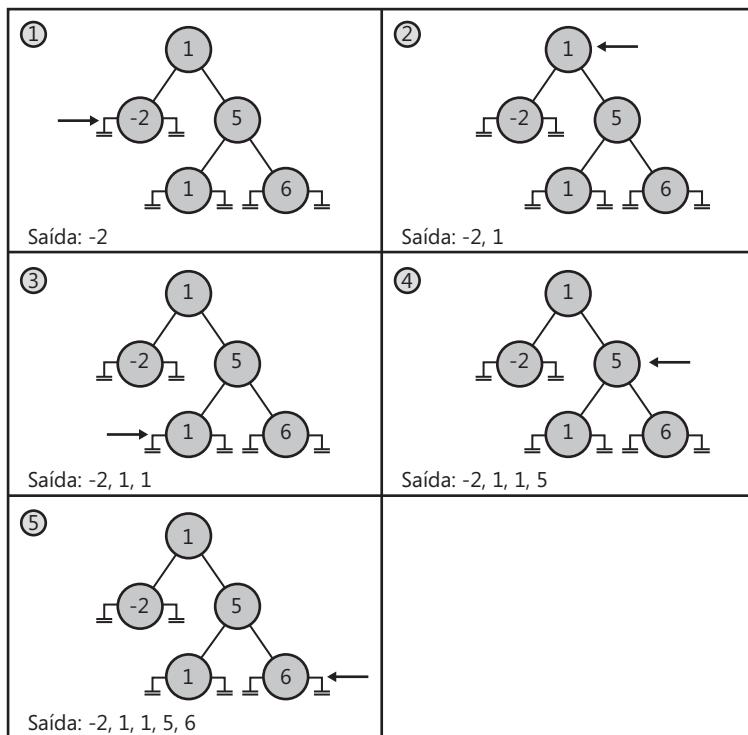


Depois de construir uma árvore binária ordenada, você pode exibir seu conteúdo em sequência visitando um nó de cada vez e imprimindo o valor encontrado. O algoritmo para executar essa tarefa também é recursivo:

```

Se a subárvore esquerda não está vazia
Então
    Exiba o conteúdo da subárvore esquerda
Fim_Se
Exiba o valor do nó
Se a árvore direita não está vazia
Então
    Exiba o conteúdo da subárvore direita
Fim_Se
  
```

A imagem a seguir mostra os passos no processo de gerar a saída da árvore. Observe que os inteiros agora são exibidos em ordem crescente.



Construa uma classe de árvore binária com genéricos

No exercício a seguir, você utilizará genéricos para definir uma classe de árvore binária capaz de armazenar quase todos os tipos de dados. A única restrição é que os tipos de dados devem fornecer uma maneira de comparar valores entre diferentes instâncias.

A classe de árvore binária pode ser útil em várias aplicações diferentes. Você a implementará como uma biblioteca de classes, em vez de um aplicativo independente. Você pode reutilizar essa classe em qualquer lugar sem ter de copiar o código-fonte e recompilá-lo. Uma *biblioteca de classes* é um conjunto de classes compiladas (e outros tipos como estruturas e delegates) armazenadas em um assembly. *Assembly* é um arquivo que normalmente tem o sufixo .dll. Outros projetos e aplicativos podem fazer uso dos itens de uma biblioteca de classes adicionando uma referência ao seu assembly e então trazendo seus namespaces para o escopo empregando instruções *using*. Você fará isso quando testar a classe de árvore binária.

As interfaces *System.IComparable* e *System.IComparable<T>*

O algoritmo para inserir um nó em uma árvore binária exige que você compare o valor do nó sendo inserido com os nós já existentes na árvore. Ao utilizar um tipo numérico, como o *int*, você pode usar os operadores *<*, *>* e *==*. Entretanto, se usar outro tipo, como *Mammal* ou *Circle* descritos em capítulos anteriores, como você compara os objetos?

Se precisar criar uma classe que exija a comparação de valores de acordo com alguma ordem natural (ou possivelmente não natural), você deve implementar a interface *IComparable*. Essa interface contém um método chamado *CompareTo*, que recebe um único parâmetro especificando o objeto a ser comparado com a instância atual e retorna um inteiro que indica o resultado da comparação, como resumido na tabela a seguir.

Valor	Significado
Menor que 0	A instância atual é menor que o valor do parâmetro.
0	A instância atual é igual ao valor do parâmetro.
Maior que 0	A instância atual é maior que o valor do parâmetro.

Como exemplo, considere a classe *Circle* descrita no Capítulo 7. Vamos vê-la novamente aqui:

```
class Circle
{
    public Circle(int initialRadius)
    {
        radius = initialRadius;
    }

    public double Area()
    {
        return Math.PI * radius * radius;
    }

    private double radius;
}
```

Você pode tornar a classe *Circle* “comparável”, implementando a interface *System.IComparable* e fornecendo o método *CompareTo*. Nesse exemplo, o método *CompareTo* compara os objetos *Circle* com base em suas áreas. Um círculo com área maior é considerado maior que um círculo com área menor.

```
class Circle : System.IComparable
{
    ...
    public int CompareTo(object obj)
    {
        Circle circObj = (Circle) obj; // faz o casting do parâmetro para seu tipo
                                      real
        if (this.Area() == circObj.Area())
            return 0;
```

```

    if (this.Area() > circObj.Area())
        return 1;

    return -1;
}
}

```

Se você examinar a interface *System.IComparable*, verá que seu parâmetro é definido como um *object*. No entanto, esse enfoque não é seguro quanto aos tipos (type-safe). Para entender a razão desse fato, considere o que pode acontecer se você tentar passar algo que não seja um *Circle* para o método *CompareTo*. A interface *System.IComparable* exige o uso de um casting para acessar o método *Area*. Se o parâmetro não for um *Circle*, mas algum outro tipo de objeto, esse casting falhará. Mas o namespace *System* também define a interface *IComparable<T>* genérica, que contém o seguinte método:

```
int CompareTo(T other);
```

Observe que esse método recebe um parâmetro de tipo (*T*) em vez de um *object* e, desse modo, é muito mais seguro do que a versão não genérica da interface. O código a seguir mostra como você pode implementar essa interface na classe *Circle*:

```

class Circle : System.IComparable<Circle>
{
    ...
    public int CompareTo(Circle other)
    {
        if (this.Area() == other.Area())
            return 0;

        if (this.Area() > other.Area())
            return 1;

        return -1;
    }
}

```

O parâmetro para o método *CompareTo* deve corresponder ao tipo especificado na interface, *IComparable<Circle>*. Em geral, é preferível implementar a interface *System.IComparable<T>*, em vez da interface *System.IComparable*. Você também pode implementar as duas da mesma maneira, como faz grande parte dos tipos no .NET Framework.

Crie a classe *Tree<TItem>*

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. No menu File, aponte para New e então clique em Project.
3. Na caixa de diálogo New Project, no painel Templates à esquerda, clique em Visual C#. No painel central, selecione o template Class Library. Na caixa Name, digite **BinaryTree**. Na caixa Location, especifique \Microsoft Press\Visual CSharp Step By Step\Chapter 17 na sua pasta Documentos e clique em OK.



Nota Utilizando o template Class Library é possível criar assemblies que podem ser reutilizados por vários aplicativos. Para usar uma classe de uma biblioteca em um aplicativo, você deve primeiramente copiar o assembly que contém o código compilado da biblioteca de classes para seu computador (se não a criou você mesmo) e, então, adicionar uma referência para esse assembly.

4. No Solution Explorer, clique com o botão direito do mouse em Class1.cs, clique em Rename e mude o nome do arquivo para **Tree.cs**. Deixe o Visual Studio mudar o nome da classe, bem como o nome do arquivo, quando solicitado.
5. Na janela Code and Text Editor, mude a definição da classe *Tree* para *Tree<TItem>*, como mostrado em negrito no código a seguir:

```
public class Tree<TItem>
{
}
```

6. Na janela Code and Text Editor, modifique a definição da classe *Tree<TItem>* para especificar que o parâmetro de tipo *TItem* deve denotar um tipo que implementa a interface genérica *IComparable<TItem>*. As alterações estão destacadas em negrito no exemplo de código a seguir.

A definição modificada da classe *Tree<TItem>* deve ficar assim:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
}
```

7. Adicione três propriedades públicas e automáticas à classe *Tree<TItem>*: uma propriedade *TItem* chamada *NodeData* e duas propriedades *Tree<TItem>* chamadas *LeftTree* e *RightTree*, como mostrado em negrito no exemplo de código a seguir:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
    public TItem NodeData { get; set; }
    public Tree<TItem> LeftTree { get; set; }
    public Tree<TItem> RightTree { get; set; }
}
```

8. Adicione um construtor à classe `Tree<TItem>` que aceita um único parâmetro `TItem` chamado `nodeValue`. No construtor, configure a propriedade `NodeData` como `nodeValue` e inicialize as propriedades `LeftTree` e `RightTree` como `null`, como mostrado em negrito no código a seguir:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
    ...
    public Tree(TItem nodeValue)
    {
        this.NodeData = nodeValue;
        this.LeftTree = null;
        this.RightTree = null;
    }
}
```



Nota Note que o nome do construtor não inclui o parâmetro de tipo; ele é chamado `Tree` e não `Tree<TItem>`.

9. Adicione um método público chamado `Insert` à classe `Tree<TItem>`, como mostrado em negrito no código a seguir. Esse método insere um valor `TItem` na árvore.

A definição do método deve ser esta:

```
public class Tree<TItem> where TItem: IComparable<TItem>
{
    ...
    public void Insert(TItem newItem)
    {
    }
}
```

O método `Insert` implementa o algoritmo recursivo descrito anteriormente para criar uma árvore binária ordenada. O programador terá utilizado o construtor para criar o nó inicial da árvore (não há um construtor padrão); portanto, o método `Insert` pode supor que a árvore não está vazia. O código a seguir é a parte do algoritmo após verificar se a árvore está vazia. Ele é reproduzido aqui para ajudá-lo a entender o código que você escreverá para o método `Insert` nos passos a seguir:

```
...
Examinar o valor do nó, N, da árvore B
Se o valor de N for maior do que o do novo item I
Então
    Se a subárvore esquerda de B está vazia
        Então
            Construa uma nova subárvore à esquerda de B, com o novo item I como o nó,
            e subárvores esquerda e direita vazias
        Senão
            Insira I na subárvore esquerda de B
        Fim_Se
    ...

```

- 10.** No método *Insert*, adicione uma instrução que declare uma variável local do tipo *TItem*, chamada *currentNodeValue*. Inicialize essa variável com o valor da propriedade *NodeData* da árvore, como mostrado em negrito no exemplo a seguir:

```
public void Insert(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;
}
```

- 11.** Adicione ao método *Insert* a seguinte instrução *if-else*, mostrada em negrito no código a seguir, depois da definição da variável *currentNodeValue*.

Essa instrução utiliza o método *CompareTo* da interface *IComparable<T>* para determinar se o valor do nó atual é maior do que o do novo item:

```
public void Insert(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;
    if (currentNodeValue.CompareTo(newItem) > 0)
    {
        // Inserir o novo item na subárvore esquerda
    }
    else
    {
        // Inserir o novo item na subárvore direita
    }
}
```

- 12.** Na parte *if* do código, imediatamente após o comentário *// Inserir o novo item na subárvore esquerda*, adicione as seguintes instruções:

```
if (this.LeftTree == null)
{
    this.LeftTree = new Tree<TItem>(newItem);
}
else
{
    this.LeftTree.Insert(newItem);
}
```

Essas instruções verificam se a subárvore esquerda está vazia. Se afirmativo, uma nova árvore será criada utilizando o novo item e será anexada como a subárvore esquerda do nó atual; caso contrário, o novo item será inserido na subárvore esquerda existente, chamando o método *Insert* recursivamente.

- 13.** Na parte *else* da instrução *if-else* externa, imediatamente após o comentário *// Inserir o novo item na subárvore direita*, adicione o código equivalente que insere o novo nó na subárvore direita:

```

if (this.RightTree == null)
{
    this.RightTree = new Tree<TItem>(newItem);
}
else
{
    this.RightTree.Insert(newItem);
}

```

- 14.** Adicione outro método público, chamado *WalkTree*, à classe *Tree<TItem>*, depois do método *Insert*.

Esse método percorre a árvore, visitando cada nó na sequência, e gera uma representação de string dos dados contidos na árvore. A definição do método deve ser esta:

```

public string WalkTree()
{
}

```

- 15.** Adicione as instruções mostradas em negrito ao código após o método *WalkTree*.

Essas instruções implementam o algoritmo descrito anteriormente para percorrer uma árvore binária. À medida que cada nó é visitado, o valor do nó é retornado para a string pelo método:

```

public string WalkTree()
{
    string result = "";

    if (this.LeftTree != null)
    {
        result = this.LeftTree.WalkTree();
    }

    result += String.Format(" {0} ", this.NodeData.ToString());

    if (this.RightTree != null)
    {
        result += this.RightTree.WalkTree();
    }

    return result;
}

```

- 16.** No menu Build, clique em Build Solution. A classe deve ser compilada inteiramente; portanto, corrija todos os erros que forem informados e recompile a solução, se necessário.

No próximo exercício, você testará a classe *Tree<TItem>* criando árvores binárias de inteiros e strings.

Teste a classe *Tree<TItem>*

- No Solution Explorer, clique com o botão direito do mouse na solução BinaryTree, aponte para Add e, então, clique em New Project.



Nota Certifique-se de clicar com o botão direito do mouse na solução BinaryTree e não no projeto BinaryTree.

- Adicione um novo projeto utilizando o template Console Application. Chame o projeto de **BinaryTreeTest**. Configure Location como \Microsoft Press\Visual CSharp Step By Step\Chapter 17 na sua pasta Documentos e clique em OK.

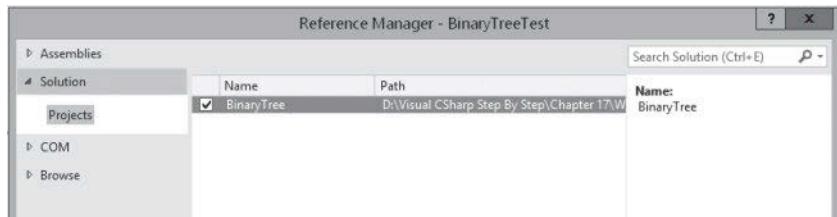


Nota Uma solução do Visual Studio 2013 pode conter mais de um projeto. Você está usando esse recurso para adicionar um segundo projeto à solução BinaryTree a fim de testar a classe *Tree<TItem>*.

- No Solution Explorer, clique com o botão direito do mouse no projeto BinaryTreeTest e, então, clique em Set As Startup Project.

O projeto BinaryTreeTest é destacado no Solution Explorer. Quando você executar o aplicativo, esse é o projeto que realmente executará.

- No Solution Explorer, clique com o botão direito do mouse no projeto BinaryTreeTest e, então, clique em Add Reference.
- No painel esquerdo da caixa de diálogo Reference Manager - BinaryTreeTest, clique em Solution. No painel central, selecione o projeto BinaryTree (certifique-se de marcar a caixa de seleção e não simplesmente clicar no assembly) e clique em OK.



Esse passo adiciona o assembly *BinaryTree* à lista de referências do projeto BinaryTreeTest no Solution Explorer. Se você examinar a pasta References do projeto BinaryTreeTest no Solution Explorer, deverá ver o assembly *BinaryTree* listado no topo. Agora você poderá criar objetos *Tree<TItem>* no projeto BinaryTreeTest.



Nota Se o projeto de biblioteca de classes não fizer parte da mesma solução que o projeto que o utiliza, você deverá adicionar uma referência ao assembly (o arquivo .dll) e não ao projeto de biblioteca de classes. Você pode fazer isso navegando até o assembly na caixa de diálogo Reference Manager. Você utilizará essa técnica no conjunto final de exercícios deste capítulo.

6. Na janela Code and Text Editor que exibe a classe *Program* no arquivo *program.cs*, adicione a seguinte diretiva *using* à lista, na parte superior da classe:

```
using BinaryTree;
```

7. Adicione ao método *Main* as instruções mostradas em negrito a seguir.

```
static void Main(string[] args)
{
    Tree<int> tree1 = new Tree<int>(10);
    tree1.Insert(5);
    tree1.Insert(11);
    tree1.Insert(5);
    tree1.Insert(-12);
    tree1.Insert(15);
    tree1.Insert(0);
    tree1.Insert(14);
    tree1.Insert(-8);
    tree1.Insert(10);
    tree1.Insert(8);
    tree1.Insert(8);

    string sortedData = tree1.WalkTree();
    Console.WriteLine("Sorted data is: {0}", sortedData);
}
```

Essas instruções criam uma nova árvore binária para armazenar *ints*. O construtor cria um nó inicial contendo o valor 10. As instruções *Insert* adicionam nós à árvore e o método *WalkTree* gera uma string representando o conteúdo da árvore, o qual deverá aparecer em ordem crescente quando essa string for exibida.



Nota Lembre-se de que a palavra-chave *int* no C# é apenas um alias para o tipo *System.Int32*; sempre que você declara uma variável *int*, na verdade está declarando uma variável *struct* do tipo *System.Int32*. O tipo *System.Int32* implementa as interfaces *IComparable* e *IComparable<T>*, sendo essa a razão pela qual é possível criar objetos *Tree<int>*. Da mesma forma, a palavra-chave *string* é um alias para *System.String*, que também implementa *IComparable* e *IComparable<T>*.

8. No menu Build, clique em Build Solution, verifique se a solução compila e corrija qualquer erro, se necessário.

- 9.** No menu Debug, clique em Start Without Debugging.

Verifique que o programa executa e exibe os valores nesta sequência:

-12 -8 0 5 5 8 8 10 10 11 14 15

- 10.** Pressione a tecla Enter para retornar ao Visual Studio 2013.

- 11.** Adicione as seguintes instruções, mostradas em negrito, ao final do método *Main* na classe *Program*, depois do código existente:

```
static void Main(string[] args)
{
    ...
    Tree<string> tree2 = new Tree<string>("Hello");
    tree2.Insert("World");
    tree2.Insert("How");
    tree2.Insert("Are");
    tree2.Insert("You");
    tree2.Insert("Today");
    tree2.Insert("I");
    tree2.Insert("Hope");
    tree2.Insert("You");
    tree2.Insert("Are");
    tree2.Insert("Feeling");
    tree2.Insert("Well");
    tree2.Insert("!");

    sortedData = tree2.WalkTree();
    Console.WriteLine("Sorted data is: {0}", sortedData);
}
```

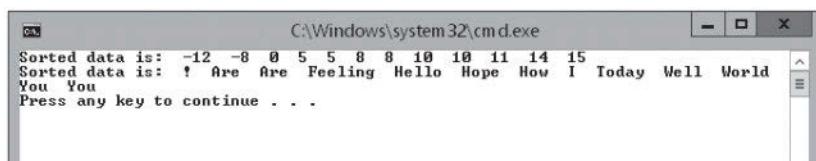
Essas instruções criam outra árvore binária para armazenar strings, preenchendo-a com alguns dados de teste e, então, a imprimem. Desta vez, os dados são ordenados alfabeticamente.

- 12.** No menu Build, clique em Build Solution, verifique se a solução compila e corrija qualquer erro, se necessário.

- 13.** No menu Debug, clique em Start Without Debugging.

Verifique que o programa executa e exibe os valores inteiros como anteriormente, seguidos pelas strings nesta sequência:

! Are Are Feeling Hello Hope How I Today Well World You You



- 14.** Pressione a tecla Enter para retornar ao Visual Studio 2013.

Crie um método genérico

Além de definir classes genéricas, você pode criar métodos genéricos.

Com um método genérico, é possível especificar os tipos dos parâmetros e o tipo de retorno utilizando um parâmetro de tipo de uma forma semelhante àquela empregada para definir uma classe genérica. Dessa maneira, você pode definir métodos generalizados que são seguros quanto ao tipo e evitar a sobrecarga do casting (e boxing, em alguns casos). Os métodos genéricos são frequentemente utilizados junto com as classes genéricas; você precisa delas para os métodos que recebem tipos genéricos como parâmetros ou que têm um tipo de retorno genérico.

Os métodos genéricos são definidos utilizando-se a mesma sintaxe de parâmetro de tipo utilizada para criar classes genéricas. (Também é possível especificar restrições.) Por exemplo, o método genérico *Swap<T>* no código a seguir troca os valores nos parâmetros. Como essa funcionalidade é útil independentemente do tipo de dado que está sendo trocado, é bom defini-la como um método genérico:

```
static void Swap<T>(ref T first, ref T second)
{
    T temp = first;
    first = second;
    second = temp;
}
```

Você chama o método especificando o tipo apropriado para seu parâmetro de tipo. Os exemplos a seguir mostram como chamar o método *Swap<T>* para permutar dois *ints* e duas *strings*:

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
...
string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2);
```



Nota Assim como instanciar uma classe genérica com diferentes parâmetros de tipo faz o compilador gerar tipos diferentes, cada uso distinto do método *Swap<T>* faz o compilador gerar uma versão diferente do método. *Swap<int>* não é o mesmo método que *Swap<string>* – os dois métodos foram gerados a partir do mesmo template genérico; portanto, mostram o mesmo comportamento, embora sobre tipos diferentes.

Defina um método genérico para criar uma árvore binária

No exercício anterior, você criou uma classe genérica para implementar uma árvore binária. A classe *Tree<TItem>* fornece o método *Insert* para adicionar itens de dados à árvore. Mas se você quer adicionar um grande número de itens, não é muito conveniente fazer chamadas repetidas ao método *Insert*. No exercício a seguir, você definirá um método genérico chamado *InsertIntoTree* que pode ser utilizado para inserir uma lista de itens de dados em uma árvore com uma única chamada de método. Você testará esse método utilizando-o para inserir uma lista de caracteres em uma árvore de caracteres.

Escreva o método *InsertIntoTree*

1. Utilizando o Visual Studio 2013, crie um novo projeto por meio do template Console Application. Na caixa de diálogo New Project, chame o projeto de **BuildTree**. Configure Location como \Microsoft Press\Visual CSharp Step By Step\Chapter 17 na sua pasta Documentos. Na lista Solution, clique em Create New Solution e depois clique em OK.
2. No menu Project, clique em Add Reference. Na caixa de diálogo Reference Manager – BuildTree, clique no botão Browse (não na guia Browse no painel à esquerda).
3. Na caixa de diálogo Select The Files To Reference, acesse a pasta Microsoft Press\Visual CSharp Step By Step\Chapter 17\BinaryTree\BinaryTree\bin\Debug na sua pasta Documentos, clique em BinaryTree.dll e então clique em Add.
4. Na caixa de diálogo Reference Manager – BuildTree, verifique se o assembly BinaryTree.dll está listado e se a caixa de seleção desse assembly está marcada; então, clique em OK.

O assembly *BinaryTree* é adicionado à lista de referências mostradas no Solution Explorer.

5. Na janela Code and Text Editor que exibe o arquivo Program.cs, adicione a seguinte diretiva *using* à parte superior do arquivo Program.cs:

```
using BinaryTree;
```

Lembre-se de que esse namespace contém a classe *Tree<TItem>*.

6. Após o método *Main*, adicione à classe *Program* um método chamado *InsertIntoTree*. Esse deve ser um método *static void* que aceita um parâmetro *Tree<TItem>* e um array *params* de elementos *TItems* chamados *data*. O parâmetro *tree* deve ser passado por referência, por motivos que serão descritos em um passo posterior.

A definição do método deve ser esta:

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree,
                                     params TItem[] data)
{ }
```

7. O tipo *TItem* utilizado para os elementos que estão sendo inseridos na árvore binária deve implementar a interface *IComparable<TItem>*. Modifique a definição do método *InsertIntoTree* e adicione a cláusula *where* mostrada em negrito no código a seguir:

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree,
                                     params TItem[] data) where TItem : IComparable<TItem>
```

8. Adicione ao método *InsertIntoTree* as instruções mostradas em negrito a seguir:

Essas instruções iteram pela lista *params*, adicionando cada item à árvore por meio do método *Insert*. Se o valor especificado pelo parâmetro *tree* for inicialmente *null*, um novo *Tree<TItem>* será criado; é por isso que o parâmetro *tree* é passado por referência.

```

static void InsertIntoTree<TItem>(ref Tree<TItem> tree,
params TItem[] data) where TItem : IComparable<TItem>
{
    foreach (TItem datum in data)
    {
        if (tree == null)
        {
            tree = new Tree<TItem>(datum);
        }
        else
        {
            tree.Insert(datum);
        }
    }
}

```

Teste o método *InsertIntoTree*

1. No método *Main* da classe *Program*, adicione as instruções mostradas em negrito a seguir, que criam uma nova *Tree* para armazenar os dados de caracteres, preenchem-na com alguns dados de exemplo utilizando o método *InsertIntoTree* e então a exibem utilizando o método *WalkTree* de *Tree*:

```

static void Main(string[] args)
{
    Tree<char> charTree = null;
    InsertIntoTree<char>(ref charTree, 'M', 'X', 'A', 'M', 'Z', 'Z', 'N');
    string sortedData = charTree.WalkTree();
    Console.WriteLine("Sorted data is: {0}", sortedData);
}

```

2. No menu Build, clique em Build Solution, verifique se a solução compila e corrija qualquer erro, se necessário.
3. No menu Debug, clique em Start Without Debugging.
O programa executa e exibe os valores de caracteres nesta ordem:
A M M N X Z Z
4. Pressione a tecla Enter para retornar ao Visual Studio 2013.

Variância e interfaces genéricas

O Capítulo 8 demonstrou que é possível utilizar o tipo *object* para armazenar um valor ou uma referência de qualquer outro tipo. Por exemplo, o código a seguir é totalmente válido:

```

string myString = "Hello";
object myObject = myString;

```

Lembre-se de que, segundo os termos da herança, a classe *String* é derivada da classe *Object*, de modo que todas as strings são objetos.

Considere agora a seguinte interface e classe genéricas:

```
interface IWrapper<T>
{
    void SetData(T data);
    T GetData();
}
class Wrapper<T> : IWrapper<T>
{
    private T storedData;

    void IWrapper<T>.SetData(T data)
    {
        this.storedData = data;
    }

    T IWrapper<T>.GetData()
    {
        return this.storedData;
    }
}
```

A classe *Wrapper*<*T*> fornece um wrapper (empacotador) simples em torno de um tipo especificado. A interface *IWrapper* define o método *SetData* que a classe *Wrapper*<*T*> implementa para armazenar os dados, e o método *GetData*, implementado por essa classe, para recuperar os dados. É possível criar uma instância dessa classe e utilizá-la para encapsular uma string, como a seguinte:

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
Console.WriteLine("Stored value is {0}", storedStringWrapper.GetData());
```

O código cria uma instância do tipo *Wrapper*<*string*>. Ela faz referência ao objeto por meio da interface *IWrapper*<*string*>, para chamar o método *SetData*. (O tipo *Wrapper*<*T*> implementa explicitamente as respectivas interfaces, de modo que você deve chamar os métodos por meio de uma referência à interface.) O código também chama o método *GetData* através da interface *IWrapper*<*string*>. Se você executar esse código, ele emitirá a mensagem "Stored value is Hello".

Dê uma olhada na seguinte linha de código:

```
IWrapper<object> storedObjectWrapper = stringWrapper;
```

Essa instrução é semelhante àquela que cria a referência *IWrapper*<*string*> no exemplo de código anterior; a diferença é que o parâmetro de tipo é *object* em vez de *string*. Esse código é válido? Lembre-se de que todas as strings são objetos (você pode atribuir um valor de *string* a uma referência a um *object*, como mostrado antes); portanto, teoricamente, essa instrução parece promissora. Entretanto, se você experimentá-la, a compilação da instrução falhará com a mensagem "Cannot implicitly convert type 'Wrapper<string>' to 'IWrapper<object>'".

Você pode experimentar um casting explícito, como este:

```
IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;
```

Esse código é compilado, mas falhará em tempo de execução com uma exceção *InvalidOperationException*. O problema é que, mesmo que todas as strings sejam objetos, o inverso não acontece. Se essa instrução fosse permitida, você poderia escrever um código como o seguinte, que, em última análise, tenta armazenar um objeto *Circle* em um campo de *string*:

```
IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;
Circle myCircle = new Circle();
storedObjectWrapper.SetData(myCircle);
```

Diz-se que a interface *IWrapper<T>* é *invariável*. Não é possível atribuir um objeto *IWrapper<A>* a uma referência de tipo *IWrapper*, mesmo que o tipo *A* seja derivado do tipo *B*. Por padrão, o C# implementa essa restrição para garantir a segurança de tipos em seu código.

Interfaces covariantes

Vamos supor que você tenha definido as interfaces *IStoreWrapper<T>* e *IRetrieveWrapper<T>*, mostradas no exemplo a seguir, no lugar de *IWrapper<T>*, e as tenha implementado na classe *Wrapper<T>*, como aqui:

```
interface IStoreWrapper<T>
{
    void SetData(T data);
}

interface IRetrieveWrapper<T>
{
    T GetData();
}

class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>
{
    private T storedData;

    void IStoreWrapper<T>.SetData(T data)
    {
        this.storedData = data;
    }

    T IRetrieveWrapper<T>.GetData()
    {
        return this.storedData;
    }
}
```

Em termos funcionais, a classe *Wrapper<T>* é a mesma de antes, exceto pelo fato de que você pode acessar os métodos *SetData* e *GetData* por meio de diferentes interfaces:

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IStoreWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;
Console.WriteLine("Stored value is {0}", retrievedStringWrapper.GetData());
```

Assim, o código a seguir é válido?

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

A resposta rápida é não, e a compilação desse código falha com o mesmo erro citado anteriormente. Mas, pensando bem, embora o compilador C# tenha considerado que essa instrução não é fortemente tipada, os motivos para essa premissa não são mais válidos. A interface *IRetrieveWrapper*<*T*> só permite ler os dados armazenados no objeto *Wrapper*<*T*> usando o método *GetData* e não oferece qualquer alternativa para alterar os dados. Em situações dessa natureza, em que o parâmetro de tipo ocorre somente como valor de retorno dos métodos em uma interface genérica, você pode informar ao compilador que algumas conversões implícitas são válidas e que não é necessário impor uma segurança de tipos rigorosa. Para isso, especifique a palavra-chave *out* ao declarar o parâmetro de tipo, como a seguir:

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

Esse recurso é chamado de *covariância*. Você pode atribuir um objeto *IRetrieveWrapper*<*A*> a uma referência a *IRetrieveWrapper*<*B*>, desde que exista uma conversão válida do tipo *A* para o tipo *B*, ou o tipo *A* derive do tipo *B*. Agora o código a seguir compila e executa como previsto:

```
// string deriva de object, de modo que isso agora é válido
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

Só é possível especificar o qualificador *out* com um parâmetro de tipo se o parâmetro de tipo ocorrer como o tipo de retorno de métodos. Se você utilizar o parâmetro de tipo para especificar o tipo de qualquer parâmetro de método, o qualificador *out* será inválido e seu código não será compilado. Além disso, a covariância funciona apenas com tipos-referência. É por isso que os tipos-valor não podem formar hierarquias de herança. Assim, o código a seguir não será compilado porque *int* é um tipo-valor:

```
Wrapper<int> intWrapper = new Wrapper<int>();
IStoreWrapper<int> storedIntWrapper = intWrapper; // válido
...
// a seguinte instrução não é válida - ints não são objects
IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;
```

Algumas das interfaces definidas pelo .NET Framework apresentam covariância, como a interface *IEnumerable*<*T*>, a qual será detalhada no Capítulo 19, “Enumeração sobre coleções”.



Nota Somente tipos de interfaces e delegates (que serão abordados no Capítulo 18) podem ser declarados como covariantes. Você não especifica o modificador *out* com classes genéricas.

Interfaces contravariantes

A contravariância segue um princípio semelhante ao da covariância, exceto pelo fato de que funciona na direção oposta; ela permite utilizar uma interface genérica para fazer referência a um objeto do tipo *B* por meio de uma referência ao tipo *A*, desde que o tipo *B* seja derivado do tipo *A*. Isso parece complicado; portanto, vale a pena examinar um exemplo da biblioteca de classes do .NET Framework.

O namespace *System.Collections.Generic* do .NET Framework dispõe de uma interface chamada *IComparer*, semelhante a esta:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Uma classe que implemente essa interface precisa definir um método *Compare*, o qual será utilizado para comparar dois objetos do tipo especificado pelo parâmetro de tipo *T*. Espera-se que o método *Compare* retorne um valor inteiro: zero, se os parâmetros *x* e *y* tiverem o mesmo valor; negativo, se *x* for menor que *y*, e positivo, se *x* for maior que *y*. O código a seguir mostra um exemplo que ordena os objetos de acordo com o respectivo código de hash. (O método *GetHashCode* é implementado pela classe *Object*. Ele simplesmente retorna um valor inteiro que identifica o objeto. Todos os tipos-referência herdam esse método e podem substituí-lo por implementações próprias.)

```
class ObjectComparer : IComparer<Object>
{
    int IComparer<Object>.Compare(Object x, Object y)
    {
        int xHash = x.GetHashCode();
        int yHash = y.GetHashCode();

        if (xHash == yHash)
            return 0;

        if (xHash < yHash)
            return -1;

        return 1;
    }
}
```

É possível criar um objeto *ObjectComparer* e chamar o método *Compare* por meio da interface *IComparer<Object>*, para comparar dois objetos, da seguinte maneira:

```
Object x = ...;
Object y = ...;
ObjectComparer objectComparer = new ObjectComparer();
IComparer<Object> objectComparator = objectComparer;
int result = objectComparator.Compare(x, y);
```

Essa é a parte mais enfadonha. O mais interessante é a possibilidade de fazer referência a esse mesmo objeto por meio de uma versão da interface *IComparer* que compara strings, como esta:

```
IComparer<String> stringComparator = objectComparator;
```

A princípio, essa instrução parece violar todas as regras imagináveis de segurança de tipos. Entretanto, se você considerar o que a interface *IComparer<T>* faz, essa estratégia fará sentido. O objetivo do método *Compare* é retornar um valor com base em uma comparação entre os parâmetros passados. Se você comparar *Objects*, certamente conseguirá comparar *Strings*, que são apenas tipos específicos de *Objects*. Afinal, uma *String* deve conseguir fazer tudo o que um *Object* pode fazer – essa é a finalidade da herança.

Como o compilador C# sabe que você não vai executar operações específicas de tipos no código do método *Compare*, que podem falhar se você chamar o método por meio de uma interface baseada em outro tipo? Reexaminando a definição da interface *IComparer*, você encontrará o qualificador *in* antes do parâmetro de tipo:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

A palavra-chave *in* informa ao compilador C# que você pode passar o tipo *T* como o tipo de parâmetro para os métodos ou pode passar qualquer tipo derivado de *T*. Não é possível utilizar *T* como tipo de retorno de qualquer método. Essencialmente, isso torna possível referenciar um objeto por meio de uma interface genérica baseada no tipo de objeto ou por meio de uma interface genérica baseada em um tipo derivado do tipo de objeto. Se um tipo *A* apresenta algumas operações, propriedades ou campos, nesse caso, se o tipo *B* for derivado do *A*, também deverá apresentar as mesmas operações (que podem se comportar de modo diferente se forem sobrescritas), propriedades e campos. Consequentemente, deve ser seguro substituir um objeto do tipo *B* por um objeto do tipo *A*.

A covariância e a contravariância podem parecer tópicos marginais no mundo dos genéricos, mas são úteis. Por exemplo, a classe de coleções genéricas *List<T>* (no namespace *System.Collections.Generic*) utiliza objetos *IComparer<T>* para implementar os métodos *Sort* e *BinarySearch*. Um objeto *List<Object>* pode conter uma coleção de objetos de qualquer tipo, de modo que os métodos *Sort* e *BinarySearch* devem ordenar objetos de qualquer tipo. Sem a contravariância, os métodos *Sort* e *BinarySearch* precisariam incluir uma lógica que determinasse os verdadeiros tipos dos itens que estão sendo ordenados ou pesquisados, e depois implementar um mecanismo de ordenação ou busca especificado de tipos. Entretanto, a menos que você seja um matemático, será muito difícil lembrar o que a covariância e a contravariância realmente fazem. Meu modo de lembrar, baseado nos exemplos desta seção, é o seguinte:

- **Exemplo de covariância** Se os métodos de uma interface genérica puderem retornar strings, também poderão retornar objetos. (Todas as strings são objetos.)
- **Exemplo de contravariância** Se os métodos de uma interface genérica puderem aceitar parâmetros de objeto, também poderão aceitar parâmetros de string. (Se você pode efetuar uma operação por meio de um objeto, poderá executar essa mesma operação por meio de uma string, porque todas as strings são objetos.)



Nota Como na covariância, somente interfaces e delegates podem ser declarados contravariantes. Você não especifica o modificador *in* com classes genéricas.

Resumo

Neste capítulo, você aprendeu a utilizar genéricos para criar classes fortemente tipadas. Vimos como instanciar um tipo genérico ao especificarmos um parâmetro de tipo. Você também viu como é possível implementar uma interface genérica e definir um método genérico. Por último, aprendeu a definir interfaces genéricas do tipo covariantes e contravariantes que podem operar com uma hierarquia de tipos.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 18.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Instanciar um objeto utilizando um tipo genérico	Especifique o parâmetro de tipo genérico apropriado. Por exemplo: <code>Queue<int> myQueue = new Queue<int>();</code>
Criar um novo tipo genérico	Defina a classe utilizando um parâmetro de tipo. Por exemplo: <code>public class Tree<TItem> { ... }</code>
Restringir o tipo que pode ser substituído para o parâmetro de tipo genérico	Especifique uma restrição utilizando uma cláusula <i>where</i> ao definir a classe. Por exemplo: <code>public class Tree<TItem> where TItem : IComparable<TItem> { ... }</code>
Definir um método genérico	Defina o método utilizando os parâmetros de tipo. Por exemplo: <code>static void InsertIntoTree<TItem> (Tree<TItem> tree, params TItem[] data) { ... }</code>

Para	Faça isto
Chamar um método genérico	Forneça tipos para cada um dos parâmetros de tipo. Por exemplo: <code>InsertIntoTree<char>(charTree, 'Z', 'X');</code>
Definir uma interface covariante	Especifique o qualificador <code>out</code> para os parâmetros do tipo covariante. Só faça referência aos parâmetros do tipo covariante como tipos de retorno de métodos e não como os tipos para parâmetros dos métodos: <code>interface IRetrieveWrapper<out T> { T GetData(); }</code>
Definir uma interface contravariante	Especifique o qualificador <code>in</code> para os parâmetros do tipo contravariante. Só faça referência aos parâmetros do tipo contravariante como os tipos de parâmetros de métodos e não como valores de retorno: <code>public interface IComparer<in T> { int Compare(T x, T y); }</code>

CAPÍTULO 18

Coleções

Neste capítulo, você vai aprender a:

- Explicar a funcionalidade fornecida nas diferentes classes de coleção disponíveis no .NET Framework.
- Criar coleções seguras quanto ao tipo.
- Preencher uma coleção com um conjunto de dados.
- Manipular e acessar os itens de dados armazenados em uma coleção.
- Procurar itens correspondentes em uma coleção baseada em lista utilizando um predicado.

O Capítulo 10, "Arrays", apresentou os arrays para armazenar conjuntos de dados. Nesse sentido, os arrays são muito úteis, mas têm suas limitações. Eles fornecem funcionalidade apenas limitada; por exemplo, não é fácil aumentar ou reduzir o tamanho de um array, e também não é simples ordenar os dados armazenados em um array. Outro problema é que os arrays só oferecem uma maneira de acessar os dados, por meio de um índice inteiro. Se seu aplicativo precisa armazenar e recuperar dados utilizando algum outro mecanismo, como uma fila FIFO (*first-in, first-out*; primeiro a entrar, primeiro a sair) descrita no Capítulo 17, "Genéricos", então os arrays talvez não sejam a estrutura de dados mais conveniente a ser usada. É aí que as coleções podem se mostrar úteis.

O que são classes de coleção?

O Microsoft .NET Framework fornece diversas classes que colecionam elementos para que um aplicativo possa acessá-los de maneiras especializadas. São as classes de coleção, mencionadas no Capítulo 17, e elas residem no namespace *System.Collections.Generic*.

Conforme o namespace indica, essas coleções são tipos genéricos; todas elas esperam que você forneça um parâmetro de tipo indicando o tipo dos dados que seu aplicativo vai armazenar nelas. Cada classe de coleção é otimizada para uma forma específica de armazenamento e acesso aos dados, e cada uma fornece métodos especializados que suportam essa funcionalidade. Por exemplo, a classe *Stack<T>* implementa um modelo LIFO (last-in, first-out; último a entrar, primeiro a sair), no qual um item é adicionado ao topo da pilha com o método *Push* e extraído do topo da pilha com o método *Pop*. O método *Pop* sempre recupera o item colocado mais recentemente e o remove da pilha. Em contraste, o tipo *Queue<T>* fornece os métodos *Enqueue* e *Dequeue*, descritos no Capítulo 17. O método *Enqueue* adiciona um item à fila, enquanto o método *Dequeue* recupera itens na mesma ordem e os remove da fila, implementando um modelo FIFO (first-in, first-out; primeiro a entrar, primeiro a sair). Diversas outras classes de coleção também estão disponíveis, e a tabela a seguir fornece um resumo das mais utilizadas.

Coleção	Descrição
<i>List<T></i>	Lista de objetos que podem ser acessados pelo índice, como um array, mas com métodos adicionais para pesquisar a lista e ordenar seu conteúdo.
<i>Queue<T></i>	Estrutura de dados do tipo primeiro a entrar, primeiro a sair, com métodos para adicionar um item em uma extremidade da fila, para remover um item da outra extremidade e para examinar um item sem removê-lo.
<i>Stack<T></i>	Estrutura de dados FILO (first-in, last-out; primeiro a entrar, último a sair), com métodos para colocar um item no topo da pilha, para extrair um item do topo da pilha e para examinar o item do topo da pilha sem removê-lo.
<i>LinkedList<T></i>	Lista ordenada de duas extremidades, otimizada para suportar inserção e remoção em ambas. Essa coleção pode atuar como uma fila ou como uma pilha, mas também suporta acesso aleatório, como uma lista.
<i>HashSet<T></i>	Conjunto não ordenado de valores, otimizado para rápida recuperação de dados. Ele fornece métodos baseados em conjunto para determinar se os itens que armazena são um subconjunto dos que estão em outro objeto <i>HashSet<T></i> e também para calcular a interseção e a união de objetos <i>HashSet<T></i> .
<i>Dictionary< TKey, TValue ></i>	Coleção de valores que podem ser identificados e recuperados pelo uso de chaves, em vez de índices.
<i>SortedList< TKey, TValue ></i>	Lista ordenada de pares chave/valor. As chaves devem implementar a interface <i>IComparable<T></i> .

As seções a seguir fornecem uma breve visão geral dessas classes de coleção. Consulte a documentação da biblioteca de classes do .NET Framework para obter mais detalhes sobre cada classe.



Nota A biblioteca de classes do .NET Framework também fornece outro conjunto de tipos de coleções no namespace *System.Collections*. São coleções não genéricas e foram projetadas antes que o C# suportasse os tipos genéricos (os genéricos foram adicionados à versão do C# desenvolvida para o .NET Framework versão 2.0). Com uma exceção, todos esses tipos armazenam referências para objetos, e você é obrigado a fazer as conversões apropriadas (casting) ao armazenar e recuperar itens. Essas classes são incluídas para compatibilidade com versões anteriores de aplicativos existentes, não sendo recomendado seu uso na compilação de novas soluções. Na verdade, essas classes não estarão disponíveis se você estiver compilando aplicativos Windows Store.

A única exceção que não armazena referências para objetos é a classe *BitArray*. Essa classe implementa um array compacto de valores booleanos utilizando um *int*; cada bit indica true (1) ou false (0). Se isso parece familiar, deveria ser mesmo, pois é muito parecido com a estrutura *IntBits* que vimos nos exemplos do Capítulo 16, “Indexadores”. A classe *BitArray* está disponível para aplicativos Windows Store.

Outro conjunto de coleções importante está disponível, e essas classes são definidas no namespace *System.Collections.Generic.Concurrent*. São classes de coleção thread-safe (ou seja, seguras para threads) que podem ser utilizadas ao se compilar aplicativos com múltiplas threads (multithreaded). O Capítulo 24, “Como melhorar o tempo de resposta empregando operações assíncronas”, fornece mais informações sobre essas classes.

A classe de coleção *List<T>*

A classe genérica *List<T>* é a mais simples das classes de coleção. Ela pode ser utilizada como um array – você pode referenciar um elemento existente em uma coleção *List<T>* utilizando a notação de array normal, com colchetes e o índice do elemento, embora não possa utilizar essa notação para adicionar novos elementos. Mas, de modo geral, a classe *List<T>* oferece mais flexibilidade do que os arrays e foi projetada para superar as seguintes restrições apresentadas pelos arrays:

- Se quiser redimensionar um array, você terá de criar um novo array, copiar os elementos (omitir alguns se o novo array for menor) e então atualizar quaisquer referências ao array original para que elas se refiram ao novo array.
- Se quiser remover um elemento de um array, você precisará mover todos os elementos finais uma posição para cima. Isso também não funciona muito bem, porque você terminará com duas cópias do último elemento.
- Se quiser inserir um elemento em um array, você terá de mover os elementos uma posição para baixo para criar um espaço vazio. Mas aí você perde o último elemento do array!

A classe de coleção *List<T>* oferece os seguintes recursos que eliminam essas restrições:

- Não é preciso especificar a capacidade de uma coleção *List<T>* ao criá-la; ela pode crescer e diminuir à medida que você adiciona elementos. Há uma sobrecarga associada a esse comportamento dinâmico e, se necessário, você pode especificar um tamanho inicial. Contudo, se esse tamanho for ultrapassado, a coleção *List<T>* simplesmente crescerá conforme a necessidade.
- Você pode remover um elemento especificado de uma coleção *List<T>* utilizando o método *Remove*. A coleção *List<T>* reordena seus elementos automaticamente e fecha a lacuna. Você também pode remover um item de uma posição especificada em uma coleção *List<T>*, utilizando o método *RemoveAt*.
- Você pode adicionar um elemento ao final de uma coleção *List<T>* utilizando o método *Add*. Você fornece o elemento a ser adicionado. A coleção *List<T>* se redimensiona automaticamente.
- Você pode inserir um elemento no meio de uma coleção *List<T>* utilizando o método *Insert*. Novamente, a coleção *List<T>* se redimensiona sozinha.
- Você pode ordenar facilmente os dados em um objeto *List<T>*, chamando o método *Sort*.



Nota Como com arrays, se utilizar *foreach* para iterar por uma coleção *List<T>*, você não poderá utilizar a variável de iteração para modificar o conteúdo da coleção. Além disso, você não pode chamar os métodos *Remove*, *Add* ou *Insert* em um loop *foreach* que itera por uma coleção *List<T>*; qualquer tentativa de fazer isso resultará em uma exceção *InvalidOperationException*.

Observe um exemplo que mostra como você pode criar, manipular e iterar pelo conteúdo de uma coleção *List<int>*:

```
using System;
using System.Collections.Generic;
...
List<int> numbers = new List<int>();

// Preenche a lista List<int> utilizando o método Add
foreach (int number in new int[12]{10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1})
{
    numbers.Add(number);
}

// Insere um elemento na penúltima posição da lista e move o último item para cima
// O primeiro parâmetro é a posição; o segundo parâmetro é o valor que está sendo inserido
numbers.Insert(numbers.Count-1, 99);

// Remove o primeiro elemento cujo valor é 7 (o 4º elemento, índice 3)
numbers.Remove(7);
// Remove o elemento que agora é o 7º, índice 6 (10)
numbers.RemoveAt(6);

// Itera pelos 11 elementos restantes utilizando uma instrução for
Console.WriteLine("Iterating using a for statement:");
for (int i = 0; i < numbers.Count; i++)
{
    int number = numbers[i]; // Observe o uso de sintaxe de array
    Console.WriteLine(number);
}

// Itera pelos mesmos 11 elementos utilizando uma instrução foreach
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Esta é a saída desse código:

```
Iterating using a for statement:
10
9
8
7
6
5
4
3
2
99
1
```

Iterating using a `foreach` statement:

```
10  
9  
8  
7  
6  
5  
4  
3  
2  
99  
1
```



Nota A maneira de determinar o número de elementos para uma coleção `List<T>` é diferente de consultar o número de itens em um array. Ao utilizar uma coleção `List<T>`, você examina a propriedade `Count` e, ao utilizar um array, examina a propriedade `Length`.

A classe de coleção `LinkedList<T>`

A classe de coleção `LinkedList<T>` implementa uma lista duplamente encadeada. Cada item da lista armazena o valor do item, junto com uma referência para o próximo item da lista (a propriedade `Next`) e para o item anterior (a propriedade `Previous`). O item do início da lista tem a propriedade `Previous` configurada como `null` e o item do final da lista tem a propriedade `Next` configurada como `null`.

Ao contrário da classe `List<T>`, `LinkedList<T>` não aceita notação de array para inserir ou examinar elementos. Em vez disso, você pode usar o método `AddFirst` para inserir um elemento no início da lista, movendo o primeiro item para cima e configurando sua propriedade `Previous` de modo a fazer referência ao novo item, ou usar o método `AddLast` para inserir um elemento no final da lista, configurando a propriedade `Next` do que anteriormente era o último item de modo a fazer referência ao novo item. Você também pode utilizar os métodos `AddBefore` e `AddAfter` para inserir um elemento antes ou depois de um item especificado na lista (é preciso primeiro recuperar o item).

O primeiro item de uma coleção `LinkedList<T>` pode ser encontrado pesquisando-se a propriedade `First`, ao passo que a propriedade `Last` retorna uma referência para o último item da lista. Para iterar por uma lista encadeada, você pode começar em uma extremidade e percorrer as referências de `Next` ou `Previous` até encontrar um item com um valor `null` para essa propriedade. Como alternativa, pode utilizar uma instrução `foreach`, a qual itera para frente em um objeto `LinkedList<T>` e para automaticamente no final.

Um item é excluído de uma coleção `LinkedList<T>` com os métodos `Remove`, `RemoveFirst` e `RemoveLast`.

O exemplo a seguir mostra uma coleção *LinkedList<T>* em ação. Observe como o código que itera pela lista, utilizando uma instrução *for*, percorre as referências de *Next* (ou *Previous*), parando somente ao encontrar uma referência *null*, que é o final da lista:

```
using System;
using System.Collections.Generic;
...
LinkedList<int> numbers = new LinkedList<int>();

// Preenche a lista List<int> utilizando o método AddFirst
foreach (int number in new int[] { 10, 8, 6, 4, 2 })
{
    numbers.AddFirst(number);
}

// Itera utilizando uma instrução for
Console.WriteLine("Iterating using a for statement:");
for (LinkedListNode<int> node = numbers.First; node != null; node = node.Next)
{
    int number = node.Value;
    Console.WriteLine(number);
}

// Itera utilizando uma instrução foreach
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// Itera para trás
Console.WriteLine("\nIterating list in reverse order:");
for (LinkedListNode<int> node = numbers.Last; node != null; node = node.Previous)
{
    int number = node.Value;
    Console.WriteLine(number);
}
```

Esta é a saída gerada por esse código:

```
Iterating using a for statement:
2
4
6
8
10

Iterating using a foreach statement:
2
4
6
8
10
```

```
Iterating list in reverse order:
10
8
6
4
2
```

A classe de coleção *Queue<T>*

A classe *Queue<T>* implementa um mecanismo FIFO (first-in, first-out; primeiro a entrar, primeiro a sair). Um elemento é inserido no final da fila (a operação *Enqueue*) e é removido no início da fila (a operação *Dequeue*).

O código a seguir é um exemplo mostrando uma coleção *Queue<int>* e suas operações comuns:

```
using System;
using System.Collections.Generic;
...
Queue<int> numbers = new Queue<int>();

// preenche a fila
Console.WriteLine("Populating the queue:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Enqueue(number);
    Console.WriteLine("{0} has joined the queue", number);
}

// itera pela fila
Console.WriteLine("\nThe queue contains the following items:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// esvazia a fila
Console.WriteLine("\nDraining the queue:");
while (numbers.Count > 0)
{
    int number = numbers.Dequeue();
    Console.WriteLine("{0} has left the queue", number);
}
```

Esta é a saída desse código:

```
Populating the queue:
9 has joined the queue
3 has joined the queue
7 has joined the queue
2 has joined the queue
```

```
The queue contains the following items:
```

```

9
3
7
2

Draining the queue:
9 has left the queue
3 has left the queue
7 has left the queue
2 has left the queue

```

A classe de coleção *Stack*<*T*>

A classe *Stack*<*T*> implementa um mecanismo LIFO (last-in, first-out; último a entrar, primeiro a sair). Um elemento entra na pilha pelo alto (a operação de inserção na pilha) e sai da pilha também por cima (a operação de remoção da pilha). Para visualizar isso, imagine uma pilha de pratos: novos pratos são adicionados à parte superior e os pratos são removidos da parte superior, fazendo com que o último prato a ser inserido na pilha seja o primeiro a ser removido. (O prato na parte inferior é pouco utilizado e inevitavelmente precisará ser lavado antes de você poder colocar qualquer comida nele, pois estará todo sujo!) Veja um exemplo — observe a ordem em que os itens são listados pelo loop *foreach*:

```

using System;
using System.Collections.Generic;
...
Stack<int> numbers = new Stack<int>();

// preenche a pilha
Console.WriteLine("Pushing items onto the stack:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Push(number);
    Console.WriteLine("{0} has been pushed on the stack", number);
}

// itera pela pilha
Console.WriteLine("\nThe stack now contains:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// esvazia a pilha
Console.WriteLine("\nPopping items from the stack:");
while (numbers.Count > 0)
{
    int number = numbers.Pop();
    Console.WriteLine("{0} has been popped off the stack", number);
}

```

Esta é a saída desse programa:

```
Pushing items onto the stack:  
9 has been pushed on the stack  
3 has been pushed on the stack  
7 has been pushed on the stack  
2 has been pushed on the stack
```

The stack now contains:

```
2  
7  
3  
9
```

```
Popping items from the stack:  
2 has been popped off the stack  
7 has been popped off the stack  
3 has been popped off the stack  
9 has been popped off the stack
```

A classe de coleção *Dictionary<TKey, TValue>*

Os tipos *array* e *List<T>* fornecem um meio de mapear um índice inteiro para um elemento. Você especifica um índice inteiro dentro de colchetes (por exemplo, [4]) e obtém de volta o elemento no índice 4 (que na realidade é o quinto elemento). Contudo, eventualmente, talvez você queira implementar um mapeamento em que o tipo a partir do qual deseja mapear não será um *int*, mas algum outro tipo qualquer, como *string*, *double* ou *Time*. Em outras linguagens, isso costuma ser chamado de *array associativo*. A classe *Dictionary<TKey, TValue>* implementa essa funcionalidade mantendo internamente dois arrays, um para as *chaves* de origem do mapeamento e uma para os *valores* de destino do mapeamento. Quando você insere um par chave/valor em uma coleção *Dictionary<TKey, TValue>*, ela determina automaticamente qual chave pertence a qual valor e torna possível recuperar de modo rápido e fácil o valor associado à chave especificada. O projeto da classe *Dictionary<TKey, TValue>* tem algumas consequências importantes:

- Uma coleção *Dictionary<TKey, TValue>* não pode conter chaves duplicadas. Se chamar o método *Add* para adicionar uma chave que já está presente no array de chaves, você obterá uma exceção. Você pode, porém, utilizar a notação de colchetes para adicionar um par chave/valor (como mostrado no exemplo a seguir), sem correr o risco de lançar uma exceção, mesmo se a chave já tiver sido adicionada; qualquer valor existente com a mesma chave será sobreescrito pelo novo valor. Você pode testar se uma coleção *Dictionary<TKey, TValue>* já contém uma chave específica utilizando o método *ContainsKey*.
- Internamente, uma coleção *Dictionary<TKey, TValue>* é uma estrutura de dados esparsa que opera de modo mais eficiente quando tem bastante memória para trabalhar. O tamanho de uma coleção *Dictionary<TKey, TValue>* pode aumentar muito rápido na memória, à medida que você insere mais elementos.

- Quando você utiliza uma instrução `foreach` para iterar por uma coleção `Dictionary< TKey, TValue >`, recebe um item `KeyValuePair< TKey, TValue >`. Essa é uma estrutura que contém uma cópia dos elementos chave e valor de um item da coleção `Dictionary< TKey, TValue >`, e você pode acessar cada elemento por meio da propriedade `Key` e das propriedades `Value`. Esses elementos são somente-leitura; não é possível utilizá-los para modificar os dados da coleção `Dictionary< TKey, TValue >`.

Veja um exemplo que associa as idades dos membros da minha família aos seus nomes e então imprime essas informações:

```
using System;
using System.Collections.Generic;
...
Dictionary<string, int> ages = new Dictionary<string, int>();

// preenche a coleção Dictionary
ages.Add("John", 47);    // utilizando o método Add
ages.Add("Diana", 46);
ages["James"] = 20;       // utilizando notação de array
ages["Francesca"] = 18;

// itera utilizando uma instrução foreach
// o iterador gera um item KeyValuePair
Console.WriteLine("The Dictionary contains:");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

Esta é a saída desse programa:

```
The Dictionary contains:
Name: John, Age: 47
Name: Diana, Age: 46
Name: James, Age: 20
Name: Francesca, Age: 18
```



Nota O namespace `System.Collections.Generic` também inclui o tipo de coleção `SortedDictionary< TKey, TValue >`. Essa classe mantém a coleção em ordem, classificada pelas chaves.

A classe de coleção `SortedList< TKey, TValue >`

A classe `SortedList< TKey, TValue >` é muito parecida com a classe `Dictionary< TKey, TValue >` no sentido de que é possível utilizá-la para associar chaves a valores. A principal diferença é que o array de chaves está sempre ordenado. (Afinal, ela se chama `SortedList`, ou seja, lista ordenada.) Na maioria dos casos, demora mais para inserir dados em um objeto `SortedList< TKey, TValue >` do que em um objeto `SortedDictionary< TKey, TValue >`, mas a recuperação de dados frequentemente é mais rápida (ou pelo menos tão rápida quanto) e a classe `SortedList< TKey, TValue >` utiliza menos memória.

Quando você insere um par chave/valor em uma coleção `SortedList<TKey, TValue>`, a chave é inserida no array de chaves no índice correto para manter as chaves ordenadas. O valor é então inserido no array de valores no mesmo índice. A classe `SortedList<TKey, TValue>` garante automaticamente que as chaves e valores mantenham o sincronismo, mesmo quando você adiciona e remove elementos. Isso significa que você pode inserir pares chave/valor em uma `SortedList<TKey, TValue>` em qualquer sequência; eles sempre são ordenados com base no valor das chaves.

Como a classe `Dictionary<TKey, TValue>`, uma coleção `SortedList<TKey, TValue>` não pode conter chaves duplicadas. Quando você utiliza uma instrução `foreach` para iterar por uma `Dictionary<TKey, TValue>`, recebe um item `KeyValuePair<TKey, TValue>`. Mas os itens de `SortedList<TKey, TValue>` serão retornados classificados pela propriedade `Key`.

Veja o mesmo exemplo que associa a idade dos membros da minha família a seus nomes e depois imprime as informações, mas esta versão foi ajustada para utilizar um objeto `SortedList<TKey, TValue>`, em vez de uma coleção `Dictionary<TKey, TValue>`:

```
using System;
using System.Collections.Generic;
...
SortedList<string, int> ages = new SortedList<string, int>();

// preenche a lista SortedList
ages.Add("John", 47);    // utilizando o método Add
ages.Add("Diana", 46);
ages["James"] = 20;      // utilizando notação de array
ages["Francesca"] = 18;

// itera utilizando uma instrução foreach
// o iterador gera um item KeyValuePair
Console.WriteLine("The SortedList contains:");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

A saída desse programa está ordenada alfabeticamente pelos nomes dos membros da minha família:

```
The SortedList contains:
Name: Diana, Age: 46
Name: Francesca, Age: 18
Name: James, Age: 20
Name: John, Age: 47
```



Importante A classe `SortedList<TKey, TValue>` não está disponível em aplicativos Windows Store. Se precisar dessa funcionalidade, você deve utilizar o tipo `SortedDictionary<TKey, TValue>`.

A classe de coleção *HashSet<T>*

A classe *HashSet<T>* é otimizada para efetuar operações de conjunto, como por exemplo para determinar a participação como membro do conjunto e gerar a união e a interseção de conjuntos.

Você insere itens em uma coleção *HashSet<T>* com o método *Add* e exclui itens com o método *Remove*. Contudo, o real poder da classe *HashSet<T>* é fornecido pelos métodos *IntersectWith*, *UnionWith* e *ExceptWith*. Esses métodos modificam uma coleção *HashSet<T>* para gerar um novo conjunto que faz interseção com, faz uma união com ou não contém os itens de uma coleção *HashSet<T>* especificada. Essas operações são destrutivas, visto que sobrescrevem o conteúdo do objeto *HashSet<T>* original com o novo conjunto de dados. Você também pode determinar se os dados de uma coleção *HashSet<T>* são um superconjunto ou um subconjunto de outra, utilizando os métodos *IsSubsetOf*, *IsSupersetOf*, *IsProperSubsetOf* e *IsProperSupersetOf*. Esses métodos retornam um valor booleano e não são destrutivos.

Internamente, uma coleção *HashSet<T>* é armazenada como uma tabela hash, permitindo uma rápida pesquisa de itens. Contudo, uma coleção *HashSet<T>* grande poderá exigir uma quantidade de memória significativa para operar rapidamente.

O exemplo a seguir mostra como se preenche uma coleção *HashSet<T>* e ilustra o uso do método *IntersectWith* para localizar dados que sobreponem dois conjuntos:

```
using System;
using System.Collections.Generic;
...
HashSet<string> employees = new HashSet<string>(new string[] {"Fred", "Bert", "Harry", "John"});
HashSet<string> customers = new HashSet<string>(new string[] {"John", "Sid", "Harry", "Diana"});

employees.Add("James");
customers.Add("Francesca");

Console.WriteLine("Employees:");
foreach (string name in employees) //Funcionários
{
    Console.WriteLine(name);
}

Console.WriteLine("\nCustomers:");
foreach (string name in customers) //Clientes
{
    Console.WriteLine(name);
}

Console.WriteLine("\nCustomers who are also employees:");
customers.IntersectWith(employees); //Clientes que também são funcionários
foreach (string name in customers)
{
    Console.WriteLine(name);
}
```

Esse código gera a seguinte saída:

Employees:

Fred
Bert
Harry
John
James

Customers:

John
Sid
Harry
Diana
Francesca

Customers who are also employees:

John
Harry



Nota O namespace *System.Collections.Generic* fornece também o tipo de coleção *SortedSet<T>*, o qual opera de maneira parecida com a classe *HashSet<T>*. A principal diferença, conforme seu nome indica, é que os dados são mantidos ordenados (sorted). As classes *SortedSet<T>* e *HashSet<T>* podem operar em conjunto; você pode obter a união de uma coleção *SortedSet<T>* com uma coleção *HashSet<T>*, por exemplo.

Inicializadores de coleção

Os exemplos nas subseções anteriores mostraram como adicionar elementos individuais a uma coleção utilizando o método mais apropriado para essa coleção (*Add* para uma coleção *List<T>*, *Enqueue* para uma coleção *Queue<T>*, *Push* para uma coleção *Stack<T>* e assim por diante). Alguns tipos de coleção também podem ser inicializados ao serem declarados, utilizando uma sintaxe parecida com aquela suportada por arrays. Por exemplo, a instrução a seguir cria e inicializa o objeto *List<int>* chamado *numbers* mostrado antes, demonstrando uma técnica alternativa para chamar o método *Add* repetidamente:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Internamente, o compilador C# converte essa inicialização em uma série de chamadas ao método *Add*. Assim, você só pode utilizar essa sintaxe para coleções que realmente suportam o método *Add*. (As classes *Stack<T>* e *Queue<T>* não suportam.)

Para coleções mais complexas que recebem pares chave/valor, como a classe *Dictionary< TKey, TValue >*, você pode especificar cada par chave/valor como um tipo anônimo na lista inicializadora, como mostrado aqui:

```
Dictionary<string, int> ages =  
    new Dictionary<string, int>(){ {"John", 44}, {"Diana", 45}, {"James", 17},  
    {"Francesca", 15}};
```

O primeiro item em cada par é a chave e o segundo é o valor.

Os métodos *Find*, predicados e expressões lambda

Utilizando as coleções baseadas em dicionário (*Dictionary< TKey, TValue >*, *SortedDictionary< TKey, TValue >* e *SortedList< TKey, TValue >*), você pode localizar um valor rapidamente, especificando a chave a procurar, e a notação de array pode ser utilizada para acessar o valor, conforme foi visto em exemplos anteriores. Outras coleções que suportam acesso aleatório sem chaves, como as classes *List< T >* e *LinkedList< T >*, não aceitam notação de array, mas, em vez disso, fornecem o método *Find* para localizar um item. Para essas classes, o argumento para o método *Find* é um predicado que especifica os critérios de busca a serem utilizados. A forma de um predicado é um método que examina cada item da coleção e retorna um valor booleano indicando se o item corresponde. No caso do método *Find*, assim que a primeira correspondência é encontrada, o item correspondente é retornado. Observe que as classes *List< T >* e *LinkedList< T >* também suportam outros métodos, como *FindLast*, que retorna o último objeto correspondente, e a classe *List< T >* fornece adicionalmente o método *FindAll*, que retorna uma coleção *List< T >* de todos os objetos correspondentes.

A maneira mais fácil de especificar o predicado é usando uma *expressão lambda*. Expressão lambda é uma expressão que retorna um método. Isso parece bastante estranho, porque a maioria das expressões que você encontrou até agora no C# retorna um valor. Se você conhece linguagens de programação funcionais, como Haskell, esse conceito deve ser fácil. Se não conhece, não se assuste: expressões lambda não são especialmente complicadas e, depois de se acostumar com a sintaxe, você verá que são bem úteis.



Nota Se estiver interessado em saber mais sobre programação funcional com Haskell, visite o site da linguagem de programação Haskell em <http://www.haskell.org/haskellwiki/Haskell>.

O Capítulo 3, “Como escrever métodos e aplicar escopo”, informa que um método típico consiste em quatro elementos: um tipo de retorno, um nome de método,

uma lista de parâmetros e um corpo de método. Uma expressão lambda contém dois desses elementos: uma lista de parâmetros e um corpo de método. As expressões lambda não definem um nome de método e o tipo de retorno (se houver algum) é inferido do contexto em que a expressão lambda é utilizada. No caso do método *Find*, o predicado processa cada item da coleção por sua vez; o corpo do predicado deve examinar o item e retornar true ou false, dependendo de corresponder aos critérios de busca. O exemplo a seguir mostra o método *Find* (destacado em negrito) em uma coleção *List<Person>*, onde *Person* é uma estrutura. O método *Find* retorna o primeiro item da lista que tem a propriedade *ID* configurada como 3:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
// Cria e preenche a lista de pessoal
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 47 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};

// Localiza o membro da lista que tem ID igual a 3
Person match = personnel.Find((Person p) => { return p.ID == 3; });

Console.WriteLine("ID: {0}\nName: {1}\nAge: {2}", match.ID, match.Name, match.Age);
```

Esta é a saída gerada por esse código:

```
ID: 3
Name: Fred
Age: 34
```

Na chamada para o método *Find*, o argumento *(Person p) => { return p.ID == 3; }* é uma expressão lambda que faz realmente o trabalho. Ela tem os seguintes itens sintáticos:

- Uma lista de parâmetros incluída entre parênteses. Como ocorre com um método normal, mesmo que o método que você esteja definindo (como no exemplo anterior) não receba parâmetro algum, você deve adicionar os parênteses. No caso do método *Find*, o predicado é fornecido com cada item da coleção por sua vez, e esse item é passado como parâmetro para a expressão lambda.
- O operador *=>*, que indica ao compilador C# que isso é uma expressão lambda.
- O corpo do método. O exemplo mostrado aqui é muito simples, contendo uma única instrução que retorna um valor booleano, indicando se o item especificado no parâmetro corresponde aos critérios de busca. Mas uma expressão lambda pode conter múltiplas instruções, e você pode formatá-la da maneira que achar mais legível. Apenas lembre-se de adicionar um ponto e vírgula após cada instrução, como você faria em um método comum.

Rigorosamente falando, o corpo de uma expressão lambda pode ser um corpo de método contendo múltiplas instruções ou pode ter uma única expressão. Se o cor-

po de uma expressão lambda só contém uma única expressão, você pode omitir as chaves e o ponto e vírgula (mas um ponto e vírgula ainda é necessário para completar a instrução inteira). Além disso, se a expressão recebe um único parâmetro, você pode omitir os parênteses em torno dele. Por fim, em muitos casos, você pode omitir o tipo dos parâmetros, pois o compilador pode inferir essa informação a partir do contexto no qual a expressão lambda é chamada. Uma forma simplificada da instrução *Find* mostrada anteriormente é parecida com a seguinte (esta instrução é muito mais fácil de ler e entender):

```
Person match = personnel.Find(p => p.ID == 3);
```

As expressões lambda são construções muito poderosas, e você pode aprender mais sobre elas no Capítulo 20, "Separação da lógica do aplicativo e tratamento de eventos".

Compare arrays e coleções

Veja um resumo das principais diferenças entre arrays e coleções:

- Uma instância de array tem um tamanho fixo e não pode aumentar nem diminuir. Uma coleção pode redimensionar dinamicamente seu tamanho, conforme a necessidade.
- Um array pode ter mais de uma dimensão. Uma coleção é linear. Entretanto, os itens de uma coleção podem eles próprios ser coleções, permitindo que você simule um array multidimensional como uma coleção de coleções.
- Você armazena e recupera um item de um array utilizando um índice. Nem todas as coleções suportam esse conceito. Por exemplo, para armazenar um item em uma coleção *List<T>*, são utilizados os métodos *Add* ou *Insert*, e para recuperar um item, é utilizado o método *Find*.
- Muitas das classes de coleção fornecem um método *ToArray* que cria e preenche um array contendo os itens da coleção. Os itens são copiados para o array e não são removidos da coleção. Além disso, essas coleções fornecem construtores que podem preencher uma coleção diretamente a partir de um array.

Utilize classes de coleção para jogar cartas

No próximo exercício, você converterá o jogo de cartas desenvolvido no Capítulo 10 para utilizar coleções em vez de arrays.

Use coleções para implementar um jogo de cartas

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto Cards, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 18\Windows X\Cards na sua pasta Documentos.

Esse projeto contém uma versão atualizada do projeto do Capítulo 10, que distribui mãos de cartas utilizando arrays. A classe *PlayingCard* foi modificada para expor o valor e o naipe de uma carta como propriedades somente-leitura.

3. Exiba o arquivo Pack.cs, na janela Code and Text Editor. Adicione a seguinte diretiva *using* ao início do arquivo:

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

4. Na classe *Pack*, mude a definição do array bidimensional *cardPack* para um objeto *Dictionary<Suit, List<Playing Card>>*, como mostrado em negrito a seguir:

```
class Pack
{
    ...
    private Dictionary<Suit, List<PlayingCard>> cardPack;
    ...
}
```

O aplicativo original utilizava um array bidimensional para representar um baralho. Este código substitui o array por um *Dictionary*, onde a chave especifica o naipe e o valor é uma lista de cartas desse naipe.

5. Localize o construtor de *Pack*. Modifique a primeira instrução nesse construtor para instanciar a variável *cardPack* como uma nova coleção *Dictionary*, em vez de um array, como mostrado em negrito a seguir:

```
public Pack()
{
    this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);
    ...
}
```

Embora a coleção *Dictionary* seja redimensionada automaticamente, quando itens são adicionados, se é improvável que a coleção mude de tamanho, você pode especificar um tamanho inicial ao instanciá-la. Isso ajuda a otimizar a alocação de memória, embora a coleção *Dictionary* ainda possa crescer, caso esse tamanho seja ultrapassado. Nesse caso, a coleção *Dictionary* conterá uma coleção de quatro listas (uma lista para cada naipe), de modo que é alocado espaço para quatro itens (*NumSuits* é uma constante com o valor 4).

6. No loop *for* externo, declare um objeto de coleção *List<PlayingCard>* chamado *cardsInSuit*, grande o suficiente para armazenar o número de cartas de cada naipe (utilize a constante *CardsPerSuit*), como a seguir, em negrito:

```
public Pack()
{
    this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);

    for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
    {
        List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
        for (Value value = Value.Two; value <= Value.Ace; value++)
        {
            ...
        }
    }
}
```

7. Mude o código no loop *for* interno para adicionar novos objetos *PlayingCard* a essa coleção, e não ao array, como mostrado em negrito no código a seguir:

```
for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
{
    List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        cardsInSuit.Add(new PlayingCard(suit, value));
    }
}
```

8. Após o loop *for* interno, adicione o objeto *List* à coleção *Dictionary* chamada *cardPack*, especificando o valor da variável *suit* como a chave para esse item:

```
for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
{
    List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        cardsInSuit.Add(new PlayingCard(suit, value));
    }
    this.cardPack.Add(suit, cardsInSuit);
}
```

9. Encontre o método *DealCardFromPack*.

Esse método escolhe uma carta aleatoriamente no baralho, remove a carta do baralho e retorna essa carta. A lógica para selecionar a carta não exige qualquer alteração, mas as instruções no final do método, que recuperam a carta do array, devem ser atualizadas de modo a utilizar, em substituição, a coleção *Dictionary*. Além disso, o código que remove a carta do array (agora ela foi distribuída) deve ser modificado; você precisa procurar a carta na lista e então removê-la. Para localizar a carta, utilize o método *Find* e especifique um predicado que localize uma carta com o valor correspondente. O parâmetro para o predicado deve ser um objeto *PlayingCard* (a lista contém itens *PlayingCard*).

As instruções atualizadas ocorrem depois da chave de fechamento do segundo loop *while*, como mostrado em negrito no código a seguir:

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
    while (this.IsEmpty(suit))
    {
        suit = (Suit)randomCardSelector.Next(NumSuits);
    }

    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
    while (this.IsCardAlreadyDealt(suit, value))
    {
        value = (Value)randomCardSelector.Next(CardsPerSuit);
    }

    List<PlayingCard> cardsInSuit = this.cardPack[suit];
    PlayingCard card = cardsInSuit.Find(c => c.CardValue == value);
    cardsInSuit.Remove(card);
    return card;
}
```

10. Localize o método *IsCardAlreadyDealt*.

Esse método determina se a carta já foi distribuída, verificando se o elemento correspondente no array foi definido com *null*. Você deve modificar esse método para determinar se existe uma carta com o valor especificado na lista para o naipe na coleção *Dictionary* chamada *cardPack*.

Para determinar se um item existe em uma coleção *List<T>*, utilize o método *Exists*. Esse método é semelhante a *Find*, visto que recebe um predicado como argumento. O predicado é passado a cada item da coleção por sua vez, e deve retornar true se o item corresponder a algum critério especificado e, caso contrário, false. Neste caso, a coleção *List<T>* armazena objetos *PlayingCard* e o critério do predicado *Exists* deve retornar true se for passado um item *PlayingCard* com naipe e valor correspondentes aos parâmetros passados para o método *IsCardAlreadyDealt*.

Atualize o método como mostrado em negrito no exemplo a seguir:

```
private bool IsCardAlreadyDealt(Suit suit, Value value)
{
    List<PlayingCard> cardsInSuit = this.cardPack[suit];
    return (!cardsInSuit.Exists(c => c.CardSuit == suit && c.CardValue == value));
}
```

11. Exiba o arquivo *Hand.cs* na janela Code and Text Editor. Adicione a seguinte diretiva *using* à lista localizada no início do arquivo:

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

12. Atualmente, a classe *Hand* utiliza um array chamado *cards* para armazenar as cartas da mão. Modifique a definição da variável *cards* de modo a ser uma coleção *List<PlayingCard>*, como mostrado aqui em negrito:

```
class Hand
{
    public const int HandSize = 13;
    private List<PlayingCard> cards = new List<PlayingCard>(HandSize);
    ...
}
```

13. Localize o método *AddCardToHand*.

Esse método verifica se a mão está cheia; se não estiver, adiciona a carta fornecida como o parâmetro do array *cards* no índice especificado pela variável *playingCardCount*.

Atualize esse método de modo a utilizar, em substituição, o método *Add* da classe *List<PlayingCard>*.

Essa mudança também evita a necessidade de controlar explicitamente a quantidade de cartas que a coleção armazena, porque você pode utilizar, em vez disso, a propriedade *Count* da coleção *cards*. Portanto, remova a variável *playingCardCount* da classe e modifique a instrução *if* que verifica se a mão está cheia, para fazer referência à propriedade *Count* da coleção *cards*.

O método completo deve se parecer com este, com as alterações destacadas em negrito:

```
public void AddCardToHand(PlayingCard cardDealt)
{
    if (this.cards.Count >= HandSize)
    {
        throw new ArgumentException("Too many cards");
    }
    this.cards.Add(cardDealt);
}
```

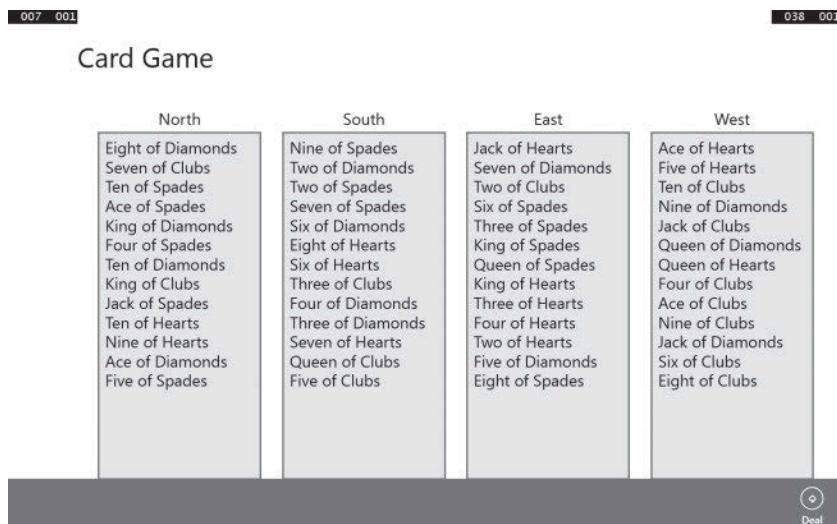
- 14.** No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
- 15.** Quando o formulário Card Game for exibido, clique em Deal.



Nota Lembre-se de que, nas versões para Windows Store desse aplicativo, o botão Deal está localizado na barra de aplicativos.

Observe se as cartas estão distribuídas e se as mãos preenchidas aparecem como antes. Clique em Deal mais uma vez para gerar outros conjuntos de mãos aleatoriamente.

A imagem a seguir mostra a versão para Windows 8.1 do aplicativo:



- 16.** Retorne ao Visual Studio 2013 e interrompa a depuração.

Resumo

Neste capítulo, você aprendeu a criar e utilizar arrays para manipular conjuntos de dados. Viu também como utilizar algumas das classes de coleção comuns para armazenar e acessar dados.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 19, “Enumeração sobre coleções”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Criar uma nova coleção	Utilize o construtor da classe de coleção. Por exemplo:
Adicionar um item a uma coleção	Utilize os métodos <i>Add</i> ou <i>Insert</i> (conforme for apropriado) para listas, conjuntos hash e coleções baseadas em dicionário. Utilize o método <i>Enqueue</i> para coleções <i>Queue<T></i> . Utilize o método <i>Push</i> para coleções <i>Stack<T></i> . Por exemplo:
Remover um item de uma coleção	Utilize o método <i>Remove</i> para listas, conjuntos hash e coleções baseadas em dicionário. Utilize o método <i>Dequeue</i> para coleções <i>Queue<T></i> . Utilize o método <i>Pop</i> para coleções <i>Stack<T></i> . Por exemplo:

Para	Faça isto
Descobrir o número de elementos em uma coleção	<p>Utilize a propriedade <code>Count</code>. Por exemplo:</p> <pre>List<PlayingCard> cards = new List<PlayingCard>(); ... int noOfCards = cards.Count;</pre>
Localizar um item em uma coleção	<p>Para coleções baseadas em dicionário, utilize notação de array. Para listas, utilize os métodos <code>Find</code>. Por exemplo:</p> <pre>Dictionary<string, int> ages = new Dictionary<string, int>(); ages.Add("John", 47); int johnsAge = ages["John"]; ... List<Person> personnel = new List<Person>(); Person match = personnel.Find(p => p.ID == 3);</pre> <p>Nota: as classes de coleção <code>Stack<T></code>, <code>Queue<T></code> e conjuntos hash não suportam buscas, embora você possa testar a participação de um item como membro de um conjunto hash utilizando o método <code>Contains</code>.</p>
Iterar pelos elementos de uma coleção	<p>Utilize uma instrução <code>for</code> ou uma instrução <code>foreach</code>. Por exemplo:</p> <pre>LinkedList<int> numbers = new LinkedList<int>(); ... for (LinkedListNode<int> node = numbers.First; node != null; node = node.Next) { int number = node.Value; Console.WriteLine(number); } ... foreach (int number in numbers) { Console.WriteLine(number); }</pre>

CAPÍTULO 19

Enumeração sobre coleções

Neste capítulo, você vai aprender a:

- Definir manualmente um enumerador que possa ser utilizado para iterar pelos elementos de uma coleção.
- Implementar um enumerador automaticamente criando um iterador.
- Fornecer iteradores adicionais que possam percorrer os elementos de uma coleção em sequências diferentes.

O Capítulo 10, “Arrays”, e o Capítulo 18, “Coleções”, abordam arrays e classes de coleções com a finalidade de armazenar sequências ou conjuntos de dados. No Capítulo 10 foi pormenorizado também a instrução *foreach*, que é possível empregar para percorrer passo a passo ou iterar pelos elementos de uma coleção. Nesses dois capítulos, utilizou-se a instrução *foreach* como uma forma rápida e conveniente de acessar o conteúdo de um array ou de uma coleção, mas agora é preciso aprender mais sobre o real funcionamento dessa instrução. Esse assunto passa a ser importante quando você define suas próprias classes de coleção, e neste capítulo descreve como tornar as coleções enumeráveis.

Enumere os elementos em uma coleção

O Capítulo 10 apresentou um exemplo do uso da instrução *foreach* para listar itens em um array simples. O código é semelhante ao seguinte:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

O construtor *foreach* fornece um mecanismo elegante que simplifica bastante o código que você precisa escrever, mas ele só pode ser executado sob certas circunstâncias – você só pode utilizar *foreach* para percorrer uma coleção *enumerável*.

Então, o que é exatamente uma coleção enumerável? A resposta rápida é que é uma coleção que implementa a interface *System.Collections.IEnumerable*.



Nota Lembre-se de que todos os arrays no C# são, na verdade, instâncias da classe *System.Array*. A classe *System.Array* é uma classe de coleção que implementa a interface *IEnumerable*.

A interface *IEnumerable* contém um método único chamado *GetEnumerator*:

```
IEnumerator GetEnumerator();
```

O método *GetEnumerator* deve retornar um objeto enumerador que implementa a interface *System.Collections.IEnumerator*. O objeto enumerador é utilizado para percorrer (enumerar) os elementos da coleção. A interface *IEnumerator* especifica os seguintes métodos e propriedade:

```
object Current { get; }
bool MoveNext();
void Reset();
```

Pense em um enumerador como um ponteiro que indica os elementos de uma lista. Inicialmente, o ponteiro aponta para *antes* do primeiro elemento. Você chama o método *MoveNext* para mover o ponteiro para baixo até o próximo item (o primeiro) da lista; o método *MoveNext* deve retornar *true* se houver realmente outro item, e *false* se não houver. A propriedade *Current* é utilizada para acessar o item que está sendo apontado no momento e o método *Reset* é utilizado para retornar o ponteiro para *antes* do primeiro item da lista. Ao criar um enumerador, por meio do método *GetEnumerator* de uma coleção, e chamar repetidamente o método *MoveNext* e recuperar o valor da propriedade *Current* utilizando o enumerador, você pode avançar, um item de cada vez, pelos elementos de uma coleção. Isso é exatamente o que a instrução *foreach* faz. Portanto, se quiser criar sua própria classe de coleção enumerável, você deve implementar a interface *IEnumerable* na sua classe de coleção e, também, fornecer uma implementação da interface *IEnumerator*, a ser retornada pelo método *GetEnumerator* da classe de coleção.



Importante À primeira vista, é fácil confundir as interfaces *IEnumerable* e *IEnumerator* devido à semelhança dos nomes. Certifique-se de não misturá-las.

Se você é observador, deve ter notado que a propriedade *Current* da interface *IEnumerator* exibe um comportamento não seguro (non-type-safe) tipado, porque retorna um *object* em vez de um tipo específico. Mas você deverá ficar contente de saber que a biblioteca de classes Microsoft .NET Framework também fornece a interface genérica *IEnumerator<T>*, que tem uma propriedade *Current* que retorna um *T*. Da mesma forma, também há uma interface *IEnumerable<T>* contendo um método *GetEnumerator* que retorna um objeto *IEnumerator<T>*. Essas duas interfaces são definidas no namespace *System.Collections.Generic*, e se estiver compilando aplicativos para o .NET Framework versão 2.0 ou superior, você deve fazer uso de interfaces genéricas ao definir coleções enumeráveis, em vez de utilizar a versão não genérica.



Nota A interface genérica *IEnumerator<T>* tem algumas diferenças adicionais em relação à interface não genérica *IEnumerator*: ela não contém um método *Reset*, mas estende a interface *IDisposable*.

Implemente manualmente um enumerador

No próximo exercício, você vai definir uma classe que implementará a interface genérica *IEnumerable<T>* e vai criar um enumerador para a classe de árvore binária demonstrada no Capítulo 17, "Genéricos".

O Capítulo 17 ilustra como é fácil percorrer uma árvore binária e exibir seu conteúdo. Portanto, você poderia estar inclinado a pensar que é simples definir um enumerador que recupera cada elemento de uma árvore binária na mesma ordem. Mas, infelizmente, você estaria enganado. O principal problema é que, ao definir um enumerador, é preciso lembrar onde você está na estrutura para que as chamadas subsequentes ao método *MoveNext* possam atualizar a posição corretamente. Os algoritmos recursivos, como aqueles utilizados para percorrer uma árvore binária, não se prestam a manter informações de estado entre chamadas de métodos de uma maneira acessível. Por essa razão, primeiro você pré-processa os dados da árvore binária em uma estrutura de dados mais acessível (uma fila) e, na verdade, enumera essa estrutura de dados. Naturalmente, esse proceder tortuoso será ocultado do usuário que está fazendo a iteração pelos elementos da árvore binária!

Crie a classe *TreeEnumerator*

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra a solução *BinaryTree*, localizada na pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 19\Windows X\BinaryTree* na sua pasta Documentos. Essa solução contém uma cópia funcional do projeto *BinaryTree* que você criou no Capítulo 17. Você vai adicionar uma nova classe a esse projeto para implementar o enumerador da classe *BinaryTree*.
3. No Solution Explorer, clique no projeto *BinaryTree*. No menu Project, clique em Add Class para abrir a caixa de diálogo Add New Item – *BinaryTree*. No painel central, selecione o template Class, digite **TreeEnumerator.cs** na caixa Name e clique em Add.

A classe *TreeEnumerator* gera um enumerador para um objeto *Tree<TItem>*. Para garantir que a classe seja fortemente tipada, você deve fornecer um parâmetro de tipo e implementar a interface *IEnumerable<T>*. Além disso, o parâmetro de tipo deve ser um tipo válido para o objeto *Tree<TItem>* que a classe enumera; portanto, ele deve ser obrigado a implementar a interface *IComparable<TItem>* (a classe *BinaryTree* exige que os itens da árvore forneçam um meio de permitir sua comparação para propósitos de ordenação).

4. Na janela Code and Text Editor que exibe o arquivo *TreeEnumerator.cs*, modifique a definição da classe *TreeEnumerator* para satisfazer esses requisitos, como mostrado em negrito no exemplo a seguir.

```
class TreeEnumerator<TItem> : IEnumerable<TItem> where TItem :  
IComparable<TItem>  
{  
}
```

5. Adicione à classe *TreeEnumerator<TItem>* as três variáveis privadas mostradas em negrito no código a seguir:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    private Tree<TItem> currentData = null;
    private TItem currentItem = default(TItem);
    private Queue<TItem> enumData = null;
}
```

A variável *currentData* será utilizada para armazenar uma referência à árvore que está sendo enumerada e a variável *currentItem* armazenará o valor retornado pela propriedade *Current*. Você preencherá a fila *enumData* com os valores extraídos dos nós da árvore, e o método *MoveNext* retornará um item de cada vez dessa fila. A palavra-chave *default* está explicada na seção “Inicializando uma variável definida com um parâmetro de tipo”, mais adiante neste capítulo.

6. Acrescente à classe *TreeEnumerator<TItem>* um construtor que aceite um único parâmetro *Tree<TItem>* chamado *data*. No corpo do construtor, adicione uma instrução que inicialize a variável *currentData* como *data*:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    ...
    public TreeEnumerator(Tree<TItem> data)
    {
        this.currentData = data;
    }
}
```

7. Acrescente o método privado a seguir, chamado *populate*, à classe *TreeEnumerator<TItem>*, depois do construtor:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    ...
    private void populate(Queue<TItem> enumQueue, Tree<TItem> tree)
    {
        if (tree.LeftTree != null)
        {
            populate(enumQueue, tree.LeftTree);
        }

        enumQueue.Enqueue(tree.NodeData);

        if (tree.RightTree != null)
        {
            populate(enumQueue, tree.RightTree);
        }
    }
}
```

Esse método percorre a árvore binária, adicionando à fila os dados contidos na árvore. O algoritmo utilizado é semelhante àquele usado pelo método *WalkTree* na classe *Tree<TItem>*, que foi descrita no Capítulo 17. A principal diferença é que, em vez de o método anexar valores *NodeData* a uma string, ele armazena esses valores na fila.

8. Retorne à definição da classe *TreeEnumerator<TItem>*. Na declaração da classe, clique com o botão direito do mouse em qualquer lugar no texto *IEnumerable<TItem>*. No menu de atalho que aparece, aponte para Implement Interface e clique em Implement Interface Explicitly.

Essa ação gera stubs (“esqueletos” de código) para os métodos da interface *IEnumerable<TItem>* e da interface *IEnumerable* e os adiciona ao final da classe. Gera também o método *Dispose* para a interface *IDisposable*.



Nota A interface *IEnumerable<TItem>* herda das interfaces *IEnumerable* e *IDisposable*, que é a razão pela qual seus métodos também aparecem. Na verdade, o único item que pertence à interface *IEnumerable<TItem>* é a propriedade *Current* genérica. Os métodos *MoveNext* e *Reset* pertencem à interface não genérica *IEnumerable*. O Capítulo 14, “Coleta de lixo e gerenciamento de recursos”, descreve a interface *IDisposable*.

9. Examine o código que foi gerado.

O corpo das propriedades e dos métodos contém uma implementação padrão que simplesmente lança uma exceção *NotImplementedException*. Você substituirá esse código por uma implementação real nos passos a seguir.

10. Atualize o corpo do método *MoveNext* com o código mostrado em negrito:

```
bool System.Collections.IEnumerable.MoveNext()
{
    if (this.enumData == null)
    {
        this.enumData = new Queue<TItem>();
        populate(this.enumData, this.currentData);
    }

    if (this.enumData.Count > 0)
    {
        this.currentItem = this.enumData.Dequeue();
        return true;
    }

    return false;
}
```

A finalidade do método *MoveNext* de um enumerador é, na verdade, dupla. Na primeira vez em que for chamado, ele deve inicializar os dados utilizados pelo enumerador e avançar para o primeiro bloco de dados a ser retornado. (Antes de *MoveNext* ser chamado pela primeira vez, o valor retornado pela propriedade *Current* é indefinido e deve resultar em uma exceção.) Nesse caso, o processo de inicialização consiste em instanciar a fila e, então, chamar o método *populate* para preencher a fila com os dados extraídos da árvore.

As chamadas subsequentes ao método *MoveNext* apenas percorrerão os itens de dados até que não reste mais item algum, removendo-os da fila até que ela esteja vazia, como nesse exemplo. É importante lembrar que *MoveNext* não retorna itens de dados – esta é a finalidade da propriedade *Current*. Tudo o que *MoveNext* faz é atualizar o estado interno no enumerador (o valor da variável *currentItem* é definida como o item de dados da fila) para ser utilizado pela propriedade *Current*, retornando *true* se houver um próximo valor, e *false* se não houver.

11. Modifique a definição do método de acesso *get* da propriedade *Current* genérica como segue em negrito:

```
TItem IEnumarator<TItem>.Current
{
    get
    {
        if (this.enumData == null)
        {
            throw new InvalidOperationException("Use MoveNext before calling Current");
        }

        return this.currentItem;
    }
}
```



Importante Certifique-se de adicionar o código à implementação correta da propriedade *Current*. Deixe a versão não genérica, *System.Collections.IEnumerable.Current*, com sua implementação padrão que lança uma exceção *NotImplementedException*.

A propriedade *Current* examina a variável *enumData* para assegurar que *MoveNext* foi chamado. (Essa variável será *null* antes da primeira chamada a *MoveNext*.) Se não for esse o caso, a propriedade lançará uma *InvalidOperationException* – esse é o mecanismo convencional utilizado pelos aplicativos do .NET Framework para indicar que uma operação não pode ser executada no estado atual. Se *MoveNext* tiver sido chamado de antemão, ele terá atualizado a variável *currentItem*; portanto, tudo o que a propriedade *Current* precisa fazer é retornar o valor nessa variável.

12. Localize o método *IDisposable.Dispose*. Transforme em comentário a instrução *throw new NotImplementedException();*, como mostrado em negrito no código a seguir. O enumerador não utiliza recurso algum que exija disponibilidade explícita; portanto, esse método não precisa fazer nada. Mas ele ainda deve estar presente. Para obter mais informações sobre o método *Dispose*, consulte o Capítulo 14.

```
void IDisposable.Dispose()
{
    // throw new NotImplementedException();
}
```

13. Compile a solução e corrija os erros relatados.

Inicializando uma variável definida com um parâmetro de tipo

Você deve ter notado que a instrução que define e inicializa a variável *currentItem* utiliza a palavra-chave *default*. A variável *currentItem* é definida pelo uso do parâmetro de tipo *TItem*. Quando o programa é escrito e compilado, o tipo real que substituirá *TItem* talvez não seja conhecido – essa questão só é resolvida quando o código é executado. Isso dificulta a especificação de como a variável será inicializada. A tentação é configurá-lo como *null*. Mas se o tipo que substituiu *TItem* for um tipo-valor, essa será uma atribuição inválida. (Você não pode configurar tipos-valor como *null*, somente tipos-referência.) Da mesma maneira, se você definir como *0*, na expectativa de que o tipo será numérico, esse procedimento será inválido se o tipo utilizado for um tipo-referência. Existem outras possibilidades também – *TItem* poderia ser um *boolean*, por exemplo. A palavra-chave *default* soluciona esse problema. O valor utilizado para inicializar a variável será determinado quando a instrução for executada. Se *TItem* for um tipo-referência, *default(TItem)* retornará *null*, se *TItem* for numérico, *default(TItem)* retornará *0* e se *TItem* for um *boolean*, *default(TItem)* retornará *false*. Se *TItem* for uma *struct*, os campos individuais na *struct* serão inicializados da mesma maneira. (Campos de referência são definidos como *null*, campos numéricos são definidos como *0* e campos *boolean* são definidos como *false*.)

Implemente a interface *IEnumerable*

No próximo exercício, você modificará a classe da árvore binária para implementar a interface *IEnumerable<T>*. O método *GetEnumerator* retornará um objeto *TreeEnumerator<TItem>*.

Implemente a interface *IEnumerable<TItem>* na classe *Tree<TItem>*

1. No Solution Explorer, dê um clique duplo no arquivo *Tree.cs* para exibir a classe *Tree<TItem>* na janela Code and Text Editor.
2. Modifique a definição da classe *Tree<TItem>* para que ela implemente a interface *IEnumerable<TItem>*, como mostrado em negrito no código a seguir:

```
public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
```

Note que as restrições são sempre colocadas no final da definição da classe.

3. Clique com o botão direito do mouse na interface *IEnumerable<TItem>* na definição da classe. No menu de atalho que aparece, aponte para Implement Interface e clique em Implement Interface Explicitly.

Essa ação gera implementações dos métodos *IEnumerable<TItem>.GetEnumerator* e *IEnumerable.GetEnumerator* e os adiciona à classe. O método da interface *IEnumerable* não genérico é implementado porque a interface genérica *IEnumerable<TItem>* herda de *IEnumerable*.

- 4.** Localize o método genérico `IEnumerable<TItem>.GetEnumerator`, próximo ao final da classe. Modifique o corpo do método `GetEnumerator()`, substituindo a instrução `throw` existente, como mostrado em negrito no exemplo a seguir:

```
IEnumerator<TItem> IEnumerable<TItem>.GetEnumerator()
{
    return new TreeEnumerator<TItem>(this);
}
```

O objetivo do método `GetEnumerator` é construir um objeto enumerador para iterar pela coleção. Nesse caso, tudo o que precisamos fazer é construir um novo objeto `TreeEnumerator<TItem>` utilizando os dados na árvore.

- 5.** Compile a solução, corrija os erros que forem relatados e recompile, se necessário.

Você agora testará a classe `Tree<TItem>` modificada, utilizando uma instrução `foreach` para iterar por uma árvore binária e exibir seu conteúdo.

Teste o enumerador

- No Solution Explorer, clique com o botão direito do mouse na solução `BinaryTree`, aponte para `Add` e, então, clique em `New Project`. Adicione um novo projeto utilizando o template `Console Application`. Chame o projeto de **EnumeratorTest**, configure `Location` como `\Microsoft Press\Visual CSharp Step By Step\Chapter 19\Windows X\BinaryTree` na sua pasta `Documentos` e clique em `OK`.



Nota Certifique-se de selecionar o template `Console Application` na lista de templates do Visual C#. Às vezes, a caixa de diálogo `Add New Project` exibe os templates do Visual Basic ou do C++ por padrão.

- Clique com o botão direito do mouse no projeto `EnumeratorTest` no Solution Explorer e, no menu de atalho que aparece, clique em `Set As StartUp Project`.
- No menu `Project`, clique em `Add Reference`. Na caixa de diálogo `Add Reference`, no painel da esquerda, clique em `Solution`; no painel do meio, selecione o projeto `BinaryTree`; então, clique em `OK`.

O assembly `BinaryTree` aparece na lista de referências do projeto `EnumeratorTest`, no Solution Explorer.

- Na janela `Code and Text Editor`, que exibe a classe `Program`, adicione a seguinte diretiva `using` à lista na parte superior do arquivo:

```
using BinaryTree;
```

- Adicione as instruções mostradas em negrito ao código, após o método `Main`. Essas instruções criam e preenchem uma árvore binária de inteiros:

```
static void Main(string[] args)
{
    Tree<int> tree1 = new Tree<int>(10);
```

```
tree1.Insert(5);
tree1.Insert(11);
tree1.Insert(5);
tree1.Insert(-12);
tree1.Insert(15);
tree1.Insert(0);
tree1.Insert(14);
tree1.Insert(-8);
tree1.Insert(10);
}
```

6. Adicione uma instrução *foreach*, como mostrado em negrito a seguir, que enumera o conteúdo da árvore e exibe os resultados:

```
static void Main(string[] args)
{
    ...
    foreach (int item in tree1)
    {
        Console.WriteLine(item);
    }
}
```

7. No menu Debug, clique em Start Without Debugging.

O programa executa e exibe os valores nesta sequência:

-12, -8, 0, 5, 5, 10, 10, 11, 14, 15

```
C:\Windows\system32
-12
-8
0
5
5
10
10
11
14
15
Press any key to continue . . .
```

8. Pressione Enter para retornar ao Visual Studio 2013.

Implemente um enumerador utilizando um iterador

Como você pode ver, o processo de criação de uma coleção enumerável pode tornar-se complexo e potencialmente propenso a erros. Para tornar a vida mais fácil, o C# fornece iteradores que podem automatizar grande parte desse processo.

Iterador é um bloco de código que produz uma sequência ordenada de valores. Um iterador não é realmente membro de uma classe enumerável; em vez disso, ele especifica a sequência que um enumerador utilizará para retornar os seus valores. Em outras palavras, um iterador é apenas uma descrição da sequência de enumeração que o compilador do C# pode usar para criar o seu próprio enumerador. Esse conceito exige um pouco de reflexão para ser compreendido corretamente; portanto, vamos considerar o exemplo básico a seguir.

Um iterador simples

A seguinte classe *BasicCollection<T>* ilustra os princípios da implementação de um iterador. A classe utiliza um objeto *List<T>* para armazenar dados e fornece o método *FillList* para preencher essa lista. Note, também, que a classe *BasicCollection<T>* implementa a interface *IEnumerable<T>*. O método *GetEnumerator* é implementado utilizando um iterador:

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

class BasicCollection<T> : IEnumerable<T>
{
    private List<T> data = new List<T>();

    public void FillList(params T [] items)
    {
        foreach (var datum in items)
        {
            data.Add(datum);
        }
    }

    I IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {
        foreach (var datum in data)
        {
            yield return datum;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Não implementado neste exemplo
        throw new NotImplementedException();
    }
}
```

O método *GetEnumerator* parece ser simples e direto, mas exige um exame mais detalhado. A primeira coisa que você deve notar é que ele não parece retornar um tipo *IEnumerator<T>*. Em vez disso, ele percorre os itens do array *data*, retornando um item de cada vez. O ponto principal é o uso da palavra-chave *yield*. A palavra-chave *yield* indica o valor que deve ser retornado por cada iteração. Se isso ajudar, você pode imaginar a instrução *yield* como a execução de uma parada temporária para o método, passando um valor de volta para o chamador. Quando o chamador precisar do valor seguinte, o método *GetEnumerator* continuará do ponto em que parou, fazendo um loop e produzindo o próximo valor. Por fim, os dados são esgotados, o loop termina e o método *GetEnumerator* se encerra. Nesse ponto a iteração estará completa.

Lembre-se de que esse não é um método normal, no sentido comum. O código no método *GetEnumerator* define um iterador. O compilador utiliza esse código para gerar uma implementação da interface *IEnumerable<T>* que contém uma propriedade *Current* e um método *MoveNext*. Essa implementação corresponderá exatamente à funcionalidade especificada pelo método *GetEnumerator*. Você, na verdade, não vê esse código gerado (a menos que descompile o assembly que contém o código compilado), mas esse é um preço pequeno a pagar pela conveniência e pela diminuição do tamanho do código que precisa ser escrito. É possível ativar o enumerador gerado pelo iterador da maneira usual, como mostrado no bloco de código a seguir, que exibe as palavras do primeiro verso do poema "Jabberwocky" (traduzido para o português como "Jaguadarte") de Lewis Carroll:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc)
{
    Console.WriteLine(word);
}
```

Esse código simplesmente envia para a saída o conteúdo do objeto *bc*, nesta ordem:

Twas, brillig, and, the, slithy, toves

Se quiser fornecer mecanismos de iteração alternativos, apresentando os dados em uma sequência diferente, você pode implementar as propriedades adicionais que implementam a interface *IEnumerable* e utilizam um iterador para retornar os dados. Por exemplo, a propriedade *Reverse* da classe *BasicCollection<T>*, mostrada a seguir, emite os dados da lista na ordem inversa:

```
class BasicCollection<T> : IEnumerable<T>
{
    ...
    public IEnumerable<T> Reverse
    {
        get
        {
            for (int i = data.Count - 1; i >= 0; i--)
            {
                yield return data[i];
            }
        }
    }
}
```

Você pode chamar essa propriedade assim:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc.Reverse)
{
    Console.WriteLine(word);
}
```

Esse código envia para a saída o conteúdo do objeto *bc* na ordem inversa:
toves, slithy, the, and, brillig, Twas

Defina um enumerador para a classe *Tree<TItem>* por meio de um iterador

No próximo exercício, você implementará o enumerador para a classe *Tree<TItem>* utilizando um iterador. Ao contrário do conjunto de exercícios anterior, o qual exigia que os dados na árvore fossem processados em uma fila pelo método *MoveNext*, você pode definir um iterador que percorre a árvore usando o mecanismo mais naturalmente recursivo, semelhante ao método *WalkTree* discutido no Capítulo 17.

Adicione um enumerador à classe *Tree<TItem>*

1. Utilizando o Visual Studio 2013, abra a solução *BinaryTree*, localizada na pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 19\Windows X\IteratorBinaryTree* na sua pasta Documentos. Essa solução contém outra cópia do projeto *BinaryTree* que você criou no Capítulo 17.
2. Abra o arquivo *Tree.cs* na janela Code and Text Editor. Modifique a definição da classe *Tree<TItem>* para que ela implemente a interface *IEnumerable<TItem>*, como mostrado em negrito no exemplo a seguir:

```
public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
{
    ...
}
```

3. Clique com o botão direito do mouse na interface *IEnumerable<TItem>* na definição da classe. No menu de atalho que aparece, aponte para Implement Interface e clique em Implement Interface Explicitly.

Os métodos *IEnumerable<TItem>.GetEnumerator* e *IEnumerable.GetIEnumerator* são adicionados ao final da classe.

4. Localize o método genérico *IEnumerable<TItem>.GetEnumerator*. Substitua o conteúdo do método *GetEnumerator*, como mostrado em negrito no código a seguir:

```
IEnumerator<TItem> IEnumerable<TItem>.GetEnumerator()
{
    if (this.LeftTree != null)
    {
        foreach (TItem item in this.LeftTree)
        {
            yield return item;
        }
    }

    yield return this.NodeData;

    if (this.RightTree != null)
    {
        foreach (TItem item in this.RightTree)
        {
            yield return item;
        }
    }
}
```

Talvez, à primeira vista, não fique evidente, mas esse código segue o mesmo algoritmo recursivo que você utilizou no Capítulo 17 para listar o conteúdo de uma árvore binária. Se *LeftTree* não estiver vazia, a primeira instrução *foreach* chamará implicitamente o método *GetEnumerator* (que você está definindo no momento) sobre ela. Esse processo continuará até que seja encontrado um nó que não tenha uma subárvore esquerda. Nesse ponto, o valor na propriedade *NodeData* é entregue (*yielded*) e a subárvore direita é examinada da mesma maneira. Depois de percorrer toda a subárvore, o processo volta ao nó pai, envia para a saída a propriedade *NodeData* do pai e examina a subárvore direita do pai. Esse curso de ação continua até que a árvore inteira tenha sido enumerada e todos os nós tenham sido enviados para a saída.

Teste o novo enumerador

1. No Solution Explorer, clique com o botão direito do mouse na solução *BinaryTree*, aponte para Add e clique em Existing Project. Na caixa de diálogo Add Existing Project, abra a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 19\Windows X\BinaryTree\EnumeratorTest, selecione o arquivo de projeto EnumeratorTest e então clique em Open.

Esse é o projeto que você criou para testar o enumerador que foi manualmente desenvolvido neste capítulo.

2. Clique com o botão direito do mouse no projeto EnumeratorTest no Solution Explorer e, no menu de atalho que aparece, clique em Set As StartUp Project.
3. No Solution Explorer, expanda a pasta References do projeto EnumeratorTest. Clique com o botão direito do mouse na referência *BinaryTree* e, no menu de atalho que aparece, clique em Remove.

4. No menu Project, clique em Add Reference. Na caixa de diálogo Add Reference, no painel da esquerda, clique em Solution; no painel do meio, selecione o projeto BinaryTree; então, clique em OK.



Nota Esses dois passos garantem que o projeto EnumeratorTest refere-se à versão do assembly *BinaryTree*. Ele deve utilizar o assembly que implementa o enumerador utilizando o iterador, em vez da versão criada no conjunto de exercícios anterior deste capítulo.

5. Exiba o arquivo Program.cs para o projeto EnumeratorTest na janela Code and Text Editor. Examine o método *Main* no arquivo Program.cs. A partir do teste do enumerador anterior, lembre-se de que esse método instancia um objeto *Tree<int>*, o preenche com alguns dados e, então, utiliza uma instrução *foreach* para exibir seu conteúdo.
6. Compile a solução, corrigindo qualquer erro, se necessário.
7. No menu Debug, clique em Start Without Debugging.
O programa executa e exibe os valores na mesma sequência anterior.
-12, -8, 0, 5, 5, 10, 10, 11, 14, 15
8. Pressione Enter para retornar ao Visual Studio 2013.

Resumo

Neste capítulo, vimos como implementar as interfaces *IEnumerable<T>* e *IEnumerator<T>* com uma classe de coleção, para permitir que os aplicativos integrassem por meio dos itens contidos na mesma coleção. Você também viu como implementar um enumerador por meio de um iterador.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 20, “Separação da lógica do aplicativo e tratamento de eventos”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Tornar uma classe de coleção enumerável, permitindo que suporte a construção <i>foreach</i>	Implemente a interface <i>IEnumerable</i> e forneça um método <i>GetEnumerator</i> que retorne um objeto <i>IEnumerator</i> . Por exemplo:
Implementar um enumerador sem utilizar um iterador	<p>Defina uma classe enumeradora que implemente a interface <i>IEnumerator</i> e que forneça a propriedade <i>Current</i> e o método <i>MoveNext</i> (e opcionalmente o método <i>Reset</i>). Por exemplo:</p> <pre>public class TreeEnumerator<TItem> : I IEnumerator<TItem> { ... TItem I Enumerator<Item>. Current { get { ... } } bool I Enumerator<Item>. MoveNext() { ... } }</pre>
Definir um enumerador usando um iterador	Implemente o enumerador para indicar quais itens devem ser retornados (utilizando a instrução <i>yield</i>) e em qual ordem. Por exemplo:

```
I Enumerator<TItem> IEnumerable<Item>
GetEnumerator()
{
    for (...)
    {
        yield return ...
    }
}
```

CAPÍTULO 20

Separação da lógica do aplicativo e tratamento de eventos

Neste capítulo, você vai aprender a:

- Declarar um tipo delegate para criar uma abstração de uma assinatura de método.
- Criar uma instância de um delegate para referenciar um método específico.
- Chamar um método por meio de um delegate.
- Definir uma expressão lambda para especificar o código a ser executado por um delegate.
- Declarar um campo de evento.
- Tratar um evento utilizando um delegate.
- Disparar um evento.

Muitos dos exemplos e exercícios deste livro deram bastante ênfase a uma definição cuidadosa de classes e estruturas para assegurar o encapsulamento. Assim, a implementação dos métodos desses tipos pode mudar sem afetar desnecessariamente os aplicativos que os utilizam. Contudo, às vezes não é possível ou desejável encapsular toda a funcionalidade de um tipo. Por exemplo, a lógica de um método em uma classe poderia depender do componente ou aplicativo ativado nesse método; talvez ela precisasse realizar algum processamento específico do aplicativo ou do componente como parte de sua operação. O problema é que, quando você compila tal classe e implementa seus métodos, talvez não saiba quais aplicativos e componentes vão utilizá-la e, assim, precisa evitar a introdução de dependências em seu código que possam restringir o uso de sua classe. Os delegates fornecem a solução ideal, tornando possível desacoplar completamente a lógica do aplicativo em seus métodos dos aplicativos que os chamam.

No C#, os eventos admitem um cenário relacionado. Grande parte do código escrito nos vários exercícios deste livro pressupõe que as instruções são executadas sequencialmente. Embora esse seja o caso mais comum, você vai perceber que algumas vezes é necessário interromper o fluxo atual da execução e executar outra tarefa mais importante. Quando a tarefa é concluída, o programa pode continuar a partir de onde foi interrompido. Os exemplos clássicos desse estilo de programa são os formulários do Windows, utilizados nos exercícios envolvendo aplicativos gráficos ao longo do livro. (Lembre-se de que, neste livro, o termo *formulário* se refere a uma página em um aplicativo Window Store ou em uma janela Microsoft Windows Presentation Foundation [WPF].) Um formulário exibe controles, como botões e caixas de texto. Ao clicar em um botão ou digitar em uma caixa de texto, você espera que o formulário responda de imediato. O aplicativo tem de parar temporariamente o que está fazendo

e manipular a sua entrada. Esse estilo de operação se aplica não apenas às interfaces gráficas do usuário (GUIs), mas também a qualquer aplicação em que uma operação precisa ser executada com urgência – desligar o reator de uma usina nuclear se estiver superaquecendo, por exemplo. Para manipular esse tipo de processamento, o runtime tem de fornecer duas coisas: um meio de indicar que algo urgente está acontecendo e uma maneira de especificar o código que deve ser executado quando o evento urgente acontecer. Em conjunto com os delegates, os eventos fornecem a infraestrutura com a qual é possível implementar sistemas que seguem essa estratégia.

Começaremos examinando os delegates.

Delegates

Delegate é uma referência para um método. Trata-se de um conceito muito simples com implicações extraordinariamente poderosas. Vou explicar.



Nota Os delegates recebem esse nome porque, quando são chamados, “delegam” o processamento para o método referenciado.

Em geral, ao escrever uma instrução que chama um método, você especifica o nome do método (e, possivelmente, o objeto ou a estrutura ao qual o método pertence). Fica claro, a partir de seu código, exatamente qual método está sendo executado e quando. Veja o exemplo simples a seguir, que chama o método *performCalculation* de um objeto *Processor* (o que esse método faz ou como a classe *Processor* é definida é irrelevante para esta discussão):

```
Processor p = new Processor();
p.performCalculation();
```

Um *delegate* é um objeto que faz referência a um método. Você pode atribuir uma referência a um método para um delegate da mesma maneira como pode atribuir um valor *int* a uma variável *int*. O exemplo a seguir cria um delegate chamado *performCalculationDelegate* que faz referência ao método *performCalculation* do objeto *Processor*. Alguns elementos da instrução que declara o delegate foram deliberadamente omitidos, pois é importante entender o conceito e não se preocupar com a sintaxe (vamos ver a sintaxe completa em breve):

```
Processor p = new Processor();
delegate ... performCalculationDelegate ...;
performCalculationDelegate = p.performCalculation;
```

É importante entender que a instrução que atribui a referência de método ao delegate não executa o método nesse ponto; não existem parênteses após o nome do método, e você não especifica nenhum parâmetro (se o método os recebe). Trata-se apenas de uma instrução de atribuição.

Tendo armazenado uma referência para o método *performCalculation* do objeto *Processor* no delegate, subsequentemente o aplicativo pode chamar o método por meio do delegate, como segue:

```
performCalculationDelegate();
```

Isso parece uma chamada de método normal; se você não soubesse que não era, poderia parecer que estaria executando um método chamado *performCalculationDelegate*. Contudo, o Common Language Runtime (CLR) sabe que se trata de um delegate; portanto, em vez disso, recupera o método referenciado pelo delegate e o executa. Posteriormente, você pode alterar o método que um delegate referencia para que uma instrução que chama um delegate possa executar um método diferente a cada vez. Além disso, um delegate pode referenciar mais de um método por vez (pense nisso como uma coleção de referências de método) e, quando você chamá-lo, todos os métodos aos quais ele faz referência serão executados.



Nota Se você conhece C++, um delegate é muito semelhante a um ponteiro de função. No entanto, ao contrário dos ponteiros de função, os delegates são fortemente tipados. Você somente pode fazer um delegate referenciar um método que corresponda à assinatura do delegate e não pode chamar um delegate que não referece um método válido.

Exemplos de delegates na biblioteca de classes do .NET Framework

A biblioteca de classes do Microsoft .NET Framework faz amplo uso de delegates para muitos de seus tipos, dos quais dois exemplos estão no Capítulo 18, “Coleções”: o método *Find* e o método *Exists* da classe *List<T>*. Se você se recorda, esses dois métodos pesquisam uma coleção *List<T>*, retornando um item correspondente ou testando a existência de um item correspondente. Quando implementaram essa classe, os projetistas da classe *List<T>* não tinham a mínima ideia do que deveria constituir uma correspondência no seu código de aplicativo; portanto, como consequência, ao contrário disso, eles permitem que você a defina, fornecendo seu próprio código em forma de predicado. Na verdade, um predicado é apenas um delegate que retorna um valor booleano.

O código a seguir deve ajudá-lo a lembrar de como o método *Find* é utilizado:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 47 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};
...
// Localiza o membro da lista que tem ID igual a 3
Person match = personnel.Find(p => p.ID == 3);
```

Outros exemplos de métodos expostos pela classe *List<T>* e que utilizam delegates para executar suas operações incluem *Average*, *Max*, *Min*, *Count* e *Sum*. Esses métodos recebem um delegate *Func* como parâmetro. Um delegate *Func* referencia um método que retorna um valor (uma função). Nos exemplos a seguir, o método *Average* é utilizado para calcular a idade média dos itens na coleção *personnel* (o delegate *Func<T>* simplesmente retorna o valor no campo *Age* de cada item da coleção), o método *Max* é usado para determinar o item com ID mais alto e o método *Count* calcula quantos itens têm *Age* entre 30 e 39, inclusive.

```
double averageAge = personnel.Average(p => p.Age);
Console.WriteLine("Average age is {0}", averageAge);
...
int id = personnel.Max(p => p.ID);
Console.WriteLine("Person with highest ID is {0}", id);
...
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
Console.WriteLine("Number of personnel in their thirties is {0}", thirties);
```

Esse código gera a seguinte saída:

```
Average age is 32.75
Person with highest ID is 4
Number of personnel in their thirties is 1
```

Você vai encontrar muitos exemplos desses e de outros tipos de delegates utilizados pela biblioteca de classes do .NET Framework no restante deste livro. Você também pode definir seus próprios delegates. A melhor maneira de entender completamente como e quando vai querer fazer isso é vê-los em ação; portanto, a seguir, vamos acompanhar um exemplo.

Os tipos delegate *Func<T, ...>* e *Action<T, ...>*

O parâmetro recebido pelos métodos *Average*, *Max*, *Count* e outros da classe *List<T>* são delegates *Func<T, TResult>* genéricos; os parâmetros de tipo fazem referência ao tipo do parâmetro passado para o delegate e ao tipo do valor de retorno. Para os métodos *Average*, *Max* e *Count* da classe *List<Person>* mostrados no texto, o primeiro parâmetro de tipo *T* é o tipo do dado na lista (a estrutura *Person*), enquanto o parâmetro de tipo *TResult* é determinado pelo contexto no qual o delegate é utilizado. No exemplo a seguir, o tipo de *TResult* é *int*, pois o valor retornado pelo método *Count* deve ser um inteiro:

```
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
```

Assim, nesse exemplo, o tipo do delegate esperado pelo método *Count* é *Func<Person, int>*.

Esse ponto pode parecer um tanto acadêmico, pois o compilador gera automaticamente o delegate, com base no tipo do *List<T>*, mas é bom conhecer esse linguajar, pois ele ocorre repetidas vezes na biblioteca de classes do .NET Framework. Na verdade, o namespace *System* define uma família inteira de tipos delegate *Func*, desde *Func<TResult>*, para funções que retornam um resultado sem receber qualquer parâmetro, até *Func<T1, T2, T3, T4, ..., T16, TResult>*, para funções que recebem 16 parâmetros. Se você se encontrar em uma situação na qual está criando seu próprio tipo delegate que corresponde a esse padrão, deve pensar em usar, em vez disso, um tipo delegate *Func* apropriado. Você encontrará os tipos delegate *Func* outra vez no Capítulo 21, "Consulta a dados na memória usando expressões de consulta".

Junto com *Func*, o namespace *System* também define uma série de tipos delegate *Action*. Um delegate *Action* é utilizado para fazer referência a um método que executa uma ação, em vez de retornar um valor (um método *void*). Novamente, está disponível uma família de tipos delegate *Action*, variando de *Action<T>* (especificando um delegate que recebe um único parâmetro) até *Action<T1, T2, T3, T4, ..., T16>*.

O cenário da fábrica automatizada

Suponha que você esteja escrevendo sistemas de controle para uma fábrica automatizada. Ela contém um grande número de diferentes máquinas, cada uma delas executando tarefas distintas na produção de artigos manufaturados – moldagem e dobradura de chapas de metal, soldagem das chapas, pintura das chapas etc. Cada máquina foi construída e instalada por um fornecedor especialista. As máquinas são controladas por computador e cada fornecedor disponibilizou um conjunto de funções para controlar sua máquina. Sua tarefa é integrar os diferentes sistemas utilizados pelas máquinas em um único programa de controle. Um aspecto no qual você decidiu se concentrar é o de fornecer um meio de desligar todas as máquinas – rapidamente, se necessário!

Cada máquina tem seu processo (e funções) controlado por computador para ser desligada de modo seguro, conforme resumido aqui:

```
StopFolding(); // Máquina de dobradura e moldagem
FinishWelding(); // Máquina de soldagem
PaintOff(); // Máquina de pintura
```

Implemente o sistema de controle da fábrica sem utilizar delegates

Uma estratégia simples para implementar a funcionalidade de desligamento no programa de controle é mostrada a seguir:

```
class Controller
{
    // Campos representando as diferentes máquinas
    private FoldingMachine folder;
    private WeldingMachine welder;
    private PaintingMachine painter;
```

```

...
public void ShutDown()
{
    folder.StopFolding();
    welder.FinishWelding();
    painter.PaintOff();
}
...
}

```

Embora essa estratégia funcione, ela não é muito extensível nem flexível. Se a fábrica comprar uma nova máquina, você precisará modificar esse código; a classe *Controller* e o código para gerenciar as máquinas estão fortemente acoplados.

Implemente a fábrica utilizando um delegate

Embora os nomes de cada método sejam diferentes, todos eles têm a mesma “forma”: não recebem parâmetros e não retornam valor. (Você vai considerar o que acontece se esse não for o caso, posteriormente – seja paciente.) Portanto, o formato geral de cada método é este:

```
void methodName();
```

É aí que um delegate pode ser útil. Um delegate que corresponda a essa forma pode ser utilizado para referenciar qualquer um dos métodos de desligamento de máquina. Você declara um delegate assim:

```
delegate void stopMachineryDelegate();
```

Note os seguintes detalhes:

- Você usa a palavra-chave *delegate*.
- Você especifica o tipo de retorno (*void*, neste exemplo), um nome para o delegate (*stopMachineryDelegate*) e todos os parâmetros (não há parâmetros neste caso).

Após ter declarado o delegate, você pode criar uma instância e fazê-la referenciar um método correspondente, utilizando o operador de atribuição composto `+=`. Você pode fazer isso no construtor da classe *Controller* como a seguir:

```

class Controller
{
    delegate void stopMachineryDelegate();           // o tipo delegate
    private stopMachineryDelegate stopMachinery; // uma instância do delegate
    ...
    public Controller()
    {
        this.stopMachinery += folder.StopFolding;
    }
    ...
}
```

Demora um pouco para se acostumar com essa sintaxe. Você *adiciona* o método ao delegate; lembre-se de que não está realmente chamando o método. O operador `+` é sobrecarregado para ter esse novo significado, quando utilizado com delegates. (Você conhecerá mais detalhes a respeito de sobrecarga de operadores no Capítulo 22 “Sobrecarga de operadores”.) Observe que você simplesmente especifica o nome do método e não inclui parênteses nem parâmetros.

É seguro utilizar o operador `+=` em um delegate não inicializado. Ele será inicializado automaticamente. Como alternativa, você pode utilizar a palavra-chave `new` para inicializar um delegate explicitamente com um método específico, como este:

```
this.stopMachinery = new stopMachineryDelegate(folder.StopFolding);
```

Você pode chamar o método ativando o delegate, como a seguir:

```
public void ShutDown()
{
    this.stopMachinery();
    ...
}
```

Para chamar um delegate, utilize a mesma sintaxe utilizada para fazer uma chamada de método. Se o método que o delegate referencia receber parâmetros, você deve especificá-los nesse momento, entre os parênteses.



Nota Se tentar chamar um delegate que não foi inicializado e que não referencia método algum, você receberá uma exceção `NullReferenceException`.

Uma vantagem importante de utilizar um delegate é que ele pode referenciar mais de um método por vez. Basta utilizar o operador `+=` para adicionar métodos ao delegate, como mostrado a seguir:

```
public Controller()
{
    this.stopMachinery += folder.StopFolding;
    this.stopMachinery += welder.FinishWelding;
    this.stopMachinery += painter.PaintOff;
}
```

Chamar `this.stopMachinery()` no método `Shutdown` da classe `Controller` faz automaticamente um método de cada vez ser chamado. O método `Shutdown` não precisa saber quantas máquinas existem ou quais são os nomes dos métodos.

Você pode remover um método de um delegate utilizando o operador de atribuição composta `-=`, como demonstrado aqui:

```
this.stopMachinery -= folder.StopFolding;
```

O esquema atual adiciona os métodos das máquinas ao delegate no construtor de *Controller*. Para tornar a classe *Controller* totalmente independente das várias máquinas, você precisa transformar *stopMachineryDelegate* em um tipo público e fornecer um meio de permitir que classes fora de *Controller* adicionem métodos ao delegate. Existem diversas opções:

- Tornar a variável do delegate, *stopMachinery*, pública:

```
public stopMachineryDelegate stopMachinery;
```

- Manter a variável do delegate *stopMachinery* privada, mas criar uma propriedade de leitura e gravação para proporcionar-lhe acesso:

```
private delegate void stopMachineryDelegate();
...
public stopMachineryDelegate StopMachinery
{
    get
    {
        return this.stopMachinery;
    }

    set
    {
        this.stopMachinery = value;
    }
}
```

- Fornecer um encapsulamento completo, implementando os métodos *Add* e *Remove* separados. O método *Add* recebe um método como um parâmetro e o adiciona ao delegate, enquanto o método *Remove* remove do delegate o método especificado (observe que você especifica um método como parâmetro utilizando um tipo delegate):

```
public void Add(stopMachineryDelegate stopMethod)
{
    this.stopMachinery += stopMethod;
}

public void Remove(stopMachineryDelegate stopMethod)
{
    this.stopMachinery -= stopMethod;
}
```

Um purista em orientação a objetos provavelmente optaria pela estratégia *Add/Remove*. Mas as outras estratégias são alternativas viáveis muito utilizadas, razão pela qual são apresentadas aqui.

Qualquer que seja a técnica escolhida, você deve remover o código que adiciona os métodos das máquinas ao delegate a partir do construtor de *Controller*. Você pode então instanciar um *Controller* e objetos representando as outras máquinas, como a seguir (este exemplo utiliza a estratégia *Add/Remove*):

```
Controller control = new Controller();
FoldingMachine folder = new FoldingMachine();
```

```
WeldingMachine welder = new WeldingMachine();
PaintingMachine painter = new PaintingMachine();
...
control.Add(folder.StopFolding);
control.Add(welder.FinishWelding);
control.Add(painter.PaintOff);
...
control.ShutDown();
...
```

Declare e utilize delegates

Nos exercícios a seguir, você concluirá um aplicativo que faz parte de um sistema de uma empresa chamada Wide World Importers. A Wide World Importers importa e comercializa materiais de construção e ferramentas, e o aplicativo em que você vai trabalhar oferece aos clientes a capacidade de navegar pelos itens que a empresa tem em estoque e fazer pedidos. O aplicativo contém um formulário que exibe a mercadoria correntemente disponível, junto com um painel que lista os itens selecionados por um cliente. Quando o cliente quer fazer um pedido, pode clicar no botão Checkout no formulário. Então, o pedido é processado e o painel é limpo.

Quando o cliente faz um pedido, atualmente várias ações ocorrem:

- Um pagamento é solicitado ao cliente.
- Os itens do pedido são examinados e, se algum deles tiver restrição de uso pela idade (como as ferramentas mecânicas), os detalhes do pedido passam por auditoria e são monitorados.
- Um aviso de despacho é gerado para propósitos de expedição. Esse aviso contém um resumo do pedido.

A lógica dos processos de auditoria e expedição é independente da lógica da checagem de saída (checkout), posto que a ordem em que eles ocorrem é irrelevante. Além disso, existe a possibilidade de que um desses elementos possa sofrer melhorias no futuro, e ser necessário mais processamento para a operação de checagem, à medida que as circunstâncias empresariais ou os requisitos normativos mudarem no futuro. Portanto, é desejável desacoplar a lógica do pagamento e de checagem de saída dos processos de auditoria e expedição para facilitar a manutenção e as atualizações. Vamos começar examinando o aplicativo e vendo como atualmente ele não cumpre esse objetivo. Então, você vai modificar a estrutura do aplicativo para eliminar as dependências entre a lógica de checagem de saída e a lógica de auditoria e expedição.

Examine o aplicativo da Wide World Importers

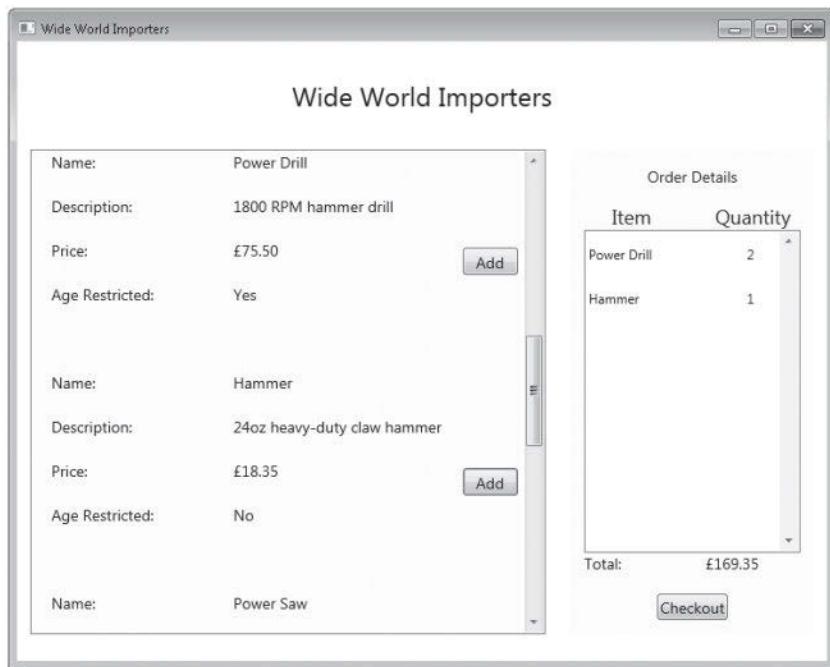
1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto Delegates, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 20\Windows X\Delegates na sua pasta Documentos.
3. No menu Debug, clique em Start Debugging.

O projeto é compilado e executado. Aparece um formulário exibindo os itens disponíveis, junto com um painel mostrando os detalhes do pedido (inicialmente ele está vazio). O formulário tem uma aparência diferente, dependendo de você estar executando o aplicativo Windows Store no Windows 8.1 ou o aplicativo WPF no Window 7 ou no Windows 8.

No Windows 8.1, o aplicativo Windows Store exibe os itens em um controle *GridView* que rola horizontalmente (esse é o estilo dos aplicativos Windows Store para exibir dados):



O aplicativo WPF executando no Windows 7 ou no Windows 8 exibe os itens em um controle *ListView* que rola verticalmente:



Fora a apresentação do aplicativo, o restante da funcionalidade é igual nos dois ambientes.

4. Selecione um ou mais itens e clique em Add para incluí-los na cesta de compras. Certifique-se de selecionar pelo menos um item com restrição à idade (*Age Restricted*).

Quando você adiciona um item, ele aparece no painel Order Details, no lado direito. Observe que, se você adiciona o mesmo item mais de uma vez, a quantidade é incrementada a cada clique (esta versão do aplicativo não implementa funcionalidade para remover itens da cesta).

5. No painel Order Details, clique em Checkout.

Aparece uma mensagem indicando que o pedido foi feito. O pedido recebe uma identificação exclusiva e essa identificação é exibida junto com o valor do pedido.

Se estiver usando o Windows 8.1, clique em Close para dispensar a mensagem. Se estiver usando o Windows 7 ou o Windows 8, clique em OK.

6. Retorne ao ambiente do Visual Studio 2013 e interrompa a depuração.

7. Utilizando o File Explorer no Windows 8 ou no Windows 8.1, ou o Windows Explorer no Windows 7, abra sua pasta Documentos. Você deverá ver dois arquivos chamados audit-*nnnnnnn.xml* (onde *nnnnnnn* é a identificação do pedido exibida anteriormente) e dispatch-*nnnnnnn.txt*. O primeiro arquivo foi gerado pelo componente de auditoria e o segundo é o aviso de despacho gerado pelo componente de expedição.



Nota Se não houver um arquivo audit-*nnnnnnn.xml*, então você não selecionou nenhum item com restrição à idade ao fazer o pedido. Nesse caso, volte para o aplicativo e crie um novo pedido, incluindo um ou mais desses itens.

8. Usando o Internet Explorer, abra o arquivo audit-*nnnnnnn.xml*. Esse arquivo contém uma lista dos itens com restrição à idade no pedido, junto com o número e a data do pedido. Ele deve ser semelhante a isto:

```
<?xml version="1.0"?>
- <Order Date="25/07/2013 18:20:18" ID="26a39a99-ebba-4dc4-a29a-6962d8638cfc">
  <Item Description="1800 RPM hammer drill" Product="Power Drill"/>
</Order>
```

Feche o Internet Explorer quando tiver terminado de examinar esse arquivo.

9. Abra o arquivo dispatch-*nnnnnnn.xml* com o Bloco de Notas. Esse arquivo contém um resumo do pedido, listando sua identificação e o valor. Ele deve ser semelhante a isto:

Order Summary:
Order ID: 26a39a99-ebba-4dc4-a29a-6962d8638cfc
Order Total: £169.35

Feche o Bloco de Notas, retorne ao Visual Studio 2013 e interrompa a depuração.

10. No Visual Studio, observe que a solução consiste nos seguintes projetos:

- **Delegates** Contém o aplicativo em si. O arquivo MainWindow.xaml define a interface do usuário e a lógica do aplicativo está contida no arquivo MainWindow.xaml.cs.

- **AuditService** Contém o componente que implementa o processo de auditoria. Ele é empacotado como uma biblioteca de classes e contém apenas uma classe, chamada *Auditor*. Essa classe expõe um único método público, chamado *AuditOrder*, que examina um pedido e gera o arquivo *audit-nnnnnn.xml*, caso o pedido contenha itens com restrição de idade.
- **DeliveryService** Contém o componente que executa a lógica de expedição, empacotada como uma biblioteca de classes. A funcionalidade de expedição está contida na classe *Shipper*, e ela fornece um método público chamado *ShipOrder*, que trata do processo de expedição e também gera o aviso de despacho.



Nota Você pode examinar o código das classes *Auditor* e *Shipper*, mas não é necessário entender o funcionamento interno desses componentes nesse aplicativo.

- **DataTypes** Contém os tipos de dados utilizados pelos outros projetos. A classe *Product* define os detalhes dos produtos exibidos pelo aplicativo e os dados dos produtos são armazenados na classe *ProductDataSource*. (Atualmente, o aplicativo utiliza um pequeno conjunto de produtos incorporados ao código. Em um sistema de produção, essa informação seria recuperada de um banco de dados ou de um web service.) As classes *Order* e *OrderItem* implementam a estrutura de um pedido; cada pedido contém um ou mais itens.

11. No projeto *Delegates*, exiba o arquivo *MainWindow.xaml.cs* na janela Code and Text Editor e examine os campos privados e o construtor *MainWindow* nesse arquivo. Os elementos importantes são como estes:

```
...
private Auditor auditor = null;
private Shipper shipper = null;

public MainWindow()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
}
```

Os campos *auditor* e *shipper* contêm referências para instâncias das classes *Auditor* e *Shipper*, e o construtor instancia esses objetos.

12. Localize o método *CheckoutButtonClicked*. Esse método é executado quando o usuário clica em Checkout para fazer um pedido. As primeiras linhas são parecidas com estas:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Executa o processamento de checagem de saída
        if (this.requestPayment())
        {
            this.auditor.AuditOrder(this.order);
            this.shipper.ShipOrder(this.order);
        }
    }
```

```

    }
    ...
}
...
}

```

Esse método implementa o processamento de checagem de saída. Ele solicita o pagamento do cliente e, então, chama o método *AuditOrder* do objeto *auditor*, seguido pelo método *ShipOrder* do objeto *shipper*. Qualquer lógica de negócio adicional exigida no futuro poderá ser adicionada aqui. O restante do código desse método, após a instrução *if*, se preocupa com o gerenciamento da interface do usuário: exibir a caixa de mensagem para o usuário e limpar o painel Order Details no lado direito do formulário.



Nota Por simplicidade, atualmente o método *requestPayment* desse aplicativo apenas retorna *true* para indicar que o pagamento foi recebido. No mundo real, esse método executaria o processamento do pagamento completo.

Embora o aplicativo funcione conforme o anunciado, os componentes Auditor e Shipper são fortemente integrados ao processamento de checagem de saída. Se esses componentes mudarem, o aplicativo precisará ser atualizado. Do mesmo modo, caso você precise incorporar mais lógica ao processo de checagem de saída, possivelmente realizado pelo uso de mais componentes, precisará corrigir essa parte do aplicativo.

No próximo exercício, você vai ver como pode desacoplar o processamento de negócios da operação de checagem de saída do aplicativo. O processamento de checagem de saída ainda precisará chamar os componentes Auditor e Shipper, mas deve ser extensível o suficiente para permitir a incorporação de mais componentes facilmente. Você faz isso criando um novo componente, chamado *CheckoutController*. O componente *CheckoutController* implementará a lógica de negócio do processo de checagem de saída e exporá um delegate que permite a um aplicativo especificar quais componentes e métodos devem ser incluídos dentro desse processo. O componente *CheckoutController* chamará esses métodos utilizando o delegate.

Crie o componente *CheckoutController*

1. No Solution Explorer, clique com o botão direito do mouse na solução *Delegates*, aponte para Add e, então, clique em New Project.
2. Na caixa de diálogo Add New Project, se estiver usando o Windows 8.1, no painel da esquerda, clique no nó Windows Store. Se estiver usando o Windows 7 ou o Windows 8, no painel da esquerda, clique no nó Windows. Nos dois casos (Windows 8.1 e Windows 7/Windows 8), no painel central, selecione o template Class Library. Na caixa Name, digite **CheckoutService** e clique em OK.

3. No Solution Explorer, expanda o projeto CheckoutService, clique com o botão direito do mouse no arquivo Class1.cs e, então, no menu de atalho que aparece, clique em Rename. Mude o nome do arquivo para **CheckoutController.cs** e pressione Enter. Quando solicitado, deixe o Visual Studio mudar o nome de todas as referências a Class1 para CheckoutController.
4. Clique com o botão direito do mouse na pasta References do projeto CheckoutService e, então, clique em Add Reference.
5. Na caixa de diálogo Reference Manager – CheckoutService, no painel da esquerda, clique em Solution. No painel central, selecione o projeto DataTypes e, então, clique em OK.

A classe *CheckoutController* utilizará a classe *Order* definida no projeto *DataTypes*.



Nota Se estiver usando o Windows 8.1 e receber um erro neste ponto, você provavelmente não criou o projeto CheckoutService com o template Class Library para aplicativos Windows Store, anteriormente neste exercício. Nesse caso, no Solution Explorer, clique com o botão direito do mouse no projeto CheckoutService e, então, clique em Remove. Usando o Windows Explorer, navegue para a pasta \Microsoft Press\Visual CSharp Step by Step\ Chapter 20\Windows 8.1\ Delegates na sua pasta Documentos, exclua a pasta CheckoutService e então volte ao passo 1 deste exercício.

6. Na janela Code and Text Editor que exibe o arquivo *CheckoutController.cs*, adicione à lista a seguinte diretiva *using* localizada no início do arquivo:

```
using DataTypes;
```

7. Adicione um tipo delegate público chamado *CheckoutDelegate* à classe *CheckoutController*, como mostrado em negrito a seguir:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
}
```

Esse tipo delegate pode ser utilizado para referenciar métodos que recebem um parâmetro *Order* e que não retornam um resultado. Isso corresponde à forma dos métodos *AuditOrder* e *ShipOrder* das classes *Auditor* e *Shipper*.

8. Adicione um delegate público, chamado *CheckoutProcessing*, baseado nesse tipo delegate, como segue:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;
}
```

9. Exiba o arquivo MainWindow.xaml.cs do projeto Delegates na janela Code and Text Editor e localize o método *requestPayment* (ele está no final do arquivo). Recorte esse método da classe *MainWindow*. Volte ao arquivo CheckoutController.cs e cole o método *requestPayment* na classe *CheckoutController*, como mostrado em negrito a seguir:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;

    private bool requestPayment()
    {
        // O processamento de pagamento fica aqui

        // A lógica de pagamento não está implementada neste exemplo
        // - simplesmente retorna true para indicar que o pagamento foi recebido
        return true;
    }
}
```

10. Adicione à classe *CheckoutController* o método *StartCheckoutProcessing* mostrado aqui em negrito:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;

    private bool requestPayment()
    {
        ...
    }

    public void StartCheckoutProcessing(Order order)
    {
        // Executa o processamento de checagem de saída
        if (this.requestPayment())
        {
            if (this.CheckoutProcessing != null)
            {
                this.CheckoutProcessing(order);
            }
        }
    }
}
```

Esse método fornece a funcionalidade de checagem de saída implementada anteriormente pelo método *CheckoutButtonClicked* da classe *MainWindow*. Ele solicita o pagamento e então examina o delegate *CheckoutProcessing*; se esse delegate não for *null* (ele referencia um ou mais métodos), chama o delegate. Todos os métodos referenciados por esse delegate serão executados nesse ponto.

11. No Solution Explorer, no projeto Delegate, clique com o botão direito do mouse na pasta References e, então, no menu de atalho que aparece, clique em Add Reference.

- 12.** Na caixa de diálogo Reference Manager – Delegates, no painel da esquerda, clique em Solution. No painel central, selecione o projeto CheckoutService e, então, clique em OK.

- 13.** Volte ao arquivo MainWindow.xaml.cs do projeto Delegates e adicione à lista a seguinte diretiva *using* na parte superior do arquivo:

```
using CheckoutService;
```

- 14.** Adicione à classe *MainWindow* uma variável privada chamada *checkoutController* de tipo *CheoutController* e inicialize-a com *null*, como mostrado em negrito a seguir:

```
public ... class MainWindow : ...
{
    ...
    private Auditor auditor = null;
    private Shipper shipper = null;
    private CheckoutController checkoutController = null;
    ...
}
```

- 15.** Localize o construtor de *MainWindow*. Após as instruções que criam os componentes *Auditor* e *Shipper*, instancie o componente *CheoutController*, como a seguir em negrito:

```
public MainWindow()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
    this.checkoutController = new CheckoutController();
}
```

- 16.** Adicione ao construtor as seguintes instruções, mostradas em negrito, após a instrução que você acabou de inserir:

```
public MainWindow()
{
    ...
    this.checkoutController = new CheckoutController();
    this.checkoutController.CheckoutProcessing += this.auditor.AuditOrder;
    this.checkoutController.CheckoutProcessing += this.shipper.ShipOrder;
}
```

Esse código adiciona referências para os métodos *AuditOrder* e *ShipOrder* dos objetos *Auditor* e *Shipper* ao delegate *CheckoutProcessing* do objeto *CheoutController*.

- 17.** Localize o método *CheckoutButtonClicked*. No bloco *try*, substitua o código existente que executa o processamento de checagem de saída (o bloco da instrução *if*) pela instrução mostrada aqui em negrito:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
```

```

{
    // Executa o processamento de checagem de saída
    this.checkoutController.StartCheckoutProcessing(this.order);

    // Exibe o resumo do pedido
    ...
}
...
}

```

Agora você desacoplou a lógica de checagem de saída dos componentes utilizados por esse processamento de checagem. A lógica de negócios na classe *MainWindow* especifica quais componentes o *CheckoutController* deve usar.

Teste o aplicativo

1. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
2. Quando o formulário da Wide World Importers aparecer, selecione alguns itens (inclua pelo menos um item com restrição de idade) e, então, clique em Checkout.
3. Quando a mensagem Order Placed aparecer, anote o número do pedido e, então, clique em Close ou em OK.
4. Troque para o Windows Explorer e acesse a sua pasta Documentos. Verifique se um novo arquivo audit-*nnnnnnn.xml* e um arquivo dispatch-*nnnnnnn.txt* foram criados, onde *nnnnnnn* é o número que identifica o novo pedido. Examine esses arquivos e verifique se eles contêm os detalhes do pedido.
5. Retorne ao Visual Studio 2013 e interrompa a depuração.

Expressões lambda e delegates

Todos os exemplos para adicionar um método a um delegate vistos até aqui utilizam o nome do método. Por exemplo, retornando ao cenário da fábrica automatizada mostrado anteriormente, você adiciona o método *StopFolding* do objeto *folder* ao delegate *stopMachinery*, desta maneira:

```
this.stopMachinery += folder.StopFolding;
```

Essa estratégia é muito útil se houver um método conveniente que corresponda à assinatura do delegate, mas e se esse não for o caso? Suponha que o método *StopFolding* tivesse na verdade a assinatura a seguir:

```
void StopFolding(int shutDownTime); // Desliga depois do número especificado de segundos
```

Essa assinatura é agora diferente daquela dos métodos *FinishWelding* e *PaintOff*e, desse modo, você não pode utilizar o mesmo delegate para lidar com os três métodos. Então, o que é possível fazer?

Crie um método adaptador

Uma maneira de contornar esse problema é criar outro método que chame *StopFolding*, mas que não tenha parâmetros, como mostrado:

```
void FinishFolding()
{
    folder.StopFolding(0); // Desliga imediatamente
}
```

Você pode então adicionar o método *FinishFolding* ao delegate *stopMachinery*, em vez do método *StopFolding*, utilizando a mesma sintaxe:

```
this.stopMachinery += folder.FinishFolding;
```

Quando o delegate *stopMachinery* é ativado, ele chama *FinishFolding* que, por sua vez, chama o método *StopFolding*, passando o parâmetro 0.



Nota O método *FinishFolding* é um exemplo clássico de adaptador: um método que converte (ou adapta) outro para dar a ele uma assinatura diferente. Esse padrão é muito comum e é um dos conjuntos de padrões documentados no livro *Design Patterns: Elements of Reusable Object-Oriented Software*, de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (Addison-Wesley Profissional, 1994).

Em muitos casos, métodos adaptadores como esse são pequenos, e é fácil perdê-los em um mar de métodos, especialmente em uma classe grande. Além disso, seu uso principal é na adaptação do método *StopFolding* para ser utilizado pelo delegate, e é pouco provável que seja chamado em outro momento. O C# fornece expressões lambda para situações como essa. As expressões lambda estão descritas no Capítulo 18, e existem mais exemplos delas anteriormente neste capítulo. No cenário da fábrica, você pode usar a seguinte expressão lambda:

```
this.stopMachinery += ((() => folder.StopFolding(0));
```

Ao chamar o delegate *stopMachinery*, ele executará o código definido pela expressão lambda, a qual, por sua vez, chamará o método *StopFolding* com o parâmetro apropriado.

As formas das expressões lambda

As expressões lambda podem assumir algumas formas sutilmente diferentes. Essas expressões, originalmente, faziam parte de uma notação matemática chamada *cálculo lambda*, o qual fornece uma notação para descrever funções. (Você pode considerar uma função como um método que retorna um valor.) Embora a linguagem C# tenha estendido a sintaxe e a semântica do cálculo lambda na implementação de expressões

lambda, boa parte dos princípios originais ainda se aplica. Veja alguns exemplos que mostram as diferentes formas de expressões lambda disponíveis no C#:

```
x => x * x // Uma expressão simples que retorna o quadrado do seu parâmetro
      // O tipo do parâmetro x é deduzido do contexto.

x => { return x * x ; } // Semanticamente igual à expressão
                        // anterior, mas usando um bloco de instruções em C# como
                        // corpo, em vez de uma expressão simples

(int x) => x / 2 // Uma expressão simples que retorna o valor do
                  // parâmetro dividido por 2
                  // O tipo do parâmetro x é declarado explicitamente.

() => folder.StopFolding() // Chamando um método
                           // A expressão não recebe parâmetros.
                           // A expressão pode ou não
                           // retornar um valor.

(x, y) => { x++; return x / y; } // Vários parâmetros; o compilador
                                   // deduz os tipos dos parâmetros.
                                   // O parâmetro x é passado por valor; portanto, o
                                   // efeito da operação ++ é
                                   // local à expressão.

(ref int x, int y) => { x++; return x / y; } // Vários parâmetros
                                              // com tipos explícitos
                                              // O parâmetro x é passado por
                                              // referência; portanto, o efeito
                                              // da operação ++ é permanente.
```

Resumindo, seguem algumas características das expressões lambda que você precisa conhecer:

- Se uma expressão lambda receber parâmetros, você os especifica nos parênteses à esquerda do operador =>. Você pode omitir os tipos dos parâmetros, e o compilador C# inferirá os tipos a partir do contexto da expressão lambda. Você pode passar parâmetros por referência (utilizando a palavra-chave *ref*), se quiser que a expressão lambda seja capaz de alterar os valores além de localmente, mas isso não é recomendável.
- As expressões lambda podem retornar valores, mas o tipo de retorno deve corresponder ao tipo do delegate ao qual eles estão sendo adicionados.
- O corpo de uma expressão lambda pode ser uma expressão simples ou um bloco de código C# composto de várias instruções, chamadas de método, definições de variáveis e outros itens de código.
- As variáveis definidas em um método de expressão lambda saem do escopo quando o método termina.
- Uma expressão lambda pode acessar e modificar todas as variáveis fora da expressão lambda que estão em escopo quando a expressão lambda é definida. Seja cuidadoso com esse recurso!

Expressões lambda e métodos anônimos

As expressões lambda foram adicionadas à linguagem C# na versão 3.0. O C# 2.0 introduziu os métodos anônimos, que podem realizar uma tarefa semelhante, mas não são tão flexíveis. Os métodos anônimos foram adicionados principalmente para que seja possível definir delegates sem criar um método nomeado – você simplesmente fornece a definição do corpo de método, em vez do nome de método, assim:

```
this.stopMachinery += delegate { folder.StopFolding(0); };
```

Você também pode passar um método anônimo como um parâmetro no lugar de um delegate, como mostrado aqui:

```
control.Add(delegate { folder.StopFolding(0); } );
```

Observe que sempre que você introduz um método anônimo, é preciso prefixá-lo com a palavra-chave *delegate*. Além disso, todos os parâmetros necessários são especificados nos parênteses após a palavra-chave *delegate*, como ilustrado no exemplo a seguir:

```
control.Add(delegate(int param1, string param2)
    { /* code that uses param1 and param2 */ ... } );
```

As expressões lambda fornecem uma sintaxe mais sucinta e natural do que a dos métodos anônimos e permeiam muitos dos aspectos mais avançados do C#, como veremos nos capítulos subsequentes deste livro. Falando em termos gerais, você deve utilizar expressões lambda em vez de métodos anônimos no seu código.

Ative notificações por meio de eventos

Você viu como declarar um tipo delegate, chamar um delegate e criar instâncias de delegate. Mas isso é apenas metade da história. Embora com delegates seja possível chamar qualquer método indiretamente, você ainda tem de ativar o delegate explicitamente. Em muitos casos, seria útil que o delegate fosse executado automaticamente quando algo significativo ocorresse. Por exemplo, no cenário da fábrica automatizada, pode ser vital conseguir ativar o delegate *stopMachinery* e interromper o equipamento se o sistema detectar o superaquecimento de uma máquina.

O .NET Framework fornece *eventos*, que você pode usar para definir e capturar ações significativas e providenciar para que um delegate seja chamado para tratar a situação. Muitas classes no .NET Framework expõem eventos. A maioria dos controles que você pode colocar em um formulário de um aplicativo Windows Store ou de um aplicativo WPF e a própria classe *Windows* utilizam eventos que permitem executar um código quando, por exemplo, o usuário clica em um botão ou digita algo em um campo. Você também pode declarar seus próprios eventos.

Declare um evento

Você declara um evento em uma classe projetada para atuar como origem de eventos. Uma *origem de eventos* normalmente é uma classe que monitora seu ambiente e dispara um evento quando algo significativo acontece. Na fábrica automatizada, uma origem de eventos pode ser uma classe que monitora a temperatura de cada máquina. A classe de monitoramento de temperatura dispararia um evento “superaquecimento da máquina” se detectasse que uma máquina excedeu seu limite de radiação térmica (isto é, esquentou demais). Um evento mantém uma lista de métodos a serem chamados quando ele é disparado. Às vezes, esses métodos são chamados de *subscribers* (assinantes). Eles devem ser preparados para tratar o “superaquecimento da máquina” e executar a ação corretiva necessária: desligar as máquinas.

Você declara um evento de maneira semelhante a como declara um campo. Mas, como os eventos serão utilizados com delegates, o tipo de um evento deve ser um delegate, e você deve iniciar a declaração com a palavra-chave *event*. Empregue a sintaxe a seguir para declarar um evento:

```
event delegateType eventName
```

Como exemplo, segue o delegate *StopMachineryDelegate* da fábrica automatizada. Ele foi realocado em uma nova classe chamada *TemperatureMonitor*, que fornece uma interface para as várias sondas que monitoram a temperatura do equipamento (esse é um local mais lógico para o evento do que a classe *Controller*):

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    ...
}
```

Você pode definir o evento *MachineOverheating* (“máquina superaquecendo”), que chamará o *stopMachineryDelegate*, como a seguir:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
}
```

A lógica (não mostrada) da classe *TemperatureMonitor* dispara o evento *MachineOverheating*, se necessário. Veremos como disparar um evento na seção “Disparando um evento”. Além disso, você adiciona métodos a um evento (um processo conhecido como *assinatura* ou inscrição para o evento), em vez de adicioná-los ao delegate no qual o evento está baseado. Examinaremos esse aspecto dos eventos a seguir.

Faça a inscrição em um evento

Como os delegates, os eventos tornam-se disponíveis para uso com um operador *+=*. Você se inscreve em um evento utilizando esse operador. Na fábrica automatizada, o software que controla cada máquina pode determinar que os métodos que desligam as máquinas sejam chamados quando o evento *MachineOverheating* for disparado, como mostrado aqui:

```

class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
}
...
TemperatureMonitor tempMonitor = new TemperatureMonitor();
...
tempMonitor.MachineOverheating += () => { folder.StopFolding(0); });
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;

```

Note que a sintaxe é a mesma empregada para adicionar um método a um delegate. Você até pode se inscrever utilizando uma expressão lambda. Quando o evento *tempMonitor.MachineOverheating* for executado, ele chamará todos os métodos inscritos e desligará as máquinas.

Cancele a inscrição em um evento

Sabendo que o operador `+=` é utilizado para anexar um delegate a um evento, provavelmente você pode imaginar que utilizará o operador `-=` para desvincular um delegate de um evento. Chamar esse operador remove o método da coleção de delegates internos do evento. Costuma-se chamar essa ação de *cancelar a inscrição* em um evento.

Dispare um evento

Você pode disparar um evento, exatamente como um delegate, chamando-o como um método. Quando um evento é disparado, todos os delegates anexados são chamados em sequência. Por exemplo, veja a classe *TemperatureMonitor* com um método *Notify* privado que dispara o evento *MachineOverheating*:

```

class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
    private void Notify()
    {
        if (this.MachineOverheating != null)
        {
            this.MachineOverheating();
        }
    }
    ...
}

```

Esse é um linguajar comum. A verificação `null` é necessária porque um campo de evento é implicitamente nulo e só se torna não nulo quando um método se inscreve nele utilizando o operador `+=`. Se tentar disparar um evento nulo, você obterá uma exceção `NullReferenceException`. Se o delegate que define o evento espera algum parâmetro, os argumentos apropriados deverão ser fornecidos quando você disparar o evento. Você verá alguns exemplos mais adiante.



Importante Os eventos têm um recurso de segurança incorporado muito útil. Um evento público (como `MachineOverheating`) só pode ser disparado por todos da classe que o define (a classe `TemperatureMonitor`). Qualquer tentativa de disparar o método fora da classe resulta em um erro de compilação.

Eventos de interface de usuário

Como mencionado anteriormente, as classes e os controles do .NET Framework utilizados para construir GUIs empregam eventos extensivamente. Por exemplo, a classe `Button` deriva da classe `ButtonBase`, herdando um evento público chamado `Click` do tipo `RoutedEventHandler`. O delegate `RoutedEventHandler` espera dois parâmetros: uma referência ao objeto que fez o evento disparar e um objeto `RoutedEventArgs` que contém informações adicionais sobre o evento:

```
public delegate void RoutedEventHandler(Object sender, RoutedEventArgs e);
```

A classe `Button` se parece com isto:

```
public class ButtonBase: ...
{
    public event RoutedEventHandler Click;
    ...
}

public class Button: ButtonBase
{
    ...
}
```

A classe `Button` dispara automaticamente o evento `Click` quando você clica no botão na tela. Esse arranjo facilita a criação de um delegate para um método escolhido e anexa esse delegate ao evento necessário. O exemplo a seguir mostra o código para um formulário WPF que contém um botão chamado `okay` e o código para conectar o evento `Click` do botão `okay` ao método `okayClick` (nos aplicativos Windows Store, os formulários operam de maneira semelhante):

```
public partial class Example : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    internal System.Windows.Controls.Button okay;
    ...
    void System.Windows.Markup.IComponentConnector.Connect(...)
    {
        ...
    }
}
```

```
        this.okay.Click += new System.Windows.RoutedEventHandler(this.okayClick);
        ...
    }
    ...
}
```

Em geral, você não vê esse código. Quando você utiliza a janela Design View no Visual Studio 2013 e configura a propriedade *Click* do botão *okay* como *okayClick* na descrição do formulário da Extensible Application Markup Language (XAML), o Visual Studio 2013 gera esse código automaticamente. Tudo o que você precisa fazer é escrever a lógica do seu aplicativo no método de tratamento de evento, *okayClick*, na parte do código à qual tem acesso, neste caso, no arquivo Example.xaml.cs:

```
public partial class Example : System.Windows.Window
{
    ...
    private void okayClick(object sender, RoutedEventArgs args)
    {
        // seu código para tratar o evento Click
    }
}
```

Os eventos gerados pelos vários controles GUI sempre seguem o mesmo padrão. São de um tipo delegate cuja assinatura tem um tipo de retorno *void* e dois argumentos. O primeiro argumento é sempre o emissor (a origem) do evento e o segundo argumento é sempre um argumento *EventArgs* (ou uma classe derivada de *EventArgs*).

Com o argumento *sender*, você pode reutilizar um único método para vários eventos. O método delegado pode examinar o argumento *sender* e responder de acordo. Por exemplo, você pode utilizar o mesmo método para inscrevê-lo ao evento *Click* de dois botões. (Você adiciona o mesmo método a dois eventos diferentes.) Quando o evento é disparado, o código no método pode examinar o argumento *sender* para se certificar de qual botão foi clicado.

Utilize eventos

No exercício anterior, você corrigiu o aplicativo da Wide World Importers para desacoplar a lógica de auditoria e expedição do processo de checagem de saída. A classe *CheckoutController* que você construiu chama os componentes de auditoria e expedição utilizando um delegate e não sabe nada sobre esses componentes ou sobre os métodos que está executando; isso é responsabilidade do aplicativo que cria o objeto *CheckoutController* e adiciona as referências apropriadas para o delegate. Mas poderia ser útil um componente alertar o aplicativo ao terminar seu processamento e permitir que este realize qualquer limpeza necessária.

À primeira vista, isso poderia parecer um pouco estranho — não é verdade que, quando o aplicativo chama o delegate no objeto *CheckoutController*, os métodos referenciados por esse delegate são executados e o aplicativo só continua na próxima instrução quando esses métodos tiverem terminado? Não necessariamente! O Capítulo 24, “Como melhorar o tempo de resposta empregando operações assíncronas”, demonstra que os métodos podem ser executados de forma assíncrona, e quando um método é chamado, talvez não tenha terminado antes que a execução continue na próxima instrução. Isso é especialmente verdade nos aplicativos Windows Store, nos quais operações prolongadas são executadas em threads de segundo plano para

permitir que a interface do usuário permaneça receptiva. No aplicativo da Wide World Importers, no método *CheckoutButtonClicked*, o código que ativa o delegate é seguido por uma instrução que exibe uma caixa de diálogo, com uma mensagem indicando que o pedido foi feito. No aplicativo Windows Store para Windows 8.1, o código é como este:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Executa o processamento de checagem de saída
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Exibe um resumo do pedido
        MessageDialog dlg = new MessageDialog(...);
        dlg.ShowAsync();
        ...
    }
    ...
}
```

O código para a versão WPF do aplicativo utilizado para Windows 7 e Windows 8 é semelhante, exceto que o WPF utiliza uma API diferente para exibir mensagens.

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Executa o processamento de checagem de saída
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Exibe um resumo do pedido
        MessageBox.Show(...);
        ...
    }
    ...
}
```

Na verdade, não há garantia de que o processamento realizado pelos métodos delegate tenha terminado quando a caixa de diálogo aparece, de modo que a mensagem poderá ser enganosa. É aí que um evento é inestimável. Os componentes Auditor e Shipper poderiam ambos publicar um evento assinado pelo aplicativo. Esse evento poderia ser disparado pelos componentes somente quando tivessem concluído seu processamento. Quando o aplicativo recebe esse evento, pode exibir a mensagem com segurança, sabendo que agora ela é precisa.

No exercício a seguir, você vai modificar as classes *Auditor* e *Shipper* para disparar um evento que ocorrerá quando elas tiverem concluído seu processamento. O aplicativo assinará o evento de cada componente e exibirá uma mensagem apropriada quando o evento ocorrer.

Adicione um evento à classe *CheckoutController*

- 1.** Retorne ao Visual Studio 2013 e exiba a solução *Delegates*.
- 2.** No projeto *AuditService*, abra o arquivo *Auditor.cs* na janela Code and Text Editor.
- 3.** Adicione um delegate público chamado *AuditingCompleteDelegate* à classe *Auditor*. Esse delegate deve especificar um método que aceita um parâmetro string chamado *message* e que retorna *void*. O código em negrito no exemplo a seguir mostra a definição desse delegate:

```
class Auditor
{
    public delegate void AuditingCompleteDelegate(string message);
    ...
}
```

- 4.** Adicione um evento público, chamado *AuditProcessingComplete*, à classe *Auditor*, após o delegate *AuditingCompleteDelegate*. Esse evento deve se basear no delegate *AuditingCompleteDelegate*, como mostrado em negrito no código a seguir:

```
class Auditor
{
    public delegate void AuditingCompleteDelegate(string message);
    public event AuditingCompleteDelegate AuditProcessingComplete;
    ...
}
```

- 5.** Localize o método *AuditOrder*. Esse é o método executado pelo delegate no objeto *CheckoutController*. Ele chama outro método privado, denominado *doAuditing*, para realmente efetuar a operação de auditoria. O método é semelhante a este:

```
public void AuditOrder(Order order)
{
    this.doAuditing(order);
}
```

- 6.** Role para baixo, até o método *doAuditing*. O código desse método está incluso em um bloco *try/catch*; ele usa as APIs XML da biblioteca de classes do .NET Framework para gerar uma representação em XML do pedido que está passando por auditoria e a salva em um arquivo. (Os detalhes exatos de como isso funciona estão fora dos objetivos deste capítulo, e variam entre a implementação de aplicativos Windows Store e a estratégia mais tradicional implementada pela versão WPF do código.)

Após o bloco *catch*, adicione um bloco *finally* que dispare o evento *AuditProcessingComplete*, como mostrado em negrito a seguir:

```
private async void doAuditing(Order order)
{
    List<OrderItem> ageRestrictedItems = findAgeRestrictedItems(order);
    if (ageRestrictedItems.Count > 0)
    {
        try
        {
            ...
        }
    }
}
```

```

        }
        catch (Exception ex)
        {
            ...
        }
    finally
    {
        if (this.AuditProcessingComplete != null)
        {
            this.AuditProcessingComplete(String.Format(
                "Audit record written for Order {0}", order.OrderID));
        }
    }
}
}

```

7. No projeto DeliveryService, abra o arquivo Shipper.cs na janela Code and Text Editor.
8. Adicione um delegate público chamado *ShippingCompleteDelegate* à classe *Shipper*. Esse delegate deve especificar um método que aceita um parâmetro string chamado *message* e que retorna *void*. O código em negrito no exemplo a seguir mostra a definição desse delegate:

```

class Shipper
{
    public delegate void ShippingCompleteDelegate(string message);
    ...
}

```

9. Adicione à classe *Shipper* um evento público chamado *ShipProcessingComplete*, baseado no delegate *ShippingCompleteDelegate*, como mostrado em negrito no código a seguir:

```

class Shipper
{
    public delegate void ShippingCompleteDelegate(string message);
    public event ShippingCompleteDelegate ShipProcessingComplete;
    ...
}

```

10. Localize o método *doShipping*, o qual executa a lógica de expedição. No método, após o bloco *catch*, adicione um bloco *finally* que dispare o evento *ShipProcessingComplete*, como mostrado em negrito aqui:

```

private async void doShipping(Order order)
{
    try
    {
        ...
    }
    catch (Exception ex)
    {
        ...
    }
    finally

```

```

    {
        if (this.ShipProcessingComplete != null)
        {
            this.ShipProcessingComplete(String.Format(
                "Dispatch note generated for Order {0}", order.OrderID));
        }
    }
}

```

- 11.** No projeto Delegates, exiba o layout do arquivo MainWindow.xaml na janela Design View. No painel XAML, role para baixo até o primeiro conjunto de itens *RowDefinition*. Se você está usando Windows 8.1, o código XAML aparece deste modo:

```

<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Grid Margin="12,0,12,0">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="10*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        ...
    </Grid>

```

Se você está usando Windows 7 ou Windows 8, o código XAML aparece deste modo:

```

<Grid Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="18*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    ...

```

- 12.** Se estiver usando o Windows 8.1, mude a propriedade *Height* do último item *RowDefinition* para **2***, como mostrado em negrito no código a seguir:

```

<Grid.RowDefinitions>
    ...
    <RowDefinition Height="10*"/>
    <RowDefinition Height="2*"/>>
</Grid.RowDefinitions>

```

Se estiver usando o Windows 7 ou o Windows 8, mude a propriedade *Height* do último item *RowDefinition* para **3***.

```

<Grid.RowDefinitions>
    ...
    <RowDefinition Height="18*"/>
    <RowDefinition Height="3*"/>>
</Grid.RowDefinitions>

```

Essa mudança no layout disponibiliza um pouco de espaço na parte inferior do formulário. Você vai usar esse espaço como uma área para exibir as mensagens recebidas dos componentes Auditor e Shipper, quando eles dispararem seus eventos. O Capítulo 25, “Implementação da interface do usuário de um aplicativo Windows Store”, fornece mais detalhes sobre a organização de interfaces com um controle *Grid*.

13. Role até a parte inferior do painel XAML. Se estiver usando o Windows 8.1, adicione os seguintes elementos *ScrollViewer* e *TextBlock*, mostrados em negrito, antes do penúltimo rótulo *</Grid>*:

```
...
</Grid>
<ScrollViewer Grid.Row="4" VerticalScrollBarVisibility="Visible">
    <TextBlock x:Name="messageBar" FontSize="18" />
</ScrollViewer>
</Grid>
</Grid>
</Page>
```

Se estiver usando o Windows 7, adicione os elementos *ScrollViewer* e *TextBlock*, mostrados no código a seguir, antes do último rótulo *</Grid>* (observe o tamanho de fonte diferente):

```
...
</Grid>
<ScrollViewer Grid.Row="4" VerticalScrollBarVisibility="Visible">
    <TextBlock x:Name="messageBar" FontSize="14" />
</ScrollViewer>
</Grid>
</Window>
```

Essa marcação adiciona um controle *TextBlock* chamado *messageBar* à área da parte inferior da tela. Você utilizará esse controle para exibir mensagens dos objetos *Auditor* e *Shipper*.

14. Exiba o arquivo *MainWindow.xaml.cs* na janela Code and Text Editor. Localize o método *CheckoutButtonClicked* e remova o código que exibe o resumo do pedido. O bloco *try* deve ser como este, após você ter excluído o código:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Executa o processamento de checagem de saída
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Limpa os detalhes do pedido para que o usuário possa começar de novo com um novo pedido
        this.order = new Order { Date = DateTime.Now, Items = new List<OrderItem>(),
            OrderID = Guid.NewGuid(), TotalValue = 0 };
        this.orderDetails.DataContext = null;
        this.orderValue.Text = String.Format("{0:C}", order.TotalValue);
        this.listViewHeader.Visibility = Visibility.Collapsed;
        this.checkout.IsEnabled = false;
    }
    catch (Exception ex)
    {
        ...
    }
}
```

15. Adicione um método privado chamado *displayMessage* à classe *MainWindow*. Esse método deve aceitar um parâmetro string chamado *message* e deve retornar *void*. No corpo desse método, adicione uma instrução que anexe o valor presente no parâmetro *message* à propriedade *Text* do controle *TextBlock* de *messageBar*, seguida de um caractere de nova linha, como mostrado em negrito a seguir:

```
private void displayMessage(string message)
{
    this.messageBar.Text += message + "\n";
}
```

Esse código faz com que a mensagem apareça na área de mensagens na parte inferior do formulário.

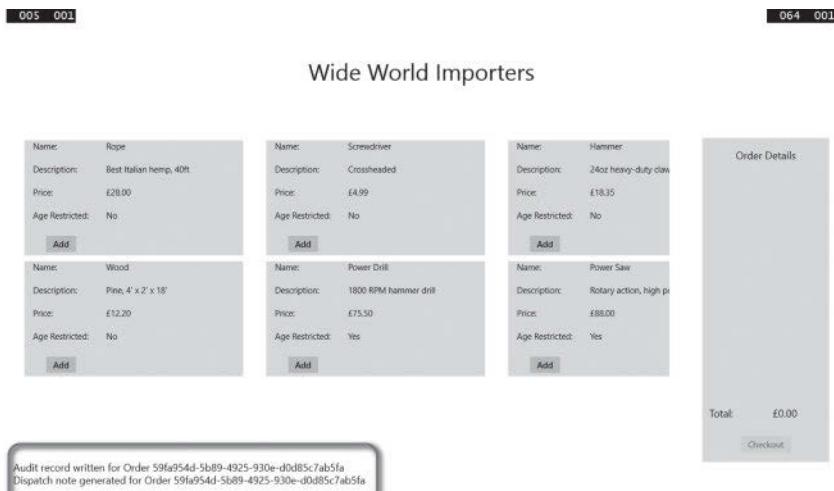
16. Localize o construtor da classe *MainWindow* e adicione o código mostrado aqui em negrito:

```
public MainWindow()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
    this.checkoutController = new CheckoutController();
    this.checkoutController.CheckoutProcessing += this.auditor.AuditOrder;
    this.checkoutController.CheckoutProcessing += this.shipper.ShipOrder;

    this.auditor.AuditProcessingComplete += this.displayMessage;
    this.shipper.ShipProcessingComplete += this.displayMessage;
}
```

Essas instruções assinam os eventos expostos pelos objetos *Auditor* e *Shipper*. Quando os eventos são disparados, o método *displayMessage* é executado. Observe que o mesmo método trata dos dois eventos.

17. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
18. Quando o formulário da Wide World Importers aparecer, selecione alguns itens (inclua pelo menos um item com restrição de idade) e, então, clique em Checkout.
19. Verifique que a mensagem “Audit record written” aparece no controle *TextBlock* na parte inferior do formulário, seguida da mensagem “Dispatch note generated”:



20. Faça mais pedidos e observe as novas mensagens que aparecem sempre que você clica em Checkout (talvez seja necessário rolar para baixo a fim de vê-las, quando a área de mensagem ficar cheia).
21. Quando terminar, retorne ao Visual Studio 2013 e interrompa a depuração.

Resumo

Neste capítulo, você aprendeu a utilizar delegates para fazer referência a métodos e chamar esses métodos. Viu também como definir expressões lambda que podem ser executadas por meio de um delegate. Por último, você aprendeu a definir e utilizar eventos para disparar a execução de um método.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 21.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Declarar um tipo delegate	<p>Escreva a palavra-chave <i>delegate</i>, seguida pelo tipo de retorno, seguida pelo nome do tipo delegate, seguida por qualquer tipo de parâmetro. Por exemplo:</p> <pre>delegate void myDelegate();</pre>

Para	Faça isto
Criar uma instância de um delegate inicializado com um único método específico	Utilize a mesma sintaxe que você utilizou para uma classe ou estrutura: escreva a palavra-chave <i>new</i> , seguida pelo nome do tipo (o nome do delegate), seguido pelo argumento entre parênteses. O argumento deve ser um método cuja assinatura corresponda exatamente à assinatura do delegate. Por exemplo:
	<pre>delegate void myDelegate(); private void myMethod() { ... } ... myDelegate del = new myDelegate(this.myMethod);</pre>
Chamar um delegate	Use a mesma sintaxe de uma chamada a um método. Por exemplo:
	<pre>myDelegate del; ... del();</pre>
Declarar um evento	Escreva a palavra-chave <i>event</i> , seguida pelo nome do tipo (o tipo deve ser um tipo delegate), seguido pelo nome do evento. Por exemplo:
	<pre>delegate void myDelegate(); class MyClass { public event myDelegate MyEvent; }</pre>
Fazer a inscrição a um evento	Crie uma instância do delegate (do mesmo tipo do evento) e vincule-a ao evento utilizando o operador <code>+=</code> . Por exemplo:
	<pre>class MyEventHandlingClass { private MyClass myClass = new MyClass(); ... public void Start() { myClass.MyEvent += new myDelegate (this.eventHandlingMethod); } private void eventHandlingMethod() { ... } }</pre>
	Você também pode fazer o compilador gerar automaticamente o novo delegate simplesmente especificando o método inscrito:
	<pre>public void Start() { myClass.MyEvent += this. eventHandlingMethod; }</pre>

Para	Faça isto
Cancelar a inscrição a um evento	<p>Crie uma instância do delegate (do mesmo tipo do evento) e desanexe a instância do delegate do evento utilizando o operador <code>--=</code>. Por exemplo:</p> <pre>class MyEventHandlingClass { private MyClass myClass = new MyClass(); ... public void Stop() { myClass.MyEvent -= new myDelegate (this.eventHandlingMethod); } ... } ou public void Stop() { myClass.MyEvent -= this. eventHandlingMethod; }</pre>
Disparar um evento	<p>Use a mesma sintaxe de uma chamada a um método. Forneça argumentos que combinem com o tipo de parâmetros esperados pelo delegate referenciado pelo evento. Não se esqueça de verificar se o evento é nulo. Por exemplo:</p> <pre>class MyClass { public event myDelegate MyEvent; ... private void RaiseEvent() { if (this.MyEvent != null) { this.MyEvent(); } } ... }</pre>

CAPÍTULO 21

Consulta a dados na memória usando expressões de consulta

Neste capítulo, você vai aprender a:

- Definir consultas em Language-Integrated Query para examinar o conteúdo de coleções enumeráveis.
- Utilizar métodos de extensão e operadores de consulta Language-Integrated Query.
- Explicar como a Language-Integrated Query posterga a avaliação de uma consulta e como você pode forçar a execução imediata e armazenar em cache os resultados de uma consulta Language-Integrated Query.

A maioria dos recursos da linguagem C# já foi mostrada. No entanto, evitamos, até este momento, um aspecto importante da linguagem que possivelmente é usado por diversos aplicativos: o suporte que o C# oferece para consultas de dados. Compreendemos que é possível definir estruturas e classes para modelar dados e que você pode utilizar coleções e arrays com a finalidade de armazenar dados temporariamente na memória. Mas como realizar tarefas comuns, como buscar itens em uma coleção que correspondam a um conjunto específico de critérios? Por exemplo, se você tiver uma coleção de objetos *Customer*, como encontrar todos os clientes (*customers*) localizados em Londres ou como poderá descobrir qual cidade tem mais clientes que adquiriram seus serviços? É possível escrever seu próprio código para iterar por uma coleção e examinar os campos em cada objeto, porém esses tipos de tarefas ocorrem com tanta frequência que os projetistas do C# optaram por incluir recursos na linguagem minimizando a quantidade de código a ser escrito. Neste capítulo, você utilizará esses recursos avançados da linguagem C# para consulta e manipulação de dados.

O que é a Language-Integrated Query?

Todos os aplicativos, exceto os triviais, precisam processar dados. Historicamente, a maioria dos aplicativos fornece uma lógica própria para efetuar essas operações. Mas essa estratégia pode fazer o código em um aplicativo tornar-se excessivamente amarrado à estrutura dos dados que processa. Se as estruturas dos dados mudarem, talvez você precise fazer um número significativo de alterações no código que trata os dados. Os projetistas do Microsoft .NET Framework pensaram bastante e por muito tempo nessas questões e decidiram facilitar a vida de um desenvolvedor de aplicativos, fornecendo recursos que abstraem o mecanismo que um aplicativo utiliza para consultar dados a partir do próprio código do aplicativo. Esses recursos são chamados de Language-Integrated Query ou LINQ.

Os criadores da LINQ fizeram um exame completo sobre como os sistemas de gerenciamento de banco de dados relacional, como o Microsoft SQL Server, separam a linguagem utilizada para consultar um banco de dados do formato interno dos dados no banco de dados. Os desenvolvedores que acessam um banco de dados SQL Server emitem instruções em Structured Query Language (SQL) para o sistema de gerenciamento de bancos de dados. A SQL fornece uma descrição de alto nível dos dados que o desenvolvedor quer recuperar, mas não indica exatamente como o sistema de gerenciamento de bancos de dados deve recuperá-los. Esses detalhes são controlados pelo próprio sistema de gerenciamento de bancos de dados. Consequentemente, um aplicativo que chama instruções de SQL não se importa com a maneira como o sistema de gerenciamento de bancos de dados armazena ou recupera fisicamente os dados. O formato empregado pelo sistema de gerenciamento de bancos de dados pode mudar (por exemplo, se uma nova versão é lançada) sem que o desenvolvedor do aplicativo precise modificar as instruções SQL utilizadas por este.

A LINQ fornece sintaxe e semântica muito semelhantes às da SQL, com vantagens parecidas. Você pode mudar a estrutura subjacente dos dados em consulta sem a necessidade de alterar o código que a realiza. Você deve estar ciente de que, embora a LINQ pareça semelhante à SQL, ela é muito mais flexível e pode tratar uma variedade mais ampla de estruturas lógicas de dados. Por exemplo, a LINQ pode tratar dados organizados hierarquicamente, como aqueles encontrados em um documento XML. Mas este capítulo se concentra no uso da LINQ de uma maneira relacional.

Como utilizar a LINQ em um aplicativo C#

Talvez a maneira mais fácil de explicar como utilizar os recursos do C# que suportam a LINQ seja trabalhar com alguns exemplos simples com base nos conjuntos de informações de clientes e endereços a seguir:

Informações de clientes

CustomerID	FirstName	LastName	CompanyName
1	Kim	Abercrombie	Alpine Ski House
2	Jeff	Hay	Coho Winery
3	Charlie	Herb	Alpine Ski House
4	Chris	Preston	Trey Research
5	Dave	Barnett	Wingtip Toys
6	Ann	Beebe	Coho Winery
7	John	Kane	Wingtip Toys
8	David	Simpson	Trey Research
9	Greg	Chapman	Wingtip Toys
10	Tim	Litton	Wide World Importers

Informações de endereços

CompanyName	City	Country
Alpine Ski House	Berne	Switzerland
Coho Winery	San Francisco	United States
Trey Research	New York	United States
Wingtip Toys	London	United Kingdom
Wide World Importers	Tetbury	United Kingdom

A LINQ exige que os dados sejam armazenados em uma estrutura de dados que implemente a interface *IEnumerable* ou *IEnumerable<T>*, como descrito no Capítulo 19, “Enumeração sobre coleções”. Não importa a estrutura utilizada (um array, um *HashSet<T>*, um *Queue<T>* ou qualquer outro tipo de coleção ou mesmo uma que você mesmo defina), contanto que seja enumerável. Mas, para facilitar, os exemplos deste capítulo supõem que as informações dos clientes e dos endereços são mantidas nos arrays *customers* e *addresses* mostrados no exemplo de código a seguir.



Nota Em um aplicativo do mundo real, você preencheria esses arrays lendo os dados a partir de um arquivo ou de um banco de dados.

```
var customers = new[] {
    new { CustomerID = 1, FirstName = "Kim", LastName = "Abercrombie",
          CompanyName = "Alpine Ski House" },
    new { CustomerID = 2, FirstName = "Jeff", LastName = "Hay",
          CompanyName = "Coho Winery" },
    new { CustomerID = 3, FirstName = "Charlie", LastName = "Herb",
          CompanyName = "Alpine Ski House" },
    new { CustomerID = 4, FirstName = "Chris", LastName = "Preston",
          CompanyName = "Trey Research" },
    new { CustomerID = 5, FirstName = "Dave", LastName = "Barnett",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 6, FirstName = "Ann", LastName = "Beebe",
          CompanyName = "Coho Winery" },
    new { CustomerID = 7, FirstName = "John", LastName = "Kane",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 8, FirstName = "David", LastName = "Simpson",
          CompanyName = "Trey Research" },
    new { CustomerID = 9, FirstName = "Greg", LastName = "Chapman",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 10, FirstName = "Tim", LastName = "Litton",
          CompanyName = "Wide World Importers" }
};

var addresses = new[] {
    new { CompanyName = "Alpine Ski House", City = "Berne",
          Country = "Switzerland" },
    new { CompanyName = "Coho Winery", City = "San Francisco",
          Country = "United States" },
    new { CompanyName = "Trey Research", City = "New York",
```

```

        Country = "United States"},  

        new { CompanyName = "Wingtip Toys", City = "London",  

            Country = "United Kingdom"},  

        new { CompanyName = "Wide World Importers", City = "Tetbury",  

            Country = "United Kingdom"}  

    };
}

```



Nota As seções “Selecione dados”, “Filtre dados”, “Ordene, agrupe e agregue dados” e “Junção de dados”, a seguir, mostram as capacidades básicas e a sintaxe para consultar dados utilizando métodos LINQ. Às vezes, a sintaxe pode tornar-se um pouco complexa, e você verá na seção “Utilizando operadores de consulta” que, na verdade, não é necessário lembrar como toda a sintaxe funciona. Mas é útil pelo menos examinar essas seções para entender como os operadores de consulta fornecidos com o C# realizam as tarefas.

Selecione dados

Suponha que você queira exibir uma lista consistindo no nome de cada cliente do array *customers*. Você pode realizar essa tarefa com o código a seguir:

```

IEnumerable<string> customerFirstNames =
    customers.Select(cust => cust.FirstName);

foreach (string name in customerFirstNames)
{
    Console.WriteLine(name);
}

```

Embora esse bloco de código seja bem curto, ele tem muitas funções e exige explicação, começando pelo uso do método *Select* do array *customers*.

Com o método *Select*, você pode recuperar dados específicos do array – neste caso, apenas o valor no campo *FirstName* de cada item no array. Como isso funciona? O parâmetro para o método *Select* é na verdade outro método que seleciona uma linha do array *customers* e retorna os dados selecionados a partir dessa linha. Você poderia definir seu próprio método personalizado para realizar essa tarefa, mas o mecanismo mais simples é utilizar uma expressão lambda para definir um método anônimo, como mostrado no exemplo anterior. Aqui, há três coisas importantes que você precisa entender:

- A variável *cust* é o parâmetro passado para o método. Você pode considerar *cust* como um alias para cada linha no array *customers*. O compilador deduz isso do fato de que você está chamando o método *Select* no array *customers*. Você pode utilizar qualquer identificador C# válido, em vez de *cust*.
- O método *Select* não recupera os dados nesse momento; ele simplesmente retorna um objeto enumerável que buscará os dados identificados pelo método *Select* quando você iterar por ele depois. Retornaremos a esse aspecto da LINQ na seção “LINQ e avaliação postergada”, mais adiante neste capítulo.

- O método *Select* não é realmente um método do tipo *Array*. É um método de extensão da classe *Enumerable*. A classe *Enumerable* está localizada no namespace *System.Linq* e fornece um conjunto substancial de métodos estáticos para consultar objetos que implementam a interface genérica *IEnumerable<T>*.

O exemplo anterior utiliza o método *Select* do array *customers* para gerar um objeto *IEnumerable<string>* chamado *customerFirstNames*. (Ele é do tipo *IEnumerable<string>* porque o método *Select* retorna uma coleção enumerável dos nomes dos clientes, que são strings.) A instrução *foreach* itera por essa coleção de strings, imprimindo o nome de cada cliente na seguinte sequência:

```
Kim  
Jeff  
Charlie  
Chris  
Dave  
Ann  
John  
David  
Greg  
Tim
```

Você pode agora exibir o nome de cada cliente. Como você busca o nome e o sobrenome de cada cliente? Essa tarefa é um pouco mais difícil. Se você examinar a definição do método *Enumerable.Select* no namespace *System.Linq* na documentação fornecida com o Microsoft Visual Studio 2013, verá que ele se parece com:

```
public static IEnumerable<TResult> Select<TSource, TResult> (  
    this IEnumerable<TSource> source,  
    Func<TSource, TResult> selector  
)
```

Na verdade, ele informa que *Select* é um método genérico que recebe dois parâmetros de tipo chamados *TSource* e *TResult* e outros dois comuns, chamados *source* e *selector*. *TSource* é o tipo da coleção que você está gerando para um conjunto enumerável de resultados (objetos *customer*, neste caso) e *TResult* é o tipo dos dados no conjunto enumerável de resultados (objetos *string*, neste caso). Lembre-se de que *Select* é um método de extensão; portanto, o parâmetro *source* é, na realidade, uma referência ao tipo que está sendo estendido (no exemplo, uma coleção genérica de objetos *customer* que implementa a interface *IEnumerable*). O parâmetro *selector* especifica um método genérico que identifica os campos a serem recuperados. (Lembre-se de que *Func* é o nome de um tipo de um delegate genérico no .NET Framework, que pode ser utilizado para encapsular um método genérico que retorna um resultado.) O método referenciado pelo parâmetro *selector* recebe um parâmetro *TSource* (neste caso, *customer*) e entrega (yield) uma coleção de objetos *TResult* (neste caso, *string*). O valor retornado pelo método *Select* é uma coleção enumerável de objetos *TResult* (novamente, *string*).



Nota O Capítulo 12, “Herança”, explica o funcionamento dos métodos de extensão e a função do primeiro parâmetro para um método de extensão.

O ponto importante a entender no parágrafo anterior é que o método *Select* retorna uma coleção enumerável com base em um único tipo. Se quiser que o enumerador retorne vários itens de dados, como o nome e o sobrenome de cada cliente, há pelo menos duas opções:

- Você pode concatenar os nomes e sobrenomes em uma única string no método *Select*, assim:

```
IEnumerable<string> customerNames =
    customers.Select(cust => String.Format("{0} {1}", cust.FirstName, cust.LastName));
```

- Você pode definir um novo tipo que envolva os nomes e sobrenomes e utilizar o método *Select* para construir instâncias desse tipo, assim:

```
class FullName
{
    public string FirstName{ get; set; }
    public string LastName{ get; set; }
}
...
IEnumerable<FullName> customerNames =
    customers.Select(cust => new FullName
    {
        FirstName = cust.FirstName,
        LastName = cust.LastName
    });

```

A segunda opção talvez seja preferível, mas se esse é o único uso que seu aplicativo faz do tipo *Names*, talvez você prefira utilizar um tipo anônimo, em vez de definir um novo tipo para uma única operação, assim:

```
var customerNames =
    customers.Select(cust => new { FirstName = cust.FirstName, LastName = cust.LastName } );
```

Observe o uso da palavra-chave *var* para definir o tipo da coleção enumerável. O tipo dos objetos na coleção é anônimo; portanto, você não pode especificar um tipo para os objetos na coleção.

Filtre dados

Com o método *Select*, você pode especificar ou *projetar* os campos que quer incluir na coleção enumerável. Mas talvez você também queira restringir as linhas que a coleção enumerável contém. Por exemplo, suponha que você queira listar os nomes de todas as empresas no array *addresses* localizadas apenas nos Estados Unidos. Para fazer isso, utilize o método *Where*, como a seguir:

```
IEnumerable<string> usCompanies =
    addresses.Where(addr => String.Equals(addr.Country, "United States"))
        .Select(usComp => usComp.CompanyName);

foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

Sintaticamente, o método *Where* é semelhante a *Select*. Ele espera um parâmetro que define um método que filtra os dados de acordo com os critérios especificados por você. Este exemplo utiliza outra expressão lambda. A variável *addr* é um alias para uma linha no array *addresses* e a expressão lambda retorna todas as linhas em que o campo *Country* corresponde à string *"United States"*. O método *Where* retorna uma coleção enumerável de linhas que contém cada campo da coleção original. O método *Select* é então aplicado a essas linhas para projetar apenas o campo *CompanyName* dessa coleção enumerável, a fim de retornar outra coleção enumerável de objetos *string*. (A variável *usComp* é um alias para o tipo de cada linha na coleção enumerável retornada pelo método *Where*.) O tipo do resultado dessa expressão completa é, portanto, *IEnumerable<string>*. É importante entender essa sequência de operações – o método *Where* é aplicado primeiro para filtrar as linhas, seguido pelo método *Select* para especificar os campos. A instrução *foreach* que itera por essa coleção exibe as seguintes empresas:

```
Coho Winery
Trey Research
```

Ordene, agrupe e agregue dados

Se estiver familiarizado com a linguagem SQL, você sabe que ela torna possível efetuar uma ampla variedade de operações relacionais, além de projeção e filtragem simples. Por exemplo, é possível especificar que você quer que os dados retornem em uma ordem específica e também agrupar as linhas retornadas de acordo com um ou mais campos-chave, sendo ainda possível calcular valores de resumo com base nas linhas em cada grupo. A LINQ fornece as mesmas funcionalidades.

Para recuperar dados em uma ordem específica, utilize o método *OrderBy*. Assim como os métodos *Select* e *Where*, *OrderBy* espera um método como argumento. Esse método identifica as expressões que você deseja utilizar para ordenar os dados. Por exemplo, você pode exibir o nome de cada empresa no array *addresses* em ordem crescente, assim:

```
IEnumerable<string> companyNames =
    addresses.OrderBy(addr => addr.CompanyName).Select(comp => comp.CompanyName);

foreach (string name in companyNames)
{
    Console.WriteLine(name);
}
```

Esse bloco de código exibe as empresas da tabela de endereços em ordem alfabética.

```
Alpine Ski House
Coho Winery
Trey Research
Wide World Importers
Wingtip Toys
```

Se quiser enumerar os dados em ordem decrescente, utilize o método *OrderByDescending*. Se quiser ordenar por mais de um valor-chave, utilize o método *ThenBy* ou *ThenByDescending* após *OrderBy* ou *OrderByDescending*.

Para agrupar os dados de acordo com valores comuns em um ou mais campos, você pode utilizar o método *GroupBy*. O exemplo a seguir mostra como agrupar as empresas no array *addresses* por país:

```
var companiesGroupedByCountry =
    addresses.GroupBy(addr => addr.Country);

foreach (var companiesPerCountry in companiesGroupedByCountry)
{
    Console.WriteLine("Country: {0}\t{1} companies",
        companiesPerCountry.Key, companiesPerCountry.Count());
    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine("\t{0}", companies.CompanyName);
    }
}
```

Agora, você deve reconhecer o padrão. O método *GroupBy* espera um método que especifica os campos pelos quais os dados são agrupados. Há, porém, algumas diferenças sutis entre o método *GroupBy* e os outros métodos que você viu até aqui.

O ponto mais interessante é que você não precisa utilizar o método *Select* para projetar os campos para o resultado. O conjunto enumerável retornado por *GroupBy* contém todos os campos da coleção-fonte original, mas as linhas são ordenadas em um conjunto de coleções enumeráveis com base no campo identificado pelo método especificado por *GroupBy*. Ou seja, o resultado do método *GroupBy* é um conjunto enumerável de grupos, cada um dos quais é um conjunto enumerável de linhas. No exemplo recém-mostrado, o conjunto enumerável *companiesGroupedByCountry* é um conjunto de países. Os próprios itens nesse conjunto são coleções enumeráveis contendo as empresas de cada país. O código que exibe as empresas em cada país utiliza um loop *foreach* para iterar pelo conjunto *companiesGroupedByCountry* a fim de entregar e exibir cada país sucessivamente. Depois, utiliza um loop *foreach* aninhado para iterar pelo conjunto de empresas em cada país. Observe no loop *foreach* externo que você pode acessar o valor em agrupamento utilizando o campo *Key* de cada item e calcular os dados de resumo para cada grupo utilizando métodos como *Count*, *Max*, *Min* e muitos outros. A saída gerada pelo código de exemplo se parece a:

```
Country: Switzerland 1 companies
        Alpine Ski House
Country: United States 2 companies
        Coho Winery
        Trey Research
```

```
Country: United Kingdom 2 companies
    Wingtip Toys
    Wide World Importers
```

Você pode utilizar vários outros métodos de resumo, como *Count*, *Max* e *Min*, diretamente sobre os resultados do método *Select*. Se quiser saber quantas empresas há no array *addresses*, utilize um bloco de código como este:

```
int numberOfCompanies = addresses.Select(addr => addr.CompanyName).Count();
Console.WriteLine("Number of companies: {0}", numberOfCompanies);
```

Observe que o resultado desses métodos é um único valor escalar, em vez de uma coleção enumerável. A saída do bloco de código anterior se parece com:

```
Number of companies: 5
```

Neste ponto, devo alertá-lo de um detalhe. Esses métodos de resumo não distinguem entre as linhas do conjunto subjacente que contêm valores duplicados nos campos que você está projetando. Isso significa que, rigorosamente falando, o exemplo anterior só mostra quantas linhas no array *addresses* contêm um valor no campo *CompanyName*. Se quiser descobrir quantos países diferentes são mencionados nessa tabela, você poderia experimentar fazer isto:

```
int numberOfCountries = addresses.Select(addr => addr.Country).Count();
Console.WriteLine("Number of countries: {0}", numberOfCountries);
```

A saída se parece com:

```
Number of countries: 5
```

De fato, há somente três diferentes países no array *addresses* – isso acontece porque United States (Estados Unidos) e United Kingdom (United Kingdom) ocorrem duas vezes. Você pode eliminar duplicatas do cálculo utilizando o método *Distinct*, assim:

```
int numberOfCountries =
    addresses.Select(addr => addr.Country).Distinct().Count();
Console.WriteLine("Number of countries: {0}", numberOfCountries);
```

A instrução *Console.WriteLine* agora gera a saída do resultado esperado:

```
Number of countries: 3
```

Junção de dados

Assim como a SQL, a LINQ oferece a capacidade de fazer junção de vários conjuntos de dados sobre um ou mais campos-chave comuns. O exemplo a seguir mostra como exibir o nome e sobrenome de cada cliente, juntamente com o nome do país onde ele está localizado:

```
var companiesAndCustomers = customers
    .Select(c => new { c.FirstName, c.LastName, c.CompanyName })
    .Join(addresses, custs => custs.CompanyName, addrs => addrs.CompanyName,
    (custs, addrs) => new {custs.FirstName, custs.LastName, addrs.Country});
```

```
foreach (var row in companiesAndCustomers)
{
    Console.WriteLine(row);
}
```

Os nomes e sobrenomes dos clientes estão disponíveis no array *customers*, mas o país de cada empresa em que os clientes trabalham é armazenado no array *addresses*. A chave comum entre o array *customers* e o array *addresses* é o nome da empresa. O método *Select* especifica os campos de interesse no array *customers* (*FirstName* e *LastName*), juntamente com o campo contendo a chave comum (*CompanyName*). Você utiliza o método *Join* para fazer a junção dos dados identificados pelo método *Select* com outra coleção enumerável. Os parâmetros para o método *Join* são:

- A coleção enumerável com a qual fazer a junção.
- Um método que identifica os campos-chave comuns a partir dos dados identificados pelo método *Select*.
- Um método que identifica os campos-chave comuns com base nos quais será feita a junção dos dados selecionados.
- Um método que especifica as colunas que você quer no conjunto de resultados enumeráveis retornado pelo método *Join*.

Neste exemplo, o método *Join* faz a junção da coleção enumerável contendo os campos *FirstName*, *LastName* e *CompanyName* do array *customers* com as linhas do array *addresses*. Os dois conjuntos de dados são unidos onde o valor no campo *CompanyName* do array *customers* corresponde ao valor no campo *CompanyName* do array *addresses*. O conjunto de resultados compreende linhas contendo os campos *FirstName* e *LastName* provenientes do array *customers* com o campo *Country* proveniente do array *addresses*. O código que dá saída aos dados da coleção *companiesAndCustomers* exibe as seguintes informações:

```
{ FirstName = Kim, LastName = Abercrombie, Country = Switzerland }
{ FirstName = Jeff, LastName = Hay, Country = United States }
{ FirstName = Charlie, LastName = Herb, Country = Switzerland }
{ FirstName = Chris, LastName = Preston, Country = United States }
{ FirstName = Dave, LastName = Barnett, Country = United Kingdom }
{ FirstName = Ann, LastName = Beebe, Country = United States }
{ FirstName = John, LastName = Kane, Country = United Kingdom }
{ FirstName = David, LastName = Simpson, Country = United States }
{ FirstName = Greg, LastName = Chapman, Country = United Kingdom }
{ FirstName = Tim, LastName = Litton, Country = United Kingdom }
```



Nota É importante lembrar que as coleções na memória não são o mesmo que as tabelas em um banco de dados relacional, e os dados que elas contêm não estão sujeitos às mesmas restrições de integridade de dados. Em um banco de dados relacional, poderia ser aceitável supor que cada cliente tem uma empresa correspondente e que cada empresa tem um endereço próprio. As coleções não impõem o mesmo nível de integridade de dados, ou seja, você pode facilmente ter um cliente que referencia uma empresa que não existe no array *addresses* e ter essa mesma empresa ocorrendo mais de uma vez no array *addresses*. Nessas situações, os resultados que você obtém talvez sejam exatos, mas inesperados. As operações de junção funcionam melhor quando você entende completamente os relacionamentos entre os dados que está usando em uma junção.

Utilize operadores de consulta

As seções anteriores mostraram muitos recursos disponíveis para consultar dados na memória utilizando os métodos de extensão para a classe *Enumerable* definida no namespace *System.Linq*. A sintaxe utiliza vários recursos avançados da linguagem C#, e o código resultante pode ser bem difícil de entender e manter. A fim de facilitar essa tarefa, os projetistas do C# adicionaram operadores de consulta à linguagem, com os quais você pode empregar recursos da LINQ utilizando uma sintaxe mais parecida com a SQL.

Como vimos nos exemplos mostrados anteriormente neste capítulo, você pode recuperar o nome de cada cliente assim:

```
IEnumerable<string> customerFirstNames =
    customers.Select(cust => cust.FirstName);
```

Você pode reformular essa instrução utilizando os operadores de consulta *from* e *select*, desta maneira:

```
var customerFirstNames = from cust in customers
                           select cust.FirstName;
```

Em tempo de compilação, o compilador C# resolve essa expressão para o método *Select* correspondente. O operador *from* define um alias para a coleção-fonte e o operador *select* especifica os campos a recuperar utilizando esse alias. O resultado é uma coleção enumerável de nomes de cliente. Se estiver familiarizado com a SQL, observe que o operador *from* ocorre antes do operador *select*.

Da mesma maneira, para recuperar os nomes e sobrenomes de cada cliente, você pode utilizar a instrução a seguir. (Talvez você queira rever o exemplo anterior da mesma instrução com base no método de extensão *Select*.)

```
var customerNames = from cust in customers
                           select new { cust.FirstName, cust.LastName };
```

Utilize o operador *where* para filtrar os dados. O exemplo a seguir mostra como retornar os nomes das empresas sediadas nos Estados Unidos a partir do array *addresses*:

```
var usCompanies = from a in addresses
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

Para ordenar os dados, utilize o operador *orderby*, desta maneira:

```
var companyNames = from a in addresses
                     orderby a.CompanyName
                     select a.CompanyName;
```

Você pode agrupar os dados empregando o operador *group* da seguinte maneira:

```
var companiesGroupedByCountry = from a in addresses
                                    group a by a.Country;
```

Observe que, como acontece com o exemplo anterior, que mostra a maneira de agrupar os dados, você não fornece o operador *select* e pode iterar pelos resultados utilizando exatamente o mesmo código, assim:

```
foreach (var companiesPerCountry in companiesGroupedByCountry)
{
    Console.WriteLine("Country: {0}\n{1} companies",
                      companiesPerCountry.Key, companiesPerCountry.Count());
    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine("\t{0}", companies.CompanyName);
    }
}
```

Você pode chamar as funções de resumo, como *Count*, na coleção retornada por uma coleção enumerável, desta maneira:

```
int numberOfCompanies = (from a in addresses
                           select a.CompanyName).Count();
```

Observe que você coloca a expressão entre parênteses. Se quiser ignorar os valores duplicados, utilize o método *Distinct*:

```
int numberOfCountries = (from a in addresses
                           select a.Country).Distinct().Count();
```



Dica Em muitos casos, você provavelmente só quer contar o número de linhas em uma coleção, em vez do número de valores em um campo ao longo de todas as linhas nessa coleção. Nesse caso, você pode chamar o método *Count* diretamente sobre a coleção original, assim:

```
int numberOfCompanies = addresses.Count();
```

Você pode utilizar o operador *join* para combinar duas coleções em uma chave comum. O exemplo a seguir mostra uma consulta que retorna clientes e endereços na coluna *CompanyName* em cada coleção, desta vez reformulada utilizando o operador *join*. Utilize a cláusula *on* com o operador *equals* para especificar como as duas coleções estão relacionadas.



Nota Atualmente, a LINQ suporta apenas equi-joins (junções baseadas na igualdade). Se você é desenvolvedor de bancos de dados acostumado com a SQL, talvez conheça as junções baseadas em outros operadores, como > e <, mas a LINQ não fornece esses recursos.

```
var countriesAndCustomers = from a in addresses
                             join c in customers
                             on a.CompanyName equals c.CompanyName
                             select new { c.FirstName, c.LastName, a.Country };
```



Nota Ao contrário da SQL, a ordem das expressões na cláusula *on* de uma expressão em LINQ é importante. Você precisa posicionar o item a partir do qual você está fazendo a junção (referenciando os dados na coleção na cláusula *from*) à esquerda do operador *equals*, e o item com o qual você está fazendo a junção (referenciando os dados na coleção na cláusula *join*) à direita.

A LINQ fornece vários outros métodos para resumir informações, fazer junção, agrupar e pesquisar dados. Esta seção abrange apenas os recursos mais comuns. Por exemplo, a LINQ fornece os métodos *Intersect* e *Union*, que você pode utilizar para efetuar operações em nível de conjuntos de dados. Ela também fornece métodos como *Any* e *All*, que você pode utilizar para determinar se pelo menos um item em uma coleção ou cada item em uma coleção corresponde a um predicado especificado. Você pode particionar os valores em uma coleção enumerável utilizando os métodos *Take* e *Skip*. Para obter mais informações, consulte a matéria da seção sobre LINQ na documentação fornecida com o Visual Studio 2013.

Consulte dados em objetos *Tree<TItem>*

Os exemplos que vimos até agora neste capítulo mostraram como consultar os dados de um array. Você pode utilizar exatamente as mesmas técnicas para qualquer classe de coleção que implemente a interface genérica *IEnumerable<T>*. No próximo exercício, você definirá uma nova classe para modelar os funcionários de uma empresa. Você criará um objeto *BinaryTree* que contém uma coleção de objetos *Employees* e então utilizará a LINQ para consultar essas informações. Primeiro, vai chamar os métodos de extensão LINQ diretamente e em seguida modificará seu código para que ele utilize operadores de consulta.

Recupere os dados de uma *BinaryTree* utilizando os métodos de extensão

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra a solução QueryBinaryTree, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 21\Windows X\QueryBinaryTree na sua pasta Documentos. O projeto contém o arquivo Program.cs, que define a classe *Program* com os métodos *Main* e *doWork* que vimos nos exercícios anteriores.
3. No Solution Explorer, clique com o botão direito do mouse no projeto QueryBinaryTree, aponte para Add e então clique em Class. Na caixa de diálogo Add New Item – QueryBinaryTree, na caixa Name, digite **Employee.cs** e clique em Add.
4. Adicione à classe *Employee* as propriedades automáticas mostradas em negrito a seguir:

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Department { get; set; }
    public int Id { get; set; }
}
```

5. Adicione o método *ToString*, mostrado em negrito, ao código após a classe *Employee*. Os tipos no .NET Framework utilizam esse método ao converter o objeto em uma representação de string, como ao exibi-lo utilizando a instrução *Console.WriteLine*.

```
class Employee
{
    ...
    public override string ToString()
    {
        return String.Format("Id: {0}, Name: {1} {2}, Dept: {3}",
            this.Id, this.FirstName, this.LastName, this.Department);
    }
}
```

6. Modifique a definição da classe *Employee* para implementar a interface *IComparable<Employee>*, como mostrado:

```
class Employee : IComparable<Employee>
{}
```

Esse passo é necessário porque a classe *BinaryTree* especifica que seus elementos devem ser “comparáveis”.

7. Clique com o botão direito do mouse na interface *IComparable<Employee>* na definição da classe, aponte para Implement Interface e então clique em Implement Interface Explicitly.

Essa ação gera uma implementação padrão do método *CompareTo*. Lembre-se de que a classe *BinaryTree* chama esse método quando precisa comparar elementos ao inseri-los na árvore.

8. Substitua o corpo do método *CompareTo* pelo código mostrado em negrito a seguir. Essa implementação do método *CompareTo* compara objetos *Employee* com base no valor do campo *Id*.

```
int IComparable<Employee>.CompareTo(Employee other)
{
    if (other == null)
    {
        return 1;
    }

    if (this.Id > other.Id)
    {
        return 1;
    }

    if (this.Id < other.Id)
    {
        return -1;
    }

    return 0;
}
```



Nota No SQL Server, o sinal de adição (+) é usado para concatenar strings.

9. No Solution Explorer, clique com o botão direito do mouse na solução *QueryBinaryTree*, aponte para Add e então clique em Existing Project. Na caixa de diálogo Add Existing Project, acesse a pasta Microsoft Press\Visual CSharp Step By Step\Chapter 21\Windows X\BinaryTree na sua pasta Documentos, clique no projeto *BinaryTree* e então clique em Open.

O projeto *BinaryTree* contém uma cópia da classe *BinaryTree* enumerável que você implementou no Capítulo 19.

10. No Solution Explorer, clique com o botão direito do mouse no projeto *QueryBinaryTree* e, então, no menu de atalho que se abre, clique em Add Reference. Na caixa de diálogo Reference Manager – *QueryBinaryTree*, no painel da esquerda, clique em Solution. No painel central, selecione o projeto *BinaryTree* e, então, clique em OK.
11. Exiba o arquivo Program.cs do projeto *QueryBinaryTree* na janela Code and Text Editor e verifique que a lista de diretivas *using* no início do arquivo contém a seguinte linha de código:

```
using System.Linq;
```

12. Adicione a seguinte diretiva *using*, que coloca o namespace *BinaryTree* no escopo, à lista localizada no início do arquivo Program.cs:

```
using BinaryTree;
```

- 13.** No método *doWork* da classe *Program*, remova o comentário `// TODO:` e adicione as seguintes instruções mostradas em negrito a fim de construir e preencher uma instância da classe *BinaryTree*:

```
static void doWork()
{
    Tree<Employee> empTree = new Tree<Employee>(
        new Employee { Id = 1, FirstName = "Kim", LastName = "Abercrombie",
            Department = "IT" });
    empTree.Insert( new Employee { Id = 2, FirstName = "Jeff", LastName = "Hay",
        Department = "Marketing" });
    empTree.Insert( new Employee { Id = 4, FirstName = "Charlie", LastName = "Herb",
        Department = "IT" });
    empTree.Insert( new Employee { Id = 6, FirstName = "Chris", LastName = "Preston",
        Department = "Sales" });
    empTree.Insert( new Employee { Id = 3, FirstName = "Dave", LastName = "Barnett",
        Department = "Sales" });
    empTree.Insert( new Employee { Id = 5, FirstName = "Tim", LastName = "Litton",
        Department="Marketing" });
}
```

- 14.** Adicione as seguintes instruções mostradas em negrito ao final do método *doWork*. Esse código chama o método *Select* para listar os departamentos encontrados na árvore binária.

```
static void doWork()
{
    ...
    Console.WriteLine("List of departments");
    var depts = empTree.Select(d => d.Department);

    foreach (var dept in depts)
    {
        Console.WriteLine("Department: {0}", dept);
    }
}
```

- 15.** No menu Debug, clique em Start Without Debugging.

O aplicativo deve enviar para a saída a seguinte lista de departamentos:

```
List of departments
Department: IT
Department: Marketing
Department: Sales
Department: IT
Department: Marketing
Department: Sales
```

Cada departamento ocorre duas vezes porque há dois funcionários em cada departamento. A ordem dos departamentos é determinada pelo método *CompareTo* da classe *Employee*, a qual utiliza a propriedade *Id* de cada funcionário para ordenar os dados. O primeiro departamento é para o funcionário com o valor de *Id* 1, o segundo departamento é para o funcionário com o valor de *Id* 2 e assim por diante.

16. Pressione Enter para retornar ao Visual Studio 2013.
17. No método *doWork* da classe *Program*, modifique a instrução que cria a coleção enumerável de departamentos, como mostrado em negrito no exemplo a seguir:

```
var depts = empTree.Select(d => d.Department).Distinct();
```

O método *Distinct* remove as linhas duplicadas da coleção enumerável.

18. No menu Debug, clique em Start Without Debugging.

Observe que o aplicativo agora exibe cada departamento somente uma vez, assim:

```
List of departments  
Department: IT  
Department: Marketing  
Department: Sales
```

19. Pressione Enter para retornar ao Visual Studio 2013.
20. Adicione as seguintes instruções mostradas em negrito ao final do método *doWork*. Esse bloco de código utiliza o método *Where* para filtrar os funcionários e retorna somente aqueles do departamento de TI. O método *Select* retorna a linha inteira, em vez de projetar colunas específicas.

```
static void doWork()  
{  
    ...  
    Console.WriteLine("\nEmployees in the IT department");  
    var ITEmployees =  
        empTree.Where(e => String.Equals(e.Department, "IT"))  
        .Select(emp => emp);  
  
    foreach (var emp in ITEmployees)  
    {  
        Console.WriteLine(emp);  
    }  
}
```

21. Adicione o código mostrado em negrito a seguir ao final do método *doWork*, após o código do passo anterior. Esse código utiliza o método *GroupBy* para agrupar os funcionários encontrados na árvore binária por departamento. A instrução *foreach* externa itera por cada grupo, exibindo o nome do departamento. A instrução *foreach* interna exibe os nomes dos funcionários de cada departamento.

```
static void doWork()  
{  
    ...  
    Console.WriteLine("\nAll employees grouped by department");  
    var employeesByDept = empTree.GroupBy(e => e.Department);
```

```

foreach (var dept in employeesByDept)
{
    Console.WriteLine("Department: {0}", dept.Key);
    foreach (var emp in dept)
    {
        Console.WriteLine("\t{0} {1}", emp.FirstName, emp.LastName);
    }
}

```

- 22.** No menu Debug, clique em Start Without Debugging. Verifique se a saída do aplicativo se parece com:

```

List of departments
Department: IT
Department: Marketing
Department: Sales

Employees in the IT department
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 4, Name: Charlie Herb, Dept: IT

All employees grouped by department
Department: IT
    Kim Abercrombie
    Charlie Herb
Department: Marketing
    Jeff Hay
    Tim Litton
Department: Sales
    Dave Barnett
    Chris Preston

```

- 23.** Pressione Enter para retornar ao Visual Studio 2013.

Recupere os dados de uma *BinaryTree* utilizando operadores de consulta

- 1.** No método *doWork*, transforme em comentário a instrução que gera a coleção enumerável dos departamentos e a substitua pela instrução equivalente mostrada em negrito, utilizando os operadores de consulta *from* e *select*:

```

// var depts = empTree.Select(d => d.Department).Distinct();
var depts = (from d in empTree
            select d.Department).Distinct();

```

- 2.** Transforme em comentário a instrução que gera a coleção enumerável dos empregados no departamento de TI e a substitua pelo seguinte código mostrado em negrito:

```
// var ITEmployees =
//   empTree.Where(e => String.Equals(e.Department, "IT"))
//   .Select(emp => emp);
var ITEmployees = from e in empTree
                  where String.Equals(e.Department, "IT")
                  select e;
```

- 3.** Transforme em comentário a instrução que gera a coleção enumerável que agrupa os funcionários por departamento e a substitua pela instrução mostrada em negrito no código a seguir:

```
// var employeesByDept = empTree.GroupBy(e => e.Department);
var employeesByDept = from e in empTree
                      group e by e.Department;
```

- 4.** No menu Debug, clique em Start Without Debugging. Observe que o programa exibe os mesmos resultados de antes.

```
List of departments
Department: IT
Department: Marketing
Department: Sales

Employees in the IT department
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 4, Name: Charlie Herb, Dept: IT

All employees grouped by department
Department: IT
    Kim Abercrombie
    Charlie Herb
Department: Marketing
    Jeff Hay
    Tim Litton
Department: Sales
    Dave Barnett
    Chris Preston
```

- 5.** Pressione Enter para retornar ao Visual Studio 2013.

LINQ e avaliação postergada

Ao utilizar a LINQ para definir uma coleção enumerável, com métodos de extensão LINQ ou com operadores de consulta, você deve lembrar que o aplicativo na verdade não constrói a coleção no momento em que o método de extensão LINQ é executado; a coleção é enumerada somente quando você itera por ela. Isso significa que os dados na coleção original podem mudar entre a execução de uma consulta LINQ e a recuperação dos dados que a consulta identifica; você sempre buscará os dados mais atualizados. Por exemplo, a consulta a seguir (vista anteriormente) define uma coleção enumerável das empresas sediadas nos Estados Unidos:

```
var usCompanies = from a in addresses
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

Os dados no array `addresses` não são recuperados e qualquer condição especificada no filtro `Where` só é avaliada quando você itera pela coleção `usCompanies`:

```
foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

Se você modificar os dados no array *addresses* entre a definição da coleção *us-Companies* e a iteração pela coleção (por exemplo, se adicionar uma nova empresa sediada nos Estados Unidos), verá estes novos dados. Essa estratégia é chamada *avaliação postergada*.

Você pode forçar a avaliação de uma consulta LINQ quando ela é definida e gerar uma coleção estática, armazenada em cache. Essa coleção é uma cópia dos dados originais e não irá mudar se os dados na coleção mudarem. A LINQ fornece o método *ToList* para construir um objeto *List* estático contendo uma cópia armazenada em cache dos dados. Você o utiliza assim:

```
var usCompanies = from a in addresses.ToList()
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

Desta vez, a lista de empresas é fixada quando você cria a consulta. Se adicionar mais empresas norte-americanas ao array *addresses*, você não vaivê-las ao iterar pela coleção *usCompanies*. A LINQ também fornece o método *ToArray*, que armazena a coleção em cache como um array.

No exercício final deste capítulo, você vai comparar os efeitos do uso da avaliação postergada de uma consulta LINQ com a geração de uma coleção armazenada em cache.

Examine os efeitos da avaliação postergada e os da armazenada em cache de uma consulta LINQ

1. Retorne ao Visual Studio 2013, exiba o projeto QueryBinaryTree e então edite o arquivo Program.cs.
 2. Transforme em comentário o conteúdo do método *doWork* separadamente das instruções que constroem a árvore binária *empTree*, como mostrado aqui:

```
empTree.Insert( new Employee { Id = 5, FirstName = "Tim", LastName = "Litton",
                                Department="Marketing" });
// transforme em comentário o restante do método
...
}
```



Dica Você pode transformar em comentário um bloco de código selecionando o bloco inteiro na janela Code and Text Editor e clicando no botão Comment Out The Selected Lines na barra de ferramentas ou pressionando Ctrl+E e depois C.

3. Adicione as seguintes instruções mostradas em negrito ao método *doWork*, após o código que cria e preenche a árvore binária *empTree*:

```
static void doWork()
{
    ...
    Console.WriteLine("All employees");
    var allEmployees = from e in empTree
                       select e;

    foreach (var emp in allEmployees)
    {
        Console.WriteLine(emp);
    }
    ...
}
```

Esse código gera uma coleção enumerável de funcionários chamada *allEmployees* e então itera por essa coleção, exibindo os detalhes de cada funcionário.

4. Adicione o código a seguir imediatamente após as instruções que você digitou no passo anterior:

```
static void doWork()
{
    ...
    empTree.Insert(new Employee
    {
        Id = 7,
        FirstName = "David",
        LastName = "Simpson",
        Department = "IT"
    });
    Console.WriteLine("\nEmployee added");

    Console.WriteLine("All employees");
    foreach (var emp in allEmployees)
    {
        Console.WriteLine(emp);
    }
    ...
}
```

Essas instruções adicionam um novo funcionário à árvore *empTree* e então iteram mais uma vez pela coleção *allEmployees*.

5. No menu Debug, clique em Start Without Debugging. Observe que a saída do aplicativo se parece com:

```
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
```

```
Employee added
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
Id: 7, Name: David Simpson, Dept: IT
```

Observe que na segunda vez em que o aplicativo itera pela coleção *allEmployees*, a lista exibida inclui David Simpson, ainda que esse funcionário só tenha sido adicionado depois de definida a coleção *allEmployees*.

6. Pressione Enter para retornar ao Visual Studio 2013.
7. No método *doWork*, altere a instrução que gera a coleção *allEmployees* para identificar e armazenar em cache os dados imediatamente, como mostrado em negrito:

```
var allEmployees = from e in empTree.ToList<Employee>()
                    select e;
```

8. No menu Debug, clique em Start Without Debugging. Verifique se a saída do aplicativo se parece com:

```
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales

Employee added
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
```

Observe que na segunda vez em que o aplicativo itera pela coleção *allEmployees*, a lista exibida não inclui David Simpson. Isso ocorre porque a consulta é avaliada e os resultados são armazenados em cache antes de David Simpson ser adicionado à árvore binária *empTree*.

9. Pressione Enter para retornar ao Visual Studio 2013.

Resumo

Neste capítulo, você aprendeu como LINQ usa a interface *IEnumerable<T>* e métodos de extensão para fornecer um mecanismo de consulta de dados. Vimos também que esses recursos aceitam a sintaxe de expressão de consultas no C#.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 22, “Sobrecarga de operadores”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Projetar campos especificados de uma coleção enumerável	<p>Utilize o método <i>Select</i> e especifique uma expressão lambda identificando os campos a projetar. Por exemplo:</p> <pre>var customerFirstNames = customers.Select(cust => cust.FirstName);</pre> <p>Ou utilize os operadores de consulta <i>from</i> e <i>select</i>. Por exemplo:</p> <pre>var customerFirstNames = from cust in customers select cust.FirstName;</pre>
Filtrar linhas de uma coleção enumerável	<p>Utilize o método <i>Where</i> e especifique uma expressão lambda contendo os critérios que a linha deve satisfazer. Por exemplo:</p> <pre>var usCompanies = addresses.Where(addr => String.Equals(addr.Country, "United States")) .Select(usComp => usComp.CompanyName);</pre> <p>Ou utilize o operador de consulta <i>where</i>. Por exemplo:</p> <pre>var usCompanies = from a in addresses where String.Equals(a.Country, "United States") select a.CompanyName;</pre>
Enumarar dados em uma ordem específica	<p>Utilize o método <i>OrderBy</i> e especifique uma expressão lambda identificando o campo a utilizar para ordenar as linhas. Por exemplo:</p> <pre>var companyNames = addresses.OrderBy(addr => addr.CompanyName) .Select(comp => comp.CompanyName);</pre> <p>Ou utilize o operador de consulta <i>orderby</i>. Por exemplo:</p> <pre>var companyNames = from a in addresses orderby a.CompanyName select a.CompanyName</pre>
Agrupar dados pelos valores de um campo	<p>Utilize o método <i>GroupBy</i> e especifique uma expressão lambda identificando o campo a utilizar para agrupar as linhas. Por exemplo:</p> <pre>var companiesGroupedByCountry = addresses.GroupBy(addr => addr.Country);</pre> <p>Ou utilize o operador de consulta <i>group by</i>. Por exemplo:</p> <pre>var companiesGroupedByCountry = from a in addresses group a by a.Country;</pre>

Para	Faça isto
Fazer a junção de dados armazenados em duas coleções diferentes	<p>Utilize o método <i>Join</i>, especificando a coleção com a qual fazer a junção, os critérios da junção e os campos para o resultado. Por exemplo:</p> <pre>var countriesAndCustomers = customers .Select(c => new { c.FirstName, c.LastName, c.CompanyName }). Join(addresses, custs => custs.CompanyName, addrs => addrs.CompanyName, (custs, addrs) => new {custs.FirstName, custs.LastName, addrs.Country });</pre> <p>Ou utilize o operador de consulta <i>join</i>. Por exemplo:</p> <pre>var countriesAndCustomers = from a in addresses join c in customers on a.CompanyName equals c.CompanyName select new { c.FirstName, c.LastName, a.Country };</pre>
Forçar a geração imediata dos resultados de uma consulta LINQ	<p>Utilize o método <i>ToList</i> ou <i>ToArray</i> para gerar uma lista ou um array contendo os resultados. Por exemplo:</p> <pre>var allEmployees = from e in empTree.ToList<Employee>() select e;</pre>

CAPÍTULO 22

Sobrecarga de operadores

Neste capítulo, você vai aprender a:

- Implementar operadores binários para seus próprios tipos.
- Implementar operadores unários para seus próprios tipos.
- Escrever operadores de incremento e decremento para seus próprios tipos.
- Entender a necessidade de implementar alguns operadores como pares.
- Implementar operadores de conversão implícita para seus próprios tipos.
- Implementar operadores de conversão explícita para seus próprios tipos.

Ao longo deste livro, os exemplos fazem amplo uso dos símbolos de operadores padrão (como `+` e `-`) para execução de operações padrão (como adição e subtração) em tipos (como `int` e `double`). A maioria dos tipos predefinidos vem com seus comportamentos predefinidos de cada operador. Também é possível definir como devem se comportar os operadores das suas estruturas e classes e isso veremos neste capítulo.

Operadores

Compensa rever alguns aspectos básicos dos operadores antes de examinar os detalhes de seu funcionamento e como sobrecarregá-los. A lista a seguir resume esses aspectos:

- Você utiliza os operadores para combinar operandos em expressões. Cada um tem sua semântica própria, dependendo do tipo com o qual trabalha. Por exemplo, o operador `+` significa “somar” quando utilizado com tipos numéricos ou “concatenar”, quando utilizado com strings.
- Cada operador tem uma *precedência*. Por exemplo, o operador `*` tem uma precedência mais alta do que o operador `+`. Isso significa que a expressão $a + b * c$ é o mesmo que $a + (b * c)$.
- Cada operador também tem uma *associatividade* para definir se é avaliado da esquerda para a direita ou da direita para a esquerda. Por exemplo, o operador `=` tem associatividade à direita (ele avalia da direita para a esquerda); portanto, $a = b = c$ é o mesmo que $a = (b = c)$.
- Um *operador unário* é um operador que tem apenas um operando. Por exemplo, o operador de incremento (`++`) é um operador unário.

- Um *operador binário* é um operador que tem dois operandos. Por exemplo, o operador de multiplicação (*) é um operador binário.

Restrições dos operadores

Este livro apresenta muitos exemplos de como o C# permite sobrerecarregar métodos quando você define seus próprios tipos. Com o C# também é possível sobrerecarregar boa parte dos símbolos de operador existentes para seus próprios tipos, embora a sintaxe seja um pouco diferente. Ao fazer isso, os operadores que você implementa caem automaticamente em uma estrutura bem definida com as regras a seguir:

- Você não pode alterar a precedência e a associatividade de um operador. A precedência e a associatividade são baseadas no símbolo de operador (por exemplo, +) e não no tipo (por exemplo, *int*) em que o símbolo de operador é utilizado. Consequentemente, a expressão $a + b * c$ é sempre igual a $a + (b * c)$, independentemente do tipo de *a*, *b* e *c*.
- Você não pode alterar a multiplicidade (o número de operandos) de um operador. Por exemplo, * (o símbolo de multiplicação) é um operador binário. Se você declarar um operador * para seu próprio tipo, ele deve ser um operador binário.
- Você não pode inventar novos símbolos de operador. Por exemplo, não é possível criar um novo símbolo de operador, como **, para elevar um número à potência de outro número. Você precisaria criar um método para fazer isso.
- Você não pode alterar o significado dos operadores quando aplicados a tipos predefinidos. Por exemplo, a expressão *1 + 2* tem um significado predefinido e você não pode redefinir esse significado. Se pudesse, as coisas poderiam se complicar!
- Há alguns símbolos de operador que não podem ser sobrerecarregados. Por exemplo, você não pode sobrerecarregar o operador ponto (), que indica acesso a um membro de classe. Novamente, se isso fosse possível, resultaria em uma complexidade desnecessária.



Dica Você pode usar os indexadores para simular [] como um operador. Da mesma forma, pode usar as propriedades para simular a atribuição (=) como um operador e pode usar delegates para imitar uma chamada de função como um operador.

Operadores sobrerecarregados

Para definir o comportamento do seu operador, você deve sobrerecarregar um operador selecionado. Você utiliza uma sintaxe do tipo método com um tipo de retorno e parâmetros, mas o nome do método é a palavra-chave *operator*, junto com o símbolo do operador que está declarando. Por exemplo, o código a seguir mostra uma estrutura especificada pelo usuário, chamada *Hour*, que define um operador + binário para somar duas instâncias de *Hour*:

```

struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }
    ...
    private int value;
}

```

Observe o seguinte:

- O operador é *public*. Todos os operadores *devem* ser públicos.
- O operador é *static*. Todos os operadores *devem* ser estáticos. Os operadores nunca são polimórficos e não podem utilizar os modificadores *virtual*, *abstract*, *override* ou *sealed*.
- Um operador binário (como o operador *+*, mostrado anteriormente) tem dois argumentos explícitos e um operador unário tem um argumento explícito. (Os programadores C++ devem observar que operadores nunca têm um parâmetro *this* oculto.)



Dica Ao se declarar uma funcionalidade altamente estilizada (como os operadores), é útil adotar uma convenção de nomes para os parâmetros. Por exemplo, os desenvolvedores costumam utilizar *lhs* e *rhs* (acrônimos para left-hand side, lado esquerdo, e right-hand side, lado direito, respectivamente) para operadores binários.

Quando você utiliza o operador *+* em duas expressões do tipo *Hour*, o compilador do C# converte automaticamente seu código em uma chamada ao método *operator +*. O compilador do C# transforma este código

```

Hour Example(Hour a, Hour b)
{
    return a + b;
}

```

em:

```

Hour Example(Hour a, Hour b)
{
    return Hour.operator +(a,b); // pseudocódigo
}

```

Note, porém, que essa sintaxe é um pseudocódigo e não é válido no C#. Você pode utilizar um operador binário somente na notação infixa padrão (com o símbolo entre os operandos).

Há uma regra final que você deve seguir ao declarar um operador: pelo menos um dos parâmetros deve ser sempre do tipo contêiner. No exemplo anterior do *operator+* para a classe *Hour*, um dos parâmetros, *a* ou *b*, deve ser um objeto *Hour*. Nesse exemplo, os dois parâmetros são objetos *Hour*. Mas pode haver ocasiões em que você queira definir implementações adicionais do *operator+* que adicione, por exemplo, um inteiro (um número de horas) a um objeto *Hour* – o primeiro parâmetro pode ser *Hour* e o segundo pode ser um inteiro. Essa regra permite que o compilador saiba onde procurar quando estiver tentando resolver uma chamada de operador e também garante que você não possa alterar o significado dos operadores predefinidos.

Crie operadores simétricos

Na seção anterior, você viu como declarar um operador + binário para somar duas instâncias do tipo *Hour*. A estrutura *Hour* também tem um construtor que cria uma *Hour* a partir de um *int*. Isso significa que é possível somar uma *Hour* e um *int*; você precisa apenas utilizar primeiramente o construtor *Hour* para converter o *int* em um *Hour*, como no exemplo a seguir:

```
Hour a = ...;
int b = ...;
Hour sum = a + new Hour(b);
```

Esse é certamente um código válido, mas não é tão claro nem tão conciso quanto somar uma *Hour* e um *int* diretamente, como a seguir:

```
Hour a = ...;
int b = ...;
Hour sum = a + b;
```

Para tornar a expressão $(a + b)$ válida, você deve especificar o que significa somar uma *Hour* (*a*, à esquerda) e um *int* (*b*, à direita). Em outras palavras, você precisa declarar um operador + binário cujo primeiro parâmetro seja uma *Hour* e o segundo seja um *int*. O código a seguir mostra a estratégia recomendada:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }
    public static Hour operator +(Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }
    ...
    private int value;
}
```

Observe que tudo o que a segunda versão do operador faz é construir uma *Hour* a partir do seu argumento *int* e, então, chamar a primeira versão. Dessa maneira, a lógica real por trás do operador é mantida em um único lugar. O ponto é que o *operator+* extra simplesmente torna a funcionalidade existente mais fácil de ser utilizada. Além disso, observe que você não deve fornecer muitas versões diferentes desse operador, cada uma com um tipo de segundo parâmetro diferente; em vez disso, apenas as forneça para os casos significativos e comuns e permita que o usuário da classe escreva alguns passos adicionais, se um caso incomum for necessário.

Esse *operator+* declara como somar uma *Hour*, como o operando da esquerda, e um *int*, como o operando da direita. Ele não declara como somar um *int*, como o operando da esquerda, e uma *Hour*, como o operando da direita:

```
int a = ...;
Hour b = ...;
Hour sum = a + b; // erro de tempo de compilação
```

Isso é contrário à intuição. Se você pode escrever a expressão *a + b*, pode esperar também ser capaz de escrever *b + a*. Portanto, você deve fornecer outra sobrecarga de *operator+*:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator +(Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }

    public static Hour operator +(int lhs, Hour rhs)
    {
        return new Hour(lhs) + rhs;
    }
    ...
    private int value;
}
```



Nota Os programadores C++ devem notar que eles mesmos devem fornecer a sobrecarga. O compilador não vai escrever a sobrecarga para você nem trocar silenciosamente a sequência dos dois operandos para encontrar um operador correspondente.

Operadores e interoperabilidade de linguagens

Nem todas as linguagens executáveis que usam o Common Language Runtime (CLR) suportam ou entendem a sobrerecarga de operadores. Se você estiver criando classes para serem utilizadas em outras linguagens, se sobrecarregar um operador, deverá oferecer um mecanismo alternativo com suporte para a mesma funcionalidade. Por exemplo, vamos supor que você implemente *operator+* para a estrutura *Hour*, como ilustrado aqui:

```
public static Hour operator +(Hour lhs, int rhs)
{
    ...
}
```

Se for necessário utilizar sua classe a partir de um aplicativo Microsoft Visual Basic, você também deverá fornecer um método *Add* que realize a mesma coisa, como demonstrado aqui:

```
public static Hour Add(Hour lhs, int rhs)
{
    ...
}
```

Avaliação da atribuição composta

Um operador de atribuição composta (como *+=*) é sempre avaliado em termos do seu operador associado simples (como *+*). Em outras palavras, a instrução

`a += b;`

é automaticamente avaliada assim:

`a = a + b;`

Em geral, a expressão *a @= b* (onde @ representa qualquer operador válido) é sempre avaliada como *a = a @ b*. Se você sobrecregou o operador simples apropriado, a versão sobrecregida será automaticamente chamada quando seu operador de atribuição composta associado for utilizado, como mostrado no exemplo a seguir:

```
Hour a = ...;
int b = ...;
a += a; // o mesmo que a = a + a
a += b; // o mesmo que a = a + b
```

A primeira expressão de atribuição composta (*a += a*) é válida porque *a* é do tipo *Hour* e o tipo *Hour* declara um *operator+* binário cujos dois parâmetros são *Hour*. Da mesma maneira, a segunda expressão de atribuição composta (*a += b*) também é válida porque *a* é do tipo *Hour* e *b* é do tipo *int*. O tipo *Hour* também declara um

operator+ binário cujo primeiro parâmetro é uma *Hour* e o segundo é um *int*. Saiba, no entanto, que você não pode escrever a expressão $b += a$, pois isso é o mesmo que $b = b + a$. Embora a soma seja válida, a atribuição não é, porque não há como atribuir uma *Hour* ao tipo *int* predefinido.

Declare operadores de incremento e decremento

Com o C#, você pode declarar sua própria versão de operadores de incremento (`++`) e decremento (`--`). As regras usuais se aplicam ao declarar esses operadores: eles precisam ser públicos, estáticos e unários (eles podem aceitar apenas um parâmetro). Observe o operador de incremento para a estrutura *Hour*:

```
struct Hour
{
    ...
    public static Hour operator ++(Hour arg)
    {
        arg.value++;
        return arg;
    }
    ...
    private int value;
}
```

Os operadores de incremento e decremento têm uma peculiaridade: podem ser utilizados nas formas de prefixo e sufixo. De forma inteligente, o C# utiliza o mesmo operador para ambas as versões de prefixo e sufixo. O resultado de uma expressão sufixada é o valor do operando *antes* que a expressão ocorra. Em outras palavras, o compilador converte o código

```
Hour now = new Hour(9);
Hour postfix = now++;
```

em:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocódigo, inválido no C#
```

O resultado da expressão prefixada é o valor de retorno do operador, de modo que o compilador do C# transforma efetivamente o código

```
Hour now = new Hour(9);
Hour prefix = ++now;
```

em:

```
Hour now = new Hour(9);
now = Hour.operator ++(now); // pseudocódigo, inválido no C#
Hour prefix = now;
```

Essa equivalência significa que o tipo de retorno dos operadores de incremento e decremento deve ser o mesmo do tipo do parâmetro.

Como comparar operadores em estruturas e classes

Saiba que a implementação do operador de incremento na estrutura *Hour* só funciona porque *Hour* é uma estrutura. Se você alterar *Hour* para uma classe, mas deixar a implementação do seu operador de incremento inalterada, verá que a versão sufixada não dará a resposta correta. Se lembrar que uma classe é um tipo-referência e se rever as traduções do compilador explicadas anteriormente, você entenderá por que no exemplo a seguir os operadores da classe *Hour* não funcionam mais como o esperado:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocódigo, inválido no C#
```

Se *Hour* for uma classe, a instrução de atribuição *postfix* = *now* fará a variável *postfix* referenciar o mesmo objeto que *now*. A atualização de *now* atualiza automaticamente *postfix*! Se *Hour* for uma estrutura, a instrução de atribuição fará uma cópia de *now* em *postfix* e todas as alterações em *now* deixarão *postfix* inalterada, que é precisamente o que você quer.

A implementação correta do operador de incremento, quando *Hour* é uma classe, é a seguinte:

```
class Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator ++(Hour arg)
    {
        return new Hour(arg.value + 1);
    }
    ...
    private int value;
}
```

Observe que, agora, *operator ++* cria um novo objeto baseado nos dados do original. Os dados no novo objeto são incrementados, mas os dados do original permanecem inalterados. Embora isso funcione, a tradução do operador de incremento pelo compilador resulta em um novo objeto que é criado cada vez que ele é utilizado. Isso pode ser caro, em termos de uso de memória e sobrerecarga de coleta de lixo. Portanto, é recomendável que você limite as sobrerecargas de operador ao definir os tipos. Essa recomendação se aplica a todos os operadores e não apenas ao operador de incremento.

Defina pares de operadores

Alguns operadores ocorrem naturalmente em pares. Por exemplo, se é possível comparar dois valores de *Hour* utilizando o operador *!=*, você espera ser capaz de comparar também dois valores de *Hour* utilizando o operador *==*. O compilador do C# reforça essa expectativa muito razoável insistindo em que, se você definir o operador

`==` ou *operador !=*, deve definir ambos. Essa regra, “nenhum ou ambos”, também se aplica aos operadores `<` e `>` e aos operadores `<=` e `>=`. O compilador do C# não escreve esses pares de operadores para você. Você é quem deve escrevê-los explicitamente, independentemente da aparente obviedade. Aqui estão os operadores `==` e `!=` para a estrutura *Hour*:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static bool operator ==(Hour lhs, Hour rhs)
    {
        return lhs.value == rhs.value;
    }

    public static bool operator !=(Hour lhs, Hour rhs)
    {
        return lhs.value != rhs.value;
    }
    ...
    private int value;
}
```

O tipo de retorno desses operadores não precisa ser realmente booleano. Mas você precisa ter uma boa razão para utilizar algum outro tipo ou esses operadores poderão se tornar muito confusos.

Redefinindo os operadores de igualdade

Se definir *operator ==* e *operator !=* em uma classe, você também deverá redefinir os métodos *Equals* e *GetHashCode* herdados de *System.Object* (ou de *System.ValueType*, se estiver criando uma estrutura). O método *Equals* deve exibir *exatamente* o mesmo comportamento de *operator ==*. (Você deve definir um em termos do outro.) O método *GetHashCode* é utilizado por outras classes no Microsoft .NET Framework. (Quando você usa um objeto como uma chave em uma tabela de hash, por exemplo, o método *GetHashCode* é chamado no objeto para ajudar a calcular um valor de hash. Para obter mais informações, consulte a documentação do .NET Framework fornecida com o Visual Studio 2013.) Tudo o que esse método precisa fazer é retornar um valor inteiro distinto. (Mas não retorne o mesmo inteiro, a partir do método *GetHashCode* de todos os seus objetos, pois isso anulará a eficácia dos algoritmos de hashing.)

Como implementar operadores

No exercício a seguir, você desenvolverá uma classe que simula números complexos.

Um número complexo tem dois elementos: um componente real e um componente imaginário. Geralmente, a representação de um número complexo é $(x + yi)$, onde x é o componente real e yi é o componente imaginário. Os valores de x e y são inteiros comuns, e i representa a raiz quadrada de -1 (eis o motivo pelo qual yi é imaginário). Apesar da aparência obscura e teórica, os números complexos têm muitas aplicações nas áreas da eletrônica, matemática aplicada, física e em diversos aspectos da engenharia. Se quiser mais informações sobre como e por que os números complexos são úteis, o Wikipedia fornece um artigo interessante e informativo.



Nota O Microsoft .NET Framework versão 4.0 e posteriores dispõem de um tipo chamado *Complex* no namespace *System.Numerics* que implementa números complexos, de modo que não há mais necessidade de definir sua própria versão desse tipo. Entretanto, é instrutivo acompanhar a implementação de alguns operadores comuns para esse tipo.

Você implementará números complexos como um par de inteiros que representam os coeficientes x e y dos componentes real e imaginário. Você também implementará os operadores necessários para efetuar uma operação aritmética simples com números complexos. A tabela a seguir resume como efetuar as quatro operações aritméticas básicas sobre um par de números complexos, $(a + bi)$ e $(c + di)$.

Operação	Cálculo
$(a + bi) + (c + di)$	$((a + c) + (b + d)i)$
$(a + bi) - (c + di)$	$((a - c) + (b - d)i)$
$(a + bi) * (c + di)$	$((a * c - b * d) + (b * c + a * d)i)$
$(a + bi) / (c + di)$	$((a * c + b * d) / (c * c + d * d)) + ((b * c - a * d) / (c * c + d * d))i$

Crie a classe *Complex* e implemente os operadores aritméticos

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.
2. Abra o projeto ComplexNumbers, localizado na pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 22\Windows X\ComplexNumbers` na sua pasta Documentos. Esse é um aplicativo de console que você utilizará para construir e testar seu código. O arquivo `Program.cs` contém o conhecido método `doWork`.
3. No Solution Explorer, clique no projeto ComplexNumbers. No menu Project, clique em Add Class. Na caixa de diálogo Add New Item – ComplexNumbers, na caixa Name, digite **Complex.cs** e clique em Add.

O Visual Studio cria a classe *Complex* e abre o arquivo `Complex.cs` na janela Code and Text Editor.

4. Adicione as propriedades automáticas *Real* e *Imaginary* à classe *Complex*, como mostrado em negrito no código a seguir:

```
class Complex
{
    public int Real { get; set; }
    public int Imaginary { get; set; }
}
```

Você utilizará essas duas propriedades para armazenar os componentes real e imaginário de um número complexo.

5. Adicione o construtor, mostrado em negrito a seguir, à classe *Complex*.

```
class Complex
{
    ...
    public Complex (int real, int imaginary)
    {
        this.Real = real;
        this.Imaginary = imaginary;
    }
}
```

Esse construtor aceita dois parâmetros *int* e os utiliza para preencher as propriedades *Real* e *Imaginary*.

6. Substitua o método *ToString*, como mostrado em negrito a seguir.

```
class Complex
{
    ...
    public override string ToString()
    {
        return String.Format("{0} + {1}i", this.Real, this.Imaginary);
    }
}
```

Esse método retorna uma string que representa o número complexo, na forma $(x + yi)$.

7. Adicione à classe *Complex* o operador + sobrecarregado, mostrado em negrito no código a seguir:

```
class Complex
{
    ...
    public static Complex operator +(Complex lhs, Complex rhs)
    {
        return new Complex(lhs.Real + rhs.Real, lhs.Imaginary + rhs.Imaginary);
    }
}
```

Esse é o operador de adição binária. Ele utiliza dois objetos *Complex* e soma esses objetos, efetuando o cálculo apresentado na tabela apresentada no início do exercício. O operador retorna um novo objeto *Complex* que contém os resultados desses cálculos.

- 8.** Adicione o operador – sobrecregadão à classe *Complex*.

```
class Complex
{
    ...
    public static Complex operator -(Complex lhs, Complex rhs)
    {
        return new Complex(lhs.Real - rhs.Real, lhs.Imaginary - rhs.Imaginary);
    }
}
```

Esse operador segue a mesma forma do operador + sobrecregadão.

- 9.** Implemente os operadores * e /.

```
class Complex
{
    ...
    public static Complex operator *(Complex lhs, Complex rhs)
    {
        return new Complex(lhs.Real * rhs.Real - lhs.Imaginary * rhs.Imaginary,
                           lhs.Imaginary * rhs.Real + lhs.Real * rhs.Imaginary);
    }

    public static Complex operator /(Complex lhs, Complex rhs)
    {
        int realElement = (lhs.Real * rhs.Real + lhs.Imaginary * rhs.Imaginary) /
                           (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);

        int imaginaryElement = (lhs.Imaginary * rhs.Real - lhs.Real * rhs.Imaginary) /
                               (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);

        return new Complex(realElement, imaginaryElement);
    }
}
```

Esses dois operadores seguem o mesmo formato dos dois operadores anteriores, embora os cálculos sejam um pouco mais complicados. (O cálculo para o operador / foi decomposto em duas etapas para evitar linhas de código muito longas.)

- 10.** Exiba o arquivo Program.cs na janela Code and Text Editor. Adicione as seguintes instruções, mostradas em negrito, ao final do método *doWork* da classe *Program* e exclua o comentário `// TODO:`:

```
static void doWork()
{
    Complex first = new Complex(10, 4);
    Complex second = new Complex(5, 2);

    Console.WriteLine("first is {0}", first);
    Console.WriteLine("second is {0}", second);

    Complex temp = first + second;
    Console.WriteLine("Add: result is {0}", temp);

    temp = first - second;
    Console.WriteLine("Subtract: result is {0}", temp);
```

```

temp = first * second;
Console.WriteLine("Multiply: result is {0}", temp);

temp = first / second;
Console.WriteLine("Divide: result is {0}", temp);
}

```

Esse código cria dois objetos *Complex* que representam os valores complexos $(10 + 4i)$ e $(5 + 2i)$. O código os exibe e testa cada um dos operadores que você acabou de definir, exibindo os resultados em cada caso.

- No menu Debug, clique em Start Without Debugging.

Verifique se o aplicativo exibe os resultados mostrados na imagem a seguir:

```

first is <10 + 4i>
second is <5 + 2i>
Add: result is <15 + 6i>
Subtract: result is <5 + 2i>
Multiply: result is <42 + 40i>
Divide: result is <2 + 8i>
Press any key to continue . . .

```

- Feche o aplicativo e retorne ao ambiente de programação do Visual Studio 2013.

Você acabou de criar um tipo que modela números complexos e suporta operações aritméticas básicas. No próximo exercício, você estenderá a classe *Complex* e fornecerá os operadores de igualdade, ***==*** e ***!=***.

Implemente operadores de igualdade

- No Visual Studio 2013, exiba o arquivo Complex.cs na janela Code and Text Editor.
- Adicione os operadores ***==*** e ***!=*** à classe *Complex*, como mostrado em negrito no exemplo a seguir.

```

class Complex
{
    ...
    public static bool operator ==(Complex lhs, Complex rhs)
    {
        return lhs.Equals(rhs);
    }

    public static bool operator !=(Complex lhs, Complex rhs)
    {
        return !(lhs.Equals(rhs));
    }
}

```

Observe que esses dois operadores utilizam o método *Equals*. O método *Equals* compara uma instância de uma classe com outra instância especificada como argumento. Ele retornará *true* se tiverem valores equivalentes, e *false*, caso contrário.

3. No menu Build, clique em Rebuild Solution.

A janela Error List exibe as seguintes mensagens de aviso:

```
'ComplexNumbers.Complex' defines operator == or operator != but does not override
Object.Equals(object o)
'ComplexNumbers.Complex' defines operator == or operator != but does not override
Object.GetHashCode()
```

Se você definir os operadores *!=* e *==*, também deverá sobreescrivê-los os métodos *Equals* e *GetHashCode*, herdados de *System.Object*.



Nota Se a janela Error List não estiver visível, no menu View, clique em Error List.

4. Substitua o método *Equals* na classe *Complex*, como mostrado aqui em negrito:

```
class Complex
{
    ...
    public override bool Equals(Object obj)
    {
        if (obj is Complex)
        {
            Complex compare = (Complex)obj;
            return (this.Real == compare.Real) &&
                   (this.Imaginary == compare.Imaginary);
        }
        else
        {
            return false;
        }
    }
}
```

O método *Equals* recebe um *Object* como parâmetro. Esse código verifica se o tipo do parâmetro é realmente um objeto *Complex*. Em caso afirmativo, esse código compara os valores das propriedades *Real* e *Imaginary* na instância atual com o parâmetro passado. Se forem iguais, o método retornará *true*; caso contrário, retornará *false*. Se o parâmetro passado não for um objeto *Complex*, o método retornará *false*.



Importante É tentador escrever o método *Equals* assim:

```
public override bool Equals(Object obj)
{
    Complex compare = obj as Complex;
    if (compare != null)
    {
        return (this.Real == compare.Real) &&
               (this.Imaginary == compare.Imaginary);
    }
    else
    {
        return false;
    }
}
```

Entretanto, a expressão `compare != null` chama o operador `!=` da classe *Complex*, o qual chama o método *Equals* novamente, o que resulta um loop recursivo.

- Sobrescreva o método *GetHashCode*. Essa implementação apenas chama o método herdado da classe *Object*, mas você pode fornecer um mecanismo próprio para gerar um código hash para um objeto, se preferir.

```
Class Complex
{
    ...
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}
```

- No menu Build, clique em Rebuild Solution.

Verifique que a solução é compilada sem emitir qualquer aviso.

- Exiba o arquivo Program.cs na janela Code and Text Editor. Adicione o seguinte código, mostrado em negrito, ao final do método *doWork*.

```
static void doWork()
{
    ...
    if (temp == first)
    {
        Console.WriteLine("Comparison: temp == first");
    }
    else
    {
        Console.WriteLine("Comparison: temp != first");
    }

    if (temp == temp)
    {
```

```

        Console.WriteLine("Comparison: temp == temp");
    }
else
{
    Console.WriteLine("Comparison: temp != temp");
}
}

```



Nota A expressão `temp == temp` gera a mensagem de aviso “Comparison made to same variable: did you mean to compare to something else?” (Comparação efetuada com a mesma variável: você queria comparar com algo mais?). Nesse caso, você pode ignorar o aviso porque essa comparação é intencional; o objetivo é verificar se o operador `==` está funcionando como previsto.

8. No menu Debug, clique em Start Without Debugging. Verifique que as duas últimas mensagens exibidas são as seguintes:

```

Comparison: temp != first
Comparison: temp == temp

```

9. Feche o aplicativo e retorne ao Visual Studio 2013.

Operadores de conversão

Às vezes, é necessário converter uma expressão de um tipo em outro. Por exemplo, o método a seguir é declarado com um único parâmetro `double`:

```

class Example
{
    public static void MyDoubleMethod(double parameter)
    {
        ...
    }
}

```

É razoável supor que apenas os valores do tipo `double` são utilizados como argumentos quando o método `MyDoubleMethod` é chamado, mas esse não é o caso. O compilador do C# também permite que `MyDoubleMethod` seja chamado com um argumento de algum outro tipo, mas somente se o valor do argumento puder ser convertido em um `double`. Por exemplo, se você fornece um argumento `int`, o compilador gera um código que converte o valor do argumento em um `double` quando o método é chamado.

Forneça conversões predefinidas

Os tipos predefinidos têm algumas conversões predefinidas. Por exemplo, como já mencionamos, um *int* pode ser implicitamente convertido em um *double*. Uma conversão implícita não exige qualquer sintaxe especial e nunca gera uma exceção.

```
Example.MyDoubleMethod(42); // conversão implícita de int em double
```

Uma conversão implícita é algumas vezes chamada *conversão de alargamento* (widening conversion), porque o resultado é *mais abrangente* que o valor original – ele contém as mesmas informações que o valor original e nada é perdido. No caso de *int* e *double*, o intervalo de *double* é maior que o de *int*, e todos os valores *int* têm um valor *double* equivalente. Contudo, o inverso não vale, e um valor *double* não pode ser implicitamente convertido em um *int*:

```
class Example
{
    public static void MyIntMethod(int parameter)
    {
        ...
    }
}
...
Example.MyIntMethod(42.0); // erro de tempo de compilação
```

A conversão de um *double* em um *int* tem o risco de perda das informações; portanto, a conversão não será feita automaticamente. (Considere o que aconteceria se o argumento para *MyIntMethod* fosse 42.5: como isso deveria ser convertido?) Um *double* pode ser convertido em um *int*, mas a conversão exige uma notação explícita (um casting):

```
Example.MyIntMethod((int)42.0);
```

Uma conversão explícita é algumas vezes chamada de *conversão de estreitamento* (narrowing conversion), porque o resultado é *mais restrito* do que o valor original (ele pode conter menos informações) e pode lançar uma exceção *OverflowException*, caso o valor resultante esteja fora do intervalo do tipo de destino. No C#, você pode criar operadores de conversão para seus próprios tipos definidos pelo usuário a fim de verificar se há sentido em converter valores em outros tipos, sendo também possível especificar se essas conversões são implícitas ou explícitas.

Implemente operadores de conversão definidos pelo usuário

A sintaxe para declarar um operador de conversão definido pelo usuário tem algumas semelhanças com àquela utilizada para declarar um operador sobre carregado, mas também algumas diferenças importantes. Veja um operador de conversão que permite a um objeto *Hour* ser implicitamente convertido em um *int*:

```
struct Hour
{
    ...
    public static implicit operator int (Hour from)
    {
        return from.value;
    }

    private int value;
}
```

Um operador de conversão deve ser *public* e também deve ser *static*. O tipo de origem da conversão é declarado como parâmetro (nesse caso, *Hour*) e o tipo de destino da conversão é declarado como o nome do tipo, após a palavra-chave *operator* (neste caso, *int*). Não há qualquer tipo de retorno especificado antes da palavra-chave *operator*.

Ao declarar seus operadores de conversão, você deve especificar se eles são implícitos ou explícitos. Você faz isso utilizando as palavras-chave *implicit* e *explicit*. Por exemplo, o operador de conversão de *Hour* para *int*, mencionado anteriormente, é implícito; ou seja, o compilador do C# pode utilizá-lo sem exigir um casting:

```
class Example
{
    public static void MyOtherMethod(int parameter) { ... }
    public static void Main()
    {
        Hour lunch = new Hour(12);
        Example.MyOtherMethod(lunch); // conversão implícita de Hour em int
    }
}
```

Se o operador de conversão tivesse sido declarado como *explicit*, o exemplo anterior não teria compilado porque um operador de conversão explícito exige um casting.

```
Example.MyOtherMethod((int)lunch); // conversão explícita de Hour em int
```

Quando você deve declarar um operador de conversão como explícito ou implícito? Se uma conversão for sempre segura, não tiver risco de perda de informação e não puder lançar uma exceção, então ela pode ser definida como uma conversão *implícita*. Caso contrário, deve ser declarada como uma conversão *explícita*. A conversão de uma *Hour* em um *int* é sempre segura – cada *Hour* tem um valor *int* correspondente; portanto, faz sentido que ela seja implícita. Um operador que converte uma *string* em uma *Hour* deve ser explícito, porque nem todas as strings representam *Hours* válidas. (Embora a string "7" seja adequada, como você converteria a string "Hello, World" em uma *Hour*?)

Crie operadores simétricos, uma retomada do assunto

Os operadores de conversão proporcionam um modo alternativo para resolver o problema de fornecer operadores simétricos. Por exemplo, em vez de fornecer três versões de *operator+* (*Hour + Hour*, *Hour + int* e *int + Hour*) para a estrutura *Hour*, como mostrado anteriormente, você pode fornecer uma única versão de *operator+* (que aceita dois parâmetros *Hour*) e uma conversão implícita de *int* em *Hour*, como esta:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }

    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }

    public static implicit operator Hour (int from)
    {
        return new Hour (from);
    }
    ...
    private int value;
}
```

Se você adicionar uma *Hour* a um *int* (em qualquer ordem), o compilador do C# converterá automaticamente o *int* em *Hour* e, então, chamará *operator+* com dois argumentos *Hour*, como demonstrado aqui:

```
void Example(Hour a, int b)
{
    Hour eg1 = a + b; // b convertido em Hour
    Hour eg2 = b + a; // b convertido em Hour
}
```

Escreva operadores de conversão

No último exercício deste capítulo, você adicionará outros operadores à classe *Complex*. Para começar, escreva um par de operadores de conversão que operem entre os tipos *int* e *Complex*. A conversão de um *int* em um objeto *Complex* sempre é um processo seguro e nunca perde informações (porque, na realidade, um *int* é apenas um número *Complex* sem um elemento imaginário). Sendo assim, você implementará tudo isso como um operador de conversão implícito. Entretanto, o inverso não se aplica – para converter um objeto *Complex* em um *int*, você deve descartar o elemento imaginário. Por conseguinte, implementará esse operador de conversão como explícito.

Implemente os operadores de conversão

1. Retorne ao Visual Studio 2013 e exiba o arquivo Complex.cs na janela Code and Text Editor. Adicione à classe *Complex* o construtor mostrado em negrito no código a seguir, imediatamente após o construtor existente e antes do método *ToString*. Esse novo construtor aceita um único parâmetro *int*, o qual utiliza para inicializar a propriedade *Real*. A propriedade *Imaginary* é definida como *0*.

```
class Complex
{
    ...
    public Complex(int real)
    {
        this.Real = real;
        this.Imaginary = 0;
    }
    ...
}
```

2. Adicione o seguinte operador de conversão implícita à classe *Complex*.

```
class Complex
{
    ...
    public static implicit operator Complex(int from)
    {
        return new Complex(from);
    }
    ...
}
```

Esse operador converte de um *int* para um objeto *Complex*, retornando uma nova instância da classe *Complex*, construída por meio do construtor criado na etapa anterior.

3. Adicione o seguinte operador de conversão explícita, mostrado em negrito, à classe *Complex*.

```
class Complex
{
    ...
    public static explicit operator int(Complex from)
    {
        return from.Real;
    }
    ...
}
```

Esse operador aceita um objeto *Complex* e retorna o valor da propriedade *Real*. Essa conversão descarta o elemento imaginário do número complexo.

4. Exiba o arquivo Program.cs na janela Code and Text Editor. Adicione o seguinte código, mostrado em negrito, ao final do método *doWork*.

```

static void doWork()
{
    ...
    Console.WriteLine("Current value of temp is {0}", temp);

    if (temp == 2)
    {

        Console.WriteLine("Comparison after conversion: temp == 2");
    }
    else
    {
        Console.WriteLine("Comparison after conversion: temp != 2");
    }

    temp += 2;
    Console.WriteLine("Value after adding 2: temp = {0}", temp);
}

```

Essas instruções testam o operador implícito que converte um *int* em um objeto *Complex*. A instrução *if* compara um objeto *Complex* com um *int*. O compilador gera um código que primeiro converte o *int* em um objeto *Complex* e depois chama o operador *==* da classe *Complex*. A instrução que soma 2 à variável *temp* converte o valor *int* 2 em um objeto *Complex* e depois utiliza o operador *+* da classe *Complex*.

- Adicione as seguintes instruções ao final do método *doWork*.

```

static void doWork()
{
    ...
    int tempInt = temp;
    Console.WriteLine("Int value after conversion: tempInt == {0}", tempInt);
}

```

A primeira instrução tenta atribuir um objeto *Complex* a uma variável *int*.

- No menu Build, clique em Rebuild Solution.

A compilação da solução falha e o compilador informa o seguinte erro na janela Error List:

Cannot implicitly convert type 'ComplexNumbers.Complex' to 'int'. An explicit conversion exists (are you missing a cast?)

O operador que converte de um objeto *Complex* para um *int* é um operador de conversão explícita, de modo que você deve fazer um casting.

- Modifique a instrução que tenta armazenar um valor *Complex* em uma variável *int* para utilizar um casting, como a seguir:

```
int tempInt = (int)temp;
```

- 8.** No menu Debug, clique em Start Without Debugging. Verifique que a solução agora compila e que as quatro últimas mensagens são as seguintes:

```
Current value of temp is (2 + 0i)
Comparison after conversion: temp == 2
Value after adding 2: temp = (4 + 0i)
Int value after conversion: tempInt == 4
```

- 9.** Feche o aplicativo e retorne ao Visual Studio 2013.

Resumo

Neste capítulo, você aprendeu a sobre carregar operadores e a fornecer uma funcionalidade específica para uma classe ou estrutura. Você implementou alguns operadores aritméticos comuns e também criou operadores com os quais pode comparar instâncias de uma classe. Por último, aprendeu a criar operadores de conversão implícita e explícita.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 23, “Como melhorar o desempenho usando tarefas”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Implementar um operador	<p>Escreva as palavras-chave <i>public</i> e <i>static</i>, seguidas pelo tipo de retorno, seguido pela palavra-chave <i>operator</i>, seguida pelo símbolo do operador que está sendo declarado, seguido pelos parâmetros apropriados entre parênteses. Implemente a lógica do operador no corpo do método. Por exemplo:</p> <pre>class Complex { ... public static bool operator==(Complex lhs, Complex rhs) { ... // Implemente a lógica para o operador == } ... }</pre>

Para	Faça isto
Definir um operador de conversão	<p>Escreva as palavras-chave <i>public</i> e <i>static</i>, seguidas pela palavra-chave <i>implicit</i> ou <i>explicit</i>, seguida pela palavra-chave <i>operator</i>, seguida pelo tipo de destino da conversão, seguido pelo tipo de origem da conversão como um único parâmetro entre parênteses. Por exemplo:</p> <pre>class Complex { ... public static implicit operator Complex(int from) { ... // código para converter a partir de um int } ... }</pre>

PARTE IV

Construção de aplicativos Windows 8.1 profissionais com C#

CAPÍTULO 23	Como melhorar o desempenho usando tarefas	527
CAPÍTULO 24	Como melhorar o tempo de resposta empregando operações assíncronas.....	570
CAPÍTULO 25	Implementação da interface do usuário de um aplicativo Windows Store	611
CAPÍTULO 26	Exibição e busca de dados em um aplicativo Windows Store.....	661
CAPÍTULO 27	Acesso a um banco de dados remoto em um aplicativo Windows Store	709

Esta página foi deixada em branco intencionalmente.

CAPÍTULO 23

Como melhorar o desempenho usando tarefas

Neste capítulo, você vai aprender a:

- Descrever os benefícios propiciados pela implementação de operações paralelas em um aplicativo.
- Utilizar a classe *Task* para criar e executar operações paralelas em um aplicativo.
- Utilizar a classe *Parallel* para parallelizar alguns blocos de programação comuns.
- Cancelar tarefas de execução demorada e tratar as exceções levantadas pelas operações paralelas.

De modo geral, nos capítulos anteriores, você aprendeu a utilizar o C# para escrever programas que executam em uma única thread (*single-threaded*). Por *single-threaded*, entendemos que, em qualquer ponto no tempo, um programa executou uma única instrução. Essa estratégia não vai ser sempre a mais eficiente para um aplicativo. Os aplicativos que executam várias operações simultaneamente podem utilizar os recursos disponíveis em um computador de modo mais eficiente. Alguns processos podem ser executados com mais velocidade, se for possível dividi-los em caminhos paralelos de execução simultânea. Este capítulo trata de como melhorar o desempenho de seus aplicativos, maximizando o uso do poder de processamento disponível. Especificamente, você vai aprender a utilizar os objetos *Task* para empregar multitarefa efetiva em aplicativos que utilizam muito poder de computação.

Por que fazer multitarefa por meio de processamento paralelo?

Há dois motivos principais pelos quais você empregaria multitarefa em um aplicativo:

- **Aumentar a rapidez de resposta** Transmitir ao usuário do aplicativo a impressão de que o programa está executando mais de uma tarefa de cada vez, ao dividir o programa em threads de execução paralela e ao permitir que, por sua vez, cada thread seja executada durante um curto intervalo de tempo. Esse é o modelo cooperativo convencional, com o qual muitos desenvolvedores experientes em Windows estão acostumados. Entretanto, isso não é a verdadeira multitarefa, uma vez que o processador é compartilhado entre as threads, e a natureza cooperativa dessa estratégia exige que o código executado por cada thread se comporte de modo adequado. Se uma thread dominar a CPU e os recursos disponíveis, driblando as outras threads, as vantagens dessa estratégia deixarão de existir. Às vezes, é difícil escrever aplicativos bem comportados,

que sigam esse modelo de modo consistente. No entanto, um objetivo importante do Windows 8 e do Windows 8.1 é oferecer uma plataforma que trate desses problemas, e o Windows Runtime (WinRT) que implementa o ambiente de execução do Windows 8.1 fornece muitas APIs direcionadas a esse modo de operação. O Capítulo 24, “Como melhorar o tempo de resposta empregando operações assíncronas”, discute esses recursos com mais detalhes.

■ **Aumentar a escalabilidade** Para melhorar a escalabilidade, utilize os recursos de processamento de modo eficiente e aplique-os para reduzir o tempo necessário para executar as partes de um aplicativo. Um desenvolvedor pode determinar quais partes de um aplicativo serão executadas simultaneamente e organizar essa execução de forma paralela. Com a inclusão de mais recursos tecnológicos, aumenta a quantidade de operações que podem ser executadas de modo simultâneo. Há relativamente pouco tempo, esse modelo só era adequado para os sistemas que tinham várias CPUs ou que podiam distribuir o processamento por vários computadores em rede. Nos dois casos, era necessário usar um modelo que coordenasse as tarefas paralelas. A Microsoft oferece uma versão especializada do Windows, chamada High Performance Computing (HPC) Server 2008, com a qual uma empresa pode construir clusters de servidores para distribuir e executar tarefas em paralelo. Os desenvolvedores podem utilizar a implementação Microsoft do Message Passing Interface (MPI) – um conhecido protocolo de comunicação independente de linguagem – para construir aplicativos baseados em tarefas paralelas, que coordenam e colaboram entre si por meio do envio de mensagens. As soluções baseadas no Windows HPC Server 2008 e no MPI são perfeitas para os aplicativos de engenharia e científicos, de larga escala, vinculados à computação*, mas essas soluções são inadequadas para os sistemas de menor escala em execução em computadores desktop ou tablets.

Considerando essas descrições, talvez você esteja propenso a concluir que o modo mais econômico de construir soluções multitarefa para computadores desktop e dispositivos móveis é utilizar a estratégia de várias threads cooperativas. Contudo, esse mecanismo tinha a intenção de simplesmente melhorar a rapidez das respostas — garantir que computadores com um único processador fornecessem a cada tarefa uma parte suficiente desse processador. Ele não é adequado para máquinas com vários processadores, pois não é projetado para distribuir a carga entre eles e, por conta disso, não escalona bem. Quando as máquinas desktop com vários processadores eram caras (e, como resultado, relativamente raras), isso não era um problema. Contudo, essa situação mudou, como explicaremos de modo resumido.

O surgimento do processador multinúcleo

Há pouco mais de 12 anos, o preço de um bom computador pessoal estava na faixa de 800 a 1.500 dólares. Atualmente, um computador pessoal aceitável ainda custa quase a mesma coisa, mesmo depois de 12 anos de inflação. Hoje, a especificação de um computador comum provavelmente inclui um processador com uma velocidade entre 2 e 3 GHz, de 500 a 1.000 GB de espaço de armazenamento em disco rígido, de 4 a 8 GB de RAM, gráficos de alta resolução e alta velocidade, e uma unidade de gravação de DVD. Há pouco mais de 10 anos, a velocidade do processador em uma máquina comum estava entre 500 MHz e 1 GHz, um disco rígido grande tinha 80 GB, o Win-

* N. de R.T.: Do inglês, *compute-bound* ou *CPU-intensive*. A expressão se refere a um aplicativo ou tarefa que faz uso intensivo de recursos de CPU e pouco consumo de recursos de E/S ou memória.

dows funcionava bem com 256 MB de RAM ou menos, e as unidades de gravação de CD custavam muito mais de 100 dólares. (As unidades de gravação de DVD eram raras e muito caras.) Essa é a compensação do avanço tecnológico: itens de hardware mais velozes e mais poderosos a preços cada vez mais reduzidos.

Não se trata de uma tendência nova. Em 1965, Gordon E. Moore, cofundador da Intel, escreveu uma publicação técnica, intitulada "Cramming More Components onto Integrated Circuits" (Amontoando mais componentes em circuitos integrados), que abordava como o crescimento da miniaturização dos componentes permitia a incorporação de uma quantidade maior de transistores em um chip de silício, e como a queda dos custos da produção – à medida que a tecnologia se tornava mais acessível – levaria à compressão, por questões econômicas, de 65.000 componentes em um único chip, até 1975. As observações de Moore levaram à criação da conhecida Lei de Moore, a qual afirma basicamente que o número de transistores que podem ser dispostos, a baixo custo, em um circuito integrado aumentará exponencialmente, dobrando a cada dois anos. (Na realidade, no início, Gordon Moore foi mais otimista, e postulou que o volume de transistores provavelmente dobraria a cada ano, mas depois ele mudou seus cálculos.) A possibilidade de empacotar transistores deu margem à possibilidade de transferir dados entre eles com mais rapidez. Ou seja, estava previsto que os fabricantes de chips produziriam microprocessadores mais velozes e mais poderosos, praticamente de modo ininterrupto, permitindo que os desenvolvedores de software escrevessem blocos de software cada vez mais complexos, que seriam executados com mais velocidade.

A Lei de Moore relacionada à miniaturização de componentes eletrônicos ainda se aplica, mesmo depois de quase 50 anos. Entretanto, a física já começou a entrar em ação. Há um limite a partir do qual não é possível transmitir sinais mais rapidamente entre transistores em um único chip, a despeito de seu pequeno tamanho ou densa compactação. Para um desenvolvedor de software, o resultado mais importante dessa limitação é o fato de que a velocidade dos processadores parou de aumentar. Há sete ou oito anos, um processador veloz executava a 3 GHz. Hoje, um processador veloz ainda executa a 3 GHz.

O limite imposto à velocidade com a qual os processadores podem transmitir dados entre componentes levou os fabricantes de chips a vislumbrar mecanismos alternativos para aumentar o volume de trabalho em um processador. O resultado é que, atualmente, a maioria dos processadores modernos tem dois ou mais *núcleos*. Na prática, os fabricantes de chips inseriram vários processadores no mesmo chip e incluíram a lógica necessária para que eles se comunicassem e se coordenassesem. Processadores quad-core (quatro núcleos) e eight-core já são lugar-comum. Existem chips com 16, 32 e 64 núcleos, e o preço dos processadores dual-core e quad-core agora é suficientemente baixo para que sejam um elemento esperado em computadores laptop e em tablets. Assim, embora a velocidade dos processadores tenha deixado de aumentar, já é possível esperar mais deles em um único chip.

O que isso significa para um desenvolvedor que escreve aplicativos em C#?

Antes do surgimento dos processadores multicore (multinúcleo), era possível aumentar a velocidade de um aplicativo single-threaded ao executá-lo em um processador mais veloz. Com os processadores multicore, isso não acontece mais. Um aplicativo single-threaded será executado com a mesma velocidade em um processador equipado com um, dois ou quarto núcleos, que possuem a mesma frequência de clock. No que tange ao seu aplicativo, a diferença é que, em um processador dual-core, um dos núcleos do processador ficará inativo, e em um processador quad-core, três dos quatro núcleos ficarão simplesmente esperando para trabalhar. Para fazer o uso mais eficiente dos processadores multicore, escreva seus aplicativos de modo a tirar proveito da multitarefa.

Como implementar multitarefa com o Microsoft .NET Framework

Multitarefa é a capacidade de fazer mais de uma coisa ao mesmo tempo. Ela representa um daqueles conceitos de fácil descrição, mas que, até pouco tempo, tinha uma implementação bem difícil.

Em termos ideais, um aplicativo em execução em um processador multicore executa tantas tarefas simultâneas quanto a quantidade de núcleos de processador disponíveis, e mantém cada núcleo ocupado. Entretanto, para implementar simultaneidade (ou "concorrência"), existem várias questões a serem consideradas:

- Como dividir um aplicativo em um conjunto de operações simultâneas?
- Como fazer um conjunto de operações ser executado simultaneamente em vários processadores?
- Como ter certeza de que você está executando apenas a quantidade de operações correspondente ao número de processadores disponíveis?
- Se uma operação estiver bloqueada (por exemplo, enquanto aguarda o término da entrada/saída), como é possível detectar essa situação e fazer o processador executar outra operação em vez de permanecer ocioso?
- Como saber se uma ou mais operações simultâneas foram concluídas?

Para um desenvolvedor de aplicativos, a primeira pergunta é uma questão de design. As perguntas restantes dependem da infraestrutura de programação. A Microsoft fornece a classe *Task* e uma coleção de tipos associados no namespace *System.Threading.Tasks* para ajudar no tratamento desses problemas.

Tarefas, threads e o *ThreadPool*

A classe *Task* é uma abstração de uma operação simultânea. Você cria um objeto *Task* para executar um bloco de código. É possível instanciar vários objetos *Task* e iniciar a sua execução em paralelo, se existirem processadores ou núcleos de processador suficientes disponíveis.



Nota De agora em diante, empregaremos o termo *processador* tanto para uma referência a um processador de um só núcleo quanto para a um único núcleo de processamento em um processador multinúcleo.

Internamente, o Common Language Runtime (CLR) implementa tarefas e agenda* a sua execução por meio de objetos *Thread* e da classe *ThreadPool*. O multithreading e os pools de threads estão disponíveis no .NET Framework desde a versão 1.0 e, se estiver compilando aplicativos de desktop tradicionais, você pode utilizar a classe *Thread* do namespace *System.Threading* diretamente em seu código. Contudo, a classe *Thread* não está disponível para aplicativos Windows Store; em vez disso, você utiliza a classe *Task*.

* N. de R.T.: Neste livro, optou-se por traduzir a expressão *task scheduling* como *agendamento de tarefas*. Em algumas publicações, *scheduling/scheduler* é traduzido como *escalonamento/escalonador*.

A classe *Task* fornece uma abstração poderosa para threads, com a qual você pode distinguir facilmente entre o grau de paralelização em um aplicativo (as tarefas) e as unidades de paralelização (as threads). Em um computador com um único processador, em geral, esses itens são idênticos. Entretanto, em um computador com vários processadores ou com um processador multicore, eles são diferentes. Se você projetar um programa baseado diretamente em threads, descobrirá que seu aplicativo pode não escalar muito bem; o programa usará o número de threads que você criar explicitamente, e o sistema operacional agendará apenas esse número de threads. Isso poderá resultar em uma sobrecarga e um tempo de resposta insuficiente se o número de threads ultrapassar muito o número de processadores disponíveis, ou em uma taxa de transferência inefficiente ou inadequada se o número de threads for inferior ao número de processadores.

O CLR otimiza o número de threads necessárias para implementar um conjunto de tarefas simultâneas e as agenda de modo eficiente, de acordo com o número de processadores disponíveis. Ele implementa um mecanismo de enfileiramento para distribuir a carga de trabalho entre um conjunto de threads alocadas em um pool de threads (implementado com um objeto *ThreadPool*). Quando um programa cria um objeto *Task*, a tarefa é adicionada a uma fila global. Assim que uma thread se torna disponível, a tarefa é removida da fila global e é executada por essa thread. A classe *ThreadPool* implementa algumas otimizações e usa um algoritmo de “roubo de trabalho” para garantir que as threads sejam agendadas (ou “escalonadas”) de maneira eficiente.



Nota A classe *ThreadPool* já estava disponível nas edições anteriores do .NET Framework, mas foi significativamente aprimorada no .NET Framework 4.0 para suportar as *Tasks*.

Observe que o número de threads criadas pelo CLR para lidar com as suas tarefas não é necessariamente idêntico ao número de processadores. De acordo com a natureza da carga de trabalho, um ou mais processadores podem estar ocupados, executando um trabalho de alta prioridade para outros aplicativos ou serviços. Assim, o número ideal de threads para seu aplicativo pode ser inferior ao número de processadores existentes na máquina. Como alternativa, uma ou mais threads em um aplicativo podem estar aguardando um acesso demorado à memória, E/S ou o término de uma operação da rede, deixando os respectivos processadores disponíveis. Nesse caso, o número ideal de threads pode ser maior que o número de processadores disponíveis. O CLR segue uma estratégia iterativa, conhecida como algoritmo *hill-climbing*, para determinar dinamicamente o número perfeito de threads para a carga de trabalho atual.

O mais importante é que tudo o que você precisa fazer em seu código é dividir, ou particionar, seu aplicativo em tarefas, que podem ser executadas simultaneamente. O CLR se encarrega da criação do número adequado de threads, com base na arquitetura do processador e na carga de trabalho de seu computador, associa suas tarefas a essas threads e gerencia a sua execução de modo eficiente. Não importa se você particionar seu trabalho em muitas tarefas, porque o CLR tentará executar apenas a quantidade viável de threads simultâneas; na realidade, é recomendável que você *particione bastante* seu trabalho, porque isso ajuda a garantir que seu aplicativo suporte ganhos de escala se você o mover para um computador equipado com mais processadores disponíveis.

Crie, execute e controle tarefas

Para criar objetos *Task*, utilize o seu construtor. Esse construtor é sobrecarregado, mas todas as versões esperam que você forneça um delegate *Action* como parâmetro. O Ca-

pítulo 20, “Separação da lógica do aplicativo e tratamento de eventos”, ilustra que um delegate *Action* faz referência a um método que não retorna um valor. Um objeto *Task* chama esse delegate no momento em que sua execução é agendada. O exemplo a seguir cria um objeto *Task* que utiliza um delegate para executar o método chamado *doWork*:

```
Task task = new Task(doWork);  
...  
private void doWork()  
{  
    // A tarefa executa este código quando iniciada  
    ...  
}
```



Dica O tipo padrão de *Action* faz referência a um método que não aceita qualquer parâmetro. Outras sobrecargas do construtor de *Task* aceitam um parâmetro *Action<object>* representando um delegate que faz referência a um método que aceita um único parâmetro *object*. Com essas sobrecargas é possível passar dados para o método executado pela tarefa. O código a seguir apresenta um exemplo:

```
Action<object> action;  
action = doWorkWithObject;  
object parameterData = ...;  
Task task = new Task(action, parameterData);  
...  
private void doWorkWithObject(object o)  
{  
    ...  
}
```

Após criar um objeto *Task*, você pode iniciar a sua execução por meio do método *Start*, como a seguir:

```
Task task = new Task(...);  
task.Start();
```

O método *Start* também é sobre carregado, e é possível especificar opcionalmente um objeto *TaskScheduler* para controlar o nível de simultaneidade e outras opções de agendamento (escalonamento). É possível obter uma referência ao objeto padrão *TaskScheduler* ao usar a propriedade estática *Default* da classe *TaskScheduler*. A classe *TaskScheduler* também fornece a propriedade estática *Current*, que retorna uma referência ao objeto *TaskScheduler* atualmente utilizado. (Esse objeto *TaskScheduler* será utilizado se você não especificar explicitamente um scheduler.) Uma tarefa poderá oferecer dicas ao *TaskScheduler* padrão sobre como agendar e executar a tarefa se você especificar um valor da enumeração *TaskCreationOptions* no construtor de *Task*.



Mais informações Para obter mais informações sobre a classe *TaskScheduler* e a enumeração *TaskCreationOptions*, consulte a documentação que descreve a biblioteca de classes do .NET Framework fornecida com o Visual Studio.

Criar e executar uma tarefa é um processo muito comum, e a classe *Task* fornece o método estático *Run* com o qual é possível combinar essas operações. O método

Run recebe um delegate *Action* especificando a operação a ser executada (tal como o construtor de *Task*), mas começa a executar a tarefa imediatamente. Ele retorna uma referência para o objeto *Task*. Você pode utilizá-lo assim:

```
Task task = Task.Run(() => doWork());
```

Quando o método executado pela tarefa terminar, a tarefa estará concluída, e a thread utilizada para executá-la poderá ser reciclada para executar outra tarefa.

Quando uma tarefa termina, você pode fazer com que outra seja agendada imediatamente, criando uma *continuação*. Para isso, chame o método *ContinueWith* de um objeto *Task*. Quando a ação executada pelo objeto *Task* terminar, o scheduler criará automaticamente um novo objeto *Task* para executar a ação especificada pelo método *ContinueWith*. O método informado pela continuação espera um parâmetro *Task* e o scheduler passa para o método uma referência à tarefa finalizada. O valor retornado por *ContinueWith* é uma referência ao novo objeto *Task*. O exemplo de código a seguir cria um objeto *Task* que executa o método *doWork* e especifica uma continuação que executa o método *doMoreWork* em uma nova tarefa quando a primeira terminar:

```
Task task = new Task(doWork);
task.Start();
Task newTask = task.ContinueWith(doMoreWork);
...
private void doWork()
{
    // A tarefa executa este código quando iniciada
    ...
}
...
private void doMoreWork(Task task)
{
    // A continuação executará esse código quando doWork terminar
    ...
}
```

O método *ContinueWith* possui muitas sobrecargas, e você pode fornecer alguns parâmetros que especifiquem outros itens, como o *TaskScheduler* a ser utilizado e um valor de *TaskContinuationOptions*. O tipo *TaskContinuationOptions* é uma enumeração que contém um superconjunto de valores da enumeração *TaskCreationOptions*. Os valores adicionais disponíveis são os seguintes:

- **NotOnCanceled** e **OnlyOnCanceled** A opção *NotOnCanceled* especifica que a continuação só deve ser executada se a ação anterior for concluída e não cancelada, e a opção *OnlyOnCanceled* especifica que a continuação só deve ser executada se a ação anterior for cancelada. A seção “Cancele tarefas e trate exceções”, apresentada mais adiante neste capítulo, descreve como cancelar uma tarefa.

- **NotOnFaulted** e **OnlyOnFaulted** A opção *NotOnFaulted* indica que a continuação só deve ser executada se a ação anterior for concluída e não lançar uma exceção não tratada. A opção *OnlyOnFaulted* instrui a execução da continuação somente se a ação anterior lançar uma exceção não tratada. A seção “Cancele tarefas e trate exceções” fornece mais informações sobre como gerenciar exceções em uma tarefa.

■ **NotOnRanToCompletion** e **OnlyOnRanToCompletion** A opção *NotOnRanToCompletion* especifica que a continuação só deve ser executada se a ação anterior não for concluída com êxito; ela deve ter sido cancelada ou deve ter lançado uma exceção. *OnlyOnRanToCompletion* instrui a execução da continuação somente se a ação anterior for concluída com êxito.

O exemplo de código a seguir mostra como adicionar uma continuação a uma tarefa que só será executada se a ação inicial não lançar uma exceção não tratada:

```
Task task = new Task(doWork);
task.ContinueWith(doMoreWork, TaskContinuationOptions.NotOnFaulted);
task.Start();
```

Uma exigência comum dos aplicativos que ativam operações simultaneamente é sincronizar as tarefas. A classe *Task* disponibiliza o método *Wait*, que implementa um mecanismo simples de coordenação de tarefas. Com esse método, você pode suspender a execução da thread atual, até que a tarefa especificada seja concluída, como a seguir:

```
task2.Wait(); // Espera nesse ponto até que task2 termine
```

Você pode esperar um conjunto de tarefas com os métodos estáticos *WaitAll* e *WaitAny* da classe *Task*. Os dois métodos recebem um array *params* que contém um conjunto de objetos *Task*. O método *WaitAll* aguarda o término de todas as tarefas especificadas e *WaitAny* espera até que pelo menos uma das tarefas especificadas tenha terminado. Você os utiliza da seguinte maneira:

```
Task.WaitAll(task, task2); // Aguarda o término de task e de task2
Task.WaitAny(task, task2); // Aguarda o término de task ou de task2
```

Utilize a classe *Task* para implementar paralelismo

No próximo exercício, você utilizará a classe *Task* para paralelizar o código que usa intensamente o processador em um aplicativo e constatará que essa paralelização reduz o tempo necessário para a execução do aplicativo ao distribuir os cálculos pelos diversos núcleos do processador.

O aplicativo, chamado *GraphDemo*, consiste em uma página que utiliza um controle *Image* para exibir um gráfico. O aplicativo plota os pontos do gráfico, efetuando um cálculo complexo.



Nota Os exercícios deste capítulo são destinados à execução em um computador equipado com um processador multinúcleo. Se você tiver apenas uma CPU de um único núcleo, não perceberá os mesmos efeitos. Além disso, você não deve inicializar outros programas ou serviços entre os exercícios, porque eles podem afetar os resultados obtidos.

Examine e execute o aplicativo single-threaded *GraphDemo*

1. Inicialize o Microsoft Visual Studio 2013 se ele ainda não estiver em execução.

2. Abra a solução GraphDemo, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\ Chapter 23\GraphDemo de sua pasta Documents. Esse é um aplicativo Windows Store.
3. No Solution Explorer, no projeto GraphDemo, clique duas vezes no arquivo GraphWindow.xaml para exibir o formulário na janela Design View.

Além do controle *Grid* que define o layout, o formulário contém os seguintes controles importantes:

- Um controle *Image* chamado *graphImage*. Esse controle de imagem exibe o gráfico processado pelo aplicativo.
- Um controle *Button* chamado *plotButton*. O usuário clica nesse botão para gerar os dados do gráfico e exibi-los no controle *graphImage*.



Nota Para manter simples a operação do aplicativo deste exercício, o botão é exibido na página. Em um aplicativo Windows Store de produção, botões como esse devem ficar na barra de ferramentas do aplicativo.

- Um controle *TextBlock* chamado *duration*. O aplicativo exibe o tempo necessário para gerar e processar os dados do gráfico nesse rótulo.
4. No Solution Explorer, expanda o arquivo GraphWindow.xaml e clique duas vezes em GraphWindow.xaml.cs para exibir o código do formulário na janela Code and Text Editor.

O formulário utiliza um objeto *WriteableBitmap* (definido no namespace *Windows.UI.Xaml.Media.Imaging*), chamado *graphBitmap*, para processar o gráfico. As variáveis *pixelWidth* e *pixelHeight* especificam a resolução horizontal e vertical, respectivamente, do objeto *WriteableBitmap*:

```
public partial class GraphWindow : Window
{
    // Reduza pixelWidth e pixelHeight se não houver espaço suficiente
    private int pixelWidth = 12000;
    private int pixelHeight = 7500;

    private WriteableBitmap graphBitmap = null;
    ...
}
```



Nota Esse aplicativo foi desenvolvido e testado em um computador desktop com 4 GB de memória. Se seu computador tiver menos memória do que isso disponível, talvez seja necessário reduzir os valores nas variáveis *pixelWidth* e *pixelHeight*; caso contrário, o aplicativo poderá gerar exceções *OutOfMemoryException*. Do mesmo modo, se você tiver muito mais memória disponível, talvez queira aumentar os valores dessas variáveis para ver todos os efeitos desse exercício.

5. Examine as três últimas linhas do construtor de *GraphWindow*, que são as seguintes:

```

public GraphWindow()
{
    ...
    int dataSize = bytesPerPixel * pixelWidth * pixelHeight;
    data = new byte[dataSize];

    graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight);
}

```

As duas primeiras linhas instanciam um array de bytes que armazenará os dados do gráfico. O tamanho desse array depende da resolução do objeto *WriteableBitmap*, determinada pelos campos *pixelWidth* e *pixelHeight*. Além disso, esse tamanho precisa ser escalonado pelo volume de memória exigida para processar cada pixel; a classe *WriteableBitmap* utiliza 4 bytes para cada pixel, os quais especificam a intensidade relativa de vermelho, verde e azul de cada pixel e o valor da fusão alfa do pixel (o valor da fusão alfa – *alpha blending* – determina a transparência e o brilho do pixel).

A última instrução cria o objeto *WriteableBitmap* com a resolução especificada.

6. Examine o código do método *plotButton_Click*:

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);

    Stopwatch watch = Stopwatch.StartNew();
    generateGraphData(data);

    duration.Text = string.Format("Duration (ms): {0}", watch.
ElapsedMilliseconds);

    Stream pixelStream = graphBitmap.PixelBuffer.AsStream();
    pixelStream.Seek(0, SeekOrigin.Begin);
    pixelStream.Write(data, 0, data.Length);
    graphBitmap.Invalidate();
    graphImage.Source = graphBitmap;
}

```

Esse método é executado quando o usuário clica no botão *plotButton*.

Você vai clicar nesse botão várias vezes, posteriormente no exercício, para ver que uma nova versão do gráfico é desenhada sempre que esse método gera um conjunto aleatório de valores para a intensidade de vermelho, verde e azul dos pontos desenhados (o gráfico terá uma cor diferente a cada vez que você clicar nesse botão).

A variável *watch* é um objeto *System.Diagnostics.Stopwatch*. O tipo *Stopwatch* é útil para cronometrar operações. O método estático *StartNew* do tipo *Stopwatch* cria uma nova instância de um objeto *Stopwatch* e inicia sua execução. Para consultar o tempo da execução de um objeto *Stopwatch*, examine a propriedade *ElapsedMilliseconds*.

O método *generateGraphData* preenche o array *data* com os dados do gráfico a ser exibido pelo objeto *WritableDatabaseBitmap*. Você examinará esse método no próximo passo.

Quando o método *generateGraphMethod* tiver terminado, o tempo decorrido (em milissegundos) aparecerá no controle *TextBox* duration.

O último bloco de código recebe as informações armazenadas no array *data* e as copia no objeto *WritableDatabaseBitmap* para processamento. A técnica mais simples é criar um fluxo na memória (stream) que possa ser utilizado para preencher a propriedade *PixelBuffer* do objeto *WritableDatabaseBitmap*. Então, você pode usar o método *Write* desse fluxo para copiar o conteúdo do array *data* nesse buffer. O método *Invalidate* da classe *WritableDatabaseBitmap* pede para que o sistema operacional redesenhe o bitmap utilizando as informações armazenadas no buffer. A propriedade *Source* de um controle *Image* especifica os dados que esse controle deve processar. A última instrução define a propriedade *Source* como o objeto *WritableDatabaseBitmap*.

7. Examine o código do método *generateGraphData*, mostrado aqui:

```
private void generateGraphData(byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    for (int x = 0; x < a; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        }
    }
}
```

Este método efetua uma série de cálculos para plotar os pontos de um gráfico bem complexo. (O cálculo em si não é importante – ele apenas gera um gráfico de visual atraente.) Ao calcular cada ponto, ele chama o método *plotXY* para definir os bytes adequados no array *data* correspondente a esses pontos. Os pontos do gráfico são refletidos em torno do eixo x, de modo que o método *plotXY* é chamado duas vezes para cada cálculo: uma vez para o valor positivo da coordenada x e outra para o valor negativo.

8. Examine o método *plotXY*:

```
private void plotXY(byte[] data, int x, int y)
{
    int pixelIndex = (x + y * pixelWidth) * bytesPerPixel;
```

```
    data[pixelIndex] = blueValue;
    data[pixelIndex + 1] = greenValue;
    data[pixelIndex + 2] = redValue;
    data[pixelIndex + 3] = 0xBF;
}
```

Esse método define os bytes apropriados no array *data* correspondentes às coordenadas x e y passadas como parâmetros. Cada ponto desenhado corresponde a um pixel e cada pixel consiste em 4 bytes, conforme descrito anteriormente. Os pixels sem definição serão exibidos na cor preta. O valor 0xBF do byte de fusão alfa indica que o pixel correspondente deve ser exibido com intensidade moderada; se você diminuir esse valor, o pixel se tornará mais desbotado, ao passo que configurar o valor como 0xFF (o máximo para um byte) exibirá o pixel em sua intensidade mais brilhante.

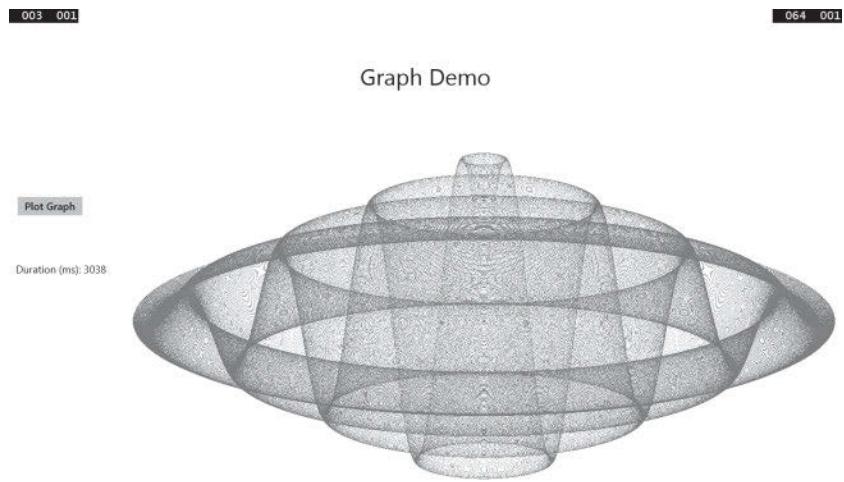
9. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.

10. Quando a janela Graph Demo for exibida, clique em Plot Graph e então aguarde.

Seja paciente. O aplicativo leva vários segundos para gerar e exibir o gráfico, e não responderá enquanto isso ocorrer (o Capítulo 24 explica o motivo disso e também informa como evitar esse comportamento). A imagem a seguir mostra o gráfico. Observe o valor no rótulo Duration (ms) na figura a seguir. Nesse caso, foram necessários 3.038 milissegundos (ms) para o aplicativo plotar o gráfico. Observe que essa duração não inclui o tempo necessário para desenhar o gráfico, o que pode levar mais alguns segundos.



Nota O aplicativo foi executado em um computador com 4 GB de memória e equipado com um processador quad-core executando a 2,40 GHz. Os tempos em seu sistema podem variar com um processador mais lento ou mais rápido, ou em um computador com mais ou menos memória.

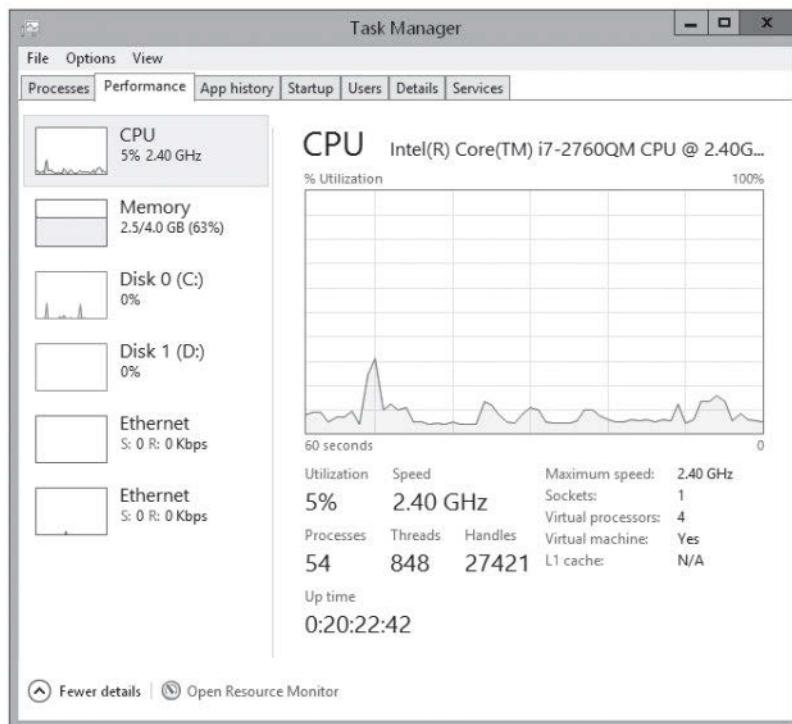


- 11.** Clique em Plot Graph novamente e anote o tempo decorrido. Repita esta ação várias vezes para obter o valor médio.



Nota Ocasionalmente, você poderá verificar que leva mais tempo para o gráfico aparecer (mais de 30 segundos). Isso tende a ocorrer se quase toda a memória de seu computador esteja sendo utilizada e o Windows 8.1 precise paginar dados entre a memória e o disco. Caso se depare com esse fenômeno, descarte esse tempo e não o inclua no cálculo de sua média.

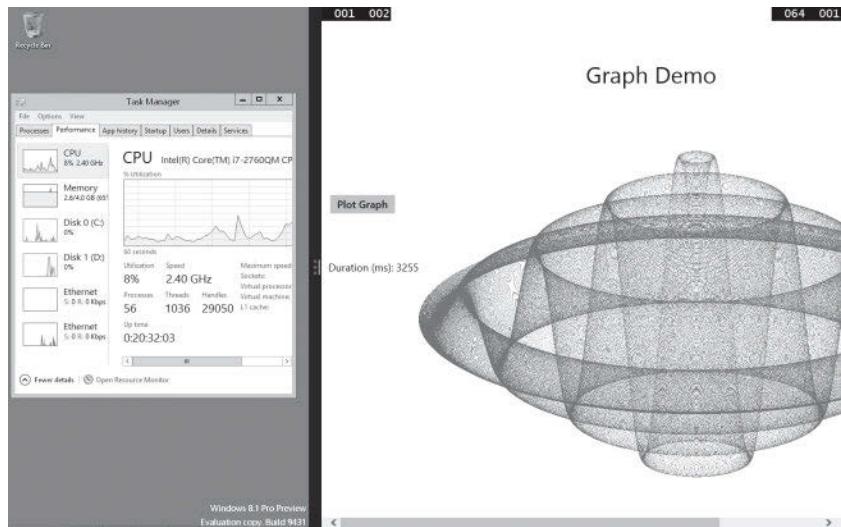
- 12.** Saia do aplicativo e alterne para a área de trabalho. Clique com o botão direito do mouse em uma área vazia da barra de tarefas e, no menu de atalho que aparece, clique em Gerenciador de Tarefas.
- 13.** Na janela Gerenciador de Tarefas, clique na guia Desempenho e exiba a utilização da CPU. Se a guia Desempenho não estiver visível, clique em Mais Detalhes (ela deverá aparecer). Clique com o botão direito do mouse no gráfico Uso de CPU, aponte para Alterar Gráfico Para e, então, clique em Utilização Geral. Essa ação faz o Gerenciador de Tarefas exibir em um único gráfico a utilização de todos os núcleos de processador em execução no seu computador. A imagem a seguir mostra a guia Desempenho do Gerenciador de Tarefas configurada dessa maneira:



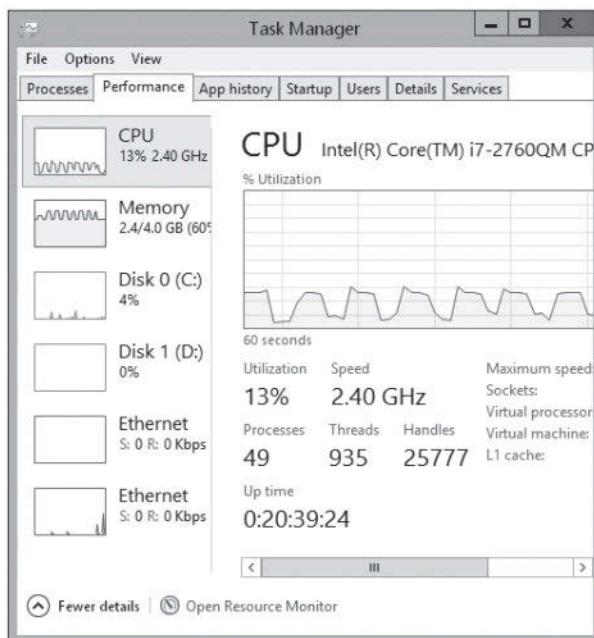
14. Volte ao aplicativo Graph Demo e ajuste a exibição para mostrá-lo na parte principal da tela, com a área de trabalho aparecendo no lado esquerdo. Certifique-se de que você possa ver a janela do Gerenciador de Tarefas exibindo a utilização da CPU.



Dica Para exibir o aplicativo GraphDemo e a Área de Trabalho do Windows lado a lado, com o aplicativo executando em tela cheia, clique no canto superior esquerdo da tela e arraste a imagem da Área de Trabalho para a metade esquerda da tela. Você pode então mover e redimensionar as janelas de acordo com sua tela.



15. Espere que a utilização da CPU estabilize e, então, na janela Graph Demo, clique em Plot Graph.
16. Espere que a utilização da CPU estabilize novamente e, então, clique em Plot Graph outra vez.
17. Repita o passo 16 várias vezes, esperando que a utilização da CPU estabilize entre os cliques.
18. Alterne para a janela do Gerenciador de Tarefas e examine a utilização da CPU. Seus resultados podem ser diferentes, mas em um processador dual-core a utilização da CPU provavelmente estará em torno de 50–55% enquanto o gráfico está sendo gerado. Em uma máquina quad-core, a utilização da CPU provavelmente ficará entre 25 e 30%, conforme mostra a imagem a seguir. Observe que outros fatores, como o tipo de placa gráfica de seu computador, também podem influenciar o desempenho.



19. Retorne ao Visual Studio 2013 e interrompa a depuração.

Você já tem um parâmetro sobre o tempo necessário ao aplicativo para efetuar seus cálculos. Entretanto, fica evidente, com base no uso da CPU exibido pelo Gerenciador de Tarefas, que o aplicativo não está utilizando plenamente os recursos de processamento disponíveis. Em uma máquina dual-core, ele está usando um pouco acima da metade da potência da CPU, e, em uma máquina quad-core, ele emprega um pouco mais de um quarto da CPU. Esse fenômeno ocorre porque o aplicativo possui uma única thread, e, em um aplicativo Windows, uma única thread pode ocupar apenas um único núcleo em um processador multicore. Para distribuir a carga por todos os núcleos disponíveis, divida o aplicativo em tarefas e faça cada tarefa ser executada em uma thread separada, executando em um núcleo diferente. É exatamente isso que você fará no próximo exercício.

Modifique o aplicativo GraphDemo para utilizar objetos Task

1. Retorne ao Visual Studio 2013 e exiba o arquivo GraphWindow.xaml.cs na janela Code and Text Editor, se ele ainda não estiver aberto.
2. Examine o método *generateGraphData*.

O objetivo desse método é preencher os itens no array *data*. Ele faz uma iteração pelo array, utilizando o loop *for* externo baseado na variável de controle de loop *x*, destacado em negrito no exemplo a seguir:

```

private void generateGraphData(byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    for (int x = 0; x < a; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        }
    }
}

```

O cálculo efetuado por uma iteração desse loop é independente dos cálculos efetuados pelas demais iterações. Portanto, compensa particionar o trabalho executado por esse loop e executar diferentes iterações em processadores separados.

3. Modifique a definição do método *generateGraphData* de modo a receber dois parâmetros *int* adicionais, chamados *partitionStart* e *partitionEnd*, como mostrado em negrito no exemplo a seguir:

```

private void generateGraphData(byte[] data, int partitionStart, int partitionEnd)
{
    ...
}

```

4. No método *generateGraphData*, mude o loop *for* externo para realizar iterações entre os valores de *partitionStart* e *partitionEnd*, como mostrado aqui em negrito:

```

private void generateGraphData(byte[] data, int partitionStart, int partitionEnd)
{
    ...
    for (int x = partitionStart; x < partitionEnd; x++)
    {
        ...
    }
}

```

5. Na janela Code and Text Editor, adicione a seguinte diretiva *using* à lista localizada no início do arquivo GraphWindow.xaml.cs:

```
using System.Threading.Tasks;
```

6. No método *plotButton_Click*, transforme em comentário a instrução que chama o método *generateGraphData* e adicione a instrução mostrada em negrito no código a seguir, que cria um objeto *Task* e o coloca em execução:

```
...
Stopwatch watch = Stopwatch.StartNew();
// generateGraphData(data);
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));
...
```

A tarefa executa o código especificado pela expressão lambda. Os valores dos parâmetros *partitionStart* e *partitionEnd* indicam que o objeto *Task* calcula os dados da primeira metade do gráfico. (Os dados do gráfico completo consistem em pontos plotados para os valores entre 0 e *pixelWidth / 2*.)

7. Adicione outra instrução que cria e executa um segundo objeto *Task* em outra thread, como mostrado no código destacado em negrito a seguir:

```
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4, pixelWidth / 2));
...
```

Esse objeto *Task* chama o método *generateGraph* e calcula os dados dos valores entre *pixelWidth / 4* e *pixelWidth / 2*.

8. Adicione a seguinte instrução, mostrada em negrito, que aguarda os dois objetos *Task* terminarem seu trabalho para continuar:

```
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4, pixelWidth / 2));
Task.WaitAll(first, second);
...
```

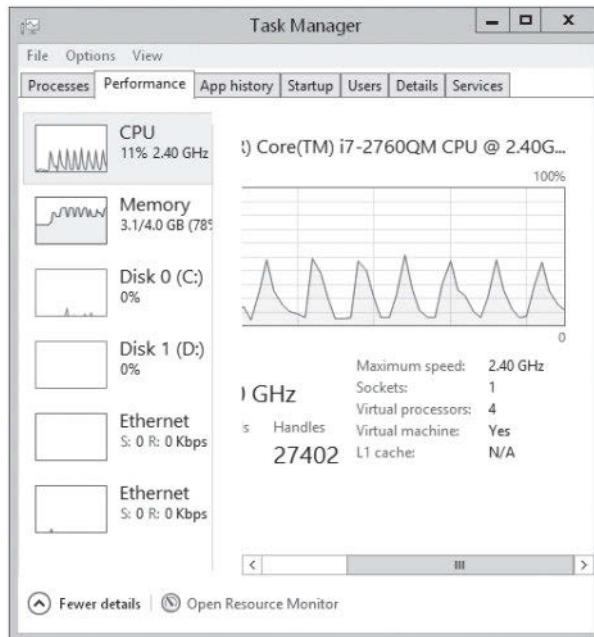
9. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo. Ajuste a exibição para mostrar o aplicativo na parte principal da tela, com a área de trabalho aparecendo no lado esquerdo. Como antes, certifique-se de que você possa ver a janela do Gerenciador de Tarefas exibindo a utilização da CPU no modo dividido.
10. Na janela Graph Demo, clique em Plot Graph. Na janela do Gerenciador de Tarefas, espere que a utilização da CPU estabilize.
11. Repita o passo 10 vezes mais, esperando que a utilização da CPU estabilize entre os cliques. Tome nota da duração registrada a cada vez que você clica no botão e calcule a média.

Você perceberá que a velocidade de execução do aplicativo é muito mais alta do que antes. Em meu computador, o tempo típico caiu para 2.951 milissegundos – uma redução em torno de 40%.

Na maioria dos casos, o tempo exigido para efetuar os cálculos será reduzido pela metade, mas o aplicativo ainda tem alguns elementos single-threaded, como a lógica que exibe o gráfico depois que os dados foram gerados. É por isso que o tempo global ainda é mais do que a metade do tempo exigido pela versão anterior do aplicativo.

12. Alterne para a janela Gerenciador de Tarefas.

Você deverá observar que o aplicativo utiliza mais núcleos da CPU. Em minha máquina quad-core, a utilização da CPU chegou a um máximo de aproximadamente 50% cada vez que cliquei em Plot Graph. Isso porque as duas tarefas foram executadas em núcleos separados, mas os dois núcleos restantes ficaram desocupados. Se você tem uma máquina dual-core, provavelmente verá uma utilização de processador atingir brevemente 100%, sempre que o gráfico for gerado.



Caso tenha um computador quad-core, você pode aumentar a utilização da CPU e reduzir o tempo ainda mais, adicionando mais dois objetos *Task* e dividindo o trabalho em quatro partes no método *plotButton_Click*, como mostrado aqui em negrito:

```
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 8));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 8,
pixelWidth / 4));
Task third = Task.Run(() => generateGraphData(data, pixelWidth / 4,
pixelWidth * 3 / 8));
Task fourth = Task.Run(() => generateGraphData(data, pixelWidth * 3 / 8,
pixelWidth / 2));
Task.WaitAll(first, second, third, fourth);
...
```

Se você tiver apenas um processador dual-core, ainda poderá experimentar essa modificação, e deverá observar um pequeno efeito positivo em relação ao tempo. Isso se deve basicamente à maneira pela qual os algoritmos utilizados pelo CLR otimizam o agendamento das threads de cada tarefa.

Abstraia tarefas com a classe *Parallel*

Ao utilizar a classe *Task*, você tem controle total sobre o número de tarefas que seu aplicativo cria. Entretanto, você teve que modificar o projeto do aplicativo para acomodar o uso de objetos *Task*. Também precisou adicionar código para sincronizar as operações; o aplicativo só pode desenhar o gráfico quando todas as tarefas estiverem concluídas. Em um aplicativo complexo, a sincronização das tarefas pode se tornar um processo complicado e é fácil cometer erros.

Com a classe *Parallel*, você pode paralelizar algumas construções (blocos) de programação comuns, sem exigir uma reformulação do aplicativo. Internamente, a classe *Parallel* cria um conjunto próprio de objetos *Task* e sincroniza automaticamente essas tarefas quando finalizadas. A classe *Parallel* está localizada no namespace *System.Threading.Tasks* e dispõe de um pequeno conjunto de métodos estáticos para indicar que o código deve ser executado em paralelo, se possível. Os métodos são os seguintes:

- **Parallel.For** Use este método no lugar da instrução *for* do C#. Ele define um loop no qual as iterações podem ocorrer em paralelo por meio de tarefas. Esse método é intensamente sobrecarregado (existem nove variações), mas o princípio geral é o mesmo para cada um deles: você especifica um valor inicial, um valor final e uma referência a um método que aceita um parâmetro de inteiro. O método é executado para todo valor entre o valor inicial e um abaixo do valor final especificado, e o parâmetro é preenchido com um inteiro que especifica o valor atual. Por exemplo, considere o seguinte loop *for* simples, que executa cada iteração em sequência:

```
for (int x = 0; x < 100; x++)  
{  
    // Processamento do loop  
}
```

Dependendo do processamento executado pelo corpo do loop, você poderá substituir esse loop por uma construção *Parallel.For* que pode fazer iterações em paralelo, como a seguir:

```
Parallel.For(0, 100, performLoopProcessing);  
...  
private void performLoopProcessing(int x)  
{  
    // Processamento do loop  
}
```

Com as sobrecargas do método *Parallel.For* é possível fornecer dados locais que são privados para cada thread, especificar várias opções para criar as tarefas executadas pelo método *For* e criar um objeto *ParallelLoopState* que pode ser utilizado para passar informações de estado para outras iterações simultâneas do loop. (O uso de um objeto *ParallelLoopState* será descrito posteriormente neste capítulo.)

- **Parallel.ForEach<T>** Use este método no lugar da instrução *foreach* do C#. Como no método *For*, *ForEach* define um loop no qual as iterações podem ocorrer em paralelo. Especifique uma coleção que implementa a interface genérica *IEnumerable<T>* e uma referência a um método que aceita um único parâmetro do tipo *T*. O método é executado para cada item da coleção e o item é passado como parâmetro para o método. Existem sobrecargas disponíveis que permitem

fornecer dados privados da thread local e especificar opções para criar as tarefas executadas pelo método *ForEach*.

- **Parallel.Invoke** Você pode utilizar esse método para executar um conjunto de chamadas a métodos sem parâmetros como tarefas paralelas. Especifique uma lista das chamadas aos métodos delegados (ou expressões lambda) que não aceitam parâmetros nem retornam valores. Cada chamada ao método pode ser executada em uma thread separada, em qualquer sequência. Por exemplo, o código a seguir faz uma série de chamadas de método:

```
doWork();
doMoreWork();
doYetMoreWork;
```

Você pode substituir essas instruções pelo código a seguir, que chama esses métodos utilizando uma série de tarefas:

```
Parallel.Invoke(
    doWork,
    doMoreWork,
    doYetMoreWork
);
```

Convém lembrar que a classe *Parallel* determina o nível real de paralelismo adequado ao ambiente e à carga de trabalho do computador. Por exemplo, se você utilizar *Parallel.For* para implementar um loop que executa 1.000 iterações, a classe *Parallel* não criará necessariamente 1.000 tarefas simultâneas (a menos que exista em seu sistema um processador excepcionalmente poderoso, com 1.000 núcleos). Em vez disso, a classe *Parallel* gerará o que considera o número ideal de tarefas que equilibra os recursos disponíveis em relação à exigência de manter os processadores ocupados. Uma única tarefa pode fazer diversas iterações, e as tarefas se coordenam entre si para determinar quais iterações cada uma executará. Uma consequência importante disso é a impossibilidade de garantir a sequência de execução das iterações, de modo que você deve assegurar que não existam dependências entre as iterações; caso contrário, poderá encontrar resultados imprevistos, como veremos mais adiante neste capítulo.

No próximo exercício, você retornará à versão original do aplicativo GraphData e utilizará a classe *Parallel* para executar operações simultaneamente.

Use a classe *Parallel* para paralelizar operações no aplicativo GraphData

1. No Visual Studio 2013, abra a solução GraphDemo, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 23\Parallel GraphDemo na sua pasta Documentos.

Essa é uma cópia do aplicativo original GraphDemo. Ela não emprega tarefas ainda.

2. No Solution Explorer, no projeto GraphDemo, expanda o nó GraphWindow.xaml e clique duas vezes em GraphWindow.xaml.cs para exibir o código do formulário na janela Code and Text Editor.
3. Adicione a seguinte diretiva *using* à lista localizada no início do arquivo:

```
using System.Threading.Tasks;
```

4. Localize o método *generateGraphData*. Ele é semelhante ao seguinte:

```
private void generateGraphData(byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    for (int x = 0; x < a; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        }
    }
}
```

O loop *for* externo, que itera pelos valores da variável de inteiros *x*, é um excelente candidato à paralelização. Você também pode considerar o loop interno baseado na variável *i*, mas esse loop exige mais esforços para paralelizar devido ao tipo da variável *i*. (Os métodos da classe *Parallel* esperam que a variável de controle seja um inteiro.) Além disso, se existirem loops aninhados, como os que ocorrem nesse código, é recomendável paralelizar os loops externos primeiramente e então verificar se o desempenho do aplicativo é suficiente. Se não for, faça o que é necessário com os loops aninhados e paralelize-os dos loops externos para os internos, e teste o desempenho após modificar cada um deles. Você perceberá que, em vários cenários, a paralelização dos loops externos surte o impacto máximo sobre o desempenho, enquanto os efeitos da modificação dos loops internos são mais sutis.

5. Recorte o código do corpo do loop *for* e crie um novo método privado *void*, chamado *calculateData*, com esse código. O método *calculateData* deve receber um parâmetro *int* chamado *x* e um array de bytes chamado *data*. Além disso, mova as instruções que declaram as variáveis locais *a*, *b* e *c* do método *generateGraphData* para o início do método *calculateData*. O código a seguir mostra o método *generateGraphData* com esse código removido e o método *calculateData* (não tente compilar esse código ainda):

```
private void generateGraphData(byte[] data)
{
    for (int x = 0; x < a; x++)
    {
    }
}
```

```

private void calculateData(int x, byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    int s = x * x;
    double p = Math.Sqrt(b - s);
    for (double i = -p; i < p; i += 3)
    {
        double r = Math.Sqrt(s + i * i) / a;
        double q = (r - 1) * Math.Sin(24 * r);
        double y = i / 3 + (q * c);
        plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
    }
}

```

- 6.** No método *generateGraphData*, mude o loop *for* para uma instrução que chama o método estático *Parallel.For*, como mostrado em negrito no código a seguir:

```

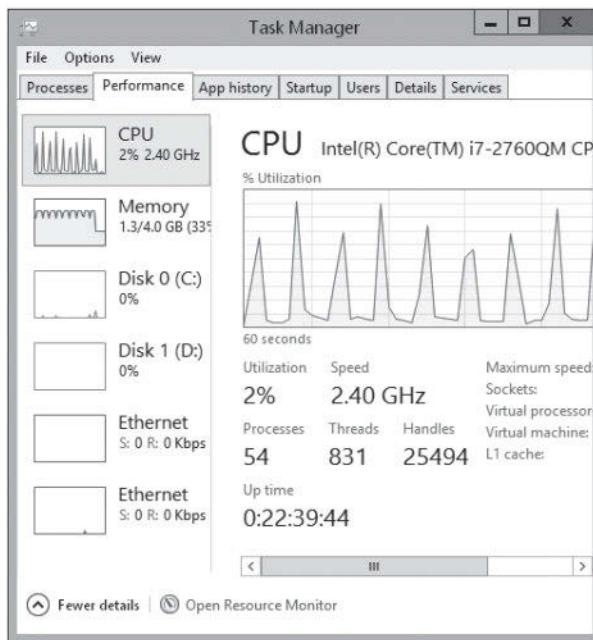
private void generateGraphData(byte[] data)
{
    Parallel.For(0, pixelWidth / 2, x => calculateData(x, data));
}

```

Esse código é o equivalente paralelo do loop *for* original. Ele itera pelos valores de 0 a *pixelWidth / 2 – 1*, inclusive. Cada chamada é executada com uma tarefa e cada tarefa pode executar mais de uma iteração. O método *Parallel.For* só terminará quando todas as tarefas por ele criadas concluírem seu trabalho. Lembre-se de que o método *Parallel.For* espera que o último parâmetro seja um método que aceita um único parâmetro inteiro. Ele chama esse método passando o índice atual do loop como parâmetro. Neste exemplo, o método *calculateData* não corresponde à assinatura necessária, pois aceita dois parâmetros: um inteiro e um array de bytes. Por esse motivo, o código utiliza uma expressão lambda que atua como um adaptador que chama o método *calculateData* com os argumentos apropriados.

- 7.** No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
8. Na janela Graph Demo, clique em Plot Graph. Quando o gráfico aparecer na janela Graph Demo, registre o tempo necessário para gerá-lo. Repita esta ação várias vezes para obter o valor médio.

Observe que o aplicativo funciona a uma velocidade comparável à da versão anterior, a qual utiliza objetos *Task* (e, possivelmente, um pouco mais veloz, dependendo do número de CPUs disponíveis). Se examinar o Gerenciador de Tarefas, você deverá notar que a utilização da CPU atinge picos próximos a 100%, independentemente de ter um computador dual-core ou quad-core.



9. Retorne ao Visual Studio e interrompa a depuração.

Quando não utilizar a classe *Parallel*

Saiba que, apesar das aparências e dos melhores esforços da equipe de desenvolvimento do .NET Framework na Microsoft, a classe *Parallel* não faz mágica – você não pode utilizá-la sem a devida consideração e esperar apenas que seus aplicativos funcionem, repentinamente, com muito mais velocidade, e gerem os mesmos resultados. O objetivo da classe *Parallel* é paralelizar as áreas de seu código que são independentes e vinculadas à computação.

Se seu código não for relacionado a alguma computação, é possível que a respectiva paralelização não melhore o desempenho. A sobrecarga de criar uma tarefa, executar essa tarefa em uma thread separada e aguardar o término da tarefa, provavelmente é maior do que o custo de executar esse método diretamente. A sobrecarga adicional pode contabilizar apenas alguns milissegundos, sempre que um método for chamado, mas você deve ter em mente o número de vezes que um método é executado. Se a chamada ao método está localizada em um loop aninhado e é executada milhares de vezes, todos esses pequenos custos de sobrecarga se acumulam. A regra geral é usar o *Parallel.Invoke* somente quando compensar. *Parallel.Invoke* deve ser reservado para operações que utilizam bastante poder de computação; caso contrário, a sobrecarga de criar e gerenciar tarefas pode na verdade diminuir a velocidade de um aplicativo.

A outra consideração importante sobre o uso da classe *Parallel* é que as operações paralelas devem ser independentes. Por exemplo, se você tentar utilizar *Parallel.For* para paralelizar um loop no qual as iterações são interdependentes, os resultados serão imprevisíveis.

Para saber o que isso significa, examine o seguinte código:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ParallelLoop
{
    class Program
    {
        private static int accumulator = 0;

        static void Main(string[] args)
        {
            for (int i = 0; i < 100; i++)
            {
                AddToAccumulator(i);
            }
            Console.WriteLine("Accumulator is {0}", accumulator);
        }

        private static void AddToAccumulator(int data)
        {
            if ((accumulator % 2) == 0)
            {
                accumulator += data;
            }
            else
            {
                accumulator -= data;
            }
        }
    }
}
```

Esse programa itera pelos valores de 0 a 99 e chama o método *AddToAccumulator* com um valor de cada vez. O método *AddToAccumulator* examina o valor atual da variável *accumulator* e, se esse valor for par, adiciona o valor do parâmetro à variável *accumulator*; caso contrário, subtrai o valor do parâmetro. Quando o programa termina, o resultado é exibido. Você encontrará esse aplicativo na solução *ParallelLoop*, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\ Chapter 23\ParallelLoop de sua pasta Documentos. Se você executar esse programa, o valor emitido deverá ser -100.

Para aumentar o nível de paralelismo nesse aplicativo simples, talvez você ficasse com vontade de substituir o loop *for* no método *Main* por *Parallel.For*, como a seguir:

```
static void Main(string[] args)
{
    Parallel.For(0, 100, AddToAccumulator);
    Console.WriteLine("Accumulator is {0}", accumulator);
}
```

Entretanto, não é possível garantir que as tarefas criadas para executar as diversas chamadas do método *AddToAccumulator* sejam executadas em uma sequência específica. (O código também não está protegido contra threads – não é thread-safe – porque várias threads executando as tarefas podem tentar modificar a variável *accumulator* paralelamente.) O valor calculado pelo método *AddToAccumulator* depende da sequência mantida, de modo que o resultado dessa modificação é que o aplicativo

pode agora gerar valores diferentes cada vez que é executado. Nesse caso simples, é provável que você não perceba qualquer diferença no valor calculado, porque o método *AddToAccumulator* opera muito rapidamente e o .NET Framework pode optar por executar cada chamada em sequência, utilizando a mesma thread. Contudo, se você fizer a seguinte mudança, mostrada em negrito, no método *AddToAccumulator*, obterá outros resultados:

```
private static void AddToAccumulator(int data)
{
    if ((accumulator % 2) == 0)
    {
        accumulator += data;
    Thread.Sleep(10); // espera por 10 milissegundos
    }
    else
    {
        accumulator -= data;
    }
}
```

O método *Thread.Sleep* simplesmente faz a thread atual aguardar o intervalo de tempo especificado. Essa modificação simula a thread executando um processamento adicional e afeta o modo como a classe *Parallel* agenda as tarefas já executadas em threads diferentes, o que resulta em uma sequência diferente.

A regra geral é só utilizar *Parallel.For* e *Parallel.ForEach* se você tiver certeza de que cada iteração do loop será independente e testar seu código a fundo. Uma consideração semelhante se aplica a *Parallel.Invoke*: use essa construção para fazer chamadas a métodos somente se forem independentes e se o aplicativo não depender da respectiva execução em uma sequência específica.

Cancele tarefas e trate exceções

Uma exigência comum dos aplicativos que efetuam operações demoradas é a possibilidade de interromper essas operações, se necessário. Entretanto, você não deve simplesmente abortar uma tarefa, porque isso poderia deixar os dados do seu aplicativo em um estado indeterminado. Em vez disso, a classe *Task* implementa uma estratégia de cancelamento cooperativo. Um cancelamento cooperativo permite que uma tarefa selecione um ponto adequado no qual interromper o processamento, e também permite que ela desfaça qualquer trabalho executado antes do cancelamento, se necessário.

Mecânica do cancelamento cooperativo

O cancelamento cooperativo se baseia no conceito de *token de cancelamento*. Um token de cancelamento é uma estrutura que representa uma solicitação para cancelar uma ou mais tarefas. O método que uma tarefa executa deve incluir um parâmetro *System.Threading.CancellationToken*. Para cancelar a tarefa, um aplicativo define a propriedade booleana *IsCancellationRequested* desse parâmetro com *true*. O método em execução na tarefa pode consultar essa propriedade em vários momentos, ao longo do processamento. Se essa propriedade for definida como *true* em qualquer

momento, ele reconhecerá que o aplicativo solicitou o cancelamento da tarefa. Além disso, o método reconhece todo o trabalho executado até então e pode desfazer qualquer alteração realizada, se necessário, e depois encerrar. Como alternativa, o método pode simplesmente ignorar a solicitação e continuar a execução.



Dica Você deve examinar com frequência o token de cancelamento em uma tarefa, mas não com tanta frequência a ponto de impactar o desempenho da tarefa. Se possível, verifique o cancelamento a cada 10 milissegundos, e não a cada milissegundo.

Para obter um *CancellationToken*, o aplicativo deve criar um objeto *System.Threading.CancellationTokenSource* e consultar a propriedade *Token* desse objeto. O aplicativo pode, então, passar esse objeto *CancellationToken* como parâmetro para qualquer método disparado por tarefas que o aplicativo cria e executa. Para cancelar as tarefas, o aplicativo deve chamar o método *Cancel* do objeto *CancellationTokenSource*. Esse método define a propriedade *IsCancellationRequested* do *CancellationToken* passado para todas as tarefas.

O exemplo de código a seguir mostra como criar um token de cancelamento e utilizá-lo para cancelar uma tarefa. O método *initiateTasks* instancia a variável *cancellationTokenSource* e obtém uma referência ao objeto *CancellationToken* disponível por meio dessa variável. Em seguida, o código cria e executa uma tarefa que executa o método *doWork*. Mais adiante, o código chama o método *Cancel* da origem (*source*) do token de cancelamento, que define esse token. O método *doWork* consulta a propriedade *IsCancellationRequested* do token de cancelamento. Se a propriedade estiver definida, o método será encerrado; caso contrário, continuará em execução.

```
public class MyApplication
{
    ...
    // Método que cria e gerencia uma tarefa
    private void initiateTasks()
    {
        // Cria a origem do token de cancelamento e obtém um token de cancelamento
        CancellationTokenSource cancellationTokenSource = new
        CancellationTokenSource();
        CancellationToken cancellationToken = cancellationTokenSource.Token;
        // Cria uma tarefa e a inicia para executar o método doWork
        Task myTask = Task.Run(() => doWork(cancellationToken));
        ...
        if (...)
        {
            // Cancela a tarefa
            cancellationTokenSource.Cancel();
        }
        ...
    }

    // Método executado pela tarefa
    private void doWork(CancellationToken token)
    {
        ...
        // Se o aplicativo definiu o token de cancelamento, encerra o processamento
        if (token.IsCancellationRequested)
        {
```

```
// Limpeza e encerramento
...
return;
}
// Se a tarefa não foi cancelada, continua a execução normalmente
...
}
```

Além de propiciar um alto grau de controle sobre o processamento de cancelamento, essa estratégia suporta escalabilidade com qualquer número de tarefas; você pode iniciar várias tarefas e passar o mesmo objeto *CancellationToken* para cada uma delas. Se você chamar *Cancel* a partir do objeto *CancellationTokenSource*, cada tarefa verá que a propriedade *IsCancellationRequested* foi definida e procederá adequadamente.

Você também pode registrar um método de retorno de chamada (na forma de um delegate *Action*) com o token de cancelamento, por meio do método *Register*. Quando um aplicativo chamar o método *Cancel* do objeto *CancellationTokenSource* correspondente, esse retorno de chamada (callback) será executado. Entretanto, não é possível garantir quando esse método executará; pode ser antes ou depois de as tarefas terem executado seus próprios processamentos de cancelamento, ou até mesmo durante esse processo.

No próximo exercício, você adicionará a funcionalidade de cancelamento ao aplicativo GraphDemo.

Adicione a funcionalidade de cancelamento ao aplicativo GraphDemo

1. No Visual Studio 2013, abra a solução GraphDemo, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 23\GraphDemo With Cancellation de sua pasta Documentos.

Essa é uma cópia completa do aplicativo GraphDemo do exercício anterior, que utiliza tarefas para melhorar o desempenho de saída. A interface do usuário também contém um botão chamado *cancelButton* que o usuário poderá utilizar para interromper as tarefas que calculam os dados do gráfico.

2. No Solution Explorer, no projeto GraphDemo, clique duas vezes em *GraphWindow.xaml* para exibir o formulário na janela Design View. Observe o botão *Cancel* que aparece no painel esquerdo do formulário.
3. Abra o arquivo *GraphWindow.xaml.cs* na janela Code and Text Editor. Localize o método *cancelButton_Click*.

Esse método é executado quando o usuário clica em *Cancel*. Ele estará vazio.

4. Adicione a seguinte diretiva *using* à lista localizada no início do arquivo:

```
using System.Threading;
```

Os tipos utilizados pelo cancelamento cooperativo residem nesse namespace.

5. Adicione um campo *CancellationTokenSource*, chamado *tokenSource*, à classe *GraphWindow* e inicialize-o com *null*, como mostrado em negrito no código a seguir:

```
public class GraphWindow : Page
{
    ...
    private byte redValue, greenValue, blueValue;
private CancellationTokenSource tokenSource = null;
    ...
}
```

6. Localize o método *generateGraphData* e adicione um parâmetro do tipo *CancellationToken*, chamado *token*, à definição do método, como mostrado aqui em negrito:

```
private void generateGraphData(byte[] data, int partitionStart, int partitionEnd,
CancellationToken token)
{
    ...
}
```

7. No método *generateGraphData*, no início do loop *for* interno, adicione o código mostrado em negrito a seguir para verificar se foi solicitado um cancelamento. Em caso afirmativo, retorne do método; caso contrário, continue calculando valores e desenhando o gráfico.

```
private void generateGraphData(byte[] data, int partitionStart, int partitionEnd,
CancellationToken token)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    for (int x = partitionStart; x < partitionEnd; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            if (token.IsCancellationRequested)
            {
                return;
            }
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        }
    }
}
```

8. No método *plotButton_Click*, adicione as seguintes instruções, mostradas em negrito, que instanciam a variável *tokenSource* e recuperam o objeto *CancellationToken* em uma variável chamada *token*:

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);

    tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;
    ...
}

```

- 9.** Modifique as instruções que criam e executam as duas tarefas e passe a variável *token* como o último parâmetro para o método *generateGraphData*:

```

...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4,
    token));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
    pixelWidth / 2, token));
...

```

- 10.** Edite a definição do método *plotButton_Click* e adicione o modificador *async*, como mostrado aqui em negrito:

```

private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
}

```

- 11.** No corpo do método *plotButton_Click*, transforme em comentário a instrução *Task.WaitAll* que aguarda o término das tarefas e a substitua pelas instruções mostradas em negrito a seguir, que utilizam o operador *await*.

```

...
// Task.WaitAll(first, second);
await first;
await second;

duration.Text = string.Format(...);
...

```

As alterações desses dois passos são necessárias devido à natureza single-threaded da interface de usuário do Windows. Sob circunstâncias normais, quando uma rotina de tratamento de evento de um componente da interface do usuário, como um botão, começa a executar, as rotinas de tratamento de evento dos outros componentes da interface são bloqueados, até que a primeira termine (mesmo que a rotina de tratamento de evento esteja utilizando tarefas). Nesse exemplo, o uso do método *Task.WaitAll* para aguardar o término das tarefas tornaria o botão Cancel inútil, pois a rotina de tratamento de evento do botão Cancel não seria executada até que a rotina de tratamento do botão Plot Graph terminasse, no caso em que não faria sentido tentar cancelar a operação. Na verdade, como mencionado antes, quando você clica no botão Plot Graph, a interface do usuário fica totalmente sem reação até que o gráfico apareça e o método *plotButton_Click* termine.

O operador *await* foi projetado para tratar de situações como essa. Esse operador só pode ser utilizado dentro de um método marcado como *async*. Seu objetivo é liberar a thread atual e aguardar pelo término da tarefa em segundo plano. Quando a tarefa termina, o controle volta para o método, o qual continua com a próxima instrução. Nesse exemplo, as duas instruções *await* simplesmente permitem que cada uma das tarefas termine em segundo plano. Depois que a segunda tarefa termina, o método continua, exibindo no *TextBlock duration* o tempo gasto para a conclusão dessas tarefas. Observe que não é um erro aguardar por uma tarefa já concluída; o operador *await* apenas retornará imediatamente e passará o controle para a instrução seguinte.



Mais informações O Capítulo 24 discute o modificador *async* e o operador *await* com detalhes.

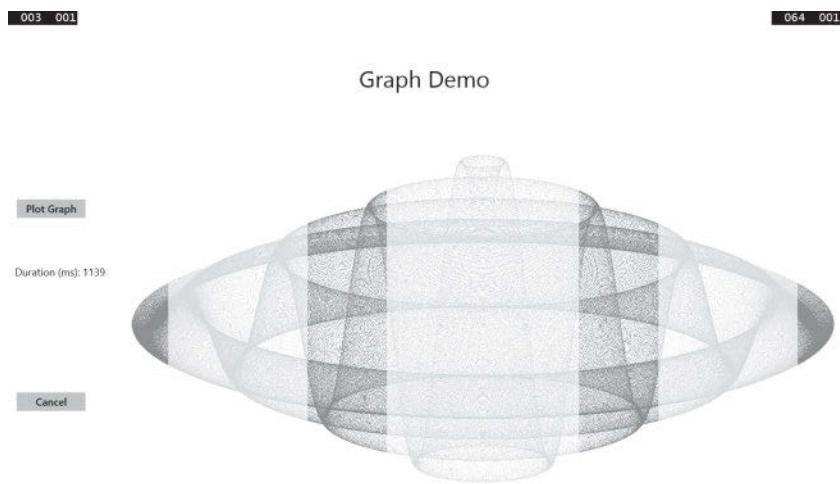
12. Localize o método *cancelButton_Click*. Adicione a esse método o código mostrado aqui em negrito:

```
private void cancelButton_Click(object sender, RoutedEventArgs e)
{
    if (tokenSource != null)
    {
        tokenSource.Cancel();
    }
}
```

Esse código verifica se a variável *tokenSource* foi instanciada. Caso tenha sido, o código chama o método *Cancel* a partir dessa variável.

13. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
14. Na janela GraphDemo, clique em Plot Graph e verifique que o gráfico é exibido como anteriormente. Mas você deve observar que demora um pouco mais do que antes para gerar o gráfico. Isso acontece por causa da verificação adicional realizada pelo método *generateGraphData*.
15. Clique em Plot Graph de novo, e clique rapidamente em Cancel.

Se você clicar rapidamente em Cancel, antes da geração dos dados do gráfico, essa ação instruirá que os métodos sendo executados pelas tarefas devem retornar. Os dados não estarão completos, de modo que o gráfico será exibido com “lacunas”, como mostra a figura a seguir. (O gráfico anterior ainda deve estar visível onde essas lacunas ocorrem, e o tamanho das lacunas dependerá da rapidez com que você clicar em Cancel.)



16. Retorne ao Visual Studio e interrompa a depuração.

Para saber se uma tarefa foi concluída ou cancelada, examine a propriedade *Status* do objeto *Task*. Essa propriedade contém um valor da enumeração *System.Threading.Tasks.TaskStatus*. A lista a seguir descreve alguns dos valores de status que você pode encontrar frequentemente (existem outros):

- **Created** Esse é o estado inicial de uma tarefa. Ela foi criada, mas ainda não foi agendada para execução.
- **WaitingToRun** A tarefa foi agendada, mas a sua execução ainda não foi iniciada.
- **Running** A tarefa está em execução no momento por uma thread.
- **RanToCompletion** A tarefa foi concluída com êxito, sem qualquer exceção não tratada.
- **Canceled** A tarefa foi cancelada antes do início de sua execução ou reconheceu o cancelamento e finalizou sem lançar uma exceção.
- **Faulted** A tarefa foi encerrada devido a uma exceção.

No próximo exercício, você informará o status de cada tarefa para que possa ver quando elas foram concluídas ou canceladas.

Cancelando um loop *Parallel.For* ou *ForEach*

Os métodos *Parallel.For* e *Parallel.ForEach* não fornecem acesso direto aos objetos *Task* criados. Na realidade, você nem sabe quantas tarefas estão em execução – o .NET Framework utiliza uma heurística própria para determinar o número ideal a ser utilizado com base nos recursos disponíveis e na carga de trabalho atual do computador.

Para interromper prematuramente o método *Parallel.For* ou *Parallel.ForEach*, use um objeto *ParallelLoopState*. O método especificado como corpo do loop deve incluir um parâmetro adicional, *ParallelLoopState*. A classe *Parallel* cria um objeto *ParallelLoopState* e passa-o como esse parâmetro para o método. A classe *Parallel* usa esse objeto para armazenar informações sobre cada chamada ao método. O método pode chamar o método *Stop* desse objeto para indicar que a classe *Parallel* não deve tentar realizar outras iterações além daquelas já iniciadas e finalizadas. O exemplo a seguir mostra o método *Parallel.For* chamando o método *doLoopWork* para cada iteração. O método *doLoopWork* examina a variável de iteração; se ela for maior que 600, o método chamará o método *Stop* do parâmetro *ParallelLoopState*. Isso instruirá o método *Parallel.For* a interromper a execução de outras iterações do loop. (As iterações atualmente em execução podem continuar até o seu final.)



Nota Lembre-se de que as iterações em um loop *Parallel.For* não são executadas em uma sequência específica. Consequentemente, cancelar o loop quando a variável da iteração tem o valor 600 não garante que as 599 iterações anteriores já tenham sido executadas. Do mesmo modo, algumas iterações com valores acima de 600 já podem ter sido concluídas.

```
Parallel.For(0, 1000, doLoopWork);
...
private void doLoopWork(int i, ParallelLoopState p)
{
    ...
    if (i > 600)
    {
        p.Stop();
    }
}
```

Exiba o status de cada tarefa

1. No Visual Studio, exiba o arquivo GraphWindow.xaml na janela Design View. No painel XAML, adicione a seguinte marcação à definição do formulário *GraphWindow*, antes da penúltima tag (ou marca) *</Grid>*, como mostrado em negrito a seguir:

```
<Image x:Name="graphImage" Grid.Column="1" Stretch="Fill" />
</Grid>
```

```
<TextBlock x:Name="messages" Grid.Row="4" FontSize="18"
HorizontalAlignment="Left"/>
</Grid>
</Grid>
</ScrollViewer>
</Page>
```

Essa marcação adiciona um controle *TextBlock* chamado *messages* à parte inferior do formulário.

2. Exiba o arquivo GraphWindow.xaml.cs na janela Code and Text Editor e localize o método *plotButton_Click*.
3. Adicione a esse método o código em negrito a seguir. Essas instruções geram uma string que contém o status de cada tarefa após o término de sua execução; em seguida, exibem essa string no controle *TextBlock messages*, na parte inferior do formulário.

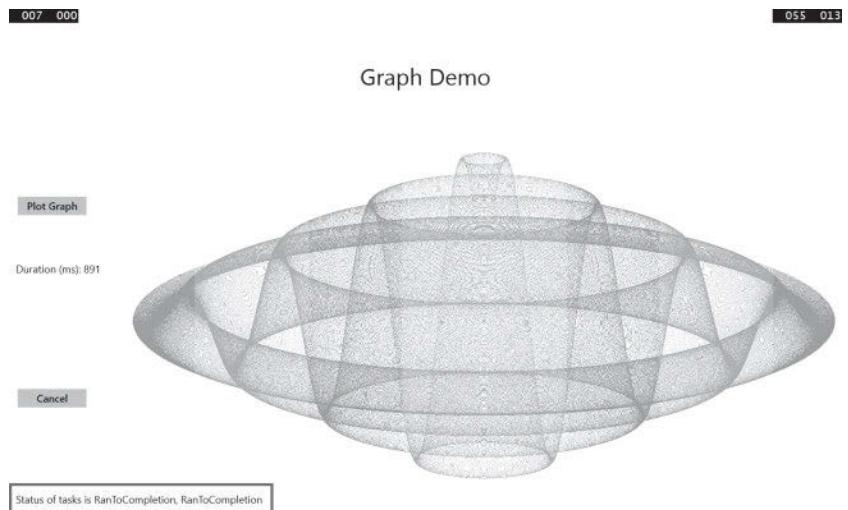
```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    await first;
    await second;

    duration.Text = string.Format(...);

    string message = string.Format("Status of tasks is {0}, {1}",
        first.Status, second.Status);
    messages.Text = message;
    ...
}
```

4. No menu Debug, clique em Start Debugging.
5. Na janela GraphDemo, clique em Plot Graph, mas não em Cancel. Verifique se a mensagem exibida informa que o status das tarefas é *RanToCompletion* (duas vezes).
6. Na janela GraphDemo, clique em Plot Graph de novo, depois, clique rapidamente em Cancel.

Surpreendentemente, a mensagem exibida ainda informa o status de cada tarefa como *RanToCompletion*, embora o gráfico apareça com lacunas.



Esse comportamento ocorre porque, embora você tenha enviado uma solicitação de cancelamento para cada tarefa com o token de cancelamento, os métodos que elas estavam executando simplesmente retornaram. O runtime do .NET Framework não sabe se as tarefas foram efetivamente canceladas ou se foram autorizadas a seguir com a execução até o final e simplesmente ignoraram as solicitações de cancelamento.

7. Retorne ao Visual Studio e interrompa a depuração.

Então, como é possível indicar que uma tarefa foi cancelada, em vez de ter sido autorizada a continuar executando até o final? A resposta está no objeto *CancellationToken* passado como parâmetro para o método que a tarefa está executando. A classe *CancellationToken* oferece um método chamado *ThrowIfCancellationRequested*. Esse método testa a propriedade *IsCancellationRequested* de um token de cancelamento; se for *true*, o método lançará uma exceção *OperationCanceledException* e abortará o método que a tarefa está executando.

O aplicativo que iniciou a thread deve estar preparado para capturar e tratar essa exceção, mas isso leva a outra questão. Se uma tarefa terminar por causa de uma exceção, ela reverterá para o estado *Faulted*. Isso ocorre mesmo que a exceção seja *OperationCanceledException*. Uma tarefa só entra no estado *Canceled* se for cancelada sem lançar uma exceção. Então, como uma tarefa pode lançar uma *OperationCanceledException* sem ser tratada como uma exceção?

Desta vez, a resposta está na própria tarefa. Para que uma tarefa reconheça que uma exceção *OperationCanceledException* é o resultado do cancelamento da tarefa de maneira controlada e não apenas por uma exceção causada por outras circunstâncias, ela precisa saber que a operação foi realmente cancelada. Ela só conseguirá fazer isso se examinar o token de cancelamento. Você passou esse token como parâmetro para o método executado pela tarefa, mas, na realidade, a tarefa não verifica esses parâmetros. Em vez disso, você especifica o token de cancelamento onde cria e executa a tarefa. O código a seguir mostra um exemplo baseado no aplicativo GraphDemo. Observe como o parâmetro *token* é passado para o método *generateGraphData* (como anteriormente), mas também como um parâmetro separado para o método *Run*:

```

tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4, token),
token);

```

Agora, quando o método que está sendo executado pela tarefa lançar uma exceção *OperationCanceledException*, a infraestrutura por trás da tarefa examinará o *CancellationToken*. Se ele indicar que a tarefa foi cancelada, a infraestrutura definirá o status da tarefa como *Canceled*. Caso esteja usando o operador *await* para aguardar a conclusão das tarefas, você também precisa estar preparado para capturar e tratar a exceção *OperationCanceledException*. É isso que você fará no próximo exercício.

Reconheça o cancelamento e trate a exceção *OperationCanceledException*

1. No Visual Studio, volte para a janela Code and Text Editor que exibe o arquivo GraphWindow.xaml.cs. No método *plotButton_Click*, modifique as instruções que criam e executam as tarefas, e especifique o objeto *CancellationToken* como o segundo parâmetro para o método *Run*, como mostrado em negrito no código a seguir:

```

private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;

    ...
    Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4,
    token), token);
    Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
    pixelWidth / 2, token), token);
    ...
}

```

2. Adicione um bloco *try* ao redor das instruções que criam e executam as tarefas, aguardam o término delas e exibem o tempo decorrido. Adicione um bloco *catch* que trata a exceção *OperationCanceledException*. Nessa rotina de tratamento de exceções, exiba o motivo da exceção relatada na propriedade *Message* do objeto de exceção, no controle *TextBlock duration*. O código mostrado em negrito a seguir destaca as alterações a serem feitas:

```

private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    try
    {
        await first;
        await second;

        duration.Text = string.Format("Duration (ms): {0}", watch.ElapsedMilliseconds);
    }
    catch (OperationCanceledException oce)

```

```

{
    duration.Text = oce.Message;
}

string message = string.Format(...);
...
}

```

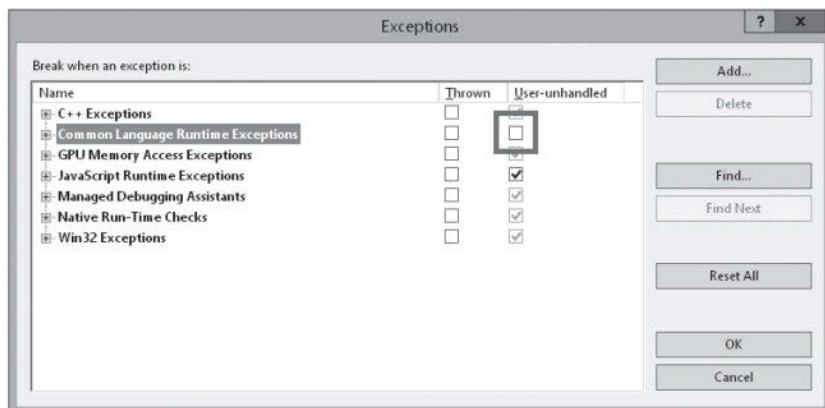
- 3.** No método *generateDataForGraph*, transforme em comentário a instrução *if* que examina a propriedade *IsCancellationRequired* do objeto *CancellationToken* e adicione uma instrução que chama o método *ThrowIfCancellationRequested*, como mostrado em negrito a seguir:

```

private void generateDataForGraph(byte[] data, int partitionStart, int partitionEnd,
CancellationToken token)
{
    ...
    for (int x = partitionStart; x < partitionEnd; x++)
    {
        ...
        for (double i = -p; I < p; i += 3)
        {
            //if (token.IsCancellationRequired)
            //{
            //    return;
            //}
            token.ThrowIfCancellationRequested();
            ...
        }
    }
    ...
}

```

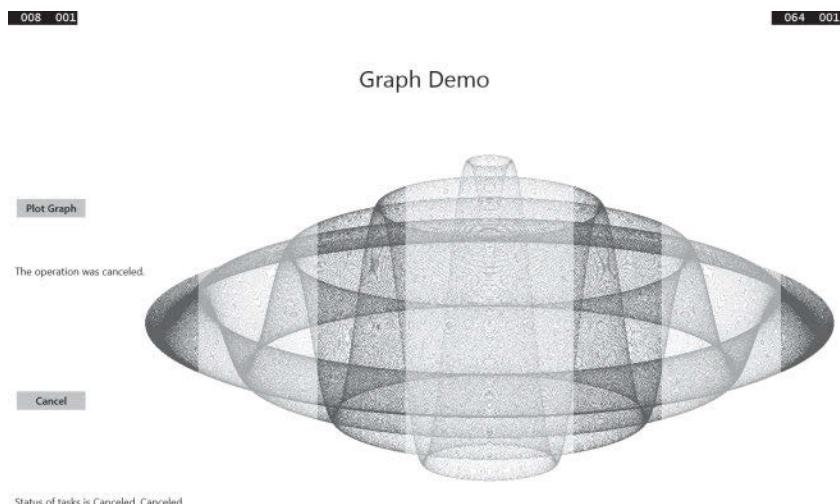
- 4.** No menu Debug, clique em Exceptions. Na caixa de diálogo Exceptions, desmarque a caixa de seleção User-unhandled do item Common Language Runtime Exceptions e, em seguida, clique em OK.



Essa configuração é necessária para impedir que o depurador do Visual Studio intercepte a exceção *OperationCanceledException* que você vai gerar quando executar o aplicativo no modo de depuração.

5. No menu Debug, clique em Start Debugging.
6. Na janela Graph Demo, clique em Plot Graph, espere que o gráfico apareça e verifique que o status das duas tarefas é relatado como *RanToCompletion* e que o gráfico é gerado.
7. Clique em Plot Graph de novo, e clique rapidamente em Cancel.

Se você conseguir clicar rapidamente, o status de uma ou de ambas as tarefas deverá ser informado como *Canceled*, o controle *TextBox duration* deverá exibir o texto "The operation was canceled" e o gráfico será exibido com lacunas. Se não foi rápido o suficiente, repita esse passo e tente novamente.



8. Retorne ao Visual Studio e interrompa a depuração.
9. No menu Debug, clique em Exceptions. Na caixa de diálogo Exceptions, marque a caixa de seleção User-unhandled do item Common Language Runtime Exceptions e, em seguida, clique em OK:

Como tratar exceções de tarefas com a classe AggregateException

Ao longo deste livro, você viu que o tratamento de exceções é um componente importante em qualquer aplicativo comercial. As construções de tratamento de exceção examinadas até agora são de fácil utilização e, se você as usar com cuidado, será uma simples questão de capturar uma exceção e determinar qual trecho do código a levantou. Entretanto, quando você começar a dividir o trabalho em várias tarefas simultâneas, o rastreamento e o tratamento das exceções se tornarão um problema mais complexo. O exercício anterior mostrou como capturar a exceção *OperationCanceledException* levantada quando uma tarefa é cancelada. No entanto, existem muitas outras exceções que também poderiam ocorrer, e diferentes tarefas podem gerar suas próprias exceções. Portanto, você precisa encontrar um jeito de capturar e tratar as diversas exceções lançadas de forma simultânea.

Se você estiver usando um dos métodos de espera de *Task* para aguardar o término de várias tarefas (utilizando o método de instância *Wait* ou os métodos estáticos *Task.WaitAll* e *Task.WaitAny*), as exceções levantadas pelos métodos que essas tarefas estão executando serão reunidas em uma única exceção, conhecida como *AggregateException*. Uma exceção *AggregateException* atua como um wrapper para uma coleção de exceções. Cada exceção da coleção pode ser lançada por diferentes tarefas. Em seu aplicativo, você pode capturar a exceção *AggregateException*, iterar sobre essa coleção e efetuar o processamento necessário. Para ajudá-lo, a classe *AggregateException* fornece o método *Handle*. O método *Handle* recebe um delegate *Func<Exception, bool>*, o qual referencia um método que recebe um objeto *Exception* como parâmetro e retorna um valor booleano. Quando você chama *Handle*, o método referenciado é executado para cada exceção existente na coleção do objeto *AggregateException*. O método referenciado pode examinar a exceção e tomar a ação adequada. Se o método referenciado tratar a exceção, ele deverá retornar *true*. Caso contrário, deverá retornar *false*. Quando o método *Handle* for concluído, todas as exceções não tratadas serão empacotadas em uma nova *AggregateException* e essa exceção será lançada. Uma rotina de tratamento de exceções externa subsequente poderá capturar essa exceção e processá-la.

O fragmento de código a seguir mostra um exemplo de método que pode ser registrado em uma rotina de tratamento de exceção para *AggregateException*. Esse método simplesmente exibirá a mensagem "Division by zero occurred", se detectar uma exceção *DivideByZeroException*, ou a mensagem "Array index out of bounds", se ocorrer uma exceção *IndexOutOfRangeException*. Outras exceções não serão tratadas.

```
private bool handleException(Exception e)
{
    if (e is DivideByZeroException)
    {
        displayErrorMessage("Division by zero occurred");
        return true;
    }

    if (e is IndexOutOfRangeException)
    {
        displayErrorMessage("Array index out of bounds");
        return true;
    }
    return false;
}
```

Ao utilizar um dos métodos de espera de *Task*, você pode capturar a exceção *AggregateException* e registrar o método *handleException*, como segue:

```
try
{
    Task first = Task.Run(...);
    Task second = Task.Run(...);
    Task.WaitAll(first, second);
}
catch (AggregateException ae)
{
    ae.Handle(handleException);
}
```

Se alguma das tarefas gerar uma exceção *DivideByZeroException* ou *IndexOutOfRangeException*, o método *handleException* exibirá uma mensagem adequada e reconhecerá a exceção como tratada. Qualquer outra exceção será classificada como não tratada e se propagará da rotina de tratamento da exceção *AggregateException* como é feito de costume.

Há mais uma complicação que você precisa conhecer. Quando uma tarefa é cancelada, vimos que o CLR lança uma exceção *OperationCanceledException*, e essa será a exceção informada, caso o operador *await* esteja sendo utilizado para aguardar pela tarefa. Contudo, se você estiver usando os métodos de espera de *Task*, essa exceção será transformada em uma exceção *TaskCanceledException*, e é esse o tipo de exceção que você deve estar preparado para tratar na rotina de tratamento de exceção para *AggregateException*.

Utilize continuações com tarefas canceladas e com falhas

Para fazer algum trabalho adicional quando uma tarefa é cancelada ou levanta uma exceção não tratada, lembre-se de que é possível utilizar o método *ContinueWith* com o valor adequado de *TaskContinuationOptions*. Por exemplo, o código a seguir cria uma tarefa que executa o método *doWork*. Se a tarefa for cancelada, o método *ContinueWith* especificará que outra tarefa deve ser criada e executará o método *doCancellationWork*. Esse método pode realizar algumas tarefas simples de registro em log ou de encerramento. Se a tarefa não for cancelada, a continuação não será executada.

```
Task task = new Task(doWork);
task.ContinueWith(doCancellationWork, TaskContinuationOptions.OnlyOnCanceled);
task.Start();
...
private void doWork()
{
    // A tarefa executa este código quando iniciada
    ...
}
...
private void doCancellationWork(Task task)
{
    // A tarefa executará esse código quando doWork terminar
    ...
}
```

De modo semelhante, você pode especificar o valor *TaskContinuationOptions.OnlyOnFaulted* para informar uma continuação que será executada se o método original executado pela tarefa levantar uma exceção não tratada.

Resumo

Neste capítulo, você aprendeu a importância de escrever aplicativos que suportam aumentos de escala usando diversos processadores e núcleos de processador. Vimos como utilizar a classe *Task* para executar operações em paralelo e como sincronizar operações simultâneas e aguardar o respectivo término. Você aprendeu a utilizar a classe *Parallel* para paralelizar algumas construções comuns de programação e também examinou quando é inadequado paralelizar o código. Você utilizou tarefas e threads juntos em uma interface gráfica de usuário a fim de melhorar a capacidade de resposta e o desempenho (throughput), e viu como é possível cancelar tarefas de modo controlado e transparente.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 24.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Criar uma tarefa e executá-la	<p>Utilize o método estático <i>Run</i> da classe <i>Task</i> para criar e executar a tarefa em um único passo:</p> <pre>Task task = Task.Run(() => doWork());</pre> <p>...</p> <pre>private void doWork() { // A tarefa executa este código quando iniciada ... }</pre> <p>Ou crie um novo objeto <i>Task</i> que faça referência a um método a executar e chame o método <i>Start</i>:</p> <pre>Task task = new Task(doWork); task.Start();</pre>
Aguardar o término de uma tarefa	<p>Chame o método <i>Wait</i> do objeto <i>Task</i>:</p> <pre>Task task = ...;</pre> <p>...</p> <pre>task.Wait();</pre> <p>Ou utilize o operador <i>await</i> (somente em um método <i>async</i>):</p> <pre>await task;</pre>
Aguardar o término de várias tarefas	<p>Chame o método estático <i>WaitAll</i> da classe <i>Task</i> e especifique as tarefas a serem aguardadas:</p> <pre>Task task1 = ...; Task task2 = ...; Task task3 = ...; Task task4 = ...; ... Task.WaitAll(task1, task2, task3, task4);</pre>
Especificando um método a executar em uma nova tarefa quando uma tarefa terminar	<p>Chame o método <i>ContinueWith</i> da tarefa e especifique-o como uma continuação:</p> <pre>Task task = new Task(doWork); task.ContinueWith(doMoreWork, TaskContinuationOptions.NotOnFaulted);</pre>
Executar iterações de loop e sequências de instruções utilizando tarefas paralelas	<p>Use os métodos <i>Parallel.For</i> e <i>Parallel.ForEach</i> para fazer as iterações do loop, por meio de tarefas:</p> <pre>Parallel.For(0, 100, performLoopProcessing); ... private void performLoopProcessing(int x) { // Processamento do loop }</pre> <p>Use o método <i>Parallel.Invoke</i> para fazer chamadas de métodos concorrentes, por meio de tarefas separadas:</p> <pre>Parallel.Invoke(doWork, doMoreWork, doYetMoreWork);</pre>

Para	Faça isto
Tratar as exceções levantadas por uma ou mais tarefas	<p>Capture a exceção <i>AggregateException</i>. Use o método <i>Handle</i> para especificar um método que possa tratar cada exceção do objeto <i>AggregateException</i>. Se o método de tratamento da exceção tratar a exceção, retorne <i>true</i>; caso contrário, retorne <i>false</i>:</p> <pre> try { Task task = Task.Run(...); task.Wait(); ... } catch (AggregateException ae) { ae.Handle(handleException); } ... private bool handleException(Exception e) { if (e is TaskCanceledException) { ... return true; } else { return false; } }</pre>
Permitir cancelamento em uma tarefa	<p>Implemente o cancelamento cooperativo, criando um objeto <i>CancellationTokenSource</i> e utilizando um parâmetro <i>CancellationToken</i> no método executado pela tarefa. No método da tarefa, chame o método <i>ThrowIfCancellationRequested</i> do parâmetro <i>CancellationToken</i> para lançar uma exceção <i>OperationCanceledException</i> e encerrar a tarefa:</p> <pre> private void generateGraphData(..., CancellationToken token) { ... token.ThrowIfCancellationRequested(); ... }</pre>

CAPÍTULO 24

Como melhorar o tempo de resposta empregando operações assíncronas

Neste capítulo, você vai aprender a:

- Definir e usar métodos assíncronos para melhorar o tempo de resposta dos aplicativos que executam operações prolongadas.
- Explicar como reduzir o tempo necessário para fazer consultas LINQ complexas utilizando paralelização.
- Utilizar as classes de coleção concorrentes para compartilhar dados entre tarefas paralelas com segurança.

O Capítulo 23, “Como melhorar o desempenho usando tarefas”, demonstra como utilizar a classe *Task* para efetuar operações em paralelo e melhorar o desempenho de aplicativos vinculados à computação (em inglês, *compute-bound*). Contudo, embora maximizar o poder de processamento disponível para um aplicativo possa fazê-lo executar mais rápido, a rapidez de resposta também é importante. Lembre-se de que a interface de usuário do Microsoft Windows funciona utilizando uma única thread de execução, mas os usuários esperam que um aplicativo responda quando clicam em um botão de um formulário, mesmo que o aplicativo esteja efetuando um cálculo grande e complexo. Além disso, algumas tarefas podem demorar um tempo considerável para executar, mesmo que não sejam vinculadas à computação (uma tarefa esperando para receber informações de um site pela rede, por exemplo), e impedir a interação do usuário enquanto esperam por um evento que poderia levar um tempo indeterminado para acontecer; isso claramente não é uma boa prática de projeto. A solução para esses dois problemas é a mesma: executar a tarefa de forma assíncrona e deixar a thread da interface livre para tratar das interações do usuário. No passado, essa estratégia era tradicionalmente cheia de complexidades e as estruturas de interface do usuário, como o Windows Presentation Foundation (WPF), tinham de implementar algumas outras soluções alternativas complicadas para suportar esse modo de trabalho. Felizmente, o Windows 8.1 e o Windows Runtime (WinRT) foram projetados com a assincronicidade em mente, e a linguagem C# foi estendida para tirar proveito dos recursos assíncronos agora fornecidos pelo Windows 8.1, tornando muito mais fácil definir operações assíncronas. Você vai conhecer esses recursos e saber como utilizá-los em conjunto com tarefas na primeira parte deste capítulo.

Os problemas com tempo de resposta não estão limitados às interfaces de usuário. Por exemplo, o Capítulo 21, “Consulta a dados na memória usando expressões de consulta”, mostra como é possível acessar os dados armazenados na memória de uma maneira declarativa, usando Language-Integrated Query (LINQ). Uma consulta LINQ comum gera um conjunto de resultados enumeráveis, e você pode iterar sequencial-

mente por esse conjunto para recuperar os dados. Se a origem dos dados utilizada para gerar o conjunto de resultados for grande, executar uma consulta LINQ pode exigir muito tempo. Muitos sistemas de gerenciamento de bancos de dados, ao enfrentarem a questão da otimização das consultas, solucionam o problema por meio de algoritmos que dividem o processo de identificação dos dados para uma consulta em uma série de tarefas, e depois executam essas tarefas em paralelo, combinando os resultados ao término das tarefas para gerar o conjunto de resultados completo. Os projetistas do Microsoft .NET Framework decidiram fornecer à LINQ um recurso semelhante, e o resultado foi a Parallel LINQ, ou PLINQ. Você estudará a PLINQ na segunda parte deste capítulo.

Contudo, nem sempre a PLINQ é a tecnologia mais adequada para utilizar em um aplicativo. Se você cria manualmente as próprias tarefas, certifique-se de que elas coordenem as respectivas atividades corretamente. A biblioteca de classes do .NET Framework dispõe de métodos com os quais é possível aguardar o término das tarefas, e você pode utilizá-los para coordenar tarefas em um nível muito pouco refinado. Mas examine o que acontece se duas tarefas tentam acessar e modificar os mesmos dados. Se ambas as tarefas forem executadas ao mesmo tempo, suas operações sobrepostas poderão danificar os dados. Essa situação pode gerar defeitos de difícil correção, basicamente devido à sua imprevisibilidade. A partir da versão 1.0, o .NET Framework disponibilizou primitivas para bloquear os dados e coordenar as threads, mas utilizá-las de modo eficiente exigia conhecer muito bem o modo de interação das threads. As versões mais recentes da biblioteca de classes do .NET Framework contêm algumas variações dessas primitivas, e ela fornece classes de coleção especializadas que podem sincronizar o acesso aos dados por meio das tarefas. Essas classes ocultam boa parte da complexidade da coordenação do acesso aos dados. Você verá como é possível utilizar as novas primitivas de sincronização e as classes de coleção na terceira parte deste capítulo.

Implemente métodos assíncronos

Um método *assíncrono* é aquele que não bloqueia a thread que está em execução. Quando um aplicativo chama um método assíncrono, existe um contrato implícito de que o método retornará o controle muito rapidamente para o ambiente que fez a chamada. A definição de *muito* não é uma quantidade matematicamente definida, mas espera-se que, se um método assíncrono efetua uma operação que pode causar um atraso perceptível para o chamador, ele deve fazer isso utilizando uma thread de segundo plano e tornar possível ao chamador continuar a executar na thread atual. Esse processo parece complicado e, na verdade, nas versões anteriores do .NET Framework, ele era. Mas agora o C# fornece o modificador de método *async* e o operador *await*, os quais delegam grande parte dessa complexidade para o compilador, significando que não é mais necessário você se preocupar com as complexidades ao lidar com multithreading.

Definição de métodos assíncronos: o problema

Você já viu como implementar operações simultâneas utilizando objetos *Task*. Recapitulando rapidamente, quando você inicia uma tarefa com o método *Start* ou *Run* do tipo *Task*, o Common Language Runtime (CLR) utiliza seu próprio algoritmo de agendamento para alocar a tarefa em uma thread e configura a execução dessa thread em um momento conveniente para o sistema operacional, quando recursos suficientes estiverem disponíveis. Esse nível de abstração isenta seu código da necessidade de en-

tender e gerenciar a carga de trabalho de seu computador. Caso seja necessário executar outra operação ao término de uma tarefa específica, você tem as seguintes escolhas:

- Pode esperar manualmente que a tarefa seja concluída, utilizando um dos métodos *Wait* expostos pelo tipo *Task*. Então, pode iniciar a nova operação, possivelmente definindo outra tarefa.
- Pode definir uma continuação. Uma continuação simplesmente especifica uma operação a ser efetuada ao término de determinada tarefa. O .NET Framework executa automaticamente a operação de continuação como uma tarefa, a qual agenda ao término da tarefa original.

Contudo, embora o tipo *Task* forneça uma generalização conveniente para uma operação, muitas vezes ainda é necessário escrever código potencialmente complicado para resolver alguns dos problemas comuns que, em geral, os desenvolvedores encontram ao compilar aplicativos que precisam executar essas operações em uma thread de segundo plano. Por exemplo, vamos supor que você defina o seguinte método para um aplicativo Windows 8.1, que envolve efetuar uma série de operações prolongadas a serem executadas de modo sequencial e, então, exibir uma mensagem em um controle *TextBox* na tela:

```
private void slowMethod()
{
    doFirstLongRunningOperation();
    doSecondLongRunningOperation();
    doThirdLongRunningOperation();
    message.Text = "Processing Completed";
}
private void doFirstLongRunningOperation()
{
    ...
}

private void doSecondLongRunningOperation()
{
    ...
}

private void doThirdLongRunningOperation()
{
    ...
}
```

Você pode tornar o método *slowMethod* mais ágil, utilizando um objeto *Task* para executar o método *doFirstLongRunningOperation*, e definir continuações para a mesma *Task* que executem os métodos *doSecondLongRunningOperation* e *doThirdLongRunningOperation* um após o outro, como segue:

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.Start();
```

```

        message.Text = "Processing Completed"; // Quando esta mensagem aparece?
    }

private void doFirstLongRunningOperation()
{
    ...
}

private void doSecondLongRunningOperation(Task t)
{
    ...
}

private void doThirdLongRunningOperation(Task t)
{
    ...
}

```

Embora essa refatoração pareça simples, existem pontos a serem observados. Especificamente, as assinaturas dos métodos *doSecondLongRunningOperation* e *doThirdLongRunningOperation* tiveram que ser alteradas para se adaptar aos requisitos das continuações (a *Task* é passada como parâmetro para um método de continuação). Mais importante, você precisa se perguntar “Quando a mensagem é exibida no controle *TextBox*”? O problema nesse segundo ponto é que, embora o método *Start* inicie uma *Task*, ele não espera que ela termine; portanto, a mensagem aparece enquanto o processamento está ocorrendo e não quando termina.

Esse é um exemplo um tanto trivial, mas o princípio geral é importante, e existem pelo menos duas soluções. A primeira é esperar pelo término da *Task* antes de exibir a mensagem, como a seguir:

```

private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.Start();
    task.Wait();
    message.Text = "Processing Completed";
}

```

Contudo, a chamada para o método *Wait* agora bloqueia a thread que executa o método *slowMethod* e anula o propósito de utilizar uma *Task*. Uma solução melhor é definir uma continuação que exiba a mensagem e providencie para que ela só seja executada quando o método *doThirdLongRunningOperation* terminar, no caso em que você pode remover a chamada para o método *Wait*. Você poderia ficar tentado a implementar essa continuação como um delegate, como mostrado em negrito no código a seguir (lembre-se de que uma continuação recebe um objeto *Task* como argumento; esse é o objetivo do parâmetro *t* para o delegate):

```

private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);

```

```

task.ContinueWith(doThirdLongRunningOperation);
task.ContinueWith((t) => message.Text = "Processing Complete");
task.Start();
}

```

Infelizmente, essa estratégia gera outro problema. Se você testar esse código, descobrirá que a última continuação gera uma exceção *System.Exception* com a mensagem bastante nebulosa "The application called an interface that was marshaled for a different thread" (O aplicativo chamou uma interface preparada para uma thread diferente). O problema é que somente a thread da interface do usuário pode manipular controles dessa interface, e agora você está tentando escrever em um controle *TextBox* a partir de uma thread diferente — a thread que está sendo utilizada para executar a *Task*. Para resolver esse problema, utilize o objeto *Dispatcher*. O objeto *Dispatcher* é um componente da infraestrutura da interface do usuário e você pode enviar a ele pedidos para realizar trabalho na thread da interface do usuário, chamando seu método *Invoke*. O método *Invoke* recebe um delegate *Action* que especifica o código a ser executado. Os detalhes do objeto *Dispatcher* e do método *Invoke* estão além dos objetivos deste livro, mas o exemplo de código a seguir mostra o que poderia ser usado para exibir a mensagem exigida pelo método *slowMethod* a partir de uma continuação:

```

private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.ContinueWith((t) => this.Dispatcher.Invoke(
        CoreDispatcherPriority.Normal,
        (sender, args) => messages.Text = "Processing Complete",
        this, null));
    task.Start();
}

```

Isso funciona, mas é complicado e difícil de manter. Agora você tem um delegado (a continuação) especificando outro delegate (o código a ser executado por *Invoke*).



Mais informações Mais informações sobre o objeto *Dispatcher* e sobre o método *Invoke* podem ser encontradas no site da Microsoft em <http://msdn.microsoft.com/en-us/library/ms615907.aspx>.

Definição de métodos assíncronos: a solução

Como você já pode ter adivinhado, o objetivo das palavras-chave *async* e *await* no C# é permitir a definição de métodos assíncronos sem se preocupar com a definição de continuações ou com agendamento de código para executar em objetos *Dispatcher* a fim de garantir que os dados sejam manipulados na thread correta. De forma muito simplificada, o modificador *async* indica que um método contém funcionalidade que pode ser executada de forma assíncrona. O operador *await* especifica os pontos em que essa funcionalidade assíncrona deve ser executada. O exemplo de código a seguir mostra o método *slowMethod* implementado como um método assíncrono, com o modificador *async* e os operadores *await*.

```
private async void slowMethod()
{
    await doFirstLongRunningOperation();
    await doSecondLongRunningOperation();
    await doThirdLongRunningOperation();
    messages.Text = "Processing Complete";
}
```

Agora esse método está muito parecido com a versão original, e esse é o poder de *async* e *await*. Na verdade, essa mágica nada mais é do que um exercício de reescrita de seu código pelo compilador do C#. Quando o compilador do C# encontra o operador *await* em um método *async*, ele efetivamente reformata o operando que vem após esse operador como uma tarefa que é executada na mesma thread do método *async*. O restante do código é convertido em uma continuação, executada ao término da tarefa – novamente, executando na mesma thread. Agora, como a thread que estava executando o método *async* era a que estava executando a interface do usuário, ela tem acesso direto aos controles da janela e, assim, pode atualizá-los diretamente, sem necessidade de um objeto *Dispatcher*.

Embora essa estratégia pareça simples à primeira vista, é importante ter em mente os seguintes pontos e evitar alguns possíveis conceitos errôneos:

- O modificador *async* não significa que um método é executado de forma assíncrona em uma thread separada. Ele apenas especifica que o código do método pode ser dividido em uma ou mais continuações. Quando essas continuações são executadas, isso acontece na mesma thread da chamada de método original.
- O operador *await* especifica o ponto no qual o compilador do C# pode dividir o código em uma continuação. O próprio operador *await* espera que seu operando seja um objeto *que possa esperar*. Um objeto *que pode esperar* é um tipo que fornece o método *GetAwaiter*, o qual retorna um objeto que, por sua vez, fornece métodos para executar código e esperar que ele termine. O compilador do C# converte seu código em instruções que utilizam esses métodos para criar uma continuação adequada.



Importante O operador *await* só pode ser utilizado em um método marcado como *async*. Fora de um método *async*, a palavra-chave *await* é tratada como um identificador normal (você pode até criar uma variável chamada *await*, embora não seja recomendado).

Além disso, você não pode usar o operador *await* nos blocos *catch* ou *finally* de uma construção *try/catch/finally* (nem mesmo em um método *async*) ou em uma expressão de consulta LINQ. Mas, se quiser fazer uma consulta LINQ utilizando várias tarefas simultâneas, você pode utilizar as extensões da PLINQ descritas mais adiante neste capítulo.

Na implementação atual do operador *await*, ele supõe que o objeto que você especificará será uma *Task*. Isso significa que algumas modificações devem ser feitas nos métodos *doFirstLongRunningOperation*, *doSecondLongRunningOperation* e *doThirdLongRunningOperation*. Especificamente, cada método deve agora criar e executar uma *Task* para fazer seu trabalho e retornar uma referência para essa *Task*. O

exemplo de código a seguir mostra uma versão corrigida do método *doFirstLongRunningOperation*:

```
private Task doFirstLongRunningOperation()
{
    Task t = Task.Run(() => { /* o código original desse método fica aqui */ });
    return t;
}
```

Também é importante considerar se existem oportunidades de dividir o trabalho feito pelo método *doFirstLongRunningOperation* em uma série de operações paralelas. Em caso positivo, você pode dividir o trabalho em um conjunto de *Tasks*, como descrito no Capítulo 23. Mas qual desses objetos *Task* você deve retornar como o resultado do método?

```
private Task doFirstLongRunningOperation()
{
    Task first = Task.Run(() => { /* código para a primeira operação */ });
    Task second = Task.Run(() => { /* código para a segunda operação */ });
    return ...; // Você retorna first ou second?
}
```

Se o método retornar *first*, o operador *await* de *slowMethod* só esperará que aquela *Task* termine e não a *Task* *second*. Uma lógica semelhante se aplica se o método retornar *second*. A solução é definir o método *doFirstLongRunningOperation* como *async* e esperar cada uma das *Tasks*, como mostrado aqui:

```
private async Task doFirstLongRunningOperation()
{
    Task first = Task.Run(() => { /* código para a primeira operação */ });
    Task second = Task.Run(() => { /* código para a segunda operação */ });
    await first;
    await second;
}
```

Lembre-se de que, quando o compilador encontra o operador *await*, ele gera código que espera a conclusão do item especificado pelo argumento, junto com uma continuação que executa a instrução seguinte. Você pode considerar o valor retornado pelo método *async* como uma referência para a *Task* que executa essa continuação (essa descrição não é totalmente precisa, mas é um modelo bom o bastante para os propósitos deste capítulo). Assim, o método *doFirstLongRunningOperation* cria e inicia as tarefas *first* e *second* executando em paralelo, o compilador reformata as instruções *await* no código que espera o término de *first*, seguido por uma continuação que espera a conclusão de *second*, e o modificador *async* faz com que o compilador retorne uma referência para essa continuação. Observe que, como agora o compilador determina o valor de retorno do método, você não especifica mais um valor de retorno (na verdade, se tentar retornar um valor, seu código não compilará).



Nota Se você não incluir uma instrução *await* em um método *async*, o método será simplesmente uma referência para uma *Task* que executa o código do corpo do método. Como resultado, ao chamar o método, ele não será executado de forma assíncrona. Nesse caso, o compilador o alertará com a mensagem “This *async* method lacks await operators and will run synchronously” (Esse método *async* não possui operadores *await* e será executado de forma síncrona).



Dica Você pode usar o modificador *async* como prefixo de um delegate. Isso torna possível criar delegates que incorporam processamento assíncrono utilizando o operador *await*.

No próximo exercício, você vai trabalhar com o aplicativo GraphDemo do Capítulo 23 e modificá-lo para gerar os dados do gráfico utilizando um método assíncrono.

Modifique o aplicativo GraphDemo para utilizar um método assíncrono

1. No Microsoft Visual Studio 2013, abra a solução GraphDemo, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 24\GraphDemo na sua pasta Documentos.
2. No Solution Explorer, expanda o nó GraphDemo.xaml e abra o arquivo GraphDemo.xaml.cs na janela Code and Text Editor.
3. Na classe *GraphWindow*, localize o método *plotButton_Click*.

O código desse método é parecido com o seguinte:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);

    tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;

    Stopwatch watch = Stopwatch.StartNew();

    try
    {
        generateGraphData(data, 0, pixelWidth / 2, token);
        duration.Text = string.Format("Duration (ms): {0}", watch.ElapsedMilliseconds);
    }

    catch (OperationCanceledException oce)
    {
        duration.Text = oce.Message;
    }

    Stream pixelStream = graphBitmap.PixelBufferAsStream();
    pixelStream.Seek(0, SeekOrigin.Begin);
```

```
pixelStream.Write(data, 0, data.Length);
graphBitmap.Invalidate();
graphImage.Source = graphBitmap;
}
```

Essa é uma versão simplificada do aplicativo do capítulo anterior. Ele chama o método *generateGraphData* diretamente da thread da interface do usuário e não utiliza objetos *Task* para gerar os dados do gráfico em paralelo.



Nota Se você reduziu o tamanho dos campos *pixelWidth* e *pixelHeight* nos exercícios do Capítulo 23 para economizar memória, faça isso novamente neste aplicativo, antes de passar para o próximo passo.

4. No menu Debug, clique em Start Debugging.
 5. Na janela GraphDemo, clique em Plot Graph. Enquanto os dados estão sendo gerados, experimente clicar em Cancel.
- Observe que a interface do usuário fica totalmente sem reação, enquanto o gráfico está sendo gerado e exibido. Isso acontece porque o método *plotButton_Click* faz todo seu trabalho de forma síncrona, inclusive a geração dos dados do gráfico.
6. Retorne ao Visual Studio e interrompa a depuração.
 7. Na janela Code and Text Editor que exibe a classe *GraphWindow*, acima do método *generateGraphData*, adicione um novo método privado, chamado *generateGraphDataAsync*.

Esse método deve receber a mesma lista de parâmetros do método *generateGraphData*, mas deve retornar um objeto *Task*, em vez de *void*. O método também deve ser marcado como *async* e ser semelhante a isto:

```
private async Task generateGraphDataAsync(byte[] data,
    int partitionStart, int partitionEnd,
    CancellationToken token)
{
}
```



Nota É uma prática recomendada colocar o sufixo *Async* no nome dos métodos assíncronos.

8. No método *generateGraphDataAsync*, adicione as instruções mostradas aqui em negrito.

```
private async Task generateGraphDataAsync(byte[] data, int partitionStart, int
partitionEnd, CancellationToken token)
{
    Task task = Task.Run(() => generateGraphData(data, partitionStart,
partitionEnd, token));
    await task;
}
```

Esse código cria um objeto *Task* que executa o método *generateGraphData* e utiliza o operador *await* para esperar o término da *Task*. A tarefa gerada pelo compilador como resultado do operador *await* é o valor retornado do método.

- 9.** Volte ao método *plotButton_Click* e altere sua definição para incluir o modificador *async*, como mostrado em negrito no código a seguir:

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
}
```

- 10.** No bloco *try* do método *plotButton_Click*, modifique a instrução que gera os dados do gráfico para chamar o método *generateGraphDataAsync* de forma assíncrona, como mostrado aqui em negrito:

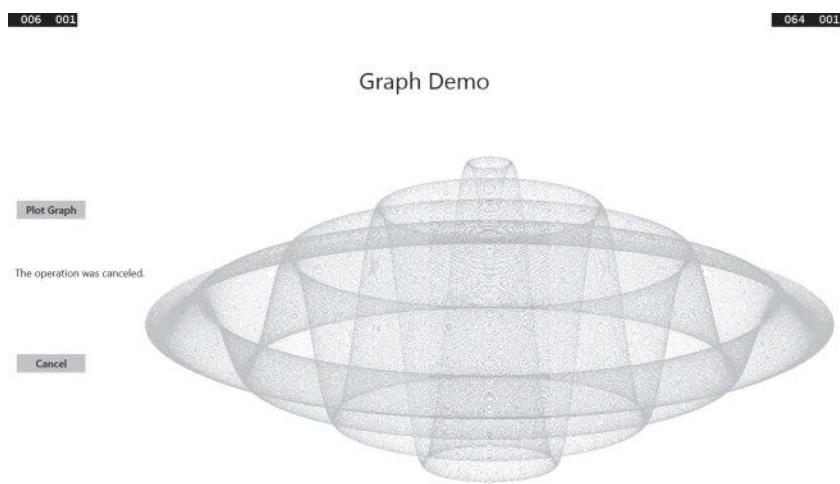
```
try
{
    await generateGraphDataAsync(data, 0, pixelWidth / 2, token);
    duration.Text = string.Format("Duration (ms): {0}", watch.ElapsedMilliseconds);
}
...
```

- 11.** No menu Debug, clique em Exceptions. Na caixa de diálogo Exceptions, expanda Common Language Runtime Exceptions, expanda System, desmarque a caixa de seleção User-unhandled da exceção *System.OperationCanceledException* e, em seguida, clique em OK.

Esse passo impede que o depurador intercepte a exceção *System.OperationCanceledException* enquanto continua a informar qualquer outra exceção que possa ocorrer.

- 12.** No menu Debug, clique em Start Debugging.
13. Na janela GraphDemo, clique em Plot Graph e verifique se o aplicativo gera o gráfico corretamente.
14. Clique em Plot Graph e, enquanto os dados estão sendo gerados, clique em Cancel.

Desta vez, a interface do usuário deve responder. Apenas parte do gráfico deve ser gerada e o *TextBlock duration* deve exibir a mensagem "The operation was cancelled".



15. Retorne ao Visual Studio e interrompa a depuração.

Defina métodos assíncronos que retornam valores

Até agora, todos os exemplos que examinamos utilizam um objeto *Task* para realizar um trabalho que não retorna um valor. Entretanto, você também pode utilizar tarefas para executar métodos que calculam um resultado. Para isso, utilize a classe genérica *Task<TResult>*, na qual o parâmetro de tipo, *TResult*, especifica o tipo do resultado.

Você cria e inicia um objeto *Task<TResult>* de modo parecido com a criação de uma *Task* normal. A principal diferença é que o código executado deve retornar um valor. Por exemplo, o método *calculateValue*, mostrado no exemplo de código a seguir, gera um resultado inteiro. Para chamar esse método por meio de uma tarefa, crie e execute um objeto *Task<int>*. Para obter o valor retornado pelo método, consulte a propriedade *Result* do objeto *Task<int>*. Se a tarefa não terminou de executar o método e o resultado ainda não estiver disponível, a propriedade *Result* bloqueará o chamar. Isso significa que não é preciso fazer qualquer sincronização e você sabe que, quando a propriedade *Result* retornar um valor, a tarefa terá finalizado seu trabalho.

```
Task<int> calculateValueTask = Task.Run(() => calculateValue(...));  
...  
int calculatedData = calculateValueTask.Result; // Bloqueia até calculateValueTask terminar  
...  
private int calculateValue(...)  
{  
    int someValue;  
    // Efetua cálculo e preenche someValue  
    ...  
    return someValue;  
}
```

O tipo genérico `Task<TResult>` também é a base do mecanismo para definir métodos assíncronos que retornam valores. Nos exemplos anteriores, vimos que os métodos `void` assíncronos são implementados pelo retorno de uma `Task`. Se um método assíncrono gera um resultado, ele deve retornar um `Task<TResult>`, como mostrado no exemplo a seguir, que cria uma versão assíncrona do método `calculateValue`:

```
private async Task<int> calculateValueAsync(...)  
{  
    // Chama calculateValue utilizando uma Task  
    Task<int> generateResultTask = Task.Run(() => calculateValue(...));  
    await generateResultTask;  
    return generateResultTask.Result;  
}
```

Esse método parece um pouco confuso, visto que o tipo de retorno é especificado como `Task<int>`, mas na verdade a instrução `return` retorna um `int`. Lembre-se de que, quando você define um método `async`, o compilador faz uma refatoração de seu código e, basicamente, retorna uma referência para `Task`, que executa a continuação da instrução `return generateResultTask.Result`. O tipo da expressão retornada por essa continuação é `int`; portanto, o tipo de retorno do método é `Task<int>`.

Para chamar um método assíncrono que retorna um valor, use o operador `await`, como segue:

```
int result = await calculateValueAsync(...);
```

O operador `await` extrai o valor da `Task` retornada pelo método `calculateValueAsync` e, neste caso, o atribui à variável `result`.

Métodos assíncronos e as APIs do Windows Runtime

Os projetistas do Windows 8 e do Windows 8.1 quiseram garantir que os aplicativos fossem os mais ágeis possível nas respostas, de modo que, ao implementar o WinRT, decidiram que qualquer operação que pudesse levar mais de 50 milissegundos para executar deveria estar disponível somente por meio de uma API assíncrona. Talvez você já tenha observado um ou dois casos dessa estratégia neste livro. Por exemplo, a fim de exibir uma mensagem para o usuário, você pode utilizar um objeto `MessageDialog`. Contudo, ao exibir essa mensagem, você deve usar o método `ShowAsync`, assim:

```
using Windows.UI.Popups;  
...  
MessageDialog dlg = new MessageDialog("Mensagem ao usuário");  
await dlg.ShowAsync();
```

O objeto `MessageDialog` exibe a mensagem e espera que o usuário clique no botão `Close` que aparece como parte dessa caixa de diálogo. Qualquer forma de interação com o usuário poderia demorar um tempo indeterminado (o usuário pode ter saído para almoçar, antes de clicar em `Close`), e frequentemente é importante não bloquear o aplicativo nem impedi-lo de executar outras operações (como responder a eventos) enquanto a caixa de diálogo está sendo exibida. A classe `MessageDialog` não fornece uma versão síncrona do método `ShowAsync`, mas se precisar exibir uma caixa de diálogo de forma síncrona, você pode simplesmente chamar `dlg.ShowAsync()` sem o operador `await`.

Outro exemplo comum de processamento assíncrono está relacionado à classe *FileOpenPicker*, a qual você utilizou no Capítulo 5, “Atribuição composta e instruções de iteração”. A classe *FileOpenPicker* exibe uma lista de arquivos e torna possível ao usuário fazer uma seleção nessa lista. Assim como na classe *MessageDialog*, ele poderia demorar um tempo considerável navegando e selecionando arquivos, de modo que essa operação não deve bloquear o aplicativo. O exemplo a seguir mostra como usar a classe *FileOpenPicker* para exibir os arquivos da pasta Documents do usuário e esperar enquanto ele seleciona um arquivo nessa lista.

```
using Windows.Storage;
using Windows.Storage.Pickers;
...
FileOpenPicker fp = new FileOpenPicker();
fp.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
fp.ViewMode = PickerViewMode.List;
fp.FileTypeFilter.Add("*");
StorageFile file = await fp.PickSingleFileAsync();
```

A principal instrução é a linha que chama o método *PickSingleFileAsync*. Esse é o método que exibe a lista de arquivos e permite que o usuário navegue pelo sistema de arquivos e selecione um arquivo (a classe *FileOpenPicker* também fornece o método *PickMultipleFilesAsync*, por meio do qual o usuário pode selecionar mais de um arquivo). O valor retornado por esse método é *Task<StorageFile>* e o operador *await* extrai o objeto *StorageFile* desse resultado. A classe *StorageFile* fornece uma abstração para um arquivo armazenado no disco rígido e, usando essa classe, você pode abrir um arquivo e ler ou gravar nele.



Nota Rigorosamente falando, o método *PickSingleFileAsync* retorna um objeto *IAsyncOperation<StorageFile>*. O WinRT utiliza sua própria abstração de operações assíncronas e mapeia os objetos *Task* do .NET Framework nessa abstração; a classe *Task* implementa a interface *IAsyncOperation*. Se estiver programando em C#, seu código não será afetado por essa transformação e você pode simplesmente utilizar objetos *Task* sem se preocupar com o modo como eles são mapeados nas operações assíncronas do WinRT.

A entrada/saída (E/S) de arquivos é outra fonte de operações potencialmente lentas, e a classe *StorageFile* implementa diversos métodos assíncronos por meio dos quais essas operações podem ser executadas sem afetar a rapidez de resposta de um aplicativo. Por exemplo, no Capítulo 5, depois que o usuário selecionava um arquivo utilizando um objeto *FileOpenPicker*, o código abria esse arquivo para leitura de forma assíncrona:

```
StorageFile file = await fp.PickSingleFileAsync();
...
var fileStream = await file.OpenAsync(FileAccessMode.Read);
```

Um último exemplo, diretamente aplicável aos exercícios vistos neste e no capítulo anterior, está relacionado à gravação em um fluxo. Talvez você tenha notado que, embora o tempo relatado para gerar os dados para o gráfico seja de alguns segundos,

pode demorar duas vezes mais para que o gráfico realmente apareça. Isso acontece por causa da maneira como os dados são gravados no bitmap. O bitmap processa os dados armazenados em um buffer como parte do objeto *WriteableBitmap*, e o método de extensão *AsStream* fornece uma interface *Stream* para esse buffer. Os dados são gravados no buffer por meio desse fluxo, utilizando o método *Write*, assim:

```
...
Stream pixelStream = graphBitmap.PixelBufferAsStream();
pixelStream.Seek(0, SeekOrigin.Begin);
pixelStream.Write(data, 0, data.Length);
...
```

A não ser que você tenha reduzido o valor dos campos *pixelWidth* e *pixelHeight* para economizar memória, o volume de dados gravados no buffer é de 366 MB ($12.000 * 8.000 * 4$ bytes), de modo que essa operação *Write* pode demorar alguns segundos. Para melhorar o tempo de resposta, execute essa operação de forma assíncrona, utilizando o método *WriteAsync*:

```
await pixelStream.WriteAsync(data, 0, data.Length);
```

Em geral, ao compilar aplicativos para o Windows 8.1, você deve procurar explorar a assincronicidade quando possível.

O padrão de projeto *IAsyncResult* nas versões anteriores do .Net Framework

Há tempos a assincronicidade é reconhecida como um elemento importante na compilação de aplicativos que respondem rapidamente com o .NET Framework, e o conceito é anterior à introdução da classe *Task* no .NET Framework versão 4.0. A Microsoft introduziu o padrão de projeto *IAsyncResult*, baseado no tipo de delegate *AsyncCallback*, para tratar dessas situações. Os detalhes exatos do funcionamento desse padrão não são apropriados neste livro, mas, do ponto de vista do programador, a implementação desse padrão significava que muitos tipos da biblioteca de classes do .NET Framework expunham operações prolongadas de duas maneiras: em uma forma síncrona, consistindo em um único método, e em uma forma assíncrona, utilizando dois métodos, chamados *BeginOperationName* e *EndOperationName*, onde *OperationName* especificava a operação sendo executada. Por exemplo, a classe *MemoryStream* no namespace *System.IO* fornece o método *Write* para escrever dados de forma síncrona em um fluxo na memória, mas também fornece os métodos *BeginWrite* e *EndWrite* para executar a mesma operação de forma assíncrona. O método *BeginWrite* inicia a operação de escrita, que é realizada em uma nova thread. Ele espera que o programador forneça uma referência para um método de retorno de chamada (call-back), o qual é executado quando a operação de escrita termina; essa referência é na forma de um delegate *AsyncCallback*. Nesse método, o programador deve implementar a limpeza apropriada e chamar o método *EndWrite* para dizer que a operação terminou. O exemplo de código a seguir mostra esse padrão em ação:

```
...
Byte[] buffer = ...; // preenchido com dados a escrever no MemoryStream
MemoryStream ms = new MemoryStream();
AsyncCallback callback = new AsyncCallback(handleWriteCompleted);
ms.BeginWrite(buffer, 0, buffer.Length, callback, ms);
...
private void handleWriteCompleted(IAsyncResult ar)
{
    MemoryStream ms = ar.AsyncState as MemoryStream;
    ... // Faz a limpeza apropriada
    ms.EndWrite(ar);
}
```

O parâmetro para o método de retorno de chamada (*handleWriteCompleted*) é um objeto *IAsyncResult* que contém informações sobre o status da operação assíncrona e qualquer outra informação de estado. Você pode passar informações definidas pelo usuário você pode passar para o retorno de chamada (call-back) nesse parâmetro; o último argumento fornecido para o método *BeginOperationName* é empacotado nesse parâmetro. Nesse exemplo, o retorno de chamada recebe uma referência para o *MemoryStream*.

Embora essa sequência funcione, ela é um paradigma complicado que torna obscura a operação que você está executando. O código da operação é dividido em dois métodos e é fácil perder o vínculo mental entre eles, se você tiver de manter esse código. Se estiver utilizando objetos *Task*, você pode simplificar esse modelo, chamando o método estático *FromAsync* da classe *TaskFactory*.

Esse método pega os métodos *BeginOperationName* e *EndOperationName* e os encapsula em um código que é executado por meio de uma *Task*. Não há necessidade de criar um delegate *AsyncCallback*, pois isso é gerado nos bastidores pelo método *FromAsync*. Assim, você pode executar a mesma operação mostrada no exemplo anterior como segue:

```
...
Byte[] buffer = ...;
MemoryStream s = new MemoryStream();
Task t = Task<int>.Factory.FromAsync(s.Beginwrite, s.EndWrite, buffer, 0,
                                         buffer.Length, null);
t.Start();
await t;
...
```

Essa técnica é útil se você precisa acessar funcionalidade assíncrona exposta pelos tipos desenvolvidos nas versões anteriores do .NET Framework.

Utilize a PLINQ para paralelizar o acesso declarativo a dados

O acesso a dados é outra área na qual o tempo de resposta é importante, especialmente se você estiver compilando aplicativos que precisam consultar estruturas de dados longas. Nos capítulos anteriores, conhecemos o poder da LINQ para recuperar

rar dados de uma estrutura de dados enumerável, mas os exemplos mostrados eram inherentemente single-threaded. A LINQ fornece o conjunto de extensões da PLINQ baseado em *Tasks*, que pode ajudá-lo a otimizar o desempenho e paralelizar algumas operações de consulta.

A PLINQ funciona dividindo um conjunto de dados em partições e utilizando tarefas para recuperar os dados que atendem aos critérios especificados pela consulta para cada partição, em paralelo. Quando as tarefas terminam, os resultados recuperados de cada partição são combinados em um único conjunto de resultados enumerável. A PLINQ é ideal nas situações que abrangem conjuntos de dados com uma grande quantidade de elementos, ou se os critérios especificados para a localização dos dados englobarem operações complexas e dispendiosas em termos de computação.

Um objetivo primordial da PLINQ é não ser invasiva o máximo possível. Se já tiver muitas consultas LINQ, você não desejará ter que modificar seu código para que elas possam ser executadas com a última versão do .NET Framework. Para isso, o .NET Framework contém o método de extensão *AsParallel*, que você pode utilizar com um objeto enumerável. O método *AsParallel* retorna um objeto *ParallelQuery* que age de modo semelhante ao objeto enumerável original, exceto pelo fato de disponibilizar implementações paralelas de vários operadores LINQ, como *join* e *where*. Essas implementações dos operadores LINQ se baseiam em tarefas e usam diversos algoritmos para executar partes de sua consulta LINQ em paralelo, sempre que possível.

Como tudo o que acontece no mundo da computação paralela, o método *AsParallel* não faz mágica. Não é possível garantir que seu código será otimizado; tudo depende da natureza de suas consultas LINQ e se as tarefas por elas executadas se prestam à paralelização. Para entender como funciona a PLINQ e conhecer as situações em que ela é útil, examinaremos alguns exemplos. Os exercícios nas seções a seguir demonstram duas situações simples.

Utilize a PLINQ para melhorar o desempenho ao iterar sobre uma coleção

O primeiro cenário é simples. Considere uma consulta LINQ que itera sobre uma coleção e recupera elementos da coleção com base em um cálculo que utiliza intensamente o processador. Esse tipo de consulta pode se beneficiar da execução paralela, desde que os cálculos sejam independentes. Os elementos da coleção podem ser divididos em diversas partições; o número exato depende da carga atual do computador e do número de CPUs disponíveis. Os elementos em cada partição podem ser processados por uma thread em separado. Quando todas as partições estiverem processadas, os resultados poderão ser combinados. Qualquer coleção com suporte para o acesso a elementos por meio de um índice, como um array ou uma coleção que implementa a interface *IList<T>*, pode ser gerenciada dessa maneira.

Paralelize uma consulta LINQ sobre uma coleção simples

1. No Visual Studio 2013, abra a solução PLINQ, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 24\PLINQ na sua pasta Documentos.

2. No Solution Explorer, clique duas vezes em Program.cs no projeto PLINQ para exibir o arquivo na janela Code and Text Editor.

Esse é um aplicativo de console. A estrutura básica do aplicativo já foi criada para você. A classe *Program* contém dois métodos, chamados *Test1* e *Test2*, que ilustram dois cenários comuns. O método *Main* chama cada um desses métodos de teste por vez.

Os dois métodos de teste têm a mesma estrutura geral: eles criam uma consulta LINQ (você vai adicionar código para fazer isso mais adiante neste conjunto de exercícios), executam-na e exibem o tempo despendido. O código de cada um desses métodos é praticamente todo separado das instruções que efetivamente criam e executam a consulta.

3. Examine o método *Test1*.

Esse método cria um grande array de inteiros e o preenche com um conjunto de números aleatórios, entre 0 e 200. O gerador de números aleatórios usa uma semente fixa, de modo que você deve obter os mesmos resultados sempre que executar o aplicativo.

4. Imediatamente após o primeiro comentário *TO DO* nesse método, adicione a consulta LINQ mostrada aqui em negrito:

```
// TO DO: criar uma consulta LINQ que recupere todos os números acima de 100
var over100 = from n in numbers
            where TestIfTrue(n > 100)
            select n;
```

Essa consulta LINQ recupera todos os itens do array *numbers* que tenham um valor acima de 100. Em si mesmo, o teste *n > 100* não é tão custoso a ponto de provar os benefícios da paralelização dessa consulta; por isso, o código chama o método *TestIfTrue*, que retarda um pouco ao executar uma operação *SpinWait*. O método *SpinWait* instrui o processador a executar continuamente um loop de instruções especiais “sem operações” durante um curto intervalo de tempo, mantendo o processador ocupado, mas sem realizar efetivamente trabalho algum. (Esse processo é conhecido como *spinning*.) O método *TestIfTrue* é parecido com este:

```
public static bool TestIfTrue(bool expr)
{
    Thread.SpinWait(1000);
    return expr;
}
```

5. Após o segundo comentário *TO DO* do método *Test1*, adicione o seguinte código, mostrado em negrito:

```
// TO DO: executar uma consulta LINQ e salvar os resultados em um objeto List<int>
List<int> numbersOver100 = new List<int>(over100);
```

Convém lembrar que as consultas LINQ usam execução adiada (deferred execution), de modo que não são executadas até que você recupere os respectivos resultados. Essa instrução cria um objeto *List<int>* e o preenche com os resultados da execução da consulta *over100*.

6. Após o terceiro comentário *TO DO* do método *Test1*, adicione a seguinte instrução, mostrada em negrito:

```
// TO DO: Exibir os resultados  
Console.WriteLine("There are {0} numbers over 100.", numbersOver100.Count);
```

7. No menu Debug, clique em Start Without Debugging. Observe o tempo necessário para executar o *Test 1* e o número de itens no array com valor acima de 100.
8. Execute o aplicativo várias vezes e tire uma média do tempo. Verifique que o número de itens maiores do que 100 é o mesmo a cada vez (o aplicativo utiliza a mesma semente numérica aleatória sempre que é executado, para garantir a possibilidade de nova execução dos testes). Quando terminar, retorne ao Visual Studio.
9. A lógica que seleciona cada item retornado pela consulta LINQ é independente da lógica de seleção de todos os outros itens; portanto, essa consulta é uma forte candidata ao particionamento. Modifique a instrução que define a consulta LINQ e especifique o método de extensão *AsParallel* para o array *numbers*, como mostrado aqui em negrito:

```
var over100 = from n in numbers.AsParallel()  
              where TestIfTrue(n > 100)  
              select n;
```



Nota Se a lógica de seleção ou os cálculos exigirem acesso a dados compartilhados, você deve sincronizar as tarefas que executam em paralelo; caso contrário, os resultados poderão ser imprevisíveis. Contudo, a sincronização pode impor uma sobrecarga e perder os benefícios da paralelização da consulta.

10. No menu Debug, clique em Start Without Debugging. Verifique que o número de itens informados por *Test 1* é idêntico ao anterior, mas que o tempo necessário para executar o teste diminui consideravelmente. Execute o teste várias vezes e tire uma média da sua duração.

Ao executar em um processador dual-core (ou em um computador com dois processadores), você constatará uma redução de 40 a 45% no tempo. Se o computador tiver mais núcleos de processamento, a redução poderá ser muito maior.

11. Feche o aplicativo e retorne ao Visual Studio.

O exercício anterior demonstra a melhoria de desempenho que é possível alcançar quando se faz uma pequena mudança em uma consulta LINQ. Entretanto, lembre-se de que você só verá resultados como esses se os cálculos efetuados pela consulta exigirem algum tempo. Eu trapaceei um pouco o processador usando spinning. Sem essa sobrecarga, a versão paralela da consulta é de fato mais lenta do que a versão serial. No próximo exercício, você verá uma consulta LINQ que une dois arrays na memória. Desta vez, o exercício utilizará volumes de dados mais reais, e não há necessidade de utilizar qualquer artifício para atrasar a consulta artificialmente.

Paralelize uma consulta LINQ que realiza *join* entre duas coleções

1. No Solution Explorer, abra o arquivo Data.cs na janela Code and Text Editor e localize a classe *CustomersInMemory*.

Essa classe contém um array de strings público, chamado *Customers*. Cada string no array *Customers* armazena os dados de um único cliente, e os campos são separados por vírgula; esse formato é típico dos dados que um aplicativo pode ler de um arquivo de texto que utiliza campos separados por vírgulas. O primeiro campo contém o ID do cliente, o segundo contém o nome da empresa que o cliente representa e os demais campos armazenam endereço, cidade, país e CEP.

2. Localize a classe *OrdersInMemory*.

Essa classe é parecida com a classe *CustomersInMemory*, mas contém um array de strings chamado *Orders*. O primeiro campo em cada string é o número do pedido, o segundo campo é o ID do cliente e o terceiro campo é a data do pedido.

3. Localize a classe *OrderInfo*. Essa classe contém quatro campos que armazenam o ID do cliente, o nome da empresa, o ID do pedido e a data do pedido. Você utilizará uma consulta LINQ para preencher uma coleção de objetos *OrderInfo* a partir dos dados contidos nos arrays *Customers* e *Orders*.
4. Exiba o arquivo Program.cs na janela Code and Text Editor e localize o método *Test2* da classe *Program*.

Nesse método, você criará uma consulta LINQ que une os arrays *Customers* e *Orders* por meio de um ID de cliente para retornar uma lista dos clientes e de todos os pedidos feitos por cada um deles. A consulta armazenará cada linha do resultado em um objeto *OrderInfo*.

5. No bloco *try* desse método, adicione o código mostrado em negrito a seguir, depois do primeiro comentário *TO DO*:

```
// TO DO: criar uma consulta LINQ que recupere clientes e pedidos a partir de arrays
// Armazenar cada linha retornada em um objeto OrderInfo
var orderInfoQuery = from c in CustomersInMemory.Customers
                      join o in OrdersInMemory.Orders
                      on c.Split(',')[0] equals o.Split(',')[1]
                      select new OrderInfo
{
    CustomerID = c.Split(',')[0],
    CompanyName = c.Split(',')[1],
    OrderID = Convert.ToInt32(o.Split(',')[0]),
    OrderDate = Convert.ToDateTime(o.Split(',')[2]),
    new CultureInfo("en-US"))
};
```

Essa instrução define a consulta LINQ. Observe que ela utiliza o método *Split* da classe *String* para dividir cada string em um array de strings. As strings são separadas pelo caractere de vírgula. (As vírgulas são retiradas.) Um fator complicador é que as datas no array são armazenadas no formato do inglês americano, de

modo que o código que as converte em objetos *DateTime* no objeto *OrderInfo* especifica o formatador do inglês americano. Se você utiliza o formatador padrão de sua localidade, é possível que as datas não sejam processadas corretamente. Em suma, essa consulta faz um volume significativo de trabalho para gerar os dados de cada item.

- 6.** No método *Test2*, adicione o seguinte código mostrado em negrito após a segunda instrução *TO DO*:

```
// TO DO: executar a consulta LINQ e salvar os resultados em um objeto List<OrderInfo>
List<OrderInfo> orderInfo = new List<OrderInfo>(orderInfoQuery);
```

Essa instrução executa a consulta e preenche a coleção *orderInfo*.

- 7.** Após a terceira instrução *TO DO*, adicione a instrução mostrada aqui em negrito:

```
// TO DO: exibir os resultados
Console.WriteLine("There are {0} orders", orderInfo.Count);
```

- 8.** No método *Main*, transforme em comentário a instrução que chama o método *Test1* e exclua as barras de comentário da instrução que chama o método *Test2*, como mostrado em negrito no código a seguir:

```
static void Main(string[] args)
{
    // Test1();
    Test2();
}
```

- 9.** No menu Debug, clique em Start Without Debugging.
- 10.** Verifique que *Test2* recupera 830 pedidos e observe a duração do teste. Execute o aplicativo várias vezes para obter uma duração média e retorne ao Visual Studio.
- 11.** No método *Test2*, modifique a consulta LINQ e adicione o método de extensão *AsParallel* aos arrays *Customers* e *Orders*, como mostrado em negrito a seguir:

```
var orderInfoQuery = from c in CustomersInMemory.Customers.AsParallel()
                      join o in OrdersInMemory.Orders.AsParallel()
                      on c.Split(',')[0] equals o.Split(',')[1]
                      select new OrderInfo
{
    CustomerID = c.Split(',')[0],
    CompanyName = c.Split(',')[1],
    OrderID = Convert.ToInt32(o.Split(',')[0]),
    OrderDate = Convert.ToDateTime(o.Split(',')[2],
        New CultureInfo("en-US"))
};
```



Aviso Quando você une duas origens de dados usando `join`, ambas devem ser objetos `IEnumerable` ou objetos `ParallelQuery`. Isso significa que, se você especificar o método `AsParallel` para uma das origens, deverá também especificá-lo para a outra. Caso contrário, seu código não será executado – ele parará com um erro.

12. Execute o aplicativo várias vezes. Observe que o tempo necessário para `Test2` deve ser muito mais curto do que antes. A PLINQ pode utilizar várias threads para otimizar operações de união, buscando simultaneamente os dados de cada parte da união.
13. Feche o aplicativo e retorne ao Visual Studio.

Esses dois exercícios simples demonstraram o poder do método de extensão `AsParallel` e da PLINQ. Observe que a PLINQ é uma tecnologia em evolução, sendo muito provável que a implementação interna mude no decorrer do tempo. Além disso, os volumes de dados e a quantidade de processamento realizada em uma consulta também influenciam na eficiência do uso da PLINQ. Portanto, não considere esses exercícios definidores de regras fixas que devem ser sempre seguidas. Em vez disso, eles ilustram o aspecto que você deve avaliar criteriosamente e analisar o provável desempenho e outros benefícios do uso da PLINQ com seus próprios dados em seu ambiente.

Cancele uma consulta PLINQ

Diferentemente das consultas LINQ comuns, uma consulta PLINQ pode ser cancelada. Para isso, especifique um objeto `CancellationToken` a partir de uma `CancellationTokenSource` e use o método de extensão `WithCancellation` de `ParallelQuery`.

```
CancellationToken tok = ...;
...
var orderInfoQuery =
    from c in CustomersInMemory.Customers.AsParallel().WithCancellation(tok)
    join o in OrdersInMemory.Orders.AsParallel()
    on ...
```

Especifique `WithCancellation` apenas uma vez em uma consulta. O cancelamento se aplica a todas as origens em uma consulta. Se o objeto `CancellationTokenSource` utilizado para gerar o `CancellationToken` for cancelado, a consulta será paralisada com uma exceção `OperationCanceledException`.

Sincronize acessos simultâneos a dados

A classe *Task* dispõe de uma estrutura poderosa, com a qual é possível elaborar e construir aplicativos que tiram proveito de vários núcleos da CPU para executar tarefas simultaneamente. Entretanto, como mencionado na introdução deste capítulo, você deve ter cuidado ao construir soluções que executam operações simultâneas, em especial se essas operações compartilharem o acesso aos mesmos dados.

A questão é que você tem pouco controle sobre o modo como as operações paralelas são agendadas, ou até mesmo sobre o grau de paralelismo que o sistema operacional pode oferecer a um aplicativo construído por meio de tarefas. Essas decisões ficam por conta das considerações do runtime e dependem da carga de trabalho e dos recursos do hardware do computador que executa seu aplicativo. Esse nível de abstração foi uma decisão deliberada de projeto por parte da equipe de desenvolvimento da Microsoft e evita a necessidade de que você conheça os detalhes de baixo nível da implementação de threads e do agendamento (scheduling), ao construir aplicativos que exigem tarefas simultâneas. Mas essa abstração tem o seu preço. Embora tudo pareça funcionar como em um passe de mágica, você deve se esforçar para saber como seu código é executado; caso contrário, poderá terminar com aplicativos que apresentam um comportamento imprevisível (e incorreto), como demonstra o exemplo a seguir:

```
using System;
using System.Threading;

class Program
{
    private const int NUMELEMENTS = 10;

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

    static void SerialTest()
    {
        int[] data = new int[NUMELEMENTS];
        int j = 0;

        for (int i = 0; i < NUMELEMENTS; i++)
        {
            j = i;
            doAdditionalProcessing();
            data[i] = j;
            doMoreAdditionalProcessing();
        }

        for (int i = 0; i < NUMELEMENTS; i++)
        {
            Console.WriteLine("Element {0} has value {1}", i, data[i]);
        }
    }

    static void doAdditionalProcessing()
    {
        Thread.Sleep(10);
    }

    static void doMoreAdditionalProcessing()
```

```

    {
        Thread.Sleep(10);
    }
}

```

O método *SerialTest* preenche um array de inteiros com um conjunto de valores (de modo muito prolixo) e depois itera sobre essa lista, imprimindo o índice de cada item do array juntamente com o valor do item correspondente. Os métodos *doAdditionalProcessing* e *doMoreAdditionalProcessing* apenas simulam a execução de operações demoradas, como parte do processamento que levaria o runtime a tomar o controle do processador. A saída do método do programa está mostrada a seguir:

```

Element 0 has value 0
Element 1 has value 1
Element 2 has value 2
Element 3 has value 3
Element 4 has value 4
Element 5 has value 5
Element 6 has value 6
Element 7 has value 7
Element 8 has value 8
Element 9 has value 9

```

Examine agora o método *ParallelTest*, mostrado a seguir. Esse método é o mesmo de *SerialTest*, mas utiliza a construção *Parallel.For* para preencher o array *data* ao executar tarefas simultâneas. O código na expressão lambda executado por cada tarefa é idêntico àquele do loop *for* inicial no método *SerialTest*.

```

using System.Threading.Tasks;
...

static void ParallelTest()
{
    int[] data = new int[NUMELEMENTS];
    int j = 0;

    Parallel.For(0, NUMELEMENTS, (i) =>
    {
        j = i;
        doAdditionalProcessing();
        data[i] = j;
        doMoreAdditionalProcessing();
    });

    for (int i = 0; i < NUMELEMENTS; i++)
    {
        Console.WriteLine("Element {0} has value {1}", i, data[i]);
    }
}

```

O objetivo do método *ParallelTest* é executar a mesma operação que o método *SerialTest*, exceto pelo fato de utilizar tarefas simultâneas e (com sorte) como consequência, é executado um pouco mais rapidamente. O problema é que nem sempre ele funciona como esperado. Veja a seguir um exemplo de saída gerada pelo método *ParallelTest*:

```

Element 0 has value 1
Element 1 has value 1
Element 2 has value 4
Element 3 has value 8
Element 4 has value 4
Element 5 has value 1
Element 6 has value 4
Element 7 has value 8
Element 8 has value 8
Element 9 has value 9

```

Nem sempre os valores atribuídos a cada item do array *data* são os mesmos gerados no método *SerialTest*. Além disso, outras execuções do método *ParallelTest* podem gerar diferentes conjuntos de resultados.

Ao examinar a lógica na construção *ParallelFor*, você logo detectará onde está o problema. A expressão lambda contém as seguintes instruções:

```

j = i;
doAdditionalProcessing();
data[i] = j;
doMoreAdditionalProcessing();

```

O código parece bastante inofensivo. Ele copia o valor atual da variável *i* (a variável de índice que identifica qual iteração do loop está em execução) para a variável *j* e, mais adiante, armazena o valor de *j* no elemento do array de dados indexado por *i*. Se *i* contém 5, então *j* recebe o valor 5 e, depois, o valor de *j* é armazenado em *data[5]*. O problema é que entre atribuir o valor a *j* e depois lê-lo de volta, o código continua trabalhando; ele chama o método *doAdditionalProcessing*. Se a execução desse método for demorada, o runtime poderá suspender a thread e agendar outra tarefa. Uma tarefa simultânea, que execute outra iteração da construção *ParallelFor*, pode executar e atribuir um novo valor a *j*. Consequentemente, quando a tarefa original for retomada, o valor de *j* que ela atribui a *data[5]* não será o valor que ela armazenou, e os dados serão corrompidos. O mais complicado é que, ocasionalmente, esse código pode funcionar como previsto e produzir os resultados corretos, e em outras ocasiões, pode não funcionar; tudo depende de quanto o computador se encontra ocupado e quando as diversas tarefas são agendadas. Assim, esses tipos de defeitos podem passar despercebidos durante o teste e, de repente, se manifestar em um ambiente de produção.

A variável *j* é compartilhada por todas as tarefas simultâneas. Se uma tarefa armazenar um valor em *j* e, mais adiante, lê-lo novamente, ela deverá se certificar de que nenhuma outra tarefa tenha modificado *j* nesse ínterim. Isso exige a sincronização do acesso à variável em todas as tarefas simultâneas que podem acessá-la. Uma maneira de sincronizar o acesso é bloquear os dados.

Bloqueie dados

A linguagem C# fornece semântica de bloqueio por meio da palavra-chave *lock*, que pode ser utilizada para garantir o acesso exclusivo a recursos. Use a palavra-chave *lock* como a seguir:

```
object myLockObject = new object();
...
lock (myLockObject)
{
    // Código que exige acesso exclusivo a um recurso compartilhado
    ...
}
```

A instrução *lock* tenta obter um bloqueio de exclusão mútua sobre o objeto especificado (na realidade, você pode utilizar qualquer tipo-referência, não apenas um *object*) e o bloqueia se esse mesmo objeto estiver atualmente bloqueado por outra thread. Quando a thread obtiver o bloqueio, o código no bloco posterior à instrução *lock* será executado. No final desse bloco, o bloqueio será liberado. Se outra thread estiver bloqueada, aguardando pelo desbloqueio, ela poderá obter o bloqueio e continuar seu processamento.

Primitivas de sincronização para coordenar tarefas

A palavra-chave *lock* é adequada a muitos cenários simples, mas, em algumas situações, você pode enfrentar exigências mais complexas. O namespace *System.Threading* dispõe de algumas primitivas de sincronização adicionais para tratar dessas situações. Essas primitivas de sincronização são classes projetadas para uso com tarefas; elas expõem mecanismos de bloqueio que restringem o acesso a um recurso enquanto uma tarefa mantiver o bloqueio. Elas têm suporte para diversas técnicas de bloqueio, que você pode utilizar para implementar diferentes estilos de acesso simultâneo, desde bloqueios exclusivos simples (onde uma única tarefa tem acesso exclusivo a um recurso) e semáforos (onde várias tarefas podem acessar simultaneamente um recurso, mas de modo controlado), até bloqueios do tipo leitor/escritor (reader/writer), que permitem a diferentes tarefas compartilhar o acesso somente leitura a um recurso, além de garantir o acesso exclusivo a uma thread que precise modificar o recurso.

A lista a seguir resume algumas dessas primitivas. Para informações e exemplos adicionais, consulte a documentação fornecida com o Visual Studio 2013.



Nota Desde a sua primeira versão, o .NET Framework tem mantido um conjunto respeitável de primitivas de sincronização. A lista a seguir descreve apenas as primitivas mais recentes incluídas no namespace *System.Threading*. Ocorre certa sobreposição entre as novas primitivas e aquelas já fornecidas. Quando houver sobreposição de funcionalidades, use as alternativas mais recentes, pois foram elaboradas e otimizadas para os computadores equipados com várias CPUs.

Uma discussão detalhada sobre a teoria de todos os possíveis mecanismos de sincronização disponíveis para construir aplicativos multitarefas está além dos objetivos deste livro. Para obter mais informações sobre a teoria geral de múltiplas threads e sincronização, consulte o tópico “Synchronizing Data for Multithreading” (Sincronizando Dados para Multithreading) na documentação fornecida com o Visual Studio 2013.

- **ManualResetEventSlim** A classe *ManualResetEventSlim* oferece uma função por meio da qual uma ou mais tarefas aguardam um evento.

Um objeto *ManualResetEventSlim* pode estar em um de dois estados: *signaled* (true) e *unsignaled* (false). Uma tarefa cria um objeto *ManualResetEventSlim* e especifica seu estado inicial. Outras tarefas podem esperar que o objeto *ManualResetEventSlim* seja sinalizado, ao chamar o método *Wait*. Se o objeto *ManualResetEventSlim* estiver no estado *unsignaled*, o método *Wait* bloqueará as tarefas. Outra tarefa pode mudar o estado do objeto *ManualResetEventSlim* para *signaled* ao chamar o método *Set*. Essa ação libera todas as tarefas que estão aguardando no objeto *ManualResetEventSlim*, o qual pode, então, continuar a sua execução. O método *Reset* altera o estado de um objeto *ManualResetEventSlim* novamente para *unsignaled*.

- **SemaphoreSlim** A classe *SemaphoreSlim* pode ser utilizada para controlar o acesso a um pool de recursos.

Um objeto *SemaphoreSlim* tem um valor inicial (um inteiro não negativo) e um valor máximo opcional. Geralmente, o valor inicial de um objeto *SemaphoreSlim* é o número de recursos existentes no pool. As tarefas que acessam os recursos no pool chamam primeiro o método *Wait*. Esse método tenta decrementar o valor do objeto *SemaphoreSlim* e, se o resultado for diferente de zero, a thread poderá prosseguir e utilizar um recurso do pool. Ao terminar, a tarefa deve chamar o método *Release* no objeto *SemaphoreSlim*. Essa ação incrementa o valor do *Semaphore*.

Se uma tarefa chamar o método *Wait* e o resultado da decrementação do valor do objeto *SemaphoreSlim* der um valor negativo, a tarefa aguardará até outra tarefa chamar o método *Release*.

A classe *SemaphoreSlim* também fornece a propriedade *CurrentCount*, a qual pode ser utilizada para determinar a probabilidade de uma operação *Wait* ter êxito imediatamente ou se resultará em bloqueio.

- **CountdownEvent** Considere a classe *CountdownEvent* como um cruzamento entre o oposto de um semáforo e um evento de redefinição (reset) manual.

Ao criar um objeto *CountdownEvent*, uma tarefa especifica um valor de estado inicial (um inteiro não negativo). Uma ou mais tarefas podem chamar o método *Wait* do objeto *CountdownEvent* e, se seu valor for diferente de zero, as tarefas serão bloqueadas. O método *Wait* não decrementa o valor do objeto *CountdownEvent*; em vez disso, outras tarefas podem chamar o método *Signal* para reduzir o valor. Quando o valor do objeto *CountdownEvent* atinge zero, todas as tarefas bloqueadas são sinalizadas e podem retomar a sua execução.

Uma tarefa pode redefinir o valor de um objeto *CountdownEvent* com o valor especificado em seu construtor por meio do método *Reset* e pode aumentar o valor ao chamar o método *AddCount*. Para saber se uma chamada ao método *Wait* provavelmente será bloqueada, examine a propriedade *CurrentCount*.

- **ReaderWriterLockSlim** A classe *ReaderWriterLockSlim* é uma primitiva de sincronização avançada, com suporte para um único escritor e vários leitores. A ideia é a de que, para modificar (escrever) um recurso, é necessário ter acesso exclusivo, mas não para ler um recurso; vários leitores podem acessar o mesmo recurso ao mesmo tempo, mas não simultaneamente com um escritor.

Para ler um recurso, uma tarefa chama o método *EnterReadLock* de um objeto *ReaderWriterLockSlim*. Essa ação captura um bloqueio de leitura no objeto.

Quando a tarefa termina de usar o recurso, ela chama o método *ExitReadLock*, o qual libera o bloqueio de leitura. Várias tarefas podem ler o mesmo recurso ao mesmo tempo, e cada tarefa obtém um bloqueio de leitura próprio.

Para modificar o recurso, uma tarefa chama o método *EnterWriteLock* do mesmo objeto *ReaderWriterLockSlim*, a fim de obter um bloqueio de escrita. Se uma ou mais tarefas tiverem atualmente um bloqueio de leitura para esse objeto, o método *EnterWriteLock* bloqueará até que todos estejam liberados. Depois que uma tarefa tem um bloqueio de escrita, pode então modificar o recurso e chamar o método *ExitWriteLock* para liberar o bloqueio.

Um objeto *ReaderWriterLockSlim* tem apenas um bloqueio de escrita. Se outra tarefa tentar obter o bloqueio de escrita, será bloqueada até que a primeira tarefa libere esse bloqueio.

Para garantir que as tarefas de escrita não fiquem bloqueadas indefinidamente, assim que uma tarefa solicitar o bloqueio de escrita, todas as chamadas subsequentes a *EnterReadLock* feitas por outras tarefas serão bloqueadas, até que o bloqueio de escrita tenha sido obtido e liberado.

- **Barrier** Com a classe *Barrier*, você pode interromper temporariamente a execução de um conjunto de tarefas em um ponto específico em um aplicativo, e só prosseguir quando todas as tarefas tiverem alcançado esse ponto. Essa classe é útil para sincronizar as tarefas que precisam executar uma série de operações simultâneas em etapas com dependências entre si.

Ao criar um objeto *Barrier*, uma tarefa especifica o número de tarefas no conjunto que será sincronizado. Considere esse valor um contador de tarefas mantido internamente dentro da classe *Barrier*. Esse valor pode ser corrigido depois, chamando-se o método *AddParticipant* ou *RemoveParticipant*. Ao atingir um ponto de sincronização, uma tarefa chama o método *SignalAndWait* do objeto *Barrier*, o qual decrementa o contador de tarefas dentro do objeto *Barrier*. Se esse contador for maior que zero, a tarefa será bloqueada. Somente quando o contador atingir zero todas as tarefas aguardando no objeto *Barrier* serão liberadas e só então poderão retomar a sua execução.

A classe *Barrier* contém a propriedade *ParticipantCount*, que especifica o número de tarefas que sincronizará, e a propriedade *ParticipantsRemaining*, que indica quantas tarefas devem chamar o método *SignalAndWait* para que a barreira seja erguida e as tarefas bloqueadas possam continuar operando.

Você também pode especificar um delegate no construtor de *Barrier*. Esse delegate pode fazer referência a um método a ser executado quando todas as tarefas alcançarem a barreira. O objeto *Barrier* é passado com parâmetro para esse método. A barreira não é levantada e as tarefas não são liberadas até o término desse método.

Cancela a sincronização

As classes *ManualResetEventSlim*, *SemaphoreSlim*, *CountdownEvent* e *Barrier* têm suporte ao cancelamento, seguindo o modelo de cancelamento descrito no Capítulo 23. As operações de espera de cada uma dessas classes podem aceitar um parâmetro opcional *CancellationToken*, recuperado de um objeto *CancellationTokenSource*. Se você

chamar o método *Cancel* do objeto *CancellationTokenSource*, cada operação de espera que estiver referenciando um *CancellationToken* gerado nessa origem será abortada com uma exceção *OperationCanceledException* (possivelmente encapsulada em uma exceção *AggregateException*, dependendo do contexto da operação de espera).

O código a seguir mostra como chamar o método *Wait* de um objeto *SemaphoreSlim* e especificar um token de cancelamento. Se a operação de espera for cancelada, a rotina de tratamento *catch OperationCanceledException* será executada.

```
CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
CancellationToken cancellationToken = cancellationTokenSource.Token;
...
// Semáforo que protege um pool de 3 recursos
SemaphoreSlim semaphoreSlim = new SemaphoreSlim(3);
...
// Aguarda o semáforo e captura a OperationCanceledException se
// outra thread chamar Cancel em cancellationTokenSource
try
{
    semaphoreSlim.Wait(cancellationToken);
}
catch (OperationCanceledException e)
{
    ...
}
```

Classes de coleção concorrentes

Um requisito comum de muitos aplicativos multithreaded é armazenar e recuperar dados em uma coleção. Por padrão, as classes de coleção padrão fornecidas no .NET Framework não são thread-safe (seguras para threads), embora você possa utilizar as primitivas de sincronização descritas na seção anterior para encapsular código que adicione, consulte e remova elementos em uma coleção. Entretanto, esse processo é propenso a erros e não suporta escalabilidade muito bem, de modo que a biblioteca de classes do .NET Framework contém um pequeno conjunto de classes de coleção e interfaces thread-safe no namespace *System.Collections.Concurrent*, projetadas especificamente para serem utilizadas com tarefas. A lista a seguir resume brevemente os principais tipos desse namespace:

- ***ConcurrentBag*<T>** Classe genérica para armazenar uma coleção não ordenada de itens. Contém métodos para inserir (*Add*), remover (*TryTake*) e examinar (*TryPeek*) itens na coleção. Esses métodos são thread-safe. A coleção também é enumerável, de modo que é possível iterar sobre seu conteúdo por meio de uma instrução *foreach*.
- ***ConcurrentDictionary*<TKey, TValue>** Essa classe implementa uma versão thread-safe da classe de coleção genérica *Dictionary*<*TKey*, *TValue*>, descrita no Capítulo 18, "Coleções". Ela fornece os métodos *TryAdd*, *ContainsKey*, *TryGetValue*, *TryRemove* e *TryUpdate*, que você pode utilizar para adicionar, consultar, remover e modificar itens no dicionário.
- ***ConcurrentQueue*<T>** Essa classe apresenta uma versão thread-safe da classe genérica *Queue*<*T*>, descrita no Capítulo 18. Ela fornece os métodos *Enqueue*, *TryDequeue* e *TryPeek*, que você pode utilizar para adicionar, remover e consultar itens na fila.

- **ConcurrentStack<T>** Implementação thread-safe da classe genérica *Stack<T>*, também descrita no Capítulo 18. Ela fornece métodos, como *Push*, *TryPop* e *TryPeek*, que você pode utilizar para empilhar, desempilhar e consultar itens na pilha.

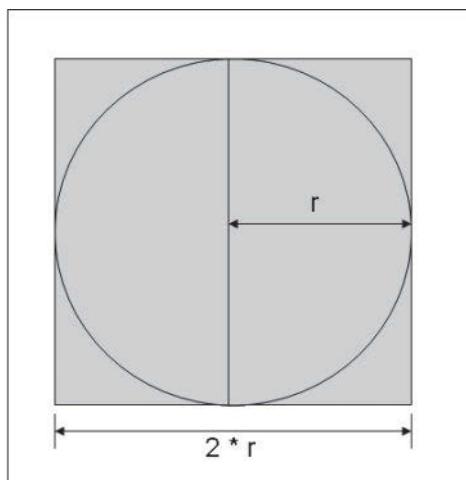


Nota Adicionar proteção quanto a threads aos métodos em uma classe de coleção impõe uma sobrecarga adicional de tempo de execução, de modo que essas classes não são tão rápidas quanto as classes de coleção comuns. Lembre-se desse aspecto ao optar por paralelizar um conjunto de operações que exigem acesso a uma coleção compartilhada.

Utilize uma coleção concorrente e um bloqueio para implementar acesso a dados thread-safe

No conjunto de exercícios a seguir, você vai implantar um aplicativo que calcula pi utilizando uma aproximação geométrica. Primeiro, você vai efetuar o cálculo ao estilo single-threaded e, depois, vai mudar o código para efetuar o cálculo por meio de tarefas simultâneas. Ao longo do processo, você vai detectar alguns problemas de sincronização de dados que podem ser solucionados através de uma classe de coleção concorrente e de um bloqueio para garantir que as tarefas coordenem suas atividades corretamente.

O algoritmo que você implementará calcula pi com base em matemática simples e amostragem estatística. Se você desenhar um círculo de raio r e um quadrado cujos lados tocam no círculo, os lados do quadrado terão $2 * r$ de comprimento, como mostrado na imagem a seguir.



A área do quadrado, Q , pode ser calculada desta maneira:

$$Q = (2 * r) * (2 * r)$$

ou

$$Q = 4 * r * r$$

A área do círculo, C , é calculada assim:

$$C = \pi * r * r$$

Rearranjando essas fórmulas, pode-se ver que

$$r * r = C / \pi$$

e

$$r * r = Q / 4$$

Portanto,

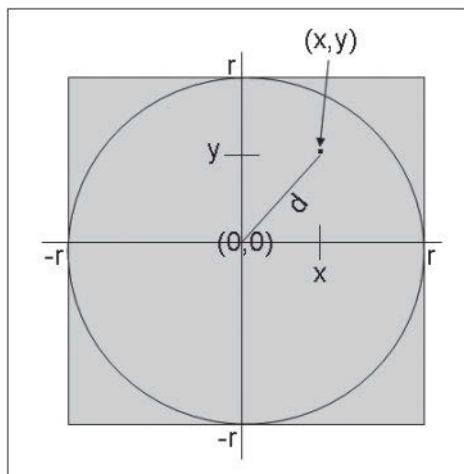
$$Q / 4 = C / \pi$$

e rearranjando essa fórmula para calcular pi, você obtém isto:

$$\pi = 4 * C / Q$$

O segredo é determinar o valor da relação da área do círculo, C , com respeito à área do quadrado, Q . É aí que entra em ação a amostragem estatística. Você pode gerar um conjunto de pontos aleatórios assentados dentro do quadrado e contar quantos desses pontos também caem dentro do círculo. Se você gerou uma amostra suficientemente grande e aleatória, a relação entre os pontos assentados dentro do círculo e os pontos dentro do quadrado (e simultaneamente no círculo) se aproxima da relação entre as áreas das duas formas, C / Q . Tudo o que você precisar fazer é contá-los.

Como saber se um ponto está situado dentro do círculo? Para ajudar a vislumbrar a solução, desenhe o quadrado em uma folha de papel milimetrado, com o centro do quadrado posicionado na origem, o ponto $(0,0)$. Então, você pode gerar pares de valores, ou coordenadas, posicionados dentro do intervalo $(-r, -r)$ a $(+r, +r)$. Para determinar se um conjunto de coordenadas (x, y) se encontra dentro do círculo, aplique o Teorema de Pitágoras para saber a distância d dessas coordenadas a partir da origem. Você pode calcular d como a raiz quadrada de $((x * x) + (y * y))$. Se d for menor ou igual a r , o raio do círculo, as coordenadas (x, y) especificam um ponto dentro do círculo, como no diagrama a seguir:



Para simplificar ainda mais, gere apenas as coordenadas posicionadas no quadrante superior direito do gráfico, de modo que você só precise gerar pares de números aleatórios entre 0 e r . Essa é a estratégia que você adotará nos exercícios.



Nota Os exercícios deste capítulo são destinados à execução em um computador equipado com um processador multicore. Se você tiver apenas uma CPU de um único núcleo, não perceberá os mesmos efeitos. Além disso, você não deve iniciar qualquer outro programa ou serviço entre os exercícios, porque isso poderá interferir nos resultados visualizados.

Calcule pi usando uma única thread

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra a solução CalculatePI, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\ Chapter 24\CalculatePI na sua pasta Documentos.
3. No Solution Explorer, no projeto CalculatePI, clique duas vezes em Program.cs para exibir o arquivo na janela Code and Text Editor.

Esse é um aplicativo de console. A estrutura básica do aplicativo já foi criada para você.

4. Vá até o final do arquivo e examine o método *Main*. Ele é semelhante ao seguinte:

```
static void Main(string[] args)
{
    double pi = SerialPI();
    Console.WriteLine("Geometric approximation of PI calculated serially: {0}", pi);

    Console.WriteLine();
    // pi = ParallelPI();
    // Console.WriteLine("Geometric approximation of PI calculated in parallel: {0}",
    pi);
}
```

Esse código chama o método *SerialPI*, o qual calculará pi por meio do algoritmo geométrico descrito anteriormente neste exercício. O valor é retornado como um *double* e exibido. O código que está como comentário chama o método *ParallelPI*, o qual efetuará o mesmo cálculo, mas por meio de tarefas simultâneas. O resultado exibido deve ser idêntico ao retornado pelo método *SerialPI*.

5. Examine o método *SerialPI*.

```
static double SerialPI()
{
    List<double> pointsList = new List<double>();
    Random random = new Random(SEED);
    int numPointsInCircle = 0;
```

```
Stopwatch timer = new Stopwatch();
timer.Start();

try
{
    // TO DO: implementar a aproximação geométrica de PI
    return 0;
}
finally
{
    long milliseconds = timer.ElapsedMilliseconds;
    Console.WriteLine("SerialPI complete: Duration: {0} ms", milliseconds);
    Console.WriteLine(
        "Points in pointsList: {0}. Points within circle: {1}",
        pointsList.Count, numPointsInCircle);
}
}
```

Esse método gera um grande conjunto de coordenadas e calcula as distâncias de cada conjunto de coordenadas a partir da origem. O tamanho do conjunto é especificado pela constante *NUMPOINTS* no início da classe *Program*. Quanto mais alto for esse valor, tanto maior será o conjunto de coordenadas e tanto mais preciso será o valor de pi calculado por esse método. Se houver memória suficiente, você pode aumentar o valor de *NUMPOINTS*. De modo semelhante, se você detectar que o aplicativo lança exceções *OutOfMemoryException* ao ser executado, você pode reduzir esse valor.

Armazene a distância de cada ponto a partir da origem na coleção *List<double> pointsList*. Os dados das coordenadas são gerados por meio da variável *random*. Esse é um objeto *Random*, provido de uma constante para gerar o mesmo conjunto de números aleatórios sempre que você executar o programa. (Isso ajuda a saber se ele está funcionando corretamente.) Você pode alterar a constante *SEED* no início da classe *Program* para fornecer um valor diferente ao gerador de números aleatórios.

Use a variável *numPointsInCircle* para contar o número de pontos na coleção *pointsList* posicionados dentro dos limites do círculo. O raio do círculo é especificado pela constante *RADIUS* no início da classe *Program*.

Para ajudá-lo a comparar o desempenho entre esse método e o método *ParallelPI*, o código gera uma variável *Stopwatch* chamada *timer* e inicia a sua execução. O bloco *finally* determina o tempo do cálculo e exibe o resultado. Por motivos que serão descritos posteriormente, o bloco *finally* também exibe o número de itens na coleção *pointsList* e o número de pontos que encontrou dentro do círculo.

Você adicionará o código que realmente efetua o cálculo ao bloco *try* nos próximos passos.

6. No bloco *try*, exclua o comentário e remova a instrução *return*. (Essa instrução foi incluída apenas para assegurar a compilação do código.) Adicione ao bloco *try* o bloco *for* e as instruções mostradas em negrito no código a seguir:

```

try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    }
}

```

Esse bloco de código gera um par de valores de coordenadas, no intervalo de 0 a *RADIUS*, e os armazena nas variáveis *xCoord* e *yCoord*. Em seguida, o código emprega o Teorema de Pitágoras para calcular a distância dessas coordenadas até a origem e adiciona o resultado à coleção *pointsList*.



Nota Embora esse bloco de código realize alguns cálculos, em um aplicativo científico real, provavelmente você incluirá cálculos muito mais complexos, que manterão o processador ocupado por muito mais tempo. Para simular essa situação, esse bloco de código chama outro método, *doAdditionalProcessing*. Tudo o que esse método faz é ocupar alguns ciclos da CPU, como no exemplo de código a seguir. Preferi adotar essa estratégia para demonstrar melhor as exigências de sincronização de dados de várias tarefas, em vez de fazê-lo escrever um aplicativo que efetue um cálculo extremamente complexo, como a Transformada Rápida de Fourier (FFT), para manter a CPU ocupada:

```

private static void doAdditionalProcessing()
{
    Thread.Sleep(SPINWAITS);
}

```

SPINWAITS é outra constante definida no início da classe *Program*.

7. No método *SerialPI*, no bloco *try*, após o bloco *for*, adicione a instrução *foreach* mostrada em negrito no exemplo a seguir.

```

try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        if (datum <= RADIUS)
        {
            numPointsInCircle++;
        }
    }
}

```

Esse código itera sobre a coleção *pointsList* e examina um valor de cada vez. Se o valor for menor ou igual ao raio do círculo, ele incrementará a variável *numPointsInCircle*. No final desse loop, *numPointsInCircle* deverá conter o número total de coordenadas posicionadas dentro dos limites do círculo.

8. Adicione ao bloco *try* as seguintes instruções mostradas em negrito, depois da instrução *foreach*:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        ...
    }

    double pi = 4.0 * numPointsInCircle / NUMPOINTS;
    return pi;
}
```

A primeira instrução calcula pi com base na proporção entre o número de pontos existentes dentro do círculo e o número total de pontos, por meio da fórmula descrita anteriormente. O valor é retornado como o resultado do método.

9. No menu Debug, clique em Start Without Debugging.

O programa é executado e exibe sua aproximação de pi, como na imagem a seguir. (Essa operação demorou apenas um pouco mais de 46 segundos em meu computador; portanto, prepare-se para esperar esse tempinho.) O tempo necessário para calcular o resultado também aparece. (Você pode ignorar os resultados do método *ParallelPI*, pois ainda não escreveu o código desse método.)

```
C:\Windows\system32\cmd.exe
SerialPI complete: Duration: 45761 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888
Press any key to continue . . .
```



Nota À exceção do tempo, seus resultados devem ser os mesmos, a menos que você tenha alterado as constantes *NUMPOINTS*, *RADIUS* ou *SEED*.

10. Feche a janela do console e retorne ao Visual Studio.

No método *SerialPI*, o código no loop *for* que gera os pontos e calcula sua distância até a origem é uma área que obviamente pode ser paralelizada. É isso que você fará no próximo exercício.

Calcule pi usando tarefas simultâneas

1. No Solution Explorer, clique duas vezes em Program.cs para exibir o arquivo na janela Code and Text Editor, se ele ainda não estiver aberto.
2. Localize o método *ParallelPI*. Ele contém exatamente o mesmo código da versão inicial do método *SerialPI*, antes de você adicionar o código ao método *try* para calcular pi.
3. No bloco *try*, exclua o comentário e remova a instrução *return*. Adicione ao bloco *try* a instrução *ParallelFor* mostrada aqui em negrito:

```
try
{
    ParallelFor (0, NUMPOINTS, (x) =>
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    });
}
```

Essa construção é o equivalente paralelo do código do loop *for* no método *SerialPI*. O corpo do loop *for* original é encapsulado em uma expressão lambda. Lembre-se de que cada iteração do loop é feita por meio de uma tarefa, e as tarefas podem ser executadas em paralelo. O nível de paralelismo depende do número de núcleos de processador e de outros recursos disponíveis em seu computador.

4. Adicione o seguinte código, mostrado em negrito, ao bloco *try*, depois da instrução *ParallelFor*. O código é exatamente igual às instruções correspondentes no método *SerialPI*.

```
try
{
    ParallelFor (...  
{  
    ...  
});  
  
foreach (double datum in pointsList)  
{  
    if (datum <= RADIUS)  
    {  
        numPointsInCircle++;  
    }
}
```

```

    }

    double pi = 4.0 * numPointsInCircle / NUMPOINTS;
    return pi;
}

```

5. No método *Main* próximo ao final do arquivo *Program.cs*, exclua as barras de comentário do código que chama o método *ParallelPI* e da instrução *Console.WriteLine* que exibe os resultados.
6. No menu *Debug*, clique em *Start Without Debugging*.

O programa é executado. A imagem a seguir mostra a saída característica:

```

C:\Windows\system32\cmd.exe
SerialPI complete: Duration: 47222 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 12480 ms
Points in pointsList: 9995392. Points within circle: 9989478
Geometric approximation of PI calculated in parallel: 3.9957912
Press any key to continue . . .

```

O valor calculado pelo método *SerialPI* deve ser exatamente igual ao anterior (o tempo pode ser um pouco diferente). Entretanto, embora tenha sido calculado um tanto mais rápido, o resultado do método *ParallelPI* parece um pouco suspeito. O gerador de números aleatórios recebeu o mesmo valor utilizado pelo método *SerialPI*, de modo que deve gerar a mesma sequência de números aleatórios com o mesmo resultado e o mesmo número de pontos dentro do círculo. Outro aspecto curioso é que a coleção *pointsList* no método *ParallelPI* parece conter bem menos pontos do que a mesma coleção no método *SerialPI*.



Nota Se a coleção *pointsList* contiver o número esperado de itens, execute o aplicativo novamente. Você deve descobrir que ela contém menos itens do que o previsto na maioria das execuções (mas não necessariamente em todas).

7. Feche a janela do console e retorne ao Visual Studio.

Então, o que deu errado no cálculo paralelo? Um bom ponto de partida é o número de itens na coleção *pointsList*. Essa coleção é um objeto *List<double>* genérico. Contudo, esse tipo não é thread-safe. O código na instrução *Parallel.For* chama o método *Add* para incluir um valor na coleção, mas lembre-se de que esse código está sendo executado pelas tarefas executadas como threads simultâneas. Consequentemente, diante do número de itens que estão sendo adicionados à coleção, é muito provável que algumas chamadas ao método *Add* interfiram entre si e causem alguns danos. Uma solução é utilizar uma das coleções do namespace *System.Collections.Concurrent*, pois elas são thread-safe. A classe genérica *ConcurrentBag<T>* nesse namespace provavelmente é a mais adequada para uso nesse exemplo.

Use uma coleção thread-safe

1. No Solution Explorer, clique duas vezes em Program.cs para exibir o arquivo na janela Code and Text Editor, se ele ainda não estiver aberto.

2. Adicione a seguinte diretiva *using* à lista localizada no início do arquivo:

```
using System.Collections.Concurrent;
```

3. Localize o método *ParallelPI*. No início desse método, substitua a instrução que instancia a coleção *List<double>* pelo código que cria uma coleção *ConcurrentBag<double>*, como mostrado em negrito no exemplo de código a seguir:

```
static double ParallelPI()  
{  
    ConcurrentBag<double> pointsList = new ConcurrentBag <double>();  
    Random random = ...;  
    ...  
}
```

Observe que não é possível especificar uma capacidade padrão para essa classe, de modo que o construtor não recebe parâmetro algum.

Não é necessário alterar outro código nesse método; você adiciona um item em uma coleção *ConcurrentBag<T>* utilizando o método *Add*, que é o mesmo mecanismo utilizado para adicionar um item a uma coleção *List<T>*.

4. No menu Debug, clique em Start Without Debugging.

O programa é executado e exibe sua aproximação de pi, por meio dos métodos *SerialPI* e *ParallelPI*. A imagem a seguir mostra a saída característica.

```
C:\Windows\system32\cmd.exe  
SerialPI complete: Duration: 47117 ms  
Points in pointsList: 10000000. Points within circle: 7853722  
Geometric approximation of PI calculated serially: 3.1414888  
  
ParallelPI complete: Duration: 14085 ms  
Points in pointsList: 10000000. Points within circle: 9989183  
Geometric approximation of PI calculated in parallel: 3.9956732  
Press any key to continue . . .
```

Desta vez, a coleção *pointsList* do método *ParallelPI* contém o número correto de pontos, mas o número de pontos dentro do círculo ainda parece ser muito alto; deveria ser idêntico ao informado pelo método *SerialPI*.

Observe também que o tempo gasto pelo método *ParallelPI* aumentou em comparação com o exercício anterior. Isso ocorre porque os métodos da classe *ConcurrentBag<T>* precisam bloquear e desbloquear dados para garantir o acesso thread-safe, e esse processo aumenta a sobrecarga de chamadas a esses métodos. Lembre-se desse aspecto ao ponderar se uma operação deve ser paralelizada ou não.

5. Feche a janela do console e retorne ao Visual Studio.

Você já tem o número correto de pontos na coleção *pointsList*, mas o valor registrado para cada um desses pontos é questionável. O código na construção *ParallelFor* chama o método *Next* de um objeto *Random*, mas como os métodos na classe

genérica *List<T>*, esse método não é thread-safe. Infelizmente, não existe uma versão concorrente da classe *Random*, e só lhe resta utilizar uma técnica alternativa para serializar as chamadas ao método *Next*. Como cada chamada é relativamente breve, compensa utilizar um bloqueio simples para proteger as chamadas a esse método.

Use um bloqueio para serializar as chamadas aos métodos

1. No Solution Explorer, clique duas vezes em *Program.cs* para exibir o arquivo na janela Code and Text Editor, se ele ainda não estiver aberto.
2. Localize o método *ParallelPI*. Modifique o código da expressão lambda na instrução *Parallel.For* para proteger as chamadas a *random.Next*, usando uma instrução de bloqueio (*lock*). Especifique a coleção *pointsList* como o alvo do bloqueio, como mostrado aqui em negrito:

```
static double ParallelPI()
{
    ...
    Parallel.For(0, NUMPOINTS, (x) =>
    {
        int xCoord;
        int yCoord;

        lock(pointsList)
        {
            xCoord = random.Next(RADIUS);
            yCoord = random.Next(RADIUS);
        }

        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    });
    ...
}
```

Observe que as variáveis *xCoord* e *yCoord* são declaradas fora da instrução de bloqueio. Isso porque a instrução de bloqueio define seu próprio escopo, e qualquer variável definida dentro do bloqueio, especificando o escopo da instrução de bloqueio, desaparece quando a construção termina.

3. No menu Debug, clique em Start Without Debugging.

Desta vez, os valores de pi calculados pelos métodos *SerialPI* e *ParallelPI* são os mesmos. A única diferença é que a execução do método *ParallelPI* é mais rápida.

```
C:\Windows\system32\cmd.exe
SerialPI complete: Duration: 47281 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 14284 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated in parallel: 3.1414888
Press any key to continue . . .
```

4. Feche a janela do console e retorne ao Visual Studio.

Resumo

Neste capítulo, vimos como definir métodos assíncronos utilizando o modificador `async` e o operador `await`. Os métodos assíncronos são baseados em tarefas, e o operador `await` especifica os pontos em que uma tarefa pode ser utilizada para realizar processamento assíncrono.

Você conheceu alguns aspectos da PLINQ e como utilizar o método de extensão `AsParallel` para paralelizar algumas consultas LINQ. Entretanto, a PLINQ é um assunto muito vasto e esse capítulo representou tão somente uma introdução. Para obter mais informações, consulte o tópico “Parallel LINQ (PLINQ)” na documentação fornecida com o Visual Studio.

Este capítulo também demonstrou como sincronizar o acesso a dados em tarefas simultâneas, por meio das primitivas de sincronização fornecidas para uso com tarefas. Você viu como é possível utilizar as classes de coleção concorrentes para manter coleções de dados de modo thread-safe.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 25, “Implementação da interface do usuário de um aplicativo Windows Store”.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Implementar um método assíncrono	<p>Defina o método com o modificador <code>async</code> e mude o tipo do método para retornar uma <code>Task</code> (ou um <code>void</code>). No corpo do método, use o operador <code>await</code> para especificar os pontos em que o processamento assíncrono pode ser realizado. Por exemplo:</p> <pre>private async Task<int> calculateValueAsync(...) { // Chama calculateValue utilizando uma Task Task<int> generateResultTask = Task.Run(() => calculateValue(...)); await generateResultTask; return generateResultTask.Result; }</pre>
Paralelizar uma consulta LINQ	<p>Especifique o método de extensão <code>AsParallel</code> com a origem de dados na consulta. Por exemplo:</p> <pre>var over100 = from n in numbers.AsParallel() where ... select n;</pre>

Para	Faça isto
Permitir cancelamento em uma consulta PLINQ	Use o método <i>WithCancellation</i> da classe <i>ParallelQuery</i> na consulta PLINQ e especifique um token de cancelamento. Por exemplo:
	<pre>CancellationToken tok = ...; ... var orderInfoQuery = from c in CustomersInMemory.Customers.AsParallel() WithCancellation(tok) join o in OrdersInMemory.Orders.AsParallel() on ...</pre>
Sincronizar uma ou mais tarefas para implementar acesso exclusivo thread-safe a dados compartilhados	Use a instrução <i>lock</i> para garantir acesso exclusivo aos dados. Por exemplo:
	<pre>object myLockObject = new object(); ... lock (myLockObject) { // Código que exige acesso exclusivo a um recurso compartilhado ... }</pre>
Sincronizar threads e fazê-las aguardar um evento	Use um objeto <i>ManualResetEventSlim</i> para sincronizar um número indeterminado de threads. Utilize um objeto <i>CountdownEvent</i> para esperar que um evento seja sinalizado um número especificado de vezes. Empregue um objeto <i>Barrier</i> para coordenar um número especificado de threads e sincronizá-las em determinado ponto de uma operação.
Sincronizar o acesso a um pool compartilhado de recursos	Utilize um objeto <i>SemaphoreSlim</i> . Especifique o número de itens do pool no construtor. Chame o método <i>Wait</i> antes de acessar um recurso do pool compartilhado. Chame o método <i>Release</i> quando terminar de utilizar o recurso. Por exemplo:
	<pre>SemaphoreSlim semaphore = new SemaphoreSlim(3); ... semaphore.Wait(); // Acessa um recurso do pool ... semaphore.Release();</pre>
Fornecer acesso de escrita exclusivo a um recurso, mas acesso de leitura compartilhado	Utilize um objeto <i>ReaderWriterLockSlim</i> . Antes de ler o recurso compartilhado, chame o método <i>EnterReadLock</i> . Quando terminar, chame o método <i>ExitReadLock</i> . Antes de escrever no recurso compartilhado, chame o método <i>EnterWriteLock</i> . Quando terminar a operação de escrita, chame o método <i>ExitWriteLock</i> . Por exemplo:
	<pre>ReaderWriterLockSlim readerWriterLock = new ReaderWriterLockSlim(); Task readerTask = Task.Factory.StartNew(() => { readerWriterLock.EnterReadLock(); // Lê recurso compartilhado readerWriterLock.ExitReadLock(); }); Task writerTask = Task.Factory.StartNew(() => { readerWriterLock.EnterWriteLock(); // Escreve em recurso compartilhado readerWriterLock.ExitWriteLock(); });</pre>

Para	Faça isto
C cancelar uma operação de espera com bloqueio	<p>Crie um token de cancelamento a partir de um objeto <code>CancellationTokenSource</code> e especifique esse token como um parâmetro para a operação de espera. Para cancelar a operação de espera, chame o método <code>Cancel</code> do objeto <code>CancellationTokenSource</code>. Por exemplo:</p> <pre>CancellationTokenSource cancellationTokenSource = new CancellationTokenSource(); CancellationToken cancellationToken = cancellationTokenSource.Token; ... // Semáforo que protege um pool de 3 recursos SemaphoreSlim semaphoreSlim = new SemaphoreSlim(3); ... // Aguarda o semáforo e lança uma OperationCanceledException se // outra thread chamar Cancel em cancellationTokenSource semaphore.Wait(cancellationToken);</pre>

CAPÍTULO 25

Implementação da interface do usuário de um aplicativo Windows Store

Neste capítulo, você vai aprender a:

- Descrever as características de um aplicativo Windows Store típico.
- Implementar uma interface de usuário escalonável para um aplicativo Windows Store que possa se adaptar a diferentes tamanhos físicos e orientações de dispositivo.
- Criar e aplicar estilos a um aplicativo Windows Store.

Os aplicativos Microsoft Windows Store são baseados em um novo paradigma que, quase indiscutivelmente, é a maior mudança no desenvolvimento de aplicativos clientes para Windows desde o advento do Microsoft .NET Framework pouco depois da virada do milênio. A interface continuamente conectada e baseada em toques do Windows 8.1, junto com o uso de contratos do Windows para interação com outros aplicativos, o suporte para sensores incorporados, e a segurança de aplicativos e o modelo do ciclo de vida atualizados, mudam o modo de usuários e aplicativos trabalharem em conjunto. Com o Microsoft Visual Studio 2013, os desenvolvedores podem projetar e implementar facilmente aplicativos capazes de tirar proveito dos novos recursos da plataforma Windows 8.1.

O objetivo deste capítulo é oferecer uma breve descrição desse novo paradigma e ajudá-lo a começar a utilização do Visual Studio 2013 para compilar aplicativos que funcionem nesse ambiente. Você vai conhecer alguns dos novos recursos e ferramentas incluídas no Visual Studio 2013 para compilar aplicativos Windows Store e vai começar a compilar um aplicativo que está de acordo com a aparência e comportamento do Windows 8.1. Vamos nos concentrar em aprender como implementar uma interface de usuário (IU) que muda de escala e se adapta a diferentes resoluções e tamanhos físicos de dispositivo, e como estilizar o aplicativo para que tenha aparência e comportamento especiais.



Nota Não há espaço suficiente em um livro como este para uma investigação abrangente sobre a compilação de aplicativos Windows Store. Em vez disso, estes últimos capítulos se concentram nos princípios básicos da compilação de um aplicativo interativo que utiliza a interface de usuário do Windows 8.1. Para obter informações detalhadas sobre como escrever aplicativos Windows Store, visite a página “Developing Windows Store apps” no site da Microsoft, em <http://msdn.microsoft.com/library/windows/apps/xaml/BR229566>.

O que é um aplicativo Windows Store?

Os aplicativos Windows Store são aplicativos gráficos que têm aparência e comportamento especiais, baseados em alguns princípios simples que você vai conhecer agora e nos Capítulos 26, “Exibição e busca de dados em um aplicativo Windows Store”, e 27, “Acesso a um banco de dados remoto em um aplicativo Windows Store”. Eles são altamente interativos, colocando o usuário no centro de suas operações. A Microsoft investiu muito tempo (e dinheiro) em estudos que examinaram como os usuários gostam de trabalhar com aplicativos e as maneiras mais eficazes de exibir dados e interagir com eles. Os projetistas da Microsoft aplicaram esse conhecimento para desenvolver o modelo que os aplicativos Windows Store devem seguir.

Ao ser executado, um aplicativo Windows Store normalmente ocupa a tela inteira, embora os aplicativos Windows Store possam operar em diferentes modos de exibição, os quais você vai conhecer neste capítulo. Eles são *chromeless*, significando que não têm bordas, menus suspensos, janelas pop-up nem muitos dos outros recursos de interface que podem distrair os usuários. Em vez disso, um aplicativo Windows é projetado para ser limpo e centrado em ajudar o usuário a executar um conjunto específico de tarefas. Além disso, a Microsoft investigou áreas como a legibilidade, e a empresa documentou diretrizes ilustrando o uso padronizado de fontes, espessuras, espaçamento e posicionamento de texto.



Nota Mais informações sobre as características de projeto de aplicativos Windows Store podem ser encontradas online no site da Microsoft. Especificamente, a página “UX Patterns,” disponível em <http://msdn.microsoft.com/library/windows/apps/hh770552.aspx>, fornece orientações sobre o planejamento dos recursos que um aplicativo Windows Store deve implementar e como projetá-lo de modo a oferecer a melhor experiência para o usuário.

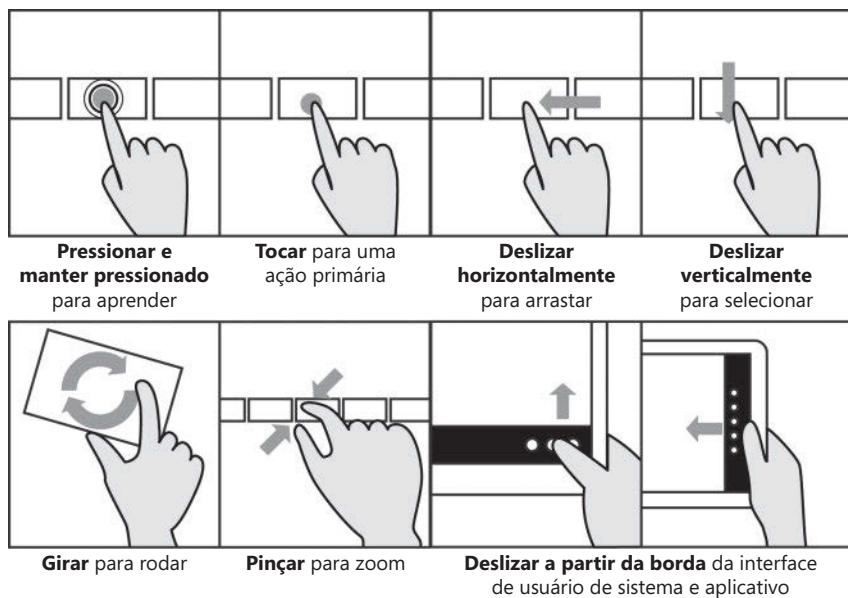
Muitos dispositivos portáteis e tablets modernos permitem que o usuário interaja com os aplicativos utilizando toques, e você deve projetar seus aplicativos Windows Store com base nesse estilo de experiência do usuário. O Windows 8.1 contém um amplo conjunto de controles baseados em toque que também funcionam com mouse e teclado, caso o usuário não possua um dispositivo sensível a toques. Não é preciso separar os recursos de toque e mouse em seus aplicativos; basta projetar para toque e os usuários ainda poderão operar utilizando o mouse e o teclado, se preferirem ou estiverem executando em um dispositivo que não aceita esse estilo de interação.

Animações sutis, fluidas e distintas também são um aspecto importante da experiência do usuário como um todo. O modo como a interface gráfica do usuário (GUI) responde aos gestos para dar retorno ao usuário podem melhorar muito a sensação profissional de seus aplicativos. Os templates de aplicativos Windows Store incluídos no Visual Studio 2013 contêm uma biblioteca de animações que você pode usar em seus aplicativos para padronizar esse retorno e adaptar perfeitamente ao sistema operacional e ao software fornecidos pela Microsoft.



Nota O termo *gesto* se refere às operações manuais baseadas em toques que o usuário pode executar. Por exemplo, o usuário pode tocar em um item com o dedo, e esse gesto normalmente responde da mesma maneira esperada para o comportamento de um clique de mouse. Contudo, os gestos podem ser bem mais expressivos que as operações simples capturadas com um mouse. Por exemplo, o gesto “*girar*” exige que o usuário coloque dois dedos na tela e “*desenhe*” um arco de círculo com eles; em um aplicativo Windows 8.1 típico, esse gesto deve fazer a interface do usuário girar o objeto selecionado na direção indicada pelo movimento dos dedos. Outros gestos incluem “*pinça*” (pinch) para ampliar um item a fim de mostrar mais detalhes, “*pressionar e manter pressionado*” para revelar mais informações sobre um item (semelhante a dar um clique com o botão direito do mouse) e “*deslizar*” para selecionar e arrastar um item na tela.

A interface do usuário do Windows 8.1 emprega diversos gestos específicos para interagir com o sistema operacional. Por exemplo, você pode “*deslizar*” (swipe) o aplicativo em execução verticalmente para a parte inferior da tela a fim de encerrá-lo, ou deslizá-lo da margem esquerda para a direita da tela para ver os ícones de sistema do Windows.



O Windows 8.1 foi projetado para executar em uma ampla variedade de dispositivos, desde computadores desktop e laptops até tablets e pequenos aparelhos portáteis e smartphones. Um conceito importante na compilação de aplicativos Windows Store é a necessidade de produzir software que se adapte ao ambiente em que está sendo executado, escalonando automaticamente de acordo com o tamanho da tela e a orientação do dispositivo. Essa estratégia abre seu software para um mercado cada vez maior. Além disso, muitos dispositivos modernos também podem detectar

sua orientação e a velocidade na qual o usuário muda essa orientação, por meio de sensores e acelerômetros incorporados. Os aplicativos Windows Store podem adaptar seu layout quando o usuário inclina ou gira um dispositivo, possibilitando à pessoa trabalhar do modo que julgar mais confortável. Você também deve entender que a mobilidade é um requisito primordial para muitos aplicativos modernos, e com os aplicativos Windows Store, os usuários podem mudar de lugar; por meio da nuvem, seus dados podem migrar para qualquer dispositivo que esteja executando seu aplicativo em dado momento.

Os aplicativos Windows Store também podem dar suporte para vários recursos que facilitam as interações com outros aplicativos e com o sistema operacional Windows 8.1. Esses recursos são baseados em um mecanismo padronizado conhecido como *contrato*. Um contrato define uma interface do Windows 8.1 por meio da qual um aplicativo pode implementar ou consumir um recurso estipulado pelo sistema operacional, como a capacidade de compartilhar dados ou aceitar pedidos de busca. Utilizando os contratos, os aplicativos Windows Store podem se comunicar sem introduzir qualquer dependência específica. Discutiremos os contratos do Windows 8.1 em mais detalhes no Capítulo 26.

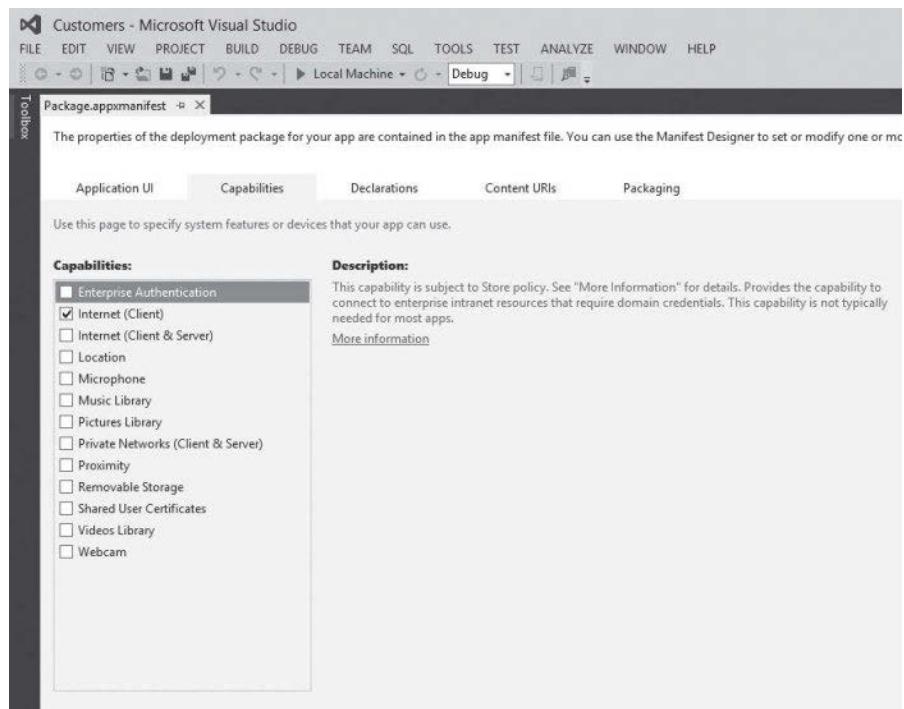
A duração de um aplicativo Windows Store é um tanto diferente da de um aplicativo de desktop tradicional. Em dado momento, o Windows 8.1 apenas executa o aplicativo que ocupa a área do foco na tela, e se você trocar para outro aplicativo, ele se tornará o foco, vindo para o primeiro plano, e o aplicativo original é suspenso. Na verdade, o Windows 8.1 pode optar por fechar um aplicativo suspenso, caso verifique que precisa liberar recursos de sistema, como a memória. Na próxima vez que o aplicativo é executado, deve ser capaz de retomar a partir de onde foi interrompido. Isso significa que você precisa estar preparado para gerenciar informações de estado de aplicativo em seu código, salvá-las no disco rígido e restaurá-las no momento apropriado.



Nota Mais informações sobre como gerenciar o ciclo de vida de um aplicativo Windows Store podem ser encontradas no site da Microsoft. Procure a página "How to suspend an app" em <http://msdn.microsoft.com/library/windows/apps/xaml/hh465115.aspx> e a página "How to resume an app" em <http://msdn.microsoft.com/library/windows/apps/xaml/hh465110.aspx>.

Ao compilar um novo aplicativo Windows Store, você pode empacotá-lo com as ferramentas fornecidas com o Visual Studio 2013 e carregá-lo no Windows Store. Então, outros usuários podem se conectar na loja, baixar seu aplicativo e instalá-lo. Você pode cobrar uma taxa por seus aplicativos ou torná-los disponíveis gratuitamente. Esse mecanismo de distribuição e implantação depende da confiabilidade de seus aplicativos e de sua obediência às políticas de segurança especificadas pela Microsoft. Quando você carrega um aplicativo no Windows Store, ele passa por várias verificações para confirmar se não contém código mal-intencionado e se obedece aos requisitos de segurança de um aplicativo Windows Store. Essas restrições de segurança determinam como seu aplicativo acessa recursos no computador em que será instalado. Por exemplo, por padrão, um aplicativo Windows Store não pode escrever diretamente no sistema de arquivos nem receber pedidos enviados pela rede (dois

dos comportamentos normalmente exibidos por vírus e outros malware). Contudo, se seu aplicativo precisa executar operações como essas, você pode especificá-las como capacidades no manifesto do aplicativo, utilizando o editor de manifesto do Visual Studio 2013. Para abrir o editor de manifesto, no Solution Explorer, clique duas vezes no arquivo Package.appxmanifest.



Nota Mais informações sobre as capacidades suportadas pelos aplicativos Windows Store podem ser encontradas na página "App capability declarations", no site da Microsoft em <http://msdn.microsoft.com/library/windows/apps/hh464936.aspx>.

Essas informações são gravadas nos metadados de seu aplicativo e avisam à Microsoft para que faça testes adicionais a fim de verificar como esses recursos são utilizados em seu aplicativo. Há também limitações no funcionamento dessas operações, destinadas a proteger os computadores nos quais seu aplicativo é instalado. Por exemplo, você pode indicar que seu aplicativo exige acesso aos arquivos da pasta Documentos, mas não pode ler nem escrever em arquivos localizados em outros lugares no computador hospedeiro.

Basta de teoria – vamos começar a compilar um aplicativo Windows Store.

Use o template Blank App para compilar um aplicativo Windows Store

O modo mais simples de compilar um aplicativo Windows Store é com os templates do Visual Studio 2013 no Windows 8.1. O Visual Studio 2013 fornece três templates principais para isso: Blank App, Grid App e Split App. Cada um desses templates permite criar rapidamente aplicativos que obedecem às diretrizes de interface de usuário do Windows 8.1, com base no modelo recomendado pela Microsoft. Além disso, eles geram um volume de código considerável, e ajuda a entender como esse código é estruturado, para que você possa adaptá-lo aos seus aplicativos e requisitos de dados. As versões para Windows 8.1 dos aplicativos vistos nos capítulos anteriores utilizavam o template Blank App, sendo esse um bom ponto de partida.

Nos exercícios a seguir, você vai projetar e implementar a interface do usuário de um aplicativo simples para uma empresa fictícia chamada Adventure Works. Essa empresa fabrica e fornece bicicletas e os acessórios associados. O aplicativo permitirá ao usuário inserir e modificar os detalhes dos clientes da Adventure Works.

Crie o aplicativo Customers da Adventure Works

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. No menu File, aponte para New e então clique em Project.
3. Na caixa de diálogo New Project, no painel da esquerda, expanda Templates, expanda Visual C# e clique em Windows Store.
4. No painel central, clique no ícone Blank App (XAML).
5. No campo Name, digite **Customers**.
6. No campo Location, digite **\Microsoft Press\Visual CSharp Step By Step\Chapter 25** na sua pasta Documentos.
7. Clique em OK.

O novo aplicativo é criado e o arquivo App.xaml.cs aparece na janela Code and Text Editor. Você pode ignorar esse arquivo por enquanto.

8. No Solution Explorer, clique duas vezes em MainPage.xaml.

A janela Design View aparece e exibe uma página em branco. Você pode arrastar controles da Toolbox para adicionar os vários controles exigidos pelo aplicativo, como demonstrado no Capítulo 1, “Bem-vindo ao C#”. No entanto, para os propósitos deste exercício, é mais instrutivo nos concentrarmos na marcação XAML que define o layout do formulário. Se você examinar essa marcação, deverá ser parecida com esta:

```
<Page  
    x:Class="Customers.MainPage"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:local="using:Customers"
```

```

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

</Grid>
</Page>
```

O formulário começa com a tag XAML `<Page>` e termina com uma tag `</Page>` de fechamento. Tudo que está entre essas tags define o conteúdo da página.

Os atributos da tag `<Page>` contêm várias declarações na forma `xmlns:id = "..."`. São declarações de namespace XAML e operam de maneira semelhante às diretivas *using* do C#, visto que colocam itens no escopo. Muitos dos controles e outros itens que podem ser adicionados a uma página são definidos nesses namespaces XAML, e você pode ignorar a maioria dessas declarações. Entretanto, existe uma declaração de aparência bastante curiosa, à qual você deve prestar atenção:

```
xmlns:local="using:Customers"
```

Essa declaração coloca no escopo os itens do namespace *Customers* do C#, com os quais você pode referenciar classes e outros tipos desse namespace em seu código XAML, prefixando-os com *local*. O namespace *Customers* é o namespace gerado pelo código de seu aplicativo.

9. No Solution Explorer, expanda `MainPage.xaml` e, então, clique duas vezes em `MainPage.xaml.cs` para exibi-lo na janela Code and Text Editor.
10. Lembre-se, dos exercícios anteriores deste livro, que esse é o arquivo do C# que contém a lógica do aplicativo e rotinas de tratamento de evento para o formulário. Ele é como este (as diretivas *using* da parte superior do arquivo foram omitidas para economizar espaço):

```
// O template de item Blank Page está documentado em http://go.microsoft.com/fwlink/?LinkId=234238
```

```

namespace Customers
{
    /// <summary>
    /// Uma página vazia que pode ser usada sozinha ou dentro de um Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Esse arquivo define os tipos do namespace *Customers*. A página é implementada por uma classe chamada *MainPage* e ela herda da classe *Page*. A classe *Page* implementa a funcionalidade padrão de uma página XAML para um aplicativo

Windows Store, de modo que basta escrever o código que define a lógica específica para seu aplicativo na classe *MainPage*.

11. Volte para o arquivo MainPage.xaml na janela Design View. Se examinar a marcação XAML da página, você deverá observar que a tag <Page> inclui o seguinte atributo:

```
x:Class="Customers.MainPage"
```

Esse atributo conecta a marcação XAML que define o layout da página à classe *MainPage* que fornece a lógica por trás da página.

Essa é a estrutura básica de um aplicativo Windows Store simples. Evidentemente, o que torna um aplicativo gráfico valioso é o modo como ele apresenta informações para o usuário. Isso nem sempre é tão simples quanto parece. Projetar uma interface gráfica atraente e fácil de usar exige habilidades de especialista que nem todos os desenvolvedores têm (sei disso, pois eu mesmo não tenho). Entretanto, muitos artistas gráficos que possuem essas habilidades não são programadores; portanto, embora eles sejam capazes de projetar uma interface de usuário maravilhosa, talvez não consigam implementar a lógica exigida para torná-la útil. Felizmente, o Visual Studio 2013 possibilita separar o projeto da interface do usuário da lógica do negócio, de modo que um artista gráfico e um desenvolvedor podem cooperar na construção de um aplicativo de excelente aparência que funciona bem. Tudo que o desenvolvedor precisa fazer é se concentrar no layout básico do aplicativo e deixar que um artista gráfico providencie a estilização.

Implemente uma interface de usuário escalonável

O segredo para fazer o layout da interface do usuário de um aplicativo Windows Store é saber como fazê-lo mudar de escala e se adaptar aos diferentes tamanhos físicos disponíveis para os dispositivos nos quais os usuários podem executar o aplicativo. Nos exercícios a seguir, você vai investigar como se obtém esse escalonamento.

Faça o layout da página do aplicativo Customers

1. No Visual Studio, examine a marcação XAML da página MainPage.

A página contém apenas um controle *Grid*:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">  
</Grid>
```



Nota Não se preocupe com o modo como a propriedade *Background* é especificada para o controle *Grid*. Esse é um exemplo de uso de um estilo, e você vai aprender a utilizar estilos mais adiante neste capítulo.

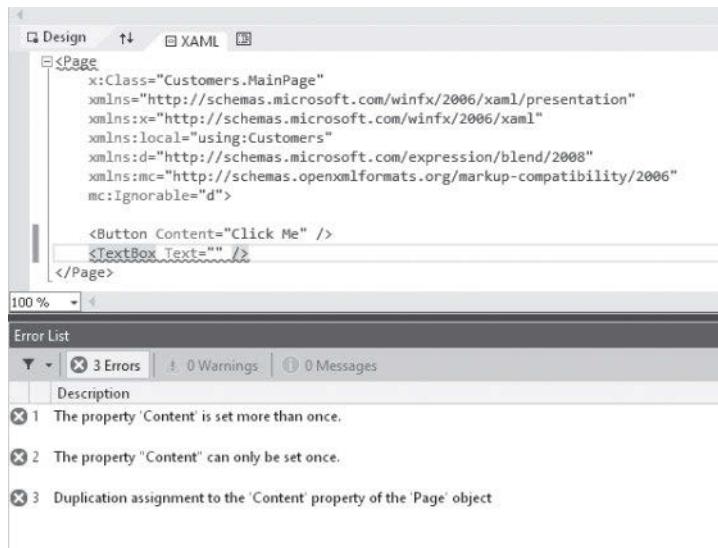
Conhecer o funcionamento do controle *Grid* é fundamental para construir interfaces de usuário escalonáveis e flexíveis. O elemento *Page* só pode conter um item e, se quiser, você pode substituir o controle *Grid* por um *Button*, como mostrado no exemplo a seguir:



Nota Não digite o código a seguir. Ele aparece somente para propósitos ilustrativos.

```
<Page
  ...
  <Button Content="Click Me"/>
</Page>
```

Contudo, o aplicativo resultante provavelmente não é muito útil — um formulário que contém um botão e não mostra mais nada com certeza não vai ganhar o prêmio de melhor aplicativo do mundo. Se você tentar adicionar um segundo controle, como um *TextBox*, à página, seu código não compilará e ocorrerão os erros mostrados na imagem a seguir:



O objetivo do controle *Grid* é facilitar a adição de vários itens a uma página. Ele é um exemplo de controle contêiner — pode conter vários outros controles e é possível especificar a posição desses outros controles dentro da grade. Também estão disponíveis outros controles contêiner. Por exemplo, o controle *StackPanel* posiciona automaticamente os controles que contêm em um arranjo vertical, com cada um posicionado diretamente abaixo do anterior.

Nesse aplicativo, você vai usar um *Grid* para conter os controles necessários para que o usuário possa inserir e ver dados de um cliente.

2. Adicione um controle *TextBlock* à página, ou arrastando-o da Toolbox ou digitando o texto <**TextBlock**> diretamente no painel XAML, na linha em branco após a tag <*Grid*> de abertura, como segue:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock />
</Grid>
```



Dica Você pode digitar o código para um controle diretamente na janela XAML de uma página; não é obrigatório arrastar controles da Toolbox.

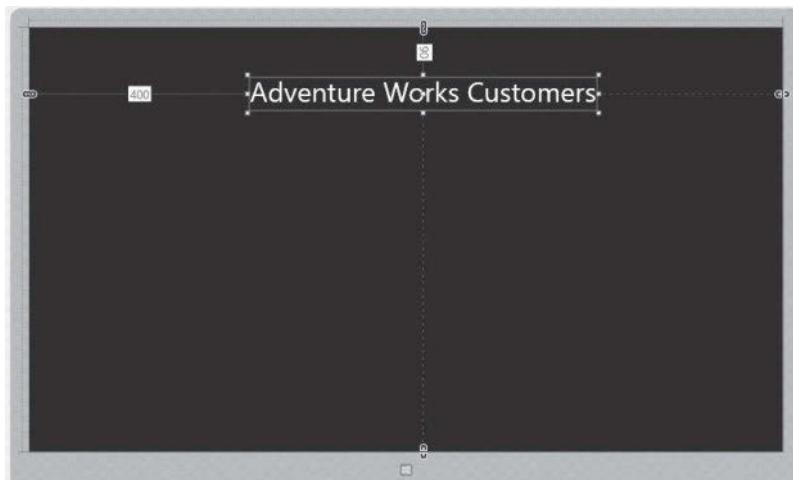
3. Esse *TextBlock* fornece o título da página. Defina as propriedades do controle *TextBlock* usando os valores da tabela a seguir.

Propriedade	Valor
HorizontalAlignment	Left
Margin	400,90,0,0
TextWrapping	Wrap
Text	Adventure Works Customers
VerticalAlignment	Top
FontSize	50

Você pode definir essas propriedades utilizando a janela Properties ou digitando a marcação XAML equivalente na janela XAML, como mostrado aqui em negrito:

```
<TextBlock HorizontalAlignment="Left" Margin="400,90,0,0" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Top" FontSize="50"/>
```

O texto resultante deve aparecer na janela Design View, deste modo:



Observe que, quando um controle é arrastado da Toolbox para um formulário, aparecem conectores especificando a distância de dois dos lados do controle a partir da borda do controle contêiner no qual é colocado. No exemplo anterior, para o controle *TextBlock* esses conectores são rotulados com os valores 400 (a partir da borda esquerda da grade) e 90 (a partir da borda superior da grade). Em tempo de execução, se o controle *Grid* for redimensionado, o *TextBlock* se moverá para manter essas distâncias e, neste caso, poderá fazer com que a distância em pixels do *TextBlock* a partir das bordas direita e inferior da *Grid* mude. Você pode especificar a borda (ou bordas) na qual um controle é ancorado, definindo as propriedades *HorizontalAlignment* e *VerticalAlignment*, e a propriedade *Margin* especifica a distância a partir das bordas ancoradas. Novamente, neste exemplo, a propriedade *HorizontalAlignment* do *TextBlock* é definida com *Left* e a propriedade *VerticalAlignment* é definida com *Top*, que é o motivo pelo qual o controle está ancorado nas bordas esquerda e superior da grade. A propriedade *Margin* contém quatro valores que especificam a distância dos lados esquerdo, superior, direito e inferior (nessa ordem) do controle a partir da borda correspondente do contêiner. Se um lado de um controle não estiver ancorado a uma borda do contêiner, você pode definir o valor correspondente na propriedade *Margin* com 0.

4. Adicione mais quatro controles *TextBlock* à página. Esses controles *TextBlock* são rótulos que ajudam o usuário a identificar os dados exibidos na página. Use os valores da tabela a seguir para definir as propriedades desses controles:

Controle	Propriedade	Valor
First Label	HorizontalAlignment	Left
	Margin	330,190,0,0
	TextWrapping	Wrap
	Text	ID
	VerticalAlignment	Top
	FontSize	20
Second Label	HorizontalAlignment	Left
	Margin	460,190,0,0
	TextWrapping	Wrap
	Text	Title
	VerticalAlignment	Top
	FontSize	20
Third Label	HorizontalAlignment	Left
	Margin	620,190,0,0
	TextWrapping	Wrap
	Text	First Name
	VerticalAlignment	Top
	FontSize	20

Controle	Propriedade	Valor
Fourth Label	HorizontalAlignment	Left
	Margin	975,190,0,0
	TextWrapping	Wrap
	Text	Last Name
	VerticalAlignment	Top
	FontSize	20

Como antes, você pode arrastar os controles da Toolbox e utilizar a janela Properties para definir suas propriedades ou digitar a marcação XAML a seguir no painel XAML, após o controle *TextBlock* existente e antes da tag *</Grid>* de fechamento:

```
<TextBlock HorizontalAlignment="Left" Margin="330,190,0,0" TextWrapping="Wrap"
Text="ID" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="460,190,0,0" TextWrapping="Wrap"
Text="Title" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="620,190,0,0" TextWrapping="Wrap"
Text="First Name" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="975,190,0,0" TextWrapping="Wrap"
Text="Last Name" VerticalAlignment="Top" FontSize="20"/>
```

5. Adicione três controles *TextBox* abaixo dos controles *TextBlock* que exibem o texto ID, First Name e Last Name. Use a tabela a seguir para definir os valores desses controles. Observe que a propriedade *Text* deve ser definida com a string vazia, *""*. Observe também que o controle *TextBox* *id* está marcado como somente leitura. Isso porque as identificações (IDs) de cliente serão geradas automaticamente no código que você vai adicionar depois:

Controle	Propriedade	Valor
First TextBox	x:Name	id
	HorizontalAlignment	Left
	Margin	300,240,0,0
	TextWrapping	Wrap
	Text	
	VerticalAlignment	Top
	FontSize	20
	IsReadOnly	True

Controle	Propriedade	Valor
Second TextBox	x:Name	firstName
	HorizontalAlignment	Left
	Margin	550,240,0,0
	TextWrapping	Wrap
	Text	
	VerticalAlignment	Top
Third TextBox	FontSize	20
	x:Name	lastName
	HorizontalAlignment	Left
	Margin	875,240,0,0
	TextWrapping	Wrap
	Text	
Text	VerticalAlignment	Top
	FontSize	20

O código a seguir mostra a marcação XAML equivalente para esses controles:

```
<TextBox x:Name="id" HorizontalAlignment="Left" Margin="300,240,0,0" TextWrapping="Wrap"
Text="" VerticalAlignment="Top" FontSize="20" IsReadOnly="True"/>
<TextBox x:Name="firstName" HorizontalAlignment="Left" Margin="550,240,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="300" FontSize="20"/>
<TextBox x:Name="lastName" HorizontalAlignment="Left" Margin="875,240,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="300" FontSize="20"/>
```

A propriedade *Name* não é exigida para um controle, mas é útil, caso você queira fazer referência ao controle no código C# do aplicativo. Observe que a propriedade *Name* tem o prefixo *x*: Isso é uma referência ao namespace XML <http://schemas.microsoft.com/winfx/2006/xaml> especificado nos atributos *Page* no início da marcação XAML. Esse namespace define a propriedade *Name* de todos os controles.



Nota Não há necessidade de entender por que a propriedade *Name* é definida dessa maneira, mas para obter mais informações, leia o artigo “*x:Name Directive*” em <http://msdn.microsoft.com/library/ms752290.aspx>.

A propriedade *Width* especifica a largura do controle e a propriedade *TextWrapping* indica o que acontece se o usuário tenta inserir informações que ultrapassam essa largura no controle. Neste caso, todos os controles *TextBox* passarão o texto para outra linha de mesma largura (o controle se expandirá verticalmente). O valor alternativo, *NoWrap*, faz o texto rolar horizontalmente, à medida que o usuário o insere.

6. Adicione um controle *ComboBox* ao formulário e posicione-o abaixo do controle *TextBlock Title*, entre os controles *TextBox id* e *firstName*. Defina as propriedades desse controle como segue.

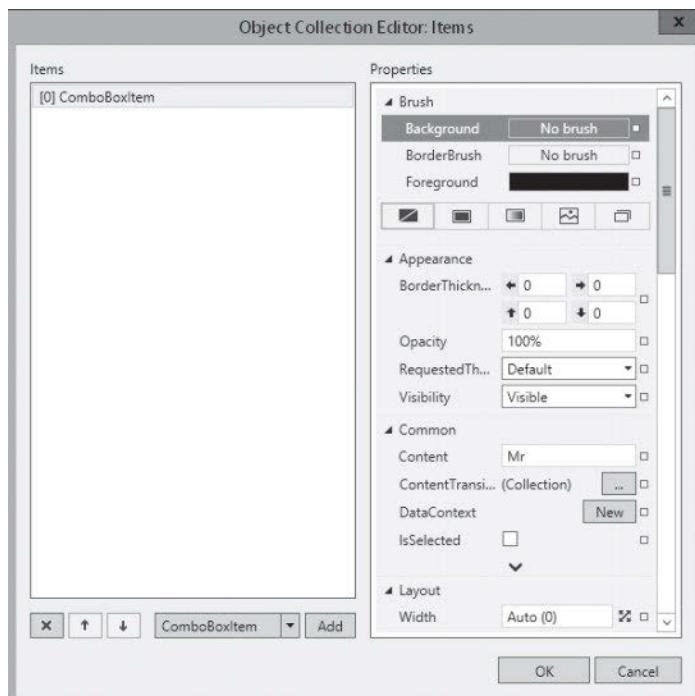
Propriedade	Valor
x:Name	title
HorizontalAlignment	Left
Margin	420,240,0,0
VerticalAlignment	Top
Width	100
FontSize	20

A marcação XAML equivalente para esse controle é a seguinte:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
VerticalAlignment="Top" Width="100" FontSize="20"/>
```

Um controle *ComboBox* é utilizado para exibir uma lista de valores para seleção do usuário.

7. Na janela Design View, clique no ComboBox title e, na janela Properties, expanda a categoria de propriedades Common, se ainda não estiver expandida. Em seguida, clique no botão de reticências que aparece ao lado da propriedade Items. A janela Object Collection Editor se abre.
8. Na lista do canto inferior esquerdo da janela, selecione ComboBoxItem e clique em Add. No painel direito que exibe as propriedades do item, expanda a seção Common, se ainda não estiver expandida, e digite **Mr** na propriedade Content.



9. Clique em OK.

O Object Collection Editor se fecha. Se você examinar a marcação XAML do *ComboBox title*, agora ela deve ser parecida com esta:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
VerticalAlignment="Top" Width="100" FontSize="20"/>
<ComboBoxItem Content="Mr"/>
</ComboBox>
```

Observe duas coisas aqui. Primeiro, a marcação *ComboBox* foi dividida em uma tag <*ComboBox*> de abertura e uma tag </*ComboBox*> de fechamento. Segundo, entre essas tags, o Visual Studio adicionou um elemento *ComboBoxItem* com a propriedade *Content* definida como *Mr*. Esse item será exibido em uma lista suspensa quando o aplicativo for executado.

10. Adicione os valores *Mrs*, *Ms* e *Miss* ao *ComboBox title*. Você pode utilizar o Object Collection Editor ou digitar a marcação XAML manualmente. A marcação resultante deve ser parecida com esta:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
VerticalAlignment="Top" Width="75" FontSize="20"/>
<ComboBoxItem Content="Mr"/>
<ComboBoxItem Content="Mrs"/>
<ComboBoxItem Content="Ms"/>
<ComboBoxItem Content="Miss"/>
</ComboBox>
```



Nota Um controle *ComboBox* pode exibir elementos simples, como um conjunto de controles *ComboBoxItem* que mostram texto, mas também pode conter elementos mais complexos, como botões, caixas de seleção e botões de opção. Se você está adicionando controles *ComboBoxItem* simples, então provavelmente é mais fácil digitar a marcação XAML manualmente, mas se está adicionando controles mais complexos, o Object Collection Editor pode se mostrar muito útil. Contudo, você deve evitar muita engenhosidade em uma caixa de combinação; os melhores aplicativos são aqueles que fornecem as interfaces de usuário mais intuitivas, e incorporar controles complexos em uma caixa de combinação pode ser confuso para o usuário.

11. Adicione mais dois controles *TextBox* e mais dois controles *TextBlock* ao formulário. Com os controles *TextBox*, o usuário poderá inserir um endereço de e-mail e o número do telefone do cliente, e os controles *TextBlock* fornecerão os rótulos das caixas de texto. Use os valores da tabela a seguir para definir as propriedades dos controles.

Controle	Propriedade	Valor
First TextBlock	HorizontalAlignment	Left
	Margin	300,390,0,0
	TextWrapping	Wrap
	Text	Email
	VerticalAlignment	Top
	FontSize	20
First TextBox	x:Name	email
	HorizontalAlignment	Left
	Margin	450,390,0,0
	TextWrapping	Wrap
	Text	Deixe vazio
	VerticalAlignment	Top
Second TextBlock	Width	400
	FontSize	20
	HorizontalAlignment	Left
	Margin	300,540,0,0
	TextWrapping	Wrap
	Text	Phone
Second TextBox	VerticalAlignment	Top
	FontSize	20
	x:Name	phone
	HorizontalAlignment	Left
	Margin	450,540,0,0
	TextWrapping	Wrap

A marcação XAML desses controles deve se parecer com esta:

```
<TextBlock HorizontalAlignment="Left" Margin="300,390,0,0" TextWrapping="Wrap"
Text="Email" VerticalAlignment="Top" FontSize="20"/>
<TextBox x:Name="email" HorizontalAlignment="Left" Margin="450,390,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="400" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="300,540,0,0" TextWrapping="Wrap"
Text="Phone" VerticalAlignment="Top" FontSize="20"/>
<TextBox x:Name="phone" HorizontalAlignment="Left" Margin="450,540,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="200" FontSize="20"/>
```

O formulário concluído na janela Design View deve ser semelhante a este:

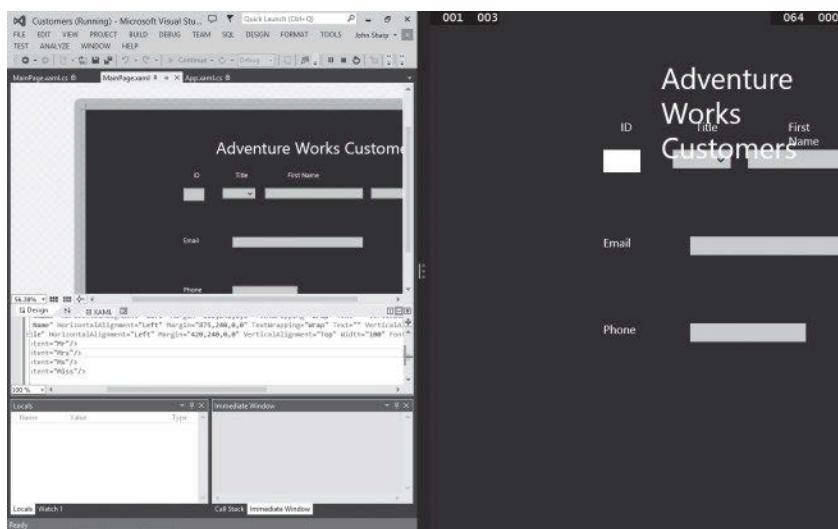


- 12.** No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.

O aplicativo inicia e exibe o formulário. Você pode inserir dados no formulário e selecionar um título na caixa de combinação, mas ainda não pode fazer muito mais do que isso.

- 13.** Enquanto o aplicativo está executando, clique no canto superior esquerdo da tela e arraste a imagem do Visual Studio em execução na área de trabalho.

O aplicativo Customers se ajusta sozinho e ocupa metade da tela. Dependendo da resolução de sua tela, o título ocupará mais de uma linha e o campo Last Name avançará para além da margem direita, como ilustrado aqui:



- 14.** Redimensione a janela que exibe o aplicativo Customers para sua largura mínima. Desta vez, grande parte do formulário desaparece (ele é exibido em uma área de apenas 500 pixels de largura). Parte do conteúdo do *TextBlock* passa para a próxima linha, mas claramente o formulário não pode ser usado nessa visualização.

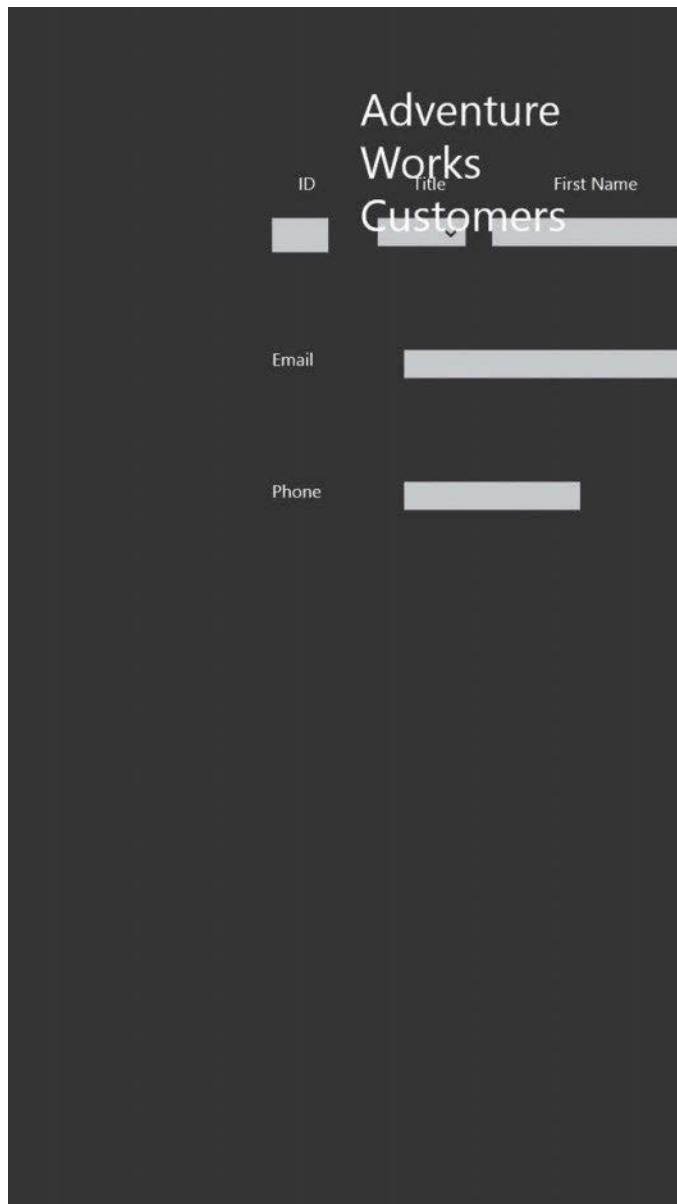
Você também pode aumentar a visualização, fazendo-a ocupar aproximadamente dois terços da tela. Mais uma vez, dependendo da resolução disponível, parte do texto poderá passar para a próxima linha e alguns controles poderão não aparecer na tela.



Nota A largura mínima padrão de um aplicativo (500 pixels) se destina a refletir o tamanho de um tablet típico, quando mantido na orientação retrato. Lembre-se de que nem todos os usuários vão executar seu aplicativo em um computador de mesa e, portanto, você deve garantir que ele apresente uma interface de usuário que funcione nessa largura. Também é possível configurar um aplicativo para suportar uma largura mínima de 320 pixels. Essa é a largura de muitos aparelhos Windows Phone. Para mudar a largura mínima padrão, edite a configuração de *Minimum Width* na página Application UI do arquivo Package.appxmanifest no editor de manifesto do Visual Studio.

- 15.** Volte para o Visual Studio e, no menu Debug, clique em Stop Debugging.

Essa foi uma lição útil sobre ser cuidadoso com o modo de fazer o layout de um aplicativo. Embora o aplicativo tivesse boa aparência ao ser executado em tela inteira, assim que você redimensionou a janela para uma visualização mais estreita, ele se tornou menos útil (ou completamente inútil). Além disso, o aplicativo presume que o usuário verá a tela em um dispositivo na orientação paisagem. Se o usuário executar o aplicativo em um tablet que suporte orientações diferentes e girar o dispositivo a fim de trocar para o modo retrato, ele ficará assim:

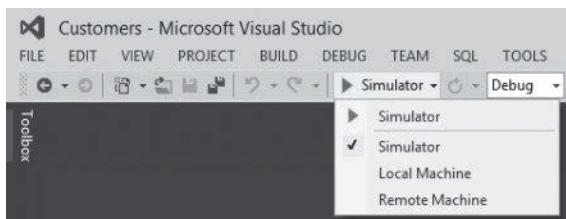


O problema é que a técnica de layout mostrada até aqui não muda de escala e não se adapta a diferentes tamanhos físicos e orientações. Felizmente, é possível utilizar as propriedades do controle *Grid* e outro recurso, chamado Visual State Manager, para resolver esses problemas.

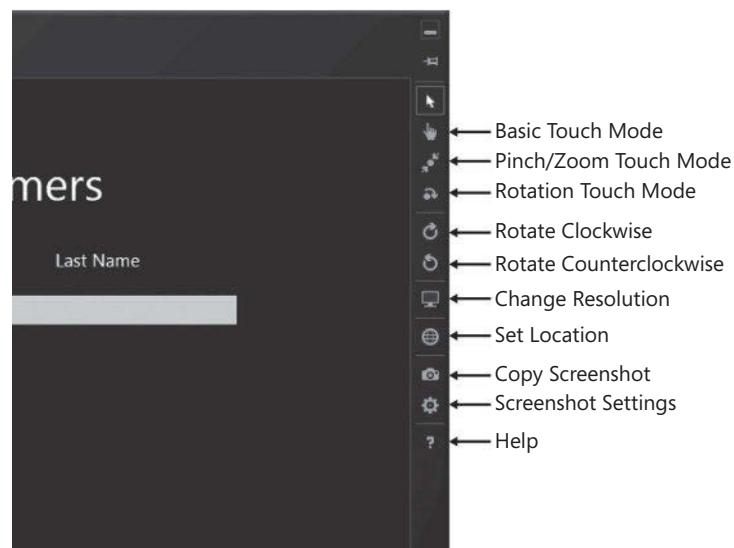
Utilizando o Simulator para testar um aplicativo Windows Store

Mesmo que não tenha um tablet, você ainda pode testar seus aplicativos Windows Store e ver como se comportam em um dispositivo móvel, utilizando o Simulator fornecido com o Visual Studio 2013. O Simulator imita um tablet, oferecendo a capacidade de simular gestos do usuário, como pinçar e deslizar objetos, assim como girar e mudar a resolução do dispositivo.

Para executar um aplicativo no Simulator, na barra de ferramentas do Visual Studio, clique na caixa de listagem Debug Target, imediatamente abaixo do menu Debug. Por padrão, o alvo da depuração é definido como Local Machine, o que faz o aplicativo ser executado em tela inteira no seu computador, mas você pode selecionar Simulator nessa lista, o que inicia o Simulator ao se depurar o aplicativo. Observe que também é possível definir o alvo da depuração com um computador diferente, caso seja necessário realizar depuração remota (quando essa opção for selecionada, será solicitado o endereço de rede do computador remoto). A imagem a seguir mostra a lista Debug Target:

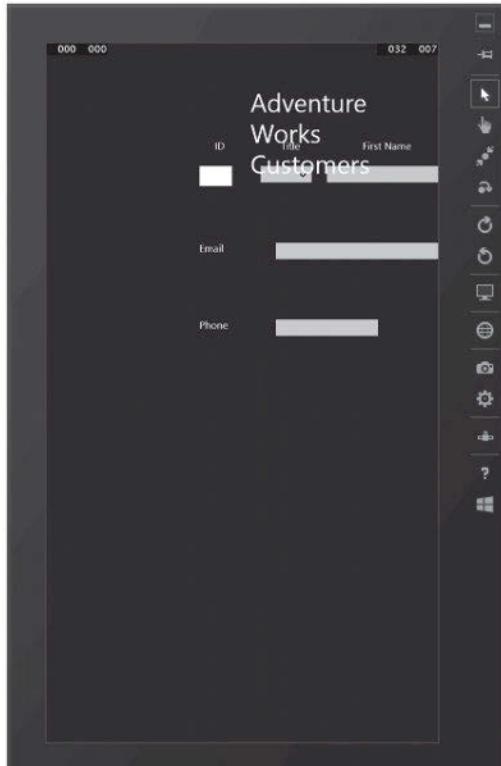


Após selecionar o Simulator, quando o aplicativo é executado a partir do menu Debug no Visual Studio, o Simulator inicia e exibe seu aplicativo. A barra de ferramentas no lado esquerdo da janela Simulator contém uma seleção de ferramentas com as quais é possível imitar os gestos do usuário com o mouse. É possível até simular a localização do usuário, caso o aplicativo exija informações sobre a posição geográfica do dispositivo. Contudo, para testar o layout de um aplicativo, as ferramentas mais importantes são Rotate Clockwise, Rotate Counterclockwise e Change Resolution. A imagem a seguir mostra o aplicativo Customers em execução no Simulator. Os rótulos no lado direito descrevem a função de cada um dos botões do Simulator.

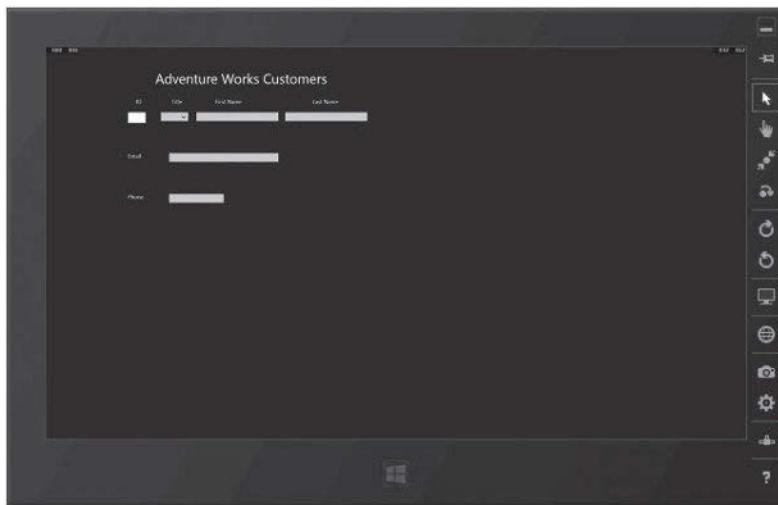


Nota As capturas de tela desta seção foram feitas em um computador com resolução de tela de 1366 × 768. Por padrão, o Simulator começa a executar na mesma resolução de sua tela. Se estiver usando uma resolução de tela diferente, talvez você precise clicar no botão Change Resolution e trocar para 1366 × 768, a fim de obter os mesmos resultados mostrados aqui.

A imagem a seguir mostra o mesmo aplicativo após o usuário ter clicado no botão Rotate Clockwise, fazendo-o executar na orientação retrato:



Você também pode tentar ver como o aplicativo se comporta se mudar a resolução do Simulator. A imagem a seguir mostra o aplicativo Customers em execução quando o Simulator está configurado com resolução alta (2560×1440 , a resolução típica de um monitor de 27 polegadas). Pode-se ver que o aplicativo está comprimido no canto superior esquerdo da tela:

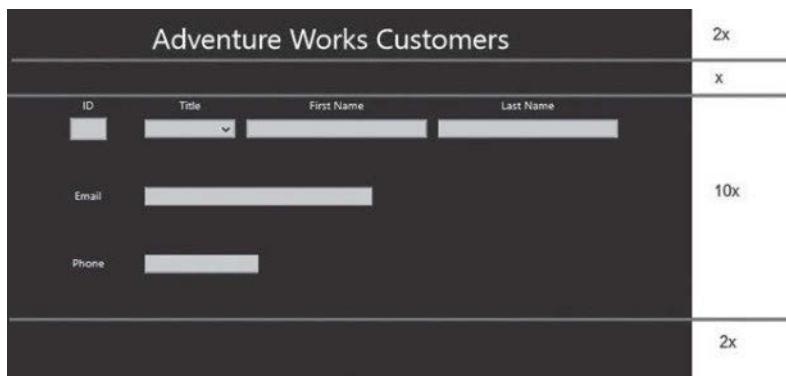


O Simulator se comporta exatamente como um computador Windows 8.1 (na verdade, ele é uma conexão de área de trabalho remota com seu próprio computador). Para parar o Simulator, selecione o charm Settings, clique em Power e, então, clique em Disconnect.

Implemente um layout tabular com um controle Grid

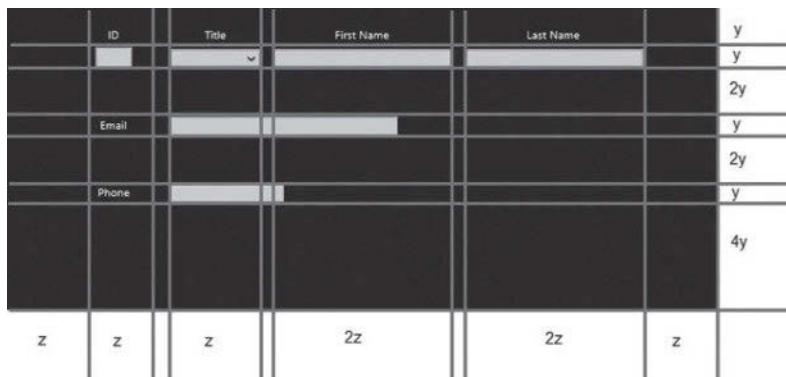
O controle *Grid* pode ser usado para implementar um layout tabular. Ele contém linhas e colunas, e você pode especificar em quais linhas e colunas outros controles devem ser colocados. A vantagem do controle *Grid* é que os tamanhos das linhas e colunas que ele contém podem ser especificados como valores relativos; à medida que a grade aumenta ou diminui para se adaptar aos diferentes tamanhos físicos e orientações para os quais os usuários podem trocar, as linhas e colunas podem aumentar e diminuir proporcionalmente à grade. O cruzamento de uma linha e uma coluna em uma grade define uma célula, e se você posicionar controles em células, eles se moverão à medida que as linhas e colunas aumentarem e diminuírem. Portanto, o segredo para implementar uma interface de usuário escalonável é decompô-la em uma coleção de células e posicionar elementos relacionados na mesma célula. Uma célula pode conter outra grade, oferecendo a capacidade de ajustar o posicionamento exato de cada elemento.

Se você examinar o aplicativo Customers, poderá ver que a interface do usuário é dividida em duas áreas principais: um cabeçalho contendo o título e o corpo contendo os detalhes do cliente. Permitindo algum espaço entre essas áreas e uma margem na parte inferior do formulário, você pode atribuir tamanhos relativos a cada uma dessas áreas, como mostrado no diagrama a seguir:



O diagrama mostra apenas aproximações grosseiras, mas a linha do cabeçalho é duas vezes mais alta que a linha do espaçador debaixo dela. A linha do corpo é dez vezes mais alta do que o espaçador e a margem inferior é duas vezes mais alta do que o espaçador.

Para conter os elementos em cada área, você pode definir uma grade com quatro linhas e posicionar os itens apropriados em cada linha. Entretanto, o corpo do formulário pode ser descrito por outra grade mais complexa, como mostrado no diagrama a seguir:



Novamente, as alturas de cada uma das linhas são especificadas em termos relativos, assim como as larguras das colunas. Além disso, pode-se ver claramente que os elementos *TextBox* para as informações de Email e Phone não se encaixam muito bem nesse padrão de grade. Se você estivesse sendo meticuloso, poderia optar por definir mais grades dentro do corpo do formulário para fazer esses itens caber. Mas deve-se ter em mente a finalidade dessa grade, que é definir o posicionamento relativo e o espaçamento dos elementos. Portanto, é aceitável um elemento ultrapassar os limites de uma célula na disposição em grade.

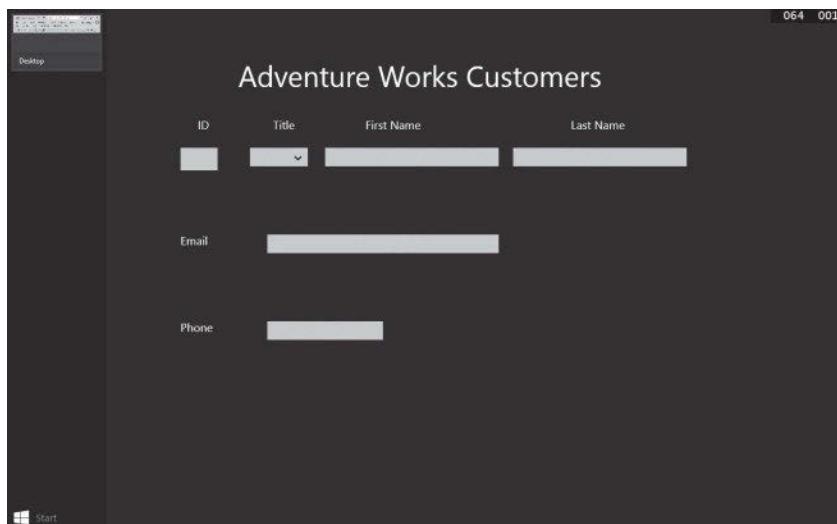
No próximo exercício, você vai modificar o layout do aplicativo Customers a fim de utilizar esse formato de grade para posicionar os controles.

Modifique o layout a fim de escalar para diferentes tamanhos físicos e orientações

1. No painel XAML do aplicativo Customers, adicione outro *Grid* dentro do elemento *Grid* existente. Dê a esse novo *Grid* uma margem de 40 pixels a partir da borda esquerda do *Grid* pai e 54 pixels a partir da parte superior, como mostrado em negrito no código a seguir:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid Margin="40,54,0,0">
        </Grid>
        <TextBlock HorizontalAlignment="Left" TextWrapping="Wrap"
Text="Adventure Works Customers" ... />
        ...
    </Grid>
```

Você poderia definir as linhas e colunas como parte do *Grid* existente, mas para manter a aparência e o comportamento coerentes com outros aplicativos Windows Store, deve deixar algum espaço vazio à esquerda e na parte superior de uma página. Essas margens tornam possível ao Windows 8.1 exibir seus vários ícones e barras de ferramentas, como os ícones que mostram os aplicativos que estão em execução e o menu Iniciar, sem sobreescriver os dados na tela. A imagem a seguir mostra um exemplo:



2. Adicione ao novo elemento *Grid* a seção *<Grid.RowDefinitions>* mostrada em negrito.

```
<Grid Margin="40,54,0,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
</Grid>
```

A seção `<Grid.RowDefinitions>` define as linhas da grade. Neste exemplo foram definidas quatro linhas. Você pode especificar o tamanho de uma linha como um valor absoluto definido em pixels, ou pode usar o operador `*` para indicar que os tamanhos são relativos e que o próprio Windows deve calcular os tamanhos de linha quando o aplicativo for executado, dependendo do tamanho físico e da resolução da tela. Os valores utilizados neste exemplo correspondem aos tamanhos de linha relativos do cabeçalho, do corpo, do espaçador e da margem inferior do formulário *Customers* mostrado no diagrama anterior.

3. Mova o controle *TextBlock* que contém o texto "Adventure Works Customers" para o *Grid*, após a tag `</Grid.RowDefinitions>` de fechamento.

4. Adicione um atributo *Grid.Row* ao controle *TextBlock* e defina o valor como *0*.

Isso indica que o *TextBlock* deve ser posicionado dentro da primeira linha do *Grid* (os controles *Grid* numeram as linhas e colunas a partir de zero).



Nota O atributo *Grid.Row* é um exemplo de propriedade anexada. Uma *propriedade anexada* é aquela que um controle recebe do controle contêiner no qual é colocado. Fora de uma grade, um *TextBlock* não tem uma propriedade *Row* (não faria sentido), mas quando posicionado dentro de uma grade, a propriedade *Row* é anexada ao *TextBlock*, e o controle *TextBlock* pode atribuir um valor a ela. Então, o controle *Grid* utiliza esse valor para determinar onde vai exibir o controle *TextBlock*.

É fácil identificar as propriedades anexadas, pois elas têm a forma *TipoDoContêiner.NomeDaPropriedade*.

5. Remova a propriedade *Margin* e defina as propriedades *HorizontalAlignment* e *VerticalAlignment* como *Center*.

Isso fará com que o *TextBlock* apareça centralizado na linha.

A marcação XAML dos controles *Grid* e *TextBlock* devem ser semelhantes a isto (as alterações no *TextBlock* estão destacadas em negrito):

```
<Grid Margin="40,54,0,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Center" FontSize="50"/>
    ...
</Grid>
```

6. Após o controle *TextBlock*, adicione outro controle *Grid* aninhado. Esse *Grid* será usado para organizar os controles no corpo do formulário e deve aparecer na terceira linha do *Grid* externo (a linha de tamanho *10**); portanto, defina a propriedade *Grid.Row* com *2*, como mostrado em negrito no código a seguir:

```
<Grid Margin="40,54,0,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" .../>
    <Grid Grid.Row="2">
        ...
    </Grid>
</Grid>
```

7. Adicione as seguintes seções *<Grid.RowDefinition>* e *<Grid.ColumnDefinition>* ao novo controle *Grid*:

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="4*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="20"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="20"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="20"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
</Grid>
```

Essas definições de linha e coluna especificam a altura e a largura de cada uma das linhas e colunas mostradas no diagrama anterior, que representava a estrutura do corpo do formulário. Existe um pequeno espaço de 20 pixels entre cada uma das colunas que conterão controles.

8. Mova os controles *TextBlock* que exibem os rótulos ID, Title, Last Name e First Name dentro do controle *Grid* aninhado, para imediatamente após a tag *<Grid.ColumnDefinitions>* de fechamento.
9. Defina a propriedade *Grid.Row* de cada controle *TextBlock* com 0 (esses rótulos aparecerão na primeira linha da grade). Defina a propriedade *Grid.Column* do rótulo ID como 1, a propriedade *Grid.Column* do rótulo Title como 3, a propriedade *Grid.Column* do rótulo First Name como 5, e a propriedade *Grid.Column* do rótulo Last Name como 7.

- 10.** Remova a propriedade *Margin* de cada um dos controles *TextBlock* e defina as propriedades *HorizontalAlignment* e *VerticalAlignment* como *Center*.

- 11.** A marcação XAML desses controles deve ser semelhante a isto (as alterações estão destacadas em negrito):

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="1" HorizontalAlignment="Center"
        TextWrapping="Wrap" Text="ID" VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="3" HorizontalAlignment="Center"
        TextWrapping="Wrap" Text="Title" VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="5" HorizontalAlignment="Center"
        TextWrapping="Wrap" Text="First Name" VerticalAlignment="Center"
        FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="7" HorizontalAlignment="Center"
        TextWrapping="Wrap" Text="Last Name" VerticalAlignment="Center" FontSize="20"/>
</Grid>
```

- 12.** Mova os controles *TextBox*, *id*, *firstName* e *lastName* e o controle *ComboBox title* dentro do controle *Grid* aninhado, para imediatamente após o controle *TextBlock Last Name*.

Posicione esses controles na linha 1 do controle *Grid*. Coloque o controle *id* na coluna 1, o controle *title* na coluna 3, o controle *firstName* na coluna 5 e o controle *lastName* na coluna 7.

Remova a propriedade *Margin* de cada um desses controles e defina a propriedade *VerticalAlignment* como *Center*. Remova a propriedade *Width* e defina a propriedade *HorizontalAlignment* como *Stretch*.

Isso faz o controle ocupar a célula inteira ao ser exibido, e o controle aumenta ou diminui, conforme o tamanho da célula muda.

A marcação XAML completa desses controles deve ser semelhante a isto, com as alterações destacadas em negrito:

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    ...
    <TextBlock Grid.Row="0" Grid.Column="7" ... Text="Last Name" .../>
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" HorizontalAlignment="Stretch"
        TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"
        IsReadOnly="True"/>
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName"
        HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
        VerticalAlignment="Center"
        FontSize="20"/>
    <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName"
```

```

HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" HorizontalAlignment="Stretch"
VerticalAlignment="Center" FontSize="20">
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>
</ComboBox>
</Grid>

```

- 13.** Mova o controle *TextBlock* do rótulo *Email* e o controle *TextBox email* para o controle *Grid* aninhado, para imediatamente após a tag de fechamento do controle *ComboBox*.

Posicione esses controles na linha 3 do controle *Grid*. Coloque o rótulo *Email* na coluna 1 e o controle *TextBox email* na coluna 3. Além disso, defina a propriedade *Grid.ColumnSpan* do controle *TextBox email* como 3; dessa maneira, a coluna pode se expandir até o valor especificado por sua propriedade *Width* ao longo de três colunas, como mostrado no diagrama anterior.

Defina a propriedade *HorizontalAlignment* do controle rótulo *Email* como *Center*, mas deixe a propriedade *HorizontalAlignment* do *TextBox email* definida como *Left*; esse controle deve permanecer alinhado à esquerda em relação à primeira coluna que abrange, em vez de ser centralizado em relação a todas elas.

Defina a propriedade *VerticalAlignment* do rótulo *Email* e do controle *TextBox email* como *Center*.

Remova a propriedade *Margin* desses dois controles.

A marcação XAML a seguir mostra as definições completas desses controles:

```

<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    ...
    <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" HorizontalAlignment="Stretch"
VerticalAlignment="Center" FontSize="20">
        ...
    </ComboBox>
    <TextBlock Grid.Row="3" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Email" VerticalAlignment="Center" FontSize="20"/>
    <TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="3" x:Name="email"
HorizontalAlignment="Left" TextWrapping="Wrap" Text="" VerticalAlignment="Center"
Width="400" FontSize="20"/>
</Grid>

```

- 14.** Mova o controle *TextBlock* do rótulo *Phone* e o controle *TextBox phone* para o controle *Grid* aninhado, imediatamente após o controle *TextBox email*.

Posicione esses controles na linha 5 do controle *Grid*. Coloque o rótulo *Phone* na coluna 1 e o controle *TextBox phone* na coluna 3. Defina a propriedade *Grid.ColumnSpan* do controle *TextBox phone* como 3.

Defina a propriedade *HorizontalAlignment* do controle rótulo Phone com *Center* e deixe a propriedade *HorizontalAlignment* do *TextBox phone* definida com *Left*.

Defina a propriedade *VerticalAlignment* dos dois controles com *Center* e remova a propriedade *Margin*.

A marcação XAML a seguir mostra as definições completas desses controles:

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    ...
    <TextBox ... x:Name="email" .../>
    <TextBlock Grid.Row="5" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Phone" VerticalAlignment="Center" FontSize="20"/>
    <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3" x:Name="phone"
HorizontalAlignment="Left" TextWrapping="Wrap" Text=""
VerticalAlignment="Center"
Width="200" FontSize="20"/>
</Grid>
```

15. Na barra de ferramentas do Visual Studio, na lista Debug Target, selecione Simulator.

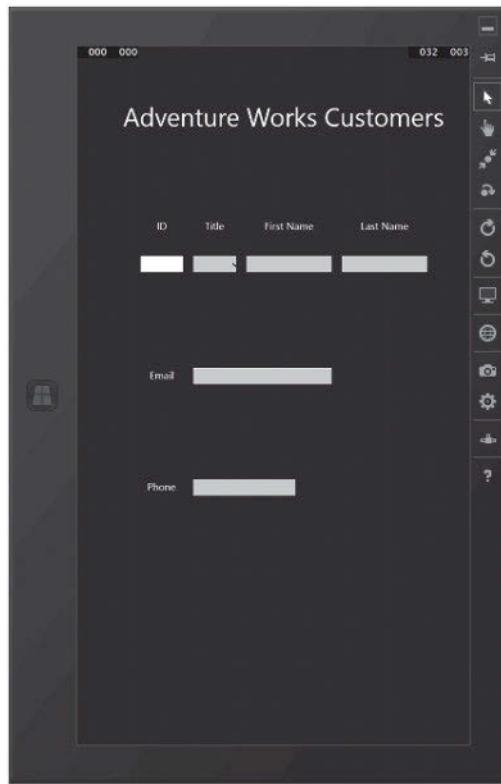
Você vai executar o aplicativo no Simulator para ver como o layout se adapta em diferentes resoluções e tamanhos físicos.

16. No menu Debug, clique em Start Debugging.

O Simulator inicia e o aplicativo Customers é executado. Clique em Change Resolution e configure o Simulator para exibir o aplicativo utilizando uma resolução de tela de 1366 × 768. Além disso, certifique-se de que o Simulator seja exibido na orientação paisagem (clique em Rotate Clockwise, se ele estiver executando na orientação retrato). Verifique que os controles estão igualmente espaçados nessa orientação.

17. Clique no botão Rotate Clockwise a fim de girar o Simulator para a orientação retrato.

O aplicativo Customers deve ajustar o layout da interface do usuário e os controles ainda devem estar igualmente espaçados e utilizáveis:

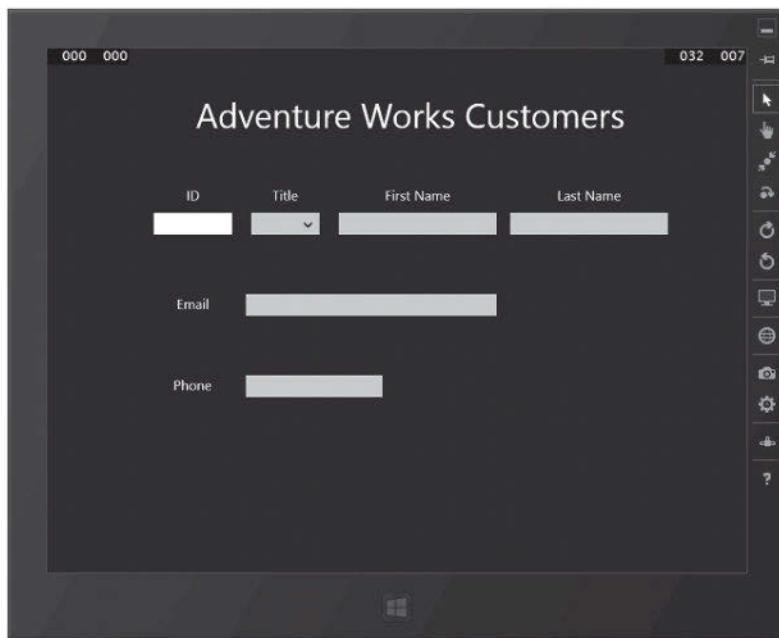


- 18.** Clique em Rotate Counterclockwise para colocar o Simulator de volta na orientação paisagem e, então, clique em Change Resolution e troque a resolução do Simulator para 2560 × 1400.

Observe que os controles permanecem igualmente espaçados no formulário, embora possa ser difícil ler os rótulos, a não ser que você tenha uma tela de 27 polegadas.

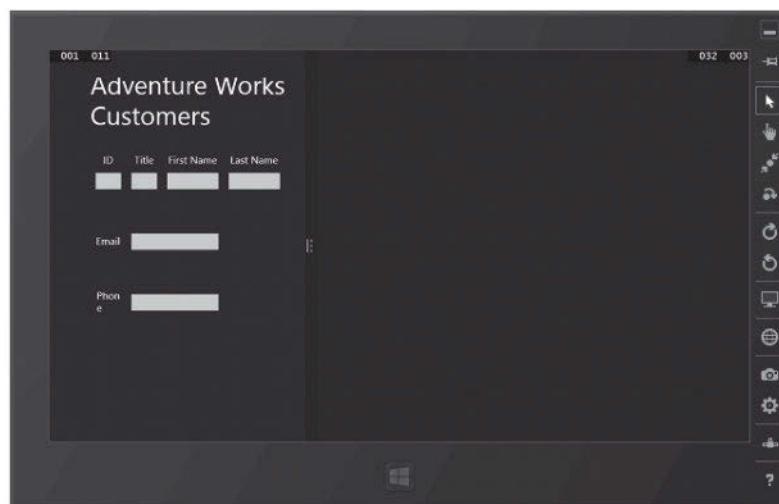
- 19.** Clique novamente em Change Resolution e troque a resolução para 1024 × 768.

Novamente, observe como o espaçamento e o tamanho dos controles são ajustados para manter o equilíbrio uniforme da interface do usuário:



20. Clique novamente em Change Resolution e volte para a resolução de 1366 × 768.
21. Clique na borda superior do formulário e arraste-a de modo que ele seja exibido na metade esquerda da tela e, então, arraste a barra no meio da tela a fim de reduzir o tamanho da visualização para sua largura mínima (500 pixels).

Todos os controles permanecem visíveis, mas o texto do rótulo Phone muda de linha, tornando-se difícil de ler, e os controles não são particularmente fáceis de usar:



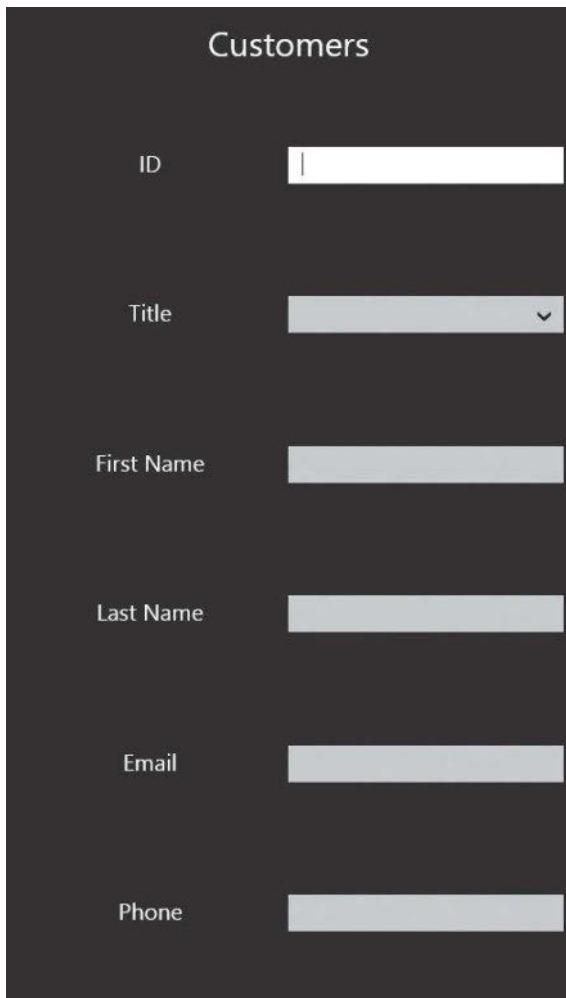
- 22.** No Simulator, exiba os charms (pressione Windows+C), clique em Settings, clique em Power e, então, clique em Disconnect.

O Simulator se fecha e você retorna ao Visual Studio.

- 23.** Na barra de ferramentas do Visual Studio, na lista Debug Target, selecione Local Machine.

Adapte o layout com o Visual State Manager

A interface do usuário do aplicativo Customers muda de escala para diferentes resoluções e tamanhos físicos, mas ainda não funciona bem se você reduz a largura da visualização. Além disso, provavelmente não ficará muito boa em um smartphone, que tem largura ainda mais estreita. Se você pensar a respeito, nesses casos o problema não é tanto uma questão de mudar a escala dos controles, mas sim organizá-los de uma maneira diferente. Por exemplo, faria mais sentido se o formulário Customers aparecesse assim em uma visualização estreita:



Esse efeito pode ser conseguido com o Visual State Manager. Todos os aplicativos Windows Store implementam um Visual State Manager, que controla o estado visual de um aplicativo. Ele pode detectar quando o aplicativo alterna entre diferentes visualizações. Você pode capturar essas transições de estado visual e utilizá-las para animar a interface do usuário – para mover controles ou para exibi-los e ocultá-los, por exemplo. É isso que você fará nos próximos exercícios. O primeiro passo é definir um layout para os dados do cliente que devem aparecer em uma visualização estreita.

Defina um layout para a visualização estreita

1. No painel XAML do aplicativo Customers, adicione ao controle *Grid* as propriedades *x:Name* e *Visibility*, mostradas em negrito no código a seguir, que definem o layout tabular dos vários controles:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid x:Name="customersTabularView" Margin="40,54,0,0" Visibility="Collapsed">
        ...
    </Grid>
</Grid>
```

Você vai fazer referência a esse controle *Grid* em outra marcação XAML, mais adiante neste conjunto de exercícios, daí a necessidade de lhe dar um nome. A propriedade *Visibility* especifica se o controle é exibido (*Visible*) ou oculto (*Collapsed*). O valor padrão é *Visible*, mas por enquanto você vai ocultar esse *Grid*, enquanto define outro para exibir os dados em um formato colunar.

2. Após a tag *</Grid>* de fechamento do controle *Grid* *customersTabularView*, adicione outro controle *Grid*. Defina a propriedade *x:Name* com *customersColumnarView*, a propriedade *Margin* com *20,10,20,10* e a propriedade *Visibility* com *Visible*.



Dica É possível expandir e contrair elementos no painel XAML da janela Design View e tornar a estrutura mais fácil de ler, clicando nos sinais + e – que aparecem na margem esquerda da marcação XAML.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid x:Name="customersTabularView" Margin="40,54,0,0" Visibility="Collapsed">
        ...
    </Grid>
    <Grid x:Name="customersColumnarView" Margin="10,20,10,20"
          Visibility="Visible">
        ...
    </Grid>
</Grid>
```

3. No controle *Grid* *customersColumnarView*, adicione as seguintes definições de linha:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
    </Grid.RowDefinitions>
</Grid>
```

Você vai usar a linha superior para exibir o título e a segunda linha, muito mais larga, para exibir os controles nos quais o usuário insere dados.

4. Após as definições de linha, adicione o controle *TextBlock* a seguir. Esse controle exibe um título truncado, Customers, na primeira linha do controle *Grid*. Defina *FontSize* com 30.

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20"
      Visibility="Visible">
    <Grid.RowDefinitions>
      ...
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
              Text="Customers" VerticalAlignment="Center" FontSize="30"/>
  </Grid>
```

5. Adicione outro controle *Grid* na linha 1 do controle *Grid customersColumnarView*, imediatamente após o controle *TextBlock* que contém o título *Customers*. Esse controle *Grid* exibirá os rótulos e os controles de entrada de dados em duas colunas; portanto, adicione a esse *Grid* as definições de linha e colunas mostradas no exemplo de código a seguir.

```
<TextBlock Grid.Row="0" ... />
<Grid Grid.Row="1">
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
</Grid>
```

Observe que, se todas as linhas ou colunas de um conjunto têm a mesma altura ou largura, você não precisa especificar seus tamanhos.

6. Copie a marcação XAML dos controles *TextBlock* ID, Title, First Name e Last Name do controle *Grid customersTabularView* para o novo controle *Grid*, imediatamente após as definições de linha que você acabou de adicionar. Coloque o controle ID na linha 0, o controle Title na linha 1, o controle First Name na linha 2 e o controle Last Name na linha 3. Coloque todos os controles na coluna 0.

```
<Grid.RowDefinitions>
  ...
</Grid.RowDefinitions>
<TextBlock Grid.Row="0" Grid.Column="0" HorizontalAlignment="Center"
          TextWrapping="Wrap" Text="ID" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center"
          TextWrapping="Wrap" Text="Title" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="2" Grid.Column="0" HorizontalAlignment="Center"
          TextWrapping="Wrap" Text="First Name" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="3" Grid.Column="0" HorizontalAlignment="Center"
          TextWrapping="Wrap" Text="Last Name" VerticalAlignment="Center" FontSize="20"/>
```

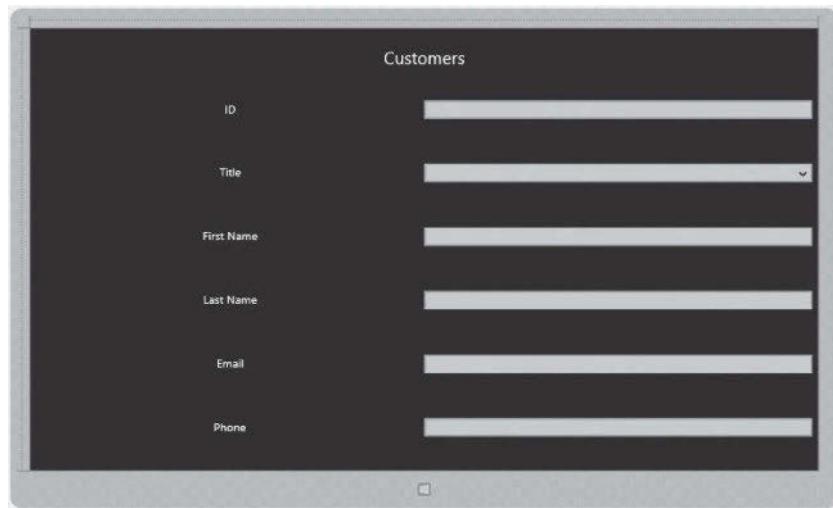
7. Copie a marcação XAML dos controles *TextBox id, title, firstName* e *lastName* e dos controles *ComboBox* do controle *Grid customersTabularView* para o novo controle *Grid*, imediatamente após os controles *TextBox*. Coloque o controle *id* na linha 0, o controle *title* na linha 1, o controle *firstName* na linha 2 e o controle *lastName* na linha 3. Coloque todos os quatro controles na coluna 1. Além disso, mude os nomes de todos os controles, prefixando-os com a letra *c* (de coluna). Essa última alteração é necessária para evitar conflito com os nomes dos controles existentes no controle *Grid customersTabularView*.

```
<TextBlock Grid.Row="3" Grid.Column="0" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Last Name" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20" IsReadOnly="True"/>
<TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="3" Grid.Column="1" x:Name="cLastName" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" HorizontalAlignment="Stretch"
VerticalAlignment="Center" FontSize="20">
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>
</ComboBox>
```

8. Copie os controles *TextBlock* e *TextBox* do endereço de e-mail e número de telefone do controle *Grid customersTabularView* para o novo controle *Grid*, após o controle *ComboBox cTitle*. Coloque os controles *TextBlock* da coluna 0 e os controles *TextBox* da coluna 1 nas linhas 4 e 5. Mude o nome do controle *TextBox email* para *cEmail* e o nome do controle *TextBox phone* para *cPhone*. Remova as propriedades *Width* dos controles *cEmail* e *cPhone*, e defina suas propriedades *HorizontalAlignment* como *Stretch*.

```
<ComboBox ...>
    ...
</ComboBox>
<TextBlock Grid.Row="4" Grid.Column="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Email" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="5" Grid.Column="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Phone" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
```

A janela Design View deve exibir o layout colunar como segue:



9. Retorne à marcação XAML do controle *Grid customersTabularView* e defina a propriedade *Visibility* com *Visible*.

```
<Grid x:Name="customersTabularView" Margin="40,54,0,0" Visibility="Visible">
```

10. Na marcação XAML do controle *Grid customersColumnarView*, defina a propriedade *Visibility* como *Collapsed*.

```
<Grid x:Name="customersColumnarView" Margin="20,10,20,10" Visibility="Collapsed">
```

A janela Design View deve exibir o layout tabular original do formulário Customers. Essa é a visualização padrão que será utilizada pelo aplicativo.

Agora você definiu o layout que aparecerá na visualização estreita. Você pode estar preocupado pelo fato de que, basicamente, tudo que fez foi duplicar muitos dos controles e organizá-los de uma maneira diferente. Se você executar o formulário e alternar entre as visualizações, como os dados de uma visualização serão transferidos para a outra? Por exemplo, se você inserir os detalhes de um cliente quando o aplicativo estiver executando em tela inteira e depois trocar para a visualização estreita, os controles recentemente exibidos não conterão os mesmos dados que acabaram de ser inseridos. Os aplicativos Windows Store tratam desse problema utilizando *vinculação de dados*. Essa técnica permite associar os mesmos dados a vários controles e, quando os dados mudam, todos os controles exibem as informações atualizadas. Vamos ver como isso funciona no Capítulo 26. Por enquanto, você precisa pensar em como vai usar o Visual State Manager para alternar entre os layouts quando a visualização mudar.

Todo aplicativo Windows Store tem um Visual State Manager. Seu objetivo é responder às mudanças no estado visual e atualizar o layout da interface do usuário. Para indicar uma mudança no estado visual, use o método *GoToState* do objeto *VisualStateManager*. As alterações feitas no layout pelo Visual State Manager podem ser especificadas pela implementação de uma série de transições de estado visual na marcação XAML de seu aplicativo. É isso que você fará no próximo exercício.

Utilize o Visual State Manager para modificar o layout

1. No painel XAML do aplicativo Customers, após a tag `</Grid>` de fechamento do controle `Grid customersColumnarView`, adicione a seguinte marcação:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    ...
</Grid>
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout"/>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Você define as transições de estado visual implementando um ou mais grupos de estado visual. Cada grupo de estado visual especifica as transições que devem ocorrer quando o Visual State Manager trocar para o estado visual descrito. Neste caso, as ações padrão ocorrerão quando o Visual State Manager trocar para o estado `TabularLayout` (esse estado indicará que o aplicativo está executando em uma janela larga; você vai definir esse estado em breve). Você já experimentou essas ações padrão – os controles ajustam suas larguras e posições relativas entre si, de acordo com as definições das várias linhas e colunas nos controles `Grid` que os contêm.

2. Adicione ao grupo de estado visual a seguinte transição de estado visual mostrada em negrito:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout"/>
        <VisualState x:Name="ColumnarLayout">
            <Storyboard>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName=
"customersTabularView" Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName=
"customersColumnarView" Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
                </ObjectAnimationUsingKeyFrames>
            </Storyboard>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Essa transição ocorrerá quando o aplicativo trocar para o estado `ColumnarLayout`, no momento em que o usuário redimensioná-lo para exibição em uma janela estreita (novamente, você vai definir esse estado em um passo posterior neste exercício).

Uma transição é descrita em termos de um quadro (storyboard) de animação. A animação em aplicativos Windows Store é um assunto enorme; infelizmente, não há espaço neste capítulo para discutirmos como ela funciona, mas o ponto importante a extrair desse código é que essa transição contém duas animações: a primeira muda a propriedade `Visibility` do controle `Grid customersTabularView` para `Collapsed` e a segunda muda a propriedade `Visibility` do controle `Grid customersColumnarView` para `Visible`.



Nota Para obter mais informações sobre o uso de animações em aplicativos Windows Store, visite a página "Quickstart: Animating your UI" no site da Microsoft, em <http://msdn.microsoft.com/library/windows/apps/xaml/hh452703.aspx>.

3. No Solution Explorer, expanda MainPage.xaml e clique duas vezes em MainPage.xaml.cs para exibir o código do formulário MainPage na janela Code and Text Editor.
4. Adicione o seguinte método à classe *MainPage*:

```
void WindowSizeChanged(object sender, Windows.UI.Core.WindowSizeChangedEventArgs e)
{
    if (e.Size.Width <= 750)
    {
        VisualStateManager.GoToState(this, "ColumnarLayout", false);
    }
    else
    {
        VisualStateManager.GoToState(this, "TabularLayout", false);
    }
}
```

Essa é uma rotina de tratamento de eventos que será executada quando a janela do aplicativo mudar de tamanho. O parâmetro *WindowSizeChangedEventArgs* contém a propriedade *Size*, a qual indica o novo tamanho da janela. A propriedade *Width* especifica a largura da janela. O código nessa rotina de tratamento de eventos consulta a propriedade *Width* e chama o método estático *GoToState* da classe *VisualStateManager*, especificando o estado para o qual o Visual State Manager deve mudar a janela. O método *GoToState* dispara uma transição no objeto especificado pelo primeiro argumento (neste caso, o formulário *Customers*), utilizando o nome de estado visual especificado pelo segundo argumento. Neste caso, se a largura da janela for menor que 750 pixels, o aplicativo trocará para o estado *ColumnarLayout*; caso contrário, mudará para o estado *TabularLayout*. Observe que os nomes de estado são simplesmente valores de string. Você pode ignorar o terceiro argumento booleano.



Nota O formulário muda para o layout colunar quando a largura cai para 750 pixels, em vez dos 500 pixels que é a largura mínima padrão de uma janela. Isso porque em larguras menores que 750 pixels, os controles e rótulos do formulário *Customers* começam a mudar de linha e se tornam difíceis de usar.

5. No construtor de *MainPage*, adicione a instrução mostrada em negrito a seguir:

```
public MainPage()
{
    this.InitializeComponent();
    Window.Current.SizeChanged += WindowSizeChanged;
}
```

Esse código realiza a inscrição do evento *SizeChanged* da janela atual; quando esse evento ocorre, o método *WindowSizeChanged* definido no passo anterior é executado.

6. No menu Debug, clique em Start Debugging.

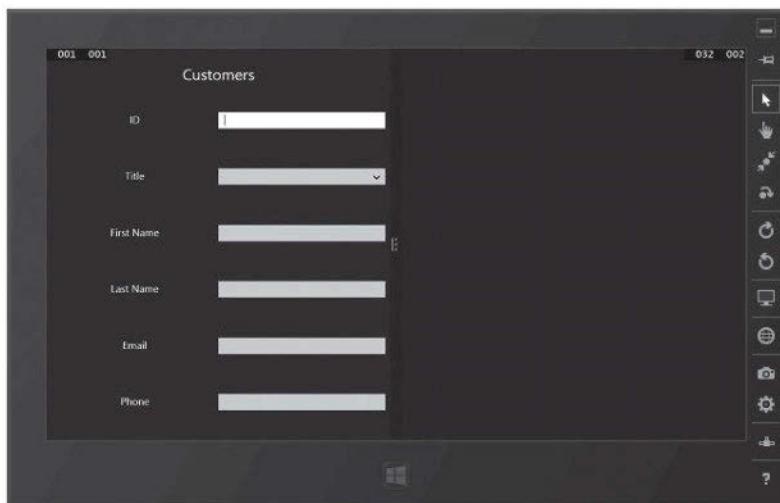
O aplicativo inicia e exibe o formulário Customer em tela inteira. Os dados são exibidos no layout tabular.



Nota Se você estiver usando uma tela com resolução de menos de 1366 × 768, inicie o aplicativo executando no Simulator como descrito anteriormente. Configure o Simulator com uma resolução de 1366 × 768.

7. Arraste a borda superior da janela que exibe o aplicativo Customer para mostrá-lo em uma visualização estreita.

Desta vez, os dados são exibidos no layout colunar.



8. Redimensione a janela que exibe o aplicativo Customer para mostrá-lo em execução em tela inteira.

O formulário Customer volta para o layout tabular.

9. Retorne ao Visual Studio e interrompa a depuração.

Aplique estilos em uma interface de usuário

Agora que você tem a mecânica do layout básico do aplicativo resolvida, o próximo passo é aplicar alguma estilização para tornar a interface do usuário mais atraente. Os controles de um aplicativo Windows Store têm uma gama variada de propriedades que podem ser utilizadas para alterar características, como a fonte, cor, tamanho e outros atributos de um elemento. Essas propriedades podem ser definidas individualmente para cada controle, mas essa estratégia pode se tornar enfadonha e repetitiva, caso você precise aplicar o mesmo estilo a vários controles. Além disso, os melhores aplicativos utilizam estilos uniformes em toda a interface do usuário, e é difícil manter

essa uniformidade se você tem de definir repetidamente as mesmas propriedades e valores ao adicionar ou alterar controles. Quanto mais vezes você tiver de fazer a mesma coisa, maiores as chances de errar pelo menos uma vez!

Nos aplicativos Windows Store, você pode definir estilos reutilizáveis. Você pode implementá-los como recursos em nível de aplicativo, criando um dicionário de recursos; então, eles ficam disponíveis para todos os controles em todas as páginas de um aplicativo. Também é possível definir recursos locais, empregados em apenas uma página, na marcação XAML dessa página. No próximo exercício, você vai definir alguns estilos simples para o aplicativo Customers e utilizar esses estilos nos controles do formulário Customers.

Defina estilos para o formulário Customers

1. No Solution Explorer, clique com o botão direito do mouse no projeto Customers, aponte para Add e clique em New Item.
2. Na caixa de diálogo Add New Item – Customers, clique em Resource Dictionary. Na caixa Name, digite **AppStyles.xaml** e clique em Add.

O arquivo AppStyles.xaml aparece na janela Code and Text Editor. Um dicionário de recursos é um arquivo XAML que contém recursos que o aplicativo pode utilizar. O arquivo AppStyles.xaml é parecido com este:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers">

</ResourceDictionary>
```

Os estilos são um exemplo de recurso, mas você também pode adicionar outros itens. Na verdade, o primeiro recurso que você vai adicionar não é realmente um estilo, mas um *ImageBrush*, que será utilizado para pintar o fundo do controle *Grid* externo no formulário Customers.

3. No Solution Explorer, clique com o botão direito do mouse no projeto Customers, aponte para Add e clique em New Folder. Mude o nome da nova pasta para **Images**.
4. Clique com o botão direito do mouse na pasta Images, aponte para Add e então clique em Existing Item.
5. Na caixa de diálogo Add Existing Item – Customers, acesse a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 25\Resources na sua pasta Documentos, clique em wood.jpg e, então, em Add.

O arquivo wood.jpg é adicionado à pasta Images no projeto Customers. Esse arquivo contém a imagem de um fundo elegante de madeira, que você vai adicionar ao formulário Customers.

6. Na janela Code and Text Editor que exibe o arquivo AppStyles.xaml, adicione a marcação XAML mostrada em negrito a seguir:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

```

xmlns:local="using:Customers">

<ImageBrush x:Key="WoodBrush" ImageSource="Images/wood.jpg"/>
</ResourceDictionary>

```

Essa marcação cria um recurso *ImageBrush* chamado *WoodBrush* que é baseado no arquivo *wood.jpg*. Você pode usar esse pincel de imagem para definir o fundo de um controle, e ele exibirá a imagem do arquivo *wood.jpg*.

- Sob o recurso *ImageBrush*, adicione ao arquivo *AppStyles.xaml* o estilo mostrado em negrito a seguir:

```

<ResourceDictionary
  ...>

  <ImageBrush x:Key="WoodBrush" ImageSource="Images/wood.jpg"/>
  <Style x:Key="GridStyle" TargetType="Grid">
    <Setter Property="Background" Value="{StaticResource WoodBrush}"/>
  </Style>
</ResourceDictionary>

```

Essa marcação mostra como definir um estilo. Um elemento *Style* deve ter um nome (para que possa ser referenciado em outra parte do aplicativo) e especificar o tipo de controle ao qual o estilo pode ser aplicado. Você vai utilizar esse estilo com o controle *Grid*.

O corpo de um estilo consiste em um ou mais elementos *Setter*. Um elemento *Setter* especifica a propriedade a ser definida e o valor com que deve ser definida. Nesse exemplo, a propriedade *Background* é definida como o recurso *ImageBrush WoodBrush*. Contudo, a sintaxe é um pouco curiosa. Em um *value*, você pode fazer referência a um dos valores adequados, definidos pelo sistema, para a propriedade (como *"Red"*, se quiser definir o fundo com uma cor vermelha uniforme), ou especificar um recurso que definiu. Para referenciar um recurso, use a palavra-chave *StaticResource* e, então, coloque a expressão inteira entre chaves.

- Antes de utilizar esse estilo, atualize o dicionário de recursos global do aplicativo no arquivo *App.xaml* e adicione uma referência ao arquivo *AppStyles.xaml*. No Solution Explorer, clique duas vezes em *App.xaml* para exibi-lo na janela Code and Text Editor. O arquivo *App.xaml* é parecido com este:

```

<Application
  x:Class="Customers.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Customers">

</Application>

```

Atualmente, o aplicativo *App.xaml* só define o objeto *app* e coloca alguns namespaces no escopo; o dicionário de recursos global está vazio.

- No arquivo *Add.xaml*, adicione o código mostrado aqui em negrito:

```

<Application
  x:Class="Customers.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Customers">
  <Application.Resources>

```

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="AppStyles.xaml"/>
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
</Application.Resources>
</Application>
```

Essa marcação adiciona os recursos definidos no arquivo AppStyles.xaml à lista de recursos disponíveis no dicionário de recursos global. Agora esses recursos estão disponíveis para uso por todo o aplicativo.

- 10.** Troque para o arquivo MainPage.xaml que exibe a interface do usuário para o formulário Customers. No painel XAML, localize o controle *Grid* externo:

```
<Grid Background="{StaticResource ApplicationBackgroundThemeBrush}">
```

Na marcação XAML desse controle, substitua a propriedade *Background* por uma propriedade *Style* que referencia o estilo *GridStyle*, como mostrado em ne-grito no código a seguir:

```
<Grid Style="{StaticResource GridStyle}">
```

O fundo do controle *Grid* na janela Design View deve ser trocado e exibir um painel de madeira, como este:





Nota Em condições ideais, você deve certificar-se de que qualquer imagem de fundo aplicada a uma página ou controle mantenha sua estética quando o tamanho físico e a orientação do dispositivo mudarem. Uma imagem que aparece bem em um monitor de 30 polegadas pode ficar distorcida e achatada em um aparelho Windows Phone. Talvez seja necessário fornecer fundos alternativos para diferentes visualizações e orientações, e utilizar o Visual State Manager para modificar a propriedade *Background* de um controle, para alternar entre eles quando o estado visual mudar.

11. Volte para AppStyles.xaml na janela Code and Text Editor e adicione o seguinte estilo *FontStyle*, após o estilo *GridStyle*:

```
<Style x:Key="GridStyle" TargetType="Grid">
    ...
</Style>
<Style x:Key="FontStyle" TargetType="TextBlock">
    <Setter Property="FontFamily" Value="Buxton Sketch"/>
</Style>
```

Esse estilo é aplicado a elementos *TextBlock* e muda a fonte para Buxton Sketch. Essa fonte lembra um estilo de escrita à mão.

Neste estágio, seria possível referenciar o estilo *FontStyle* em todo controle *TextBlock* que exigisse essa fonte, mas essa estratégia não ofereceria vantagem em relação a simplesmente definir a *FontFamily* diretamente na marcação de cada controle. O real poder dos estilos ocorre quando várias propriedades são combinadas, conforme vamos ver nos próximos passos.

12. Adicione o estilo *HeaderStyle*, mostrado aqui, ao arquivo AppStyles.xaml:

```
<Style x:Key="FontStyle" TargetType="TextBlock">
    ...
</Style>
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="Foreground" Value="SteelBlue"/>
</Style>
```

Esse é um estilo composto que define as propriedades *HorizontalAlignment*, *TextWrapping*, *VerticalAlignment* e *Foreground* de um *TextBlock*. Além disso, o estilo *HeaderStyle* referencia o estilo *FontStyle* utilizando a propriedade *BasedOn*. A propriedade *BasedOn* oferece uma forma simples de herança para estilos.

Você usará esse estilo para formatar os rótulos que aparecem na parte superior dos controles *customersTabularGrid* e *customersColumnarGrid*. Contudo, esses cabeçalhos têm tamanhos de fonte diferentes (o cabeçalho do layout tabular é maior que o do layout colunar), de modo que você vai criar mais dois estilos que estendem o estilo *HeaderStyle*.

13. Adicione os seguintes estilos ao arquivo AppStyles.xaml:

```

<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
    ...
</Style>
<Style x:Key="TabularHeaderStyle" TargetType="TextBlock"
BasedOn="{StaticResource HeaderStyle}">
    <Setter Property="FontSize" Value="70"/>
</Style>

<Style x:Key="ColumnarHeaderStyle" TargetType="TextBlock"
BasedOn="{StaticResource HeaderStyle}">
    <Setter Property="FontSize" Value="50"/>
</Style>

```

Observe que os tamanhos de fonte desses estilos são um pouco maiores que os tamanhos de fonte atualmente utilizados pelos cabeçalhos nos controles *Grid*. Isso acontece porque a fonte Buxton Sketch é menor que a fonte padrão.

14. Volte ao arquivo MainPage.xaml e localize a marcação XAML do controle *TextBlock* para o rótulo Adventure Works Customers no controle *Grid customersTabularView*:

```
<TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Center" FontSize="50"/>
```

15. Altere as propriedades desse controle para referenciar o estilo *TabularHeaderStyle*, como mostrado em negrito no código a seguir:

```
<TextBlock Grid.Row="0" Style="{StaticResource TabularHeaderStyle}"
Text="Adventure Works Customers"/>
```

O cabeçalho exibido na janela Design View deve mudar de cor, tamanho e fonte, e ser semelhante a isto:



16. Localize a marcação XAML do controle *TextBlock* para o rótulo Customers no controle *Grid customersColumnarView*:

```
<TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Customers" VerticalAlignment="Center" FontSize="30"/>
```

Modifique a marcação desse controle para referenciar o estilo *ColumnarHeaderStyle*, como mostrado aqui em negrito:

```
<TextBlock Grid.Row="0" Style="{StaticResource ColumnarHeaderStyle}"
Text="Customers"/>
```

Saiba que você não vai ver essa mudança na janela Design View, pois o controle *Grid customersColumnarView* é contraído por padrão. Contudo, você vai ver os efeitos dessa mudança quando executar o aplicativo mais adiante neste exercício.

17. Retorne ao arquivo AppStyles.xaml na janela Code and Text Editor. Modifique o estilo *HeaderStyle* com os elementos *Setter* de propriedade adicional, mostrados em negrito no exemplo a seguir:

```
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="Foreground" Value="SteelBlue"/>
    <Setter Property="RenderTransformOrigin" Value="0.5,0.5"/>
    <Setter Property="RenderTransform">
        <Setter.Value>
            <CompositeTransform Rotation="-5"/>
        </Setter.Value>
    </Setter>
</Style>
```

Esses elementos giram o texto exibido no cabeçalho em torno de seu ponto central, por um ângulo de 5 graus, utilizando uma transformação.



Nota Esse exemplo mostra uma transformação simples. Com a propriedade *RenderTransform*, você pode realizar diversas outras transformações em um item, e é possível combinar várias transformações. Por exemplo, você pode transladar (mover) um item nos eixos x e y, inclinar o item e mudar a escala de um elemento. Para obter mais informações, consulte <http://msdn.microsoft.com/library/system.windows.uielement.rendertransform.aspx>.

Observe também que o valor da propriedade *RenderTransform* é ele próprio outro par propriedade/valor (a propriedade é *Rotation* e o valor é -5). Em casos como esse, você especifica o valor utilizando a tag *<Setter.Value>*.

18. Troque para o arquivo MainPage.xaml. Na janela Design View, o título deve agora aparecer em um ângulo vistoso:



19. No arquivo AppStyles.xaml, adicione o seguinte estilo:

```
<Style x:Key="LabelStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
    <Setter Property="FontSize" Value="30"/>
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="Foreground" Value="AntiqueWhite"/>
</Style>
```

Você vai aplicar esse estilo aos elementos *TextBlock* que fornecem os rótulos para os vários controles *TextBox* e *ComboBox* empregados pelo usuário para inserir informações do cliente. O estilo referencia o mesmo estilo de fonte dos cabeçalhos, mas define as outras propriedades com valores mais adequados para rótulos.

- 20.** Volte para o arquivo MainPage.xaml. No painel XAML, modifique a marcação dos controles *TextBlock* de cada um dos rótulos nos controles *Grid customersTabularView* e *customersColumnarView*, remova as propriedades *HorizontalAlignment*, *TextWrapping*, *VerticalAlignment* e *FontSize*, e referencie o estilo *LabelStyle*, como mostrado aqui em negrito:

```
<Grid x:Name="customersTabularView" Margin="40,54,0,0" Visibility="Visible">
    ...
    <Grid Grid.Row="2">
        ...
        <TextBlock Grid.Row="0" Grid.Column="1" Style="{StaticResource LabelStyle}"
Text="ID"/>
        <TextBlock Grid.Row="0" Grid.Column="3" Style="{StaticResource LabelStyle}"
Text="Title"/>
        <TextBlock Grid.Row="0" Grid.Column="5" Style="{StaticResource LabelStyle}"
Text="First Name"/>
        <TextBlock Grid.Row="0" Grid.Column="7" Style="{StaticResource LabelStyle}"
Text="Last Name"/>
        ...
        <TextBlock Grid.Row="3" Grid.Column="1" Style="{StaticResource LabelStyle}"
Text="Email"/>
        ...
        <TextBlock Grid.Row="5" Grid.Column="1" Style="{StaticResource LabelStyle}"
Text="Phone"/>
        ...
    </Grid>
</Grid>
<Grid x:Name="customersColumnarView" Margin="20,10,20,10" Visibility="Collapsed">
    ...
    <Grid Grid.Row="1">
        ...
        <TextBlock Grid.Row="0" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="ID"/>
        <TextBlock Grid.Row="1" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Title"/>
        <TextBlock Grid.Row="2" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="First Name"/>
        <TextBlock Grid.Row="3" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Last Name"/>
        ...
        <TextBlock Grid.Row="4" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Email"/>
        ...
        <TextBlock Grid.Row="5" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Phone"/>
        ...
    </Grid>
</Grid>
```

Os rótulos do formulário devem mudar para a fonte Buxton Sketch e aparecer na cor branca, em um tamanho de fonte de 30 pontos:



21. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.



Nota Utilize o Simulator, se estiver usando uma tela com resolução de menos de 1366 × 768.

O formulário Customers deve aparecer e ser estilizado da mesma maneira que aparece na janela Design View no Visual Studio. Observe que, se você digitar qualquer texto nos vários campos do formulário, eles utilizarão a fonte e a estilização padrão dos controles *TextBox*.



Nota Embora a fonte Buxton Sketch seja boa para rótulos e títulos, não é recomendada como fonte para campos de entrada de dados, pois pode ser difícil diferenciar alguns caracteres dos outros. Por exemplo, a letra / minúscula é muito parecida com o algarismo 1 e a letra O maiúscula é quase indistinguível do algarismo 0. Por isso, faz sentido ficar com a fonte padrão para os controles *TextBox*.

22. Clique na borda superior do formulário e arraste-a de modo que ele seja exibido na metade esquerda da tela e verifique que os estilos foram aplicados aos controles na grade *customersColumnarView*. O formulário deve ser parecido com este:



23. Retorne ao Visual Studio e interrompa a depuração.

Pode-se ver que, utilizando estilos, é possível implementar com facilidade vários efeitos realmente interessantes. Além disso, o uso cuidadoso de estilos torna seu código muito mais fácil de manter do que a definição de propriedades em controles individuais. Por exemplo, se quiser trocar a fonte utilizada pelos rótulos e cabeçalhos no aplicativo Customers, você precisa fazer uma única alteração no estilo *FontStyle*. De modo geral, você deve usar estilos quando possível; além de auxiliar na manutenibilidade, o uso de estilos ajuda a manter a marcação XAML de seus formulários limpa e organizada, e o código XAML de um formulário só precisa especificar os controles e o layout, e não como os controles devem aparecer no formulário.



Nota Você também pode utilizar Microsoft Blend for Visual Studio 2013 para definir estilos complexos que podem ser integrados em um aplicativo. Artistas gráficos profissionais podem utilizar o Blend para criar estilos personalizados e fornecê-los na forma de marcação XAML para os desenvolvedores que constroem aplicativos. Tudo que o desenvolvedor precisa fazer é adicionar as tags *Style* aos elementos da interface do usuário, para referenciar os estilos adequados.

Resumo

Neste capítulo, você aprendeu a utilizar o controle *Grid* para implementar uma interface de usuário que pode mudar de escala para diferentes tamanhos físicos e orientações de dispositivo. Aprendeu também a utilizar o Visual State Manager para adaptar o layout dos controles quando o usuário muda o tamanho da janela que exibe o aplicativo. Por último, você aprendeu a criar estilos personalizados e a aplicá-los nos controles de um formulário. Agora que a interface do usuário foi definida, o próximo desafio é adicionar funcionalidade ao aplicativo, permitindo ao usuário exibir e atualizar dados, o que será feito no último capítulo.

- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 26.
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Criar um novo aplicativo Windows Store	Utilize um dos templates Windows Store no Visual Studio 2013, como o template Blank App.
Implementar uma interface de usuário que muda de escala para diferentes tamanhos físicos e orientações de dispositivo	Utilize um controle <i>Grid</i> . Divida o controle <i>Grid</i> em linhas e colunas, e coloque controles nessas linhas e colunas, em vez de especificar uma localização absoluta em relação às margens do <i>Grid</i> .
Implementar uma interface de usuário que se adapte a diferentes larguras de tela	Para cada visualização, crie diferentes layouts que exibam os controles de maneira adequada. Utilize o Visual State Manager para selecionar o layout a ser exibido quando o estado visual mudar.
Criar estilos personalizados	Adicione um dicionário de recursos ao aplicativo. Defina estilos nesse dicionário com o elemento <code><Style></code> e especifique as propriedades que cada estilo modifica. Por exemplo: <code><Style x:Key="GridStyle" TargetType="Grid"> <Setter Property="Background" Value="{StaticResource WoodBrush}"/> </Style></code>
Aplicar um estilo personalizado a um controle	Defina a propriedade <code>Style</code> do controle e refcrcie o estilo pelo nome. Por exemplo: <code><Grid Style="{StaticResource GridStyle}"></code>

CAPÍTULO 26

Exibição e busca de dados em um aplicativo Windows Store

Neste capítulo, você vai aprender a:

- Explicar como o padrão Model-View-ViewModel é utilizado para implementar a lógica de um aplicativo Windows Store.
- Utilizar vinculação de dados para exibir e modificar dados em uma visualização.
- Criar um ViewModel com o qual uma visualização pode interagir com um modelo.
- Implementar um contrato Search por meio do qual um aplicativo Windows Store se integre com a funcionalidade Search do Windows 8.1.

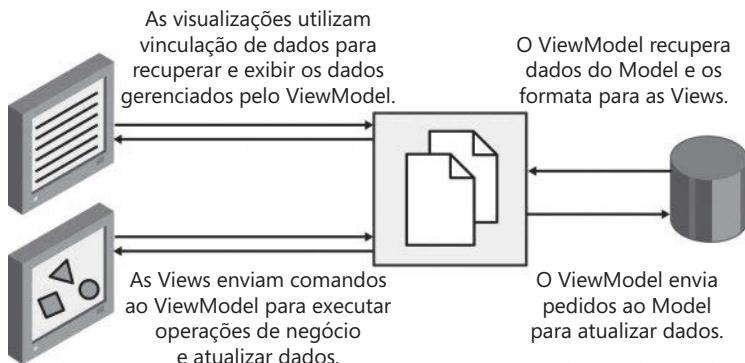
O Capítulo 25, “Implementação da interface do usuário de um aplicativo Windows Store”, demonstrou como projetar uma interface de usuário (IU) para um aplicativo Microsoft Windows Store que podia se adaptar a diferentes tamanhos físicos, orientações e visualizações do dispositivo que um cliente que estivesse executando seu aplicativo poderia usar. O aplicativo desenvolvido naquele capítulo era simples, utilizado para exibir e editar os detalhes dos clientes.

Neste capítulo, você vai aprender a exibir os dados na interface do usuário e ver os recursos fornecidos pelo Microsoft Windows 8.1 para procurar dados em um aplicativo. Ao executar essas tarefas, aprenderá também como pode estruturar um aplicativo Windows Store. Este capítulo aborda muitos assuntos; em especial, você vai ver como utilizar vinculação de dados (data-binding) para conectar a interface do usuário aos dados que ela exibe e como criar um ViewModel, que torna possível separar a lógica da interface do usuário do modelo de dados e da lógica do negócio de um aplicativo. Verá também como utilizar contratos para implementar um recurso de pesquisa que se integra no sistema operacional Windows 8.1. Ao construir esse aplicativo, você vai aprender muito mais sobre os templates fornecidos pelo Microsoft Visual Studio 2013 para ajudá-lo a compilar aplicativos Windows Store.

Implemente o padrão Model-View-ViewModel

Um aplicativo Windows Store bem estruturado separa o projeto da interface do usuário dos dados que utiliza e da lógica do negócio que compõe a funcionalidade do aplicativo. Essa separação ajuda a eliminar as dependências entre os vários componentes, removendo os obstáculos para que os diferentes elementos sejam projetados e implemen-

tados por pessoas que tenham as habilidades especializadas apropriadas. Por exemplo, um artista gráfico pode concentrar-se em projetar uma interface de usuário atraente e intuitiva, um especialista em banco de dados pode concentrar-se na implementação de um conjunto de estruturas de dados otimizada para armazenar e acessar os dados e um desenvolvedor C# pode dirigir seus esforços para a implementação da lógica do negócio do aplicativo. Esse é um objetivo comum que tem sido o alvo de muitas estratégias de desenvolvimento, não apenas para aplicativos Windows Store, e nos últimos anos muitas técnicas foram concebidas para ajudar a estruturar um aplicativo dessa maneira. Possivelmente, a estratégia mais popular é seguir o padrão de projeto Model-View-ViewModel (MVVM). Nesse padrão, Model (o modelo) fornece os dados utilizados pelo aplicativo, View (a visualização) representa o modo de exibir os dados na interface do usuário e ViewModel contém a lógica que conecta os dois, pegando a entrada do usuário e convertendo-a em comandos que efetuam operações do nível de negócios no modelo, pegando também os dados do modelo e formatando-os da maneira esperada pela visualização (view states). O diagrama a seguir mostra um relacionamento simplificado entre os elementos do padrão MVVM. Observe que um aplicativo poderia fornecer várias visualizações dos mesmos dados. Em um aplicativo Windows Store, por exemplo, você poderia implementar diferentes estados de visualização, os quais poderiam apresentar as informações utilizando diferentes layouts de tela. Uma tarefa do ViewModel é garantir que os dados do mesmo modelo possam ser exibidos e manipulados por muitas visualizações diferentes. Em um aplicativo Windows Store, a visualização pode configurar vinculação de dados para conectar aos dados apresentados pelo ViewModel. Além disso, a visualização pode pedir para que o ViewModel atualize os dados no modelo ou execute tarefas de negócios, chamando os comandos implementados pelo ViewModel.



Exiba dados utilizando vinculação de dados

Antes de começar a implementar um ViewModel para o aplicativo Customers, é bom saber um pouco mais sobre vinculação de dados e como essa técnica pode ser aplicada para exibir dados em uma interface de usuário. Com a vinculação de dados é possível associar uma propriedade de um controle a uma propriedade de um objeto; se o valor da propriedade especificada do objeto mudar, a propriedade do controle associada ao objeto também mudará. Além disso, a vinculação de dados pode ser bidirecional: se o valor de uma propriedade em um controle que utiliza vinculação de dados muda, a modificação é propagada para o objeto ao qual ela está associada. O exercício a seguir fornece uma rápida introdução ao uso de vinculação de dados para exibir dados. Ele é baseado no aplicativo Customers do Capítulo 25.

Use vinculação de dados para exibir informações dos clientes

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. Abra o projeto Customers, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 26\Data Binding na sua pasta Documentos. Essa é uma versão do aplicativo Customers desenvolvido no Capítulo 25, mas o layout da interface do usuário foi um pouco modificado – os controles são exibidos sobre um fundo azul, o qual os faz sobressair mais facilmente.



Nota O fundo azul foi criado com um controle *Rectangle* que abrange as mesmas linhas e colunas dos controles *TextBlock* e *TextBox* que exibem os cabeçalhos e os dados. O retângulo é preenchido com um *LinearGradientBrush* que muda a cor do retângulo gradualmente, de um azul médio na parte superior para um azul muito escuro na parte inferior. A marcação XAML do controle *Rectangle* exibido nas visualizações do controle Grid *customersTabularView* é parecida com esta (a marcação XAML do controle Grid *customersColumnarView* inclui um controle *Rectangle* similar, abrangendo as linhas e colunas utilizadas por esse layout):

```
<Rectangle Grid.Row="0" Grid.RowSpan="6" Grid.Column="1" Grid.ColumnSpan="7" ...>
    <Rectangle.Fill>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FF0E3895"/>
            <GradientStop Color="#FF141415" Offset="0.929"/>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

3. No Solution Explorer, clique com o botão direito do mouse no projeto Customers, selecione Add e então clique em Class.
4. Na caixa de diálogo Add New Items – Customers, certifique-se de que o template Class esteja selecionado, digite **Customer.cs** na caixa Name, e, então, clique em Add.

Você vai usar essa classe para implementar o tipo de dado *Customer* e, então, implementará vinculação de dados para exibir os detalhes dos objetos *Customer* na interface do usuário.

5. Na janela Code and Text Editor que exibe o arquivo Customer.cs, torne a classe *Customer* pública e adicione os seguintes campos e propriedades privados, mostrados em negrito, à classe *Customer*:

```
public class Customer
{
    public int _customerID;
    public int CustomerID
    {
        get { return this._customerID; }
        set { this._customerID = value; }
    }

    public string _title;
    public string Title
    {
```

```
        get { return this._title; }
        set { this._title = value; }
    }

    public string _firstName;
    public string FirstName
    {
        get { return this._firstName; }
        set { this._firstName = value; }
    }

    public string _lastName;
    public string LastName
    {
        get { return this._lastName; }
        set { this._lastName = value; }
    }

    public string _emailAddress;
    public string EmailAddress
    {
        get { return this._emailAddress; }
        set { this._emailAddress = value; }
    }

    public string _phone;
    public string Phone
    {
        get { return this._phone; }
        set { this._phone = value; }
    }
}
```



Nota Talvez você esteja se perguntando por que essas propriedades não são implementadas como automáticas, pois tudo que fazem é obter e definir o valor em um campo privado. Você vai adicionar mais código a essas propriedades em um exercício posterior.

6. No Solution Explorer, no projeto Customers, clique duas vezes no arquivo MainPage.xaml para exibir a interface do usuário do aplicativo na janela Design View.
7. No painel XAML, localize a marcação do controle *TextBox id*. Modifique a marcação XAML que define a propriedade *Text* desse controle, como mostrado aqui em negrito:

```
<TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
Text="{Binding CustomerID}" .../>
```

A sintaxe *Text="{Binding Path}"* especifica que o valor da propriedade *Text* será fornecido pelo valor da expressão *Path* em tempo de execução. Neste caso, *Path* é definida como *CustomerID*; portanto, o valor armazenado na expressão *CustomerID* será exibido por esse controle. Contudo, você precisa fornecer um pouco mais de informação para indicar que *CustomerID* é na verdade uma propriedade de um objeto *Customer*. Para isso, você pode definir a propriedade *DataContext* do controle, o que vai fazer em breve.

8. Adicione as seguintes expressões de vinculação para cada um dos outros controles de texto no formulário. Aplique vinculação de dados aos controles *TextBox* nos controles *Grid customersTabularView* e *customersColumnView*, como mostrado em negrito no código a seguir (os controles *ComboBox* exigem um tratamento um pouco diferente, o que será feito na seção "Utilize vinculação de dados com um controle *ComboBox*", mais adiante neste capítulo):

```
<Grid x:Name="customersTabularView" ...>
  ...
  <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
  Text="{Binding CustomerID}" .../>
  ...
  <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...
  Text="{Binding FirstName}" .../>
  <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
  Text="{Binding LastName}" .../>
  ...
  <TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="3"
  x:Name="email" ... Text="{Binding EmailAddress}" .../>
  ...
  <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3"
  x:Name="phone" ... Text="{Binding Phone}" .../>
</Grid>
<Grid x:Name="customersColumnarView" Margin="20,10,20,110"
Visibility="Collapsed">
  ...
  <TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
  Text="{Binding CustomerID}" .../>
  ...
  <TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
  Text="{Binding FirstName}" .../>
  <TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
  Text="{Binding LastName}" .../>
  ...
  <TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
  Text="{Binding EmailAddress}" .../>
  ...
  <TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
  Text="{Binding Phone}" .../>
</Grid>
```

9. No Solution Explorer, expanda o arquivo *MainPage.xaml* e clique duas vezes no arquivo *MainPage.xaml.cs* para exibir o código do formulário *MainPage.xaml* na janela Code and Text Editor. Adicione a instrução mostrada em negrito no código a seguir ao construtor de *MainPage*.

```
public MainPage()
{
    this.InitializeComponent();
    Window.Current.SizeChanged +=WindowSizeChanged;
```

```
Customer customer = new Customer
{
    CustomerID = 1,
    Title = "Mr",
    FirstName = "John",
    LastName = "Sharp",
    EmailAddress = "john@contoso.com",
    Phone = "111-1111"
};
```

Esse código cria uma nova instância da classe *Customer* e a preenche com alguns dados de exemplo.

10. Após o código que cria o novo objeto *Customer*, adicione a instrução mostrada em negrito a seguir:

```
Customer customer = new Customer
{
    ...
};

this.DataContext = customer;
```

Essa instrução especifica o objeto ao qual os controles do formulário *MainPage* devem se vincular. Em cada um dos controles, a marcação XAML *Text="{Binding Path}"* será resolvida em relação a esse objeto. Por exemplo, os controles *TextBox id* e *cld* especificam ambos *Text="{Binding CustomerID}"* e, assim, exibirão o valor encontrado na propriedade *CustomerID* do objeto *Customer* ao qual o formulário está vinculado.



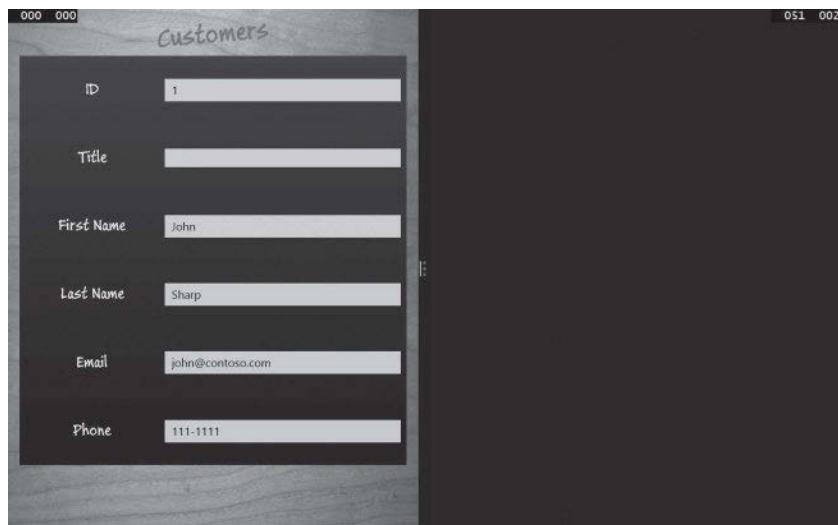
Nota Neste exemplo, você definiu a propriedade *DataContext* do formulário, de modo que a mesma vinculação de dados é aplicada automaticamente a todos os controles do formulário. Também é possível definir a propriedade *DataContext* de controles individuais, caso seja necessário vincular os diferentes controles a diferentes objetos.

11. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.

Verifique se o formulário ocupa a tela inteira e exibe os detalhes do cliente John Sharp, como na imagem a seguir:



12. Redimensione a janela do aplicativo e exiba-o na visualização estreita. Verifique se ele exibe os mesmos dados, como ilustrado aqui:



Os controles exibidos na visualização estreita estão vinculados aos mesmos dados dos controles exibidos na visualização em tela inteira.

13. Na visualização estreita, mude o endereço de e-mail para **john@treyresearch.com**.

14. Expanda a janela do aplicativo de modo a ocupar a tela inteira.

Observe que o endereço de e-mail exibido nessa visualização não mudou.

15. Retorne ao Visual Studio e interrompa a depuração.

16. No Visual Studio, exiba o código da classe Customer na janela Code and Text Editor e defina um ponto de interrupção (breakpoint) no método set da propriedade *EmailAddress*.

17. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo novamente.

18. Quando o depurador chegar ao ponto de interrupção pela primeira vez, pressione F5 para continuar a executar o aplicativo.

19. Quando a interface do usuário do aplicativo Customers aparecer, redimensione a janela do aplicativo para exibir a visualização estreita e mude o endereço de e-mail para john@treyresearch.com.

20. Expanda a janela do aplicativo de volta para a visualização em tela inteira.

Observe que o depurador não atinge o ponto de interrupção no método de acesso set da propriedade *EmailAddress*; o valor atualizado não é escrito de volta no objeto *Customer* quando o *TextBox* de e-mail perde o foco.

21. Retorne ao Visual Studio e interrompa a depuração.

22. Remova o ponto de interrupção do método de acesso set da propriedade *EmailAddress* na classe *Customer*.

Modifique dados através de vinculação de dados

No exercício anterior, você viu como é fácil exibir os dados de um objeto usando vinculação de dados. Mas, por padrão, a vinculação de dados é uma operação unidirecional, e as alterações feitas nos dados exibidos não são copiadas de volta para a origem de dados (data source). No exercício, vimos isso ao mudar o endereço de e-mail exibido na visualização estreita – quando você voltou para a visualização em tela inteira, os dados não tinham mudado. A vinculação de dados bidirecional pode ser implementada modificando-se o parâmetro *Mode* da especificação *Binding* da marcação XAML de um controle. O parâmetro *Mode* indica se a vinculação de dados é unidirecional (*one-way*, o padrão) ou bidirecional (*two-way*). Isso é o que você fará a seguir.

Implemente vinculação de dados *TwoWay* para modificar informações de um cliente

1. Exiba o arquivo MainPage.xaml na janela Design View e modifique a marcação XAML de cada um dos controles *TextBox*, como mostrado em negrito no código a seguir:

```
<Grid x:Name="customersTabularView" ...>
    ...
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
Text="{Binding CustomerID, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...
Text="{Binding FirstName, Mode=TwoWay}" .../>
```

```
<TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...  
Text="{Binding LastName, Mode=TwoWay}" ...>  
...  
<TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="3"  
x:Name="email" ... Text="{Binding EmailAddress, Mode=TwoWay}" ...>  
...  
<TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3"  
x:Name="phone" ... Text="{Binding Phone, Mode=TwoWay}" ...>  
</Grid>  
<Grid x:Name="customersColumnarView" Margin="20,10,20,110" ...>  
...  
<TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...  
Text="{Binding CustomerID, Mode=TwoWay}" ...>  
...  
<TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...  
Text="{Binding FirstName, Mode=TwoWay}" ...>  
<TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...  
Text="{Binding LastName, Mode=TwoWay}" ...>  
...  
<TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...  
Text="{Binding EmailAddress, Mode=TwoWay}" ...>  
...  
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...  
Text="{Binding Phone, Mode=TwoWay}" ...>  
</Grid>
```

O parâmetro *Mode* na especificação *Binding* indica se a vinculação de dados é unidirecional (o padrão) ou bidirecional. Definir *Mode* com *TwoWay* faz com que as alterações do usuário sejam passadas de volta para o objeto ao qual um controle está vinculado.

2. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo novamente.
3. Enquanto o aplicativo está exibido na visualização em tela inteira, mude o endereço de e-mail para **john@treyresearch.com** e, então, redimensione a janela para exibir o aplicativo na visualização estreita.

Observe que, apesar da mudança na vinculação de dados para o modo *TwoWay*, o endereço de e-mail exibido na visualização estreita não foi atualizado — ainda é *john@contoso.com*.

4. Retorne ao Visual Studio e interrompa a depuração.

Claramente, algo não está funcionando do modo correto! O problema agora não é que os dados não foram atualizados, mas sim que a visualização não está mostrando a versão mais recente deles (se você redefinir o ponto de interrupção no método de acesso *set* da propriedade *EmailAddress* da classe *Customer* e executar novamente o aplicativo no depurador, verá que o depurador atingirá o ponto de interrupção quando o valor do endereço de e-mail for mudado e tirará o foco do controle *TextBox*). Apesar das aparências, o processo de vinculação de dados não é mágico e a vinculação de dados não sabe quando os dados aos quais está vinculada foram mudados; o objeto precisa informá-la sobre qualquer modificação, enviando um evento *PropertyChanged* para a interface do usuário. Esse evento faz parte de uma interface chamada *INotifyPropertyChanged*, e todos os objetos que suportam vinculação de dados bidirecional devem implementar essa interface. Você implementará essa interface no próximo exercício.

Implemente a interface *IPropertyChanged* na classe *Customer*

1. No Visual Studio, exiba o arquivo Customer.cs na janela Code and Text Editor.
2. Adicione a seguinte diretiva *using* à lista localizada no início do arquivo:

```
using System.ComponentModel;
```

A interface *IPropertyChanged* é definida nesse namespace.

3. Modifique a definição da classe *Customer* para especificar que ela implementa a interface *IPropertyChanged*, conforme mostrado em negrito:

```
class Customer : IPropertyChanged
```

4. Adicione à classe *Customer* o evento *PropertyChanged*, mostrado em negrito no código a seguir, após a propriedade *Phone*:

```
class Customer : IPropertyChanged
{
    ...
    public string _phone;
    public string Phone {
        get { return this._phone; }
        set { this._phone = value; }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

Esse é o único item definido pela interface *IPropertyChanged*. Todos os objetos que implementam essa interface devem fornecer esse evento, e devem dispará-lo sempre que quiserem notificar o mundo exterior sobre uma mudança no valor de uma propriedade.

5. Adicione o método a seguir à classe *Customer*, após o evento *PropertyChanged*:

```
class Customer : IPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

O método *OnPropertyChanged* dispara o evento *PropertyChanged*. O parâmetro *PropertyChangedEventArgs* para o evento *PropertyChanged* deve especificar o nome da propriedade que foi alterada. Esse valor é passado como um parâmetro para o método *OnPropertyChanged*.

6. Modifique os métodos de acesso set para cada uma das propriedades da classe *Customer*, para chamarem o método *OnPropertyChanged* sempre que o valor que contém for modificado, como mostrado aqui em negrito:

```
class Customer : INotifyPropertyChanged
{
    public int _customerID;
    public int CustomerID
    {
        get { return this._customerID; }
        set
        {
            this._customerID = value;
            this.OnPropertyChanged("CustomerID");
        }
    }

    public string _title;
    public string Title
    {
        get { return this._title; }
        set
        {
            this._title = value;
            this.OnPropertyChanged("Title");
        }
    }

    public string _firstName;
    public string FirstName
    {
        get { return this._firstName; }
        set
        {
            this._firstName = value;
            this.OnPropertyChanged("FirstName");
        }
    }

    public string _lastName;
    public string LastName
    {
        get { return this._lastName; }
        set
        {
            this._lastName = value;
            this.OnPropertyChanged("LastName");
        }
    }
}
```

```

public string _emailAddress;
public string EmailAddress
{
    get { return this._emailAddress; }
    set
    {
        this._emailAddress = value;
        this.OnPropertyChanged("EmailAddress");
    }
}

public string _phone;
public string Phone
{
    get { return this._phone; }
    set
    {
        this._phone = value;
        this.OnPropertyChanged("Phone");
    }
}
...
}

```

7. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo novamente.
8. Quando o formulário Customers aparecer, mude o endereço de e-mail para **john@treyresearch.com**, e o número do telefone para **222-2222**.
9. Redimensione a janela para exibir o aplicativo na visualização estreita e verifique que o endereço de e-mail e o número do telefone mudaram.
10. Mude o nome para **James**, expanda a janela para exibir a visualização em tela inteira e verifique se o nome foi alterado.
11. Retorne ao Visual Studio e interrompa a depuração.

Utilize vinculação de dados com um controle *ComboBox*

É relativamente simples utilizar vinculação de dados com um controle como *TextBox* ou *TextBlock*. Por outro lado, os controles *ComboBox* exigem um pouco mais de atenção. O problema é que, na verdade, um controle *ComboBox* exibe duas coisas: uma lista de valores na lista suspensa na qual o usuário pode selecionar um item e o valor do item atualmente selecionado. Você pode implementar vinculação de dados para exibir uma lista de itens na lista suspensa de um controle *ComboBox*, e o valor que o usuário seleciona deve ser membro dessa lista. No aplicativo *Customers*, você pode configurar a vinculação de dados para o valor selecionado no controle *ComboBox title*, definindo a propriedade *SelectedValue*, como segue:

```
<ComboBox ... x:Name="title" ... SelectedValue="{Binding Title}" ... />
```

Mas lembre-se de que a lista de valores da lista suspensa é codificada na marcação XAML, como isto:

```
<ComboBox ... x:Name="title" ... >
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>
</ComboBox>
```

Essa marcação não é aplicada até que o controle tenha sido criado; portanto, o valor especificado pela vinculação de dados não é encontrado na lista, pois ela ainda não existe quando a vinculação é construída. O resultado é que o valor não é exibido. Pode tentar isso, se quiser — configure a vinculação para a propriedade *SelectedValue* como acabamos de mostrar e execute o aplicativo. O *ComboBox title* estará vazio ao ser exibido inicialmente, não obstante o cliente ter título de Mr.

Existem várias soluções para esse problema, mas a mais simples é criar uma origem de dados que contenha a lista de valores válidos e, então, especificar que o controle *ComboBox* deve utilizar essa lista como conjunto de valores suspensos. Também é necessário fazer isso antes que a vinculação de dados do *ComboBox* seja aplicada.

Implemente vinculação de dados para os controles *ComboBox title*

1. No Visual Studio, exiba o arquivo MainPage.xaml.cs na janela Code and Text Editor.
2. Adicione o código a seguir, mostrado em negrito, ao construtor de *MainPage*:

```
public MainPage()
{
    this.InitializeComponent();
    Window.Current.SizeChanged += WindowSizeChanged;

    List<string> titles = new List<string>
    {
        "Mr", "Mrs", "Ms", "Miss"
    };

    this.title.ItemsSource = titles;
    this.cTitle.ItemsSource = titles;

    Customer customer = new Customer
    {
        ...
    };

    this.DataContext = customer;
}
```

Esse código cria uma lista de strings contendo os títulos válidos que os clientes podem ter. Então, ele define a propriedade *ItemsSource* de ambos os controles *ComboBox title* de forma a referenciar essa lista (lembre-se de que existe um controle *ComboBox* para cada visualização).



Nota Em um aplicativo comercial, você provavelmente recuperaria a lista de valores exibida por um controle *ComboBox* de um banco de dados ou de alguma outra origem de dados, em vez de usar uma lista codificada, como mostrado nesse exemplo.

A localização desse código é importante. Ele deve ser executado antes da instrução que define a propriedade *DataContext* do formulário *MainPage*, pois é aí que ocorre a vinculação de dados nos controles do formulário.

3. Exiba o arquivo *MainPage.xaml* na janela Design View.
4. Modifique a marcação XAML dos controles *ComboBox* *title* e *cTitle*, como mostrado aqui em negrito:

```
<Grid x:Name="customersTabularView" ...>
  ...
  <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...
    SelectedValue="{Binding Title, Mode=TwoWay}">
    </ComboBox>
  ...
</Grid>
<Grid x:Name="customersColumnarView" ...>
  ...
  <ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" ...
    SelectedValue="{Binding Title, Mode=TwoWay}">
    </ComboBox>
  ...
</Grid>
```

Observe que a lista de elementos *ComboBoxItem* para cada controle foi removida e que a propriedade *SelectedValue* está configurada para utilizar vinculação de dados com o campo *Title* no objeto *Customer*.

5. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
6. Verifique se o valor do título do cliente é exibido corretamente (deve ser Mr). Clique na seta suspensa do controle *ComboBox* e verifique se ele contém os valores *Mr, Mrs, Ms* e *Miss*.
7. Redimensione a janela para exibir o aplicativo na visualização estreita e realize as mesmas verificações. Observe que você pode mudar o título e, quando volta para a visualização em tela inteira, o novo título é exibido.
8. Retorne ao Visual Studio e interrompa a depuração.

Crie um ViewModel

Agora você já viu como configurar vinculação de dados para conectar uma origem de dados aos controles de uma interface de usuário, mas a origem de dados que estava utilizando é muito simples, consistindo em apenas um cliente. No mundo real, a origem de dados provavelmente será muito mais complexa, compreendendo coleções de diferentes tipos de objetos. Lembre-se de que, em termos de MVVM, a origem de da-

dos frequentemente é fornecida pelo modelo, e a interface do usuário (a visualização) se comunica com o modelo apenas indiretamente, por meio de um objeto *ViewModel*. O fundamento lógico por trás dessa estratégia é que o modelo e as visualizações que exibem os dados fornecidos por esse modelo devem ser independentes entre si; não deve ser necessário mudar o modelo se a interface do usuário for modificada e também não deve ser necessário ajustar a interface se o modelo subjacente mudar.

O *ViewModel* fornece a conexão entre a visualização e o modelo, e também implementa a lógica do negócio do aplicativo. Novamente, essa lógica deve ser independente da visualização e do modelo. O *ViewModel* expõe a lógica de negócio para a visualização, implementando um conjunto de comandos. A interface de usuário pode disparar esses comandos com base no modo como o usuário navega pelo aplicativo. No próximo exercício, você vai ampliar o aplicativo *Customers*. Vai implementar um modelo que contém uma lista de objetos *Customer* e vai criar um *ViewModel* que fornece comandos com os quais a visualização pode se mover entre os clientes.

Crie um *ViewModel* para gerenciar informações de clientes

1. Abra o projeto *Customers*, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 26\ViewModel na sua pasta Documentos. Esse projeto contém uma versão completa do aplicativo *Customers* do conjunto de exercícios anterior; se preferir, você pode continuar usando sua versão do projeto.
2. No Solution Explorer, clique com o botão direito do mouse no projeto *Customers*, aponte para Add e então clique em Class.
3. Na caixa de diálogo Add New Item – *Customers*, na caixa Name, digite **ViewModel.cs** e clique em Add.

Essa classe fornece um *ViewModel* básico que contém uma coleção de objetos *Customer*. A interface do usuário se vinculará aos dados expostos por esse *ViewModel*.

4. Na janela Code and Text Editor que exibe o arquivo *ViewModel.cs*, marque a classe como pública e adicione à classe *ViewModel* o código mostrado em negrito no exemplo a seguir:

```
public class ViewModel
{
    private List<Customer> customers;

    public ViewModel()
    {
        this.customers = new List<Customer>
        {
            new Customer
            {
                CustomerID = 1,
                Title = "Mr",
                FirstName="John",
                LastName="Sharp",
                EmailAddress="john@contoso.com",
                Phone="111-1111"
            },
        };
    }
}
```

```
new Customer
{
    CustomerID = 2,
    Title = "Mrs",
    FirstName="Diana",
    LastName="Sharp",
    EmailAddress="diana@contoso.com",
    Phone="111-1112"
},
new Customer
{
    CustomerID = 3,
    Title = "Ms",
    FirstName="Francesca",
    LastName="Sharp",
    EmailAddress="frankie@contoso.com",
    Phone="111-1113"
}
};
```

A classe `ViewModel` utiliza um objeto `List<Customer>` como seu modelo e o construtor preenche essa lista com alguns dados de exemplo.

5. Adicione a variável privada `currentCustomer`, mostrada em negrito no código a seguir, à classe `ViewModel`, e inicialize essa variável com zero no construtor:

```
class ViewModel  
{  
    private List<Customer> customers;  
    private int currentCustomer;  
  
    public ViewModel()  
    {  
        this.currentCustomer = 0;  
        this.customers = new List<Customer>();  
        ...  
    }  
}
```

A classe *ViewModel* usará essa variável para monitorar qual objeto *Customer* a visualização está exibindo no momento.

6. Adicione a propriedade *Current*, mostrada aqui em negrito, à classe *ViewModel*, após o construtor:

```
class ViewModel  
{  
    ...  
    public ViewModel()  
}
```

```
{  
    ...  
}  
  
public Customer Current  
{  
    get { return this.customers[currentCustomer]; }  
}  
}
```

A propriedade *Current* dá acesso ao objeto *Customer* atual no modelo.



Nota É considerada uma boa prática fornecer acesso controlado a um modelo de dados; somente o *ViewModel* deve ser capaz de modificar o modelo. Contudo, essa restrição não impede a visualização de atualizar os dados apresentados pelo *ViewModel* – ela apenas não pode alterar o modelo e fazê-lo se referir a uma origem de dados diferente.

7. Abra o arquivo MainPage.xaml.cs na janela Code and Text Editor.
8. No construtor de *MainPage*, remova o código que cria o objeto *Customer* e o substitua por uma instrução que cria uma instância da classe *ViewModel*. Altere a instrução que define a propriedade *DataContext* do objeto *MainPage* para referenciar o novo objeto *ViewModel*, como mostrado aqui em negrito:

```
public MainPage()  
{  
    ...  
    this.cTitle.ItemsSource = titles;  
  
    ViewModel viewModel = new ViewModel();  
    this.DataContext = viewModel;  
}
```

9. Abra o arquivo MainPage.xaml na janela Design View.
10. No painel XAML, modifique as vinculações de dados dos controles *TextBox* e *ComboBox* para referenciar propriedades por meio do objeto *Current* apresentado pelo *ViewModel*, como mostrado em negrito no código a seguir:

```
<Grid x:Name="customersTabularView" ...>  
    ...  
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...  
        Text="{Binding Current.CustomerID, Mode=TwoWay}" .../>  
    <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...  
        SelectedValue="{Binding Current.Title, Mode=TwoWay}">  
        </ComboBox>  
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...  
        Text="{Binding Current.FirstName, Mode=TwoWay }" .../>  
    <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
```

```

Text="{Binding Current.LastName, Mode=TwoWay }" .../>
...
<TextBox Grid.Row="3" Grid.Column="3" ... x:Name="email" ...
Text="{Binding Current.EmailAddress, Mode=TwoWay }" .../>
...
<TextBox Grid.Row="5" Grid.Column="3" ... x:Name="phone" ...
Text="{Binding Current.Phone, Mode=TwoWay }" ..."/>
</Grid>
<Grid x:Name="customersColumnarView" Margin="20,10,20,110" ...>
...
<TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
Text="{Binding Current.CustomerID, Mode=TwoWay }" .../>
<ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" ...
SelectedValue="{Binding Current.Title, Mode=TwoWay}">
</ComboBox>
<TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
Text="{Binding Current.FirstName, Mode=TwoWay }" .../>
<TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
Text="{Binding Current.LastName, Mode=TwoWay }" .../>
...
<TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
Text="{Binding Current.EmailAddress, Mode=TwoWay }" .../>
...
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
Text="{Binding Current.Phone, Mode=TwoWay }" .../>
</Grid>
```

11. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
12. Verifique se o aplicativo exibe os detalhes de John Sharp (o primeiro cliente na lista de clientes).
13. Retorne ao Visual Studio e interrompa a depuração.

O ViewModel dá acesso às informações dos clientes por meio da propriedade *Current*, mas atualmente ele não fornece uma maneira de navegar entre os clientes. Você pode implementar métodos que incrementem e decrementem a variável *currentCustomer* para que a propriedade *Current* recupere diferentes clientes, mas deve fazer isso de uma maneira que não vincule a visualização ao ViewModel. A técnica mais aceita é usar o padrão Command. Nesse padrão, o ViewModel expõe métodos na forma de comandos que a visualização pode chamar. O truque é evitar uma referência explícita a esses métodos pelos seus nomes no código da visualização. Para isso, a XAML torna possível vincular comandos declarativamente às ações disparadas pelos controles na interface do usuário, conforme você vai ver nos exercícios da próxima seção.

Adicione comandos a um ViewModel

A marcação XAML que vincula a ação de um controle a um comando exige que os comandos expostos por um ViewModel implementem a interface *ICommand*. Essa interface define os seguintes itens:

- ***CanExecute*** Esse método retorna um valor booleano indicando se o comando pode ser executado. Com esse método, um ViewModel pode habilitar ou desabilitar um comando, dependendo do contexto. Por exemplo, um comando que busca o próximo cliente de uma lista deve ser executado somente se houver um próximo cliente a buscar; se não houver mais clientes, o comando deve ser desabilitado.

- **Execute** Esse método é executado quando o comando é chamado.
- **CanExecuteChanged** Esse evento é disparado quando o estado do ViewModel muda. Sob essas circunstâncias, comandos que antes podiam ser executados agora podem ser desabilitados e vice-versa. Por exemplo, se a interface do usuário chama um comando que busca o próximo cliente de uma lista, se esse é o último cliente, as chamadas subsequentes para *CanExecute* devem retornar *false*. Nesse caso, o evento *CanExecuteChanged* deve ser disparado para indicar que o comando foi desabilitado.

No próximo exercício, você vai criar uma classe genérica que implementa a interface *ICommand*.

Implemente a classe *Command*

1. No Visual Studio, clique com o botão direito do mouse no projeto *Customers*, aponte para Add e então clique em Class.
2. Na caixa de diálogo Add New Item – *Customers*, selecione o template Class, digite **Command.cs** na caixa Name e, então, clique em Add.
3. Na janela Code and Text Editor que exibe o arquivo *Command.cs*, adicione a seguinte diretiva *using* à lista localizada no início do arquivo:

```
using System.Windows.Input;
```

A interface *ICommand* é definida nesse namespace.

4. Torne a classe *Command* pública e especifique que ela implementa a interface *ICommand*, conforme mostrado em negrito:

```
public class Command : ICommand  
{  
}
```

5. Adicione os seguintes campos privados à classe *Command*:

```
public class Command : ICommand  
{  
    private Action methodToExecute = null;  
    private Func<bool> methodToDetectCanExecute = null;  
}
```

Os tipos *Action* e *Func* estão descritos brevemente no Capítulo 20, “Separação da lógica do aplicativo e tratamento de eventos”. O tipo *Action* é um delegate que pode ser usado para referenciar um método que não recebe parâmetros e não retorna um valor, e o tipo *Func<T>* também é um delegate que pode referenciar um método que não recebe parâmetros, mas retorna um valor do tipo especificado pelo parâmetro de tipo *T*. Nessa classe, você vai usar o campo *methodToExecute* para referenciar o código que o objeto *Command* executará quando for chamado pela visualização. O campo *methodToDetectCanExecute* será usado para referenciar o método que detecta se o comando pode ser executado (ele pode estar desabilitado por algum motivo, dependendo do estado do aplicativo ou dos dados).

6. Adicione um construtor à classe *Command*. Esse construtor deve receber dois parâmetros, um objeto *Action* e um objeto *Func<T>*, e atribuir esses parâmetros aos campos *methodToExecute* e *methodToDetectCanExecute*, como mostrado aqui em negrito:

```
public Command : ICommand
{
    ...
    public Command(Action methodToExecute,
                  Func<bool> methodToDetectCanExecute)
    {
        this.methodToExecute = methodToExecute;
        this.methodToDetectCanExecute =
            methodToDetectCanExecute;
    }
}
```

O ViewModel criará uma instância dessa classe para cada comando. O ViewModel fornecerá o método para executar o comando e o método para detectar se o comando deve ser habilitado quando chamar o construtor.

7. Implemente os métodos *Execute* e *CanExecute* da classe *Command* utilizando os métodos referenciados pelos campos *methodToExecute* e *methodToDetectCanExecute*, como segue:

```
public Command : ICommand
{
    ...
    public Command(Action methodToExecute,
                  Func<bool> methodToDetectCanExecute)
    {
        ...
    }

    public void Execute(object parameter)
    {
        this.methodToExecute();
    }

    public bool CanExecute(object parameter)
    {
        if (this.methodToDetectCanExecute == null)
        {
            return true;
        }
        else
        {
            return this.methodToDetectCanExecute();
        }
    }
}
```

Observe que, se o ViewModel fornece uma referência *null* para o parâmetro *methodToDetectCanExecute* do construtor, a ação padrão é supor que o comando pode ser executado, e o método *CanExecute* retorna *true*.

8. Adicione o evento público *CanExecuteChanged* à classe *Command*:

```
public Command : ICommand
{
    ...
    public bool CanExecute(object parameter)
    {
        ...
    }

    public event EventHandler CanExecuteChanged;
}
```

Quando um comando é vinculado a um controle, o controle assina esse evento automaticamente. Esse evento deve ser disparado pelo objeto *Command* se o estado do *ViewModel* for atualizado e o valor retornado por *CanExecute* mudar. Nas versões anteriores do Windows, o Windows Presentation Foundation (WPF) fornecia o objeto *CommandManager* para detectar uma mudança no estado e disparar o evento *CanExecuteChanged*, mas o objeto *CommandManager* não está disponível para aplicativos Windows Store. Com isso, você precisa implementar esse recurso manualmente. A estratégia mais simples é usar um cronômetro para disparar o evento *CanExecuteChanged* periodicamente, mas ou menos uma vez por segundo.

9. Adicione à lista no início do arquivo a diretiva *using* mostrada a seguir:

```
using Windows.UI.Xaml;
```

10. Acima do construtor, adicione o seguinte campo à classe *Command*:

```
public class Command : ICommand
{
    ...
    private Func<bool> methodToDetectCanExecute = null;
    private DispatcherTimer canExecuteChangedEventTimer = null;

    public Command(Action methodToExecute,
                  Func<bool> methodToDetectCanExecute)
    {
        ...
    }
}
```

A classe *DispatcherTimer*, definida no namespace *Windows.UI.Xaml*, implementa um cronômetro que pode disparar um evento em intervalos especificados. Você vai usar o campo *canExecuteChangedEventTimer* para disparar o evento *CanExecuteChanged* em intervalos de 1 segundo.

11. Adicione o método *canExecuteChangedEventTimer_Click*, mostrado em negrito no código a seguir, ao final da classe *Command*:

```
public class Command : ICommand
{
    ...
    public event EventHandler CanExecuteChanged;
```

```

void canExecuteChangedEventTimer_Tick(object sender, object e)
{
    if (this.CanExecuteChanged != null)
    {
        this.CanExecuteChanged(this, EventArgs.Empty);
    }
}
}

```

Esse método simplesmente dispara o evento *CanExecuteChanged* se pelo menos um controle estiver vinculado ao comando. Rigorosamente falando, antes de disparar o evento, esse método também verifica se o estado do objeto mudou. Mas você vai definir o intervalo do cronômetro com um período de tempo prolongado (em termos de processamento), para minimizar qualquer ineficiência pelo fato de não verificar uma mudança no estado.

12. No construtor de *Command*, adicione as seguintes instruções mostradas em negrito.

```

public class Command : ICommand
{
    ...
    public Command(Action methodToExecute,
                  Func<bool> methodToDetectCanExecute)
    {
        this.methodToExecute = methodToExecute;
        this.methodToDetectCanExecute = methodToDetectCanExecute;

        this.canExecuteChangedEventTimer = new DispatcherTimer();
        this.canExecuteChangedEventTimer.Tick +=
            canExecuteChangedEventTimer_Tick;
        this.canExecuteChangedEventTimer.Interval =
            new TimeSpan(0, 0, 1);
        this.canExecuteChangedEventTimer.Start();
    }
    ...
}

```

Esse código inicia o objeto *DispatcherTimer* e define o intervalo para eventos de cronômetro com 1 segundo, antes de iniciar a execução do cronômetro.

13. No menu Build, clique em Build Solution e verifique se seu aplicativo compila sem erros.

Agora você pode utilizar a classe *Command* para adicionar comandos à classe *ViewModel*. No próximo exercício, você definirá comandos com os quais uma visualização pode mover-se entre os clientes.

Adicione os comandos *NextCustomer* e *PreviousCustomer* à classe *ViewModel*

1. No Visual Studio, exiba o arquivo *ViewModel.cs* na janela Code and Text Editor.
2. Adicione a seguinte diretiva *using* ao início do arquivo e modifique a definição da classe *ViewModel* para implementar a interface *INotifyPropertyChanged*.

```
...
using System.ComponentModel;

namespace Customers
{
    public class ViewModel : INotifyPropertyChanged
    {
        ...
    }
}
```

3. Adicione o evento *PropertyChanged* e o método *OnPropertyChanged* ao final da classe *ViewModel*. Esse é o mesmo código que você incluiu na classe *Customer*.

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Lembre-se de que a visualização referencia os dados por meio da propriedade *Current* nas expressões de vinculação de dados para os vários controles que ela contém. Quando a classe *ViewModel* mudar para outro cliente, deverá disparar o evento *PropertyChanged* para notificar à visualização de que os dados a serem exibidos mudaram.

4. Após o construtor, adicione os seguintes campos e propriedades à classe *ViewModel*:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public ViewModel()
    {
        ...
    }

    private bool _isAtStart;
    public bool IsAtStart
    {
        get { return this._isAtStart; }
        set
        {
            this._isAtStart = value;
            this.OnPropertyChanged("IsAtStart");
        }
    }
}
```

```

private bool _isAtEnd;
public bool IsAtEnd
{
    get { return this._isAtEnd; }
    set
    {
        this._isAtEnd = value;
        this.OnPropertyChanged("IsAtEnd");
    }
}
...
}

```

Você utilizará essas duas propriedades para monitorar o estado do ViewModel. A propriedade *IsAtStart* será definida com *true* quando o campo *currentCustomer* no ViewModel estiver posicionado no início da coleção de clientes, e a propriedade *IsAtEnd* será definida com *true* quando o ViewModel estiver posicionado no final da coleção.

5. Modifique o construtor para definir as propriedades *IsAtStart* e *IsAtEnd*, como segue em negrito:

```

public ViewModel()
{
    this.currentCustomer = 0;
this.IsAtStart = true;
this.IsAtEnd = false;
...
}

```

6. Após a propriedade *Current*, adicione os métodos privados *Next* e *Previous*, mostrados aqui em negrito, à classe *ViewModel*:

```

public class ViewModel : INotifyPropertyChanged
{
    ...
    public Customer Current
    {
        get { return this.customers[currentCustomer]; }
    }

    private void Next()
    {
        if (this.customers.Count - 1 > this.currentCustomer)
        {
            this.currentCustomer++;
            this.OnPropertyChanged("Current");
            this.IsAtStart = false;
            this.IsAtEnd =
                (this.customers.Count - 1 == this.currentCustomer);
        }
    }

    private void Previous()
    {

```

```
        if (this.currentCustomer > 0)
    {
        this.currentCustomer--;
        this.OnPropertyChanged("Current");
        this.IsAtEnd = false;
        this.IsAtStart = (this.currentCustomer == 0);
    }
}
...
}
```



Nota A propriedade *Count* retorna o número de itens de uma coleção, mas lembre-se de que os itens de uma coleção são numerados de 0 a *Count* – 1.

Esses métodos atualizam a variável *currentCustomer* para fazer referência ao próximo cliente (ou ao anterior) na lista de clientes. Observe que esses métodos mantêm controle sobre os valores das propriedades *IsAtStart* e *IsAtEnd*, e indicam que o cliente atual mudou, disparando o evento *PropertyChanged* para a propriedade *Current*. Esses métodos são privados porque não devem ser acessíveis fora da classe *ViewModel*. Classes externas executarão esses métodos utilizando comandos, os quais você vai adicionar nos passos a seguir.

7. Adicione as propriedades automáticas *NextCustomer* e *PreviousCustomer*, mostradas aqui, à classe *ViewModel*.

```
public class ViewModel : INotifyPropertyChanged
{
    private List<Customer> customers;
    private int currentCustomer;
    public Command NextCustomer { get; private set; }
    public Command PreviousCustomer { get; private set; }
    ...
}
```

A visualização se vinculará a esses objetos *Command* para que o usuário possa navegar entre os clientes.

8. No construtor de *ViewModel*, defina as propriedades *NextCustomer* e *PreviousCustomer* para fazer referência aos novos objetos *Command*, como segue:

```
public ViewModel()
{
    this.currentCustomer = 0;
    this.IsAtStart = true;
    this.IsAtEnd = false;
    this.NextCustomer = new Command(this.Next, () =>
        { return this.customers.Count > 0 && !this.IsAtEnd; });
    this.PreviousCustomer = new Command(this.Previous, () =>
        { return this.customers.Count > 0 && !this.IsAtStart; });
    ...
}
```

O *Command NextCustomer* especifica o método *Next* como a operação a ser executada quando o método *Execute* for chamado. A expressão lambda `() => { return this.customers.Count > 0 && !this.IsAtEnd; }` é especificada como a função a ser chamada quando o método *CanExecute* executar. Essa expressão retorna *true* desde que a lista de clientes contenha pelo menos um cliente e o *ViewModel* não esteja posicionado no último cliente da lista. O *Command PreviousCustomer* segue o mesmo padrão: ele chama o método *Previous* para recuperar o cliente anterior da lista, e o método *CanExecute* referencia a expressão `() => { return this.customers.Count > 0 && !this.IsAtStart; }`, a qual retorna *true* desde que a lista de clientes contenha pelo menos um cliente e o *ViewModel* não esteja posicionado no primeiro cliente dessa lista.

9. No menu Build, clique em Build Solution e verifique se seu aplicativo compila sem erros.

Agora que você adicionou os comandos *NextCustomer* e *PreviousCustomer* ao *ViewModel*, pode vinculá-los aos botões da visualização. Quando o usuário clicar em um botão, o comando apropriado será executado.

A Microsoft publica diretrizes para a adição de botões nas visualizações em aplicativos Windows Store, e a recomendação geral é que os botões que chamam comandos devem ser colocados na barra de aplicativo (app bar). Os aplicativos Windows Store fornecem duas barras de aplicativo: uma aparece na parte superior do formulário e a outra na parte inferior. Os botões que navegam por um aplicativo ou por dados normalmente são colocados na barra de aplicativo superior, e essa é a estratégia que vamos adotar no próximo exercício.



Nota As diretrizes da Microsoft para a implementação de barras de aplicativo podem ser encontradas em <http://msdn.microsoft.com/library/windows/apps/hh465302.aspx>.

Adicione botões Next e Previous ao formulário Customers

1. Abra o arquivo MainPage.xaml na janela Design View.
2. Vá até a parte inferior do painel XAML e adicione a marcação mostrada em negrito a seguir, imediatamente acima da tag `</Page>` de fechamento:

```
...
<Page.TopAppBar>
    <AppBar IsSticky="True">
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <AppBarButton x:Name="previousCustomer" Icon="Back"
Command="{Binding Path=PreviousCustomer}"/>
            <AppBarButton x:Name="nextCustomer" Icon="Forward"
Command="{Binding Path=NextCustomer}"/>
        </StackPanel>
    </AppBar>
</Page.TopAppBar>
</Page>
```

Existem várias considerações a respeito desse fragmento de marcação XAML:

- Por padrão, a barra de aplicativo aparece quando o usuário clica com o botão direito do mouse no formulário, pressiona a tecla Windows+Z ou desliza (gesto de swipe) a partir da margem superior ou inferior do formulário. A barra de aplicativo permanece aberta até que o usuário execute outra ação no formulário – momento em que ela desaparece de novo. Você pode impedir que uma barra de aplicativo desapareça automaticamente, definindo a propriedade *IsSticky* como *true*, como mostrado nessa marcação. Nesse caso, a barra de aplicativo permanece aberta até que o usuário clique com o botão direito do mouse no formulário, pressione a tecla Windows+Z ou deslize (swipe) a barra de aplicativo a partir da margem superior do formulário. Aplicando essa configuração, o usuário pode se mover rapidamente entre os clientes, sem ter de repetir o gesto a fim de exibir a barra de aplicativo a cada vez.
- O controle *AppBar* contém um controle *StackPanel*. Assim como muitos controles, um *AppBar* pode ter apenas um conteúdo. Se precisar exibir vários itens em um controle, você deve usar um controle contêiner, como um *Grid* ou um *StackPanel*. Neste caso, um *StackPanel* é o mais conveniente, e os itens que ele exibe serão dispostos horizontalmente.
- O controle *AppBar* utiliza controles *AppBarButton*. Eles são versões estilizadas dos controles *Button* padrão, projetados especificamente para uso em uma barra de aplicativo (são circulares e respondem com um realce quando o usuário deixa o cursor sobre eles ou clica neles, embora seja possível substituir esse comportamento, definindo seus próprios layouts, transformações e transições como parte das propriedades do controle). Você pode exibir um ícone indicando a finalidade do controle *AppBarButton*. Os templates da Windows App Store contêm diversos ícones (como *Back* e *Forward* mostrados no exemplo de código anterior), e também é possível definir ícones e bitmaps personalizados.

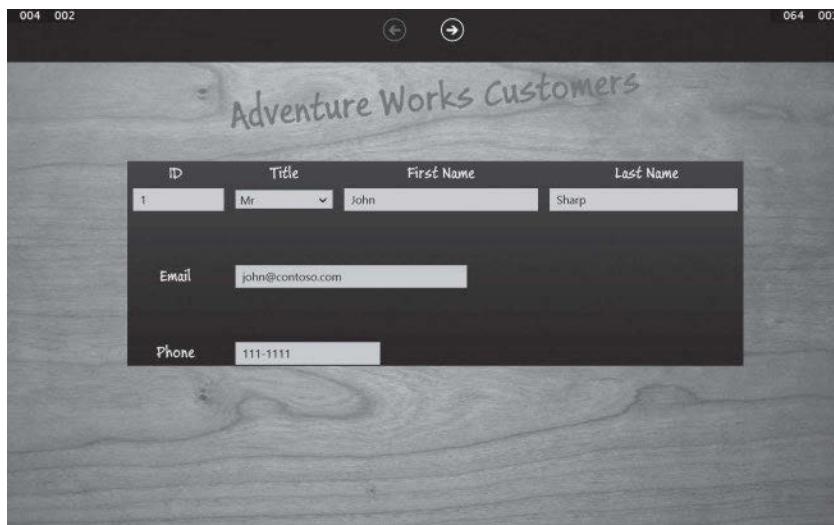


Nota Você pode ver o conjunto completo de ícones disponíveis usando a janela Properties. Na seção *Icon*, clique no ícone *Symbol*. Os ícones disponíveis são exibidos no controle *listbox* que aparece. Também é possível adicionar uma legenda textual a um controle *AppBarButton*, configurando a propriedade *Label*. Mas você deve ter cuidado ao fazer isso, especialmente se estiver projetando um aplicativo que poderá ser utilizado em outras culturas. Em vez disso, deve simplesmente manter-se fiel aos ícones, os quais foram projetados para serem reconhecidos em nível internacional, independentemente do idioma.

- Cada botão tem uma propriedade *Command*. Essa é a propriedade que você pode vincular a um objeto que implementa a interface *ICommand*. Neste aplicativo, você vinculou os botões aos comandos *PreviousCustomer* e *NextCustomer* na classe *ViewModel*. Quando o usuário clicar em um desses botões em tempo de execução, o comando correspondente será executado.
3. No menu Debug, clique em Start Without Debugging.
- O formulário *Customers* deve aparecer e exibir os detalhes de John Sharp.

- Clique com o botão direito do mouse em qualquer lugar no fundo do formulário.

A barra de aplicativo deve aparecer na parte superior do formulário e exibir os botões Next e Previous, como mostrado na imagem a seguir:



Observe que o botão *Previous* não está disponível. Isso porque a propriedade *IsAtStart* do *ViewModel* é *true* e o método *CanExecute* do objeto *Command* referenciado pelo botão *Previous* indica que o comando não pode ser executado.

- Na barra de aplicativo, clique em *Next*.

Os detalhes do cliente 2, Diana Sharp, devem aparecer e, após um curto atraso de até 1 segundo, o botão *Previous* deve se tornar disponível. A propriedade *IsAtStart* não é mais *true*; portanto, o método *CanExecute* do comando retorna *true*. Mas o botão não é notificado dessa mudança no estado até que o objeto cronômetro do comando expire e dispare o evento *CanExecuteChanged*, o que pode levar até um segundo para ocorrer.



Nota Se você precisa de uma reação mais instantânea para a mudança no estado dos comandos, pode providenciar para que o cronômetro na classe *Command* expire com mais frequência. Entretanto, evite reduzir demais o tempo, pois disparar o evento *CanExecuteChanged* com muita frequência pode afetar o desempenho da interface do usuário.

- Na barra de aplicativo, clique em *Next* novamente.
- Os detalhes do cliente 3, Francesca Sharp, devem aparecer e, após um curto atraso de até 1 segundo, o botão *Next* não deve mais estar disponível. Desta vez, a propriedade *IsAtEnd* do *ViewModel* é *true*, de modo que o método *CanExecute* do objeto *Command* do botão *Next* retorna *false* e o comando é desabilitado.

8. Redimensione a janela para exibir o aplicativo na visualização estreita e verifique se ele continua a funcionar corretamente. Os botões Next e Previous devem avançar e retroceder pela lista de clientes.
9. Clique com o botão direito do mouse em qualquer lugar no fundo do formulário. A barra de aplicativo deve desaparecer.
10. Retorne ao Visual Studio e interrompa a depuração.

Contratos do Windows 8.1

O Capítulo 25 menciona brevemente que um aplicativo Windows Store pode implementar um ou mais contratos do Windows 8.1. Um contrato define uma interface do Windows 8.1 com a qual um aplicativo pode implementar ou consumir um recurso estipulado pelo sistema operacional, como busca de informações, compartilhamento de dados ou atuar como um recurso de seleção de arquivos. Um contrato fornece um mecanismo padrão que é compartilhado por outros aplicativos Windows Store; os usuários que executam um aplicativo que implementa um contrato não precisam conhecer os procedimentos específicos do aplicativo para executar as tarefas fornecidas pelo contrato. Basicamente, os contratos tornam possível aos aplicativos Windows Store trabalharem juntos de forma transparente.

Os contratos mais utilizados são os seguintes:

- **Contrato Share Target** Com esse contrato, um aplicativo Windows Store pode se integrar na charm Compartilhar (Share) e atuar como destino para dados compartilhados.

Existem de fato dois lados no compartilhamento: um aplicativo Windows Store pode se registrar como origem de compartilhamento para especificar quais dados deseja compartilhar e em quais formatos. Um aplicativo de destino que pode consumir dados compartilhados deve implementar o contrato Share Target. Depois que um aplicativo de origem tiver registrado que pode compartilhar dados, o usuário pode utilizar a charm Compartilhar (Share) para exibir uma lista de aplicativos que implementam o contrato Share Target e selecionar um dessa lista. O contrato Share Target define eventos aos quais o aplicativo de destino deve responder; o aplicativo de destino utiliza esses eventos para solicitar e receber os dados da origem.

- **Contrato File Open Picker** Com esse contrato, um aplicativo Windows Store pode responder aos pedidos do selecionador de arquivos do Windows 8.1. Utilizando esse contrato, um aplicativo Windows Store pode fornecer acesso controlado aos dados que gerencia, aos usuários e a outros aplicativos. Esse contrato permite efetivamente que um aplicativo Windows Store se torne um parceiro de armazenamento local. Um aplicativo Windows Store que implemente o contrato tem total controle dos dados e das visualizações desses dados que apresenta ao selecionador de arquivos. O Windows 8.1 também fornece o contrato File Save Picker, com o qual um aplicativo pode controlar o modo como os dados que gerencia são armazenados.

- **Contrato Search** Com esse contrato, os usuários podem procurar os dados expostos por seu aplicativo, utilizando a charm Pesquisar (Search) do Windows 8.1 – esse é um mecanismo padrão utilizado por outros aplicativos Windows 8.1.

Implementar o contrato Search significa que os usuários não precisam aprender qualquer procedimento especial, específico de seu aplicativo, para procurar dados. O Windows 8.1 fornece a estrutura básica para fazer uma pesquisa, e tudo que você tem de fazer é fornecer a lógica que pega um pedido de busca e localiza os dados apropriados.



Nota Para mais informações sobre os contratos do Windows 8.1 e exemplos de como implementá-los, visite a página "App contracts and extensions" no site da Microsoft em <http://msdn.microsoft.com/library/windows/apps/hh464906.aspx>.

Implemente o contrato Search

O aplicativo Customers funciona bem se houver um número pequeno de registros, e você pode usar os botões Next e Previous para navegar pelas informações dos clientes. Mas, em um ambiente comercial, é improvável que você tenha um número tão pequeno de clientes (a não ser que sua empresa seja especialmente malsucedida!). Utilizar a funcionalidade de Next e Previous para percorrer uma lista de centenas de registros, a fim de encontrar os detalhes de um cliente em especial, é demorado e ineficiente. Para tornar o aplicativo mais prático, você deve oferecer um recurso de pesquisa e implementar o contrato Search.

O Visual Studio 2013 contém o template Search Contract. Esse template gera o código que faz a integração com a charm Pesquisar. Quando o usuário seleciona essa charm, os aplicativos que implementam o contrato Search são listados, junto com uma caixa de entrada de dados na qual o usuário pode especificar os dados a serem procurados. Se o usuário opta por pesquisar em seu aplicativo, os termos ou critérios de busca digitados por ele são passados para o aplicativo. Essas informações são utilizadas para filtrar os dados no aplicativo e determinar quais itens correspondem aos termos da busca. Então, seu aplicativo pode exibir uma lista de todos os registros coincidentes, utilizando uma página fornecida como parte do template Search Contract. Isso tudo parece muito complicado, mas na realidade grande parte da complexidade é implementada pelo template Search Contract. Todos os aplicativos que fornecem um contrato Search devem operar da mesma maneira; assim, a Microsoft pode fatorar grande parte do código no template, conforme você vai ver nos próximos exercícios. Neste exercício, você vai adicionar um contrato Search ao aplicativo Customers, por meio do qual o usuário poderá procurar clientes pelo nome ou pelo sobrenome.

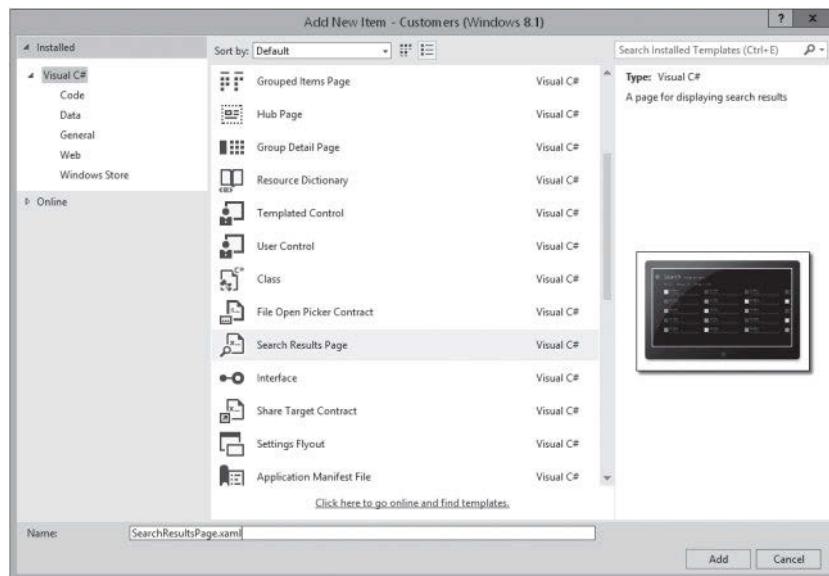
Implemente o contrato Search no aplicativo Customers

1. No Visual Studio, abra o projeto Customers, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 26\Search na sua pasta Documentos.

Essa versão do aplicativo Customers tem o mesmo ViewModel criado no exercício anterior, mas a origem de dados contém detalhes de muito mais clientes. As informações dos clientes ainda são armazenadas em um objeto *List<Customer>*, mas agora esse objeto é criado pela classe *DataSource* no arquivo *DataSource.cs*. A classe *ViewModel* referencia essa lista, em vez de criar a pequena coleção de três clientes utilizada no exercício anterior.

2. No Solution Explorer, clique com o botão direito do mouse no projeto Customers, aponte para Add e clique em New Item.

3. Na caixa de diálogo Add New Item – Customers, no painel esquerdo, clique em Windows Store. No painel central, selecione o template Search Results Page. Na caixa Name, digite **SearchResultsPage.xaml** e clique em Add, como mostrado na imagem a seguir:



O Visual Studio exibe uma caixa de mensagem com o texto "This addition depends on files that are missing from your project. Without these files you must resolve dependencies on the Common namespace manually. Add the missing files automatically?"



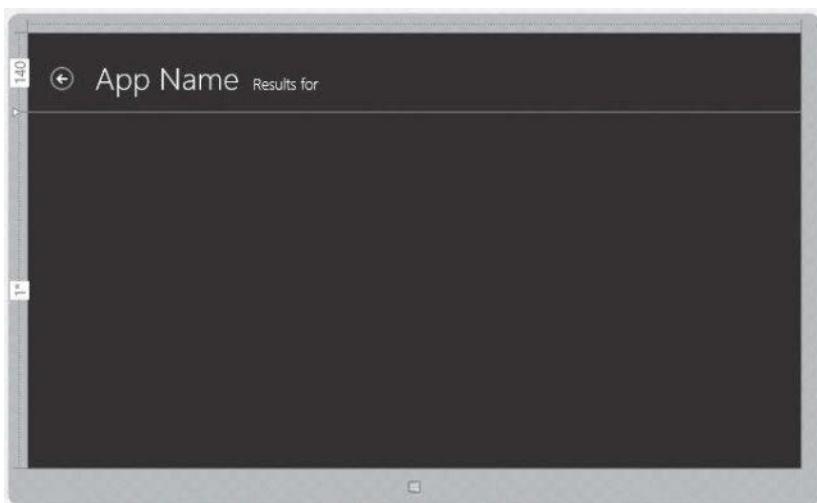
Essa mensagem ocorre porque o aplicativo Customers foi criado originalmente com o template Blank App, e esse template não contém todo o código e outros elementos exigidos pelo template Search Contract. Clique em Yes para permitir que o template Search Contract adicione esses itens.

O template Search Contract gera uma nova página XAML chamada SearchResultsPage.xaml, junto com um arquivo de código chamado SearchResultsPage.xaml.cs, o qual aparece na janela Code and Text Editor. Vários novos arquivos são adicionados à pasta Common. Esses arquivos contêm código e tipos de dados exigidos pela página SearchResultsPage.xaml.



Nota O Visual Studio também poderá relatar alguns erros no arquivo SearchResultsPage.xaml. Esses erros ocorrem porque o arquivo SearchResultsPage.xaml é criado antes que os arquivos exigidos sejam adicionados à pasta Common. Eles desaparecerão na próxima vez que você compilar o aplicativo.

4. No menu Build, clique em Build Solution.
5. No Solution Explorer, clique duas vezes no arquivo SearchResultsPage.xaml para exibi-lo na janela Design View. A página deve ser parecida com esta:



Se você examinar a marcação XAML dessa página, verá que a maior parte do conteúdo é organizada por um controle *Grid* chamado *resultsPanel*, o qual contém outro controle *Grid* chamado *typicalPanel*. O controle *Grid typicalPanel* contém os seguintes itens:

- Um controle *ItemsControl* chamado *filtersItemsControl*. Em tempo de execução, esse controle exibe uma lista de filtros com os quais o usuário pode especificar como vai aplicar os critérios de busca que digitou. No aplicativo Customers, você definirá filtros para que o usuário possa aplicar a pesquisa ao nome ou sobrenome dos clientes.
- Um controle *GridView* chamado *resultsGridView*. Os clientes que correspondem aos critérios de busca aparecem nesse controle e são formatados com um template chamado *StandardSmallIcon300x70ItemTemplate*. Esse template pode ser encontrado no arquivo StandardStyles.xaml, na pasta Common, e você vai modificá-lo para exibir dados dos clientes.

Abaixo do controle *Grid resultsPanel* existe outro controle *Grid* que contém os títulos e botões que aparecem na parte superior da página, seguido por um controle *TextBlock* chamado *noResultsTextBlock*. Esse controle *TextBlock* é exibido se o termo de busca digitado pelo usuário não corresponde a nenhum cliente.

No final do arquivo SearchResultsPage.xaml estão os grupos de estado visual utilizados pelo Visual State Manager para trocar o layout dos controles quando o usuário alterna entre as visualizações.



Nota Você pode modificar os estilos utilizados pelos elementos dessa página para que eles tenham a mesma aparência e comportamento do restante de seu aplicativo, mas não deve tentar mudar o layout da página nem adicionar ou remover controles. Para funcionar adequadamente, o contrato Search depende das definições corretas desses controles.

6. Na seção `<Page.Resources>`, próxima à parte superior do arquivo SearchResultsPage.xaml, mude o valor do recurso de string `AppName` para **Customers**, como mostrado aqui em negrito:

```
<Page.Resources>
    ...
    <!-- TODO: Update the following string to be the name of your app -->
    <x:String x:Key="AppName">Customers</x:String>
</Page.Resources>
```

O título exibido na janela Design View muda para Customers.

7. No Solution Explorer, expanda SearchResultsPage.xaml e, então, clique duas vezes em SearchResultsPage.xaml.cs para exibi-lo na janela Code and Text Editor.

Esse arquivo contém o código da classe `SearchResultsPage`. Essa classe define os seguintes métodos:

- ***navigationHelper_LoadState*** Esse método é executado quando o usuário digita um termo de busca na charm Pesquisar e seleciona o aplicativo Customers. Os critérios fornecidos pelo usuário são passados para esse método no parâmetro `LoadEventArgs`, e o código gerado por esse método extrai essas informações e as salva em uma variável local chamada `queryText`. O objetivo desse método é encontrar todos os itens que correspondem ao termo de busca e adicioná-los como coleções (denominadas filtros) ao ViewModel implementado por essa página. O código gerado pelo template cria um filtro padrão chamado All, o qual pode ser preenchido com os detalhes de cada cliente, mas neste aplicativo você vai remover o filtro All e criar filtros contendo os detalhes dos clientes com nomes ou sobrenomes que correspondam ao valor presente na variável `queryText`.



Nota A classe `SearchResultsPage` utiliza seu próprio ViewModel, chamado `DefaultViewModel`, definido como membro público da classe. Não confunda esse ViewModel com aquele que você criou anteriormente para o aplicativo Customers.

- ***Filter_Checked*** Esse método é executado quando o usuário seleciona um filtro. Os detalhes do filtro são fornecidos no parâmetro `RoutedEventArgs` e o código gerado por esse método recupera esse valor e o armazena na variável local `selectedFilter`. Nesse método, você deve atualizar o ViewModel para exibir os dados especificados por esse filtro.

Esse arquivo também contém a definição da classe *Filter* utilizada pela classe *SearchResultsPage*. Você não deve alterar o código dessa classe.

8. No Solution Explorer, expanda App.xaml e, então, clique duas vezes em App.xaml.cs para exibir o arquivo na janela Code and Text Editor.
9. Adicione o campo privado *_mainPageViewModel* e a propriedade pública *MainViewModel* ao início da classe *App*, como aparece em negrito a seguir:

```
sealed partial class App : Application
{
    private ViewModel _mainViewModel = null;
    public ViewModel MainViewModel
    {
        get { return this._mainViewModel; }
        set { this._mainViewModel = value; }
    }
    ...
}
```

Você vai usar a propriedade *MainViewModel* para permitir que a página *SearchResultsPage* acesse o ViewModel do formulário *MainPage*.

10. No Solution Explorer, expanda MainPage.xaml e, então, clique duas vezes em MainPage.xaml.cs para exibir o arquivo na janela Code and Text Editor.
11. No construtor de *MainPage*, adicione a instrução mostrada aqui em negrito:

```
public MainPage()
{
    ...
    ViewModel viewModel = new ViewModel();
    (Application.Current as App).MainViewModel = viewModel;
    this.DataContext = viewModel;
}
```

Essa instrução torna o ViewModel do formulário *MainPage* disponível por meio da propriedade *MainViewModel* da classe *App*. Observe que você pode acessar o objeto *App* do aplicativo atualmente em execução, utilizando a propriedade estática *Application.Current* e fazendo o casting do resultado como o tipo *App*.

12. No Solution Explorer, clique duas vezes no arquivo *ViewModel.cs* para exibi-lo na janela Code and Text Editor.
13. Na classe *ViewModel*, adicione a propriedade pública *AllCustomers* mostrada em negrito no código a seguir:

```
public class ViewModel : INotifyPropertyChanged
{
    private List<Customer> customers;
    public List<Customer> AllCustomers
    {
        get { return this.customers; }
    }
    ...
}
```

Essa propriedade torna a coleção de clientes utilizada pela classe *ViewModel* disponível para outras classes; você precisará acessar essa coleção na classe *SearchResultsPage*.

- 14.** Exiba o arquivo *SearchPageResults.xaml.cs* na janela Code and Text Editor. Adicione o seguinte campo privado, mostrado em negrito, ao início da classe *SearchResultsPage*:

```
public sealed partial class SearchResultsPage : ...
{
    private Dictionary<string, List<Customer>> searchResults =
        new Dictionary<string, List<Customer>>();
    ...
}
```

Essa coleção *Dictionary* conterá as listas de clientes que correspondem ao termo de busca especificado pelo usuário. Haverá duas listas nessa coleção: uma para os clientes com nomes coincidentes e outra para os clientes com sobrenomes coincidentes.

- 15.** No método *navigationHelper_LoadState* adicione a instrução mostrada em negrito a seguir, imediatamente após o comentário *TODO*:

```
protected override void navigationHelper_LoadState(object sender,
LoadStateEventArgs e)
{
    var queryText = navigationParameter as String;

    // TODO: lógica de pesquisa específica do aplicativo...
    // ...

    List<Customer> allCustomers =
        (Application.Current as App).MainViewModel.AllCustomers;
    ...
}
```

- 16.** No método *navigationHelper_LoadState*, transforme em comentário a seguinte instrução destacada em negrito:

```
...
var filterList = new List<Filter>();
// filterList.Add(new Filter("All", 0, true));
...
```

O contrato *Search* implementado pelo aplicativo *Customers* não dará suporte para a opção All.

- 17.** Após essa instrução, adicione o seguinte bloco de código mostrado em negrito:

```

var filterList = new List<Filter>();
// filterList.Add(new Filter("All", 0, true));

// Localiza todos os clientes cujos nomes
// ou sobrenomes correspondem ao texto da consulta
queryText = queryText.ToLower();
List<Customer> matchingFirstNames = new List<Customer>();
List<Customer> matchingLastNames = new List<Customer>();
foreach (Customer customer in allCustomers)
{
    string firstName = customer.FirstName.ToLower();
    string lastName = customer.LastName.ToLower();
    if (firstName.Contains(queryText))
    {
        matchingFirstNames.Add(customer);
    }
    if (lastName.Contains(queryText))
    {
        matchingLastNames.Add(customer);
    }
}

```

Esse código é o ponto crucial do contrato Search. Ele itera pela coleção *allCustomers* procurando os clientes que têm um valor na propriedade *FirstName* ou *Lastname* correspondente ao valor que está na variável *queryText*. O mecanismo de comparação elimina qualquer consideração com letras maiúsculas e minúsculas, convertendo todos os dados para minúsculas. O código adiciona uma referência para cada cliente correspondente nas coleções *matchingFirstNames* e *matchingLastNames*.

- 18.** Após o bloco anterior, adicione o seguinte código mostrado em negrito ao método *navigationHelper_LoadState*:

```

filterList.Add(new Filter(
    "Matching First Names", matchingFirstNames.Count, false));
filterList.Add(new Filter(
    "Matching Last Names", matchingLastNames.Count, false));
searchResults.Add("Matching First Names", matchingFirstNames);
searchResults.Add("Matching Last Names", matchingLastNames);

// Comunica os resultados por meio do view model
this.DefaultViewModel["QueryText"] = '\u201c' + queryText + '\u201d';

```

Esse código adiciona os detalhes das coleções *matchingFirstNames* e *matchingLastNames* à lista de filtros que será exibida pela página de resultados da busca. Essa informação consiste em um nome, junto com uma contagem do número de coincidências. As coleções em si são adicionadas à coleção *searchResults*. O nome de cada lista de clientes adicionados à coleção *searchResults* deve corresponder ao nome de cada filtro especificado na coleção *filterList*.

- 19.** No método *Filter_Checked*, imediatamente após o comentário *TODO*, adicione a instrução mostrada aqui em negrito:

```
// TODO: responder à mudança no filtro ativo ...
// ...
this.DefaultViewModel["Results"] =
    this.searchResults[selectedFilter.Name];
```

Essa instrução faz com que a lista de clientes especificada pelo filtro selecionado seja exibida na página de resultados da busca. A lista de clientes é recuperada da coleção *searchResults* pelo uso do nome do filtro (será “Matching First Names” ou “Matching Last Names”, conforme definido quando você adicionou os filtros à coleção *filterList*, no passo anterior).

20. No Solution Explorer, clique duas vezes em *SearchResultsPage.xaml* para exibi-lo na janela Code and Text Editor.
21. Localize a seção *<DataTemplate>* no controle *GridView resultsGridView*. Esse controle define o template utilizado pela *SearchResultsPage* para exibir os detalhes de cada cliente correspondente. Os controles *Image* e *TextBlock* desse controle *DataTemplate* utilizam vinculação de dados para exibir as propriedades de um objeto.

```
<DataTemplate>
<Grid Width="294" Margin="6">
    ...
    <Border ...>
        <Image Source="{Binding Image}" .../>
    </Border>
    <StackPanel Grid.Column="1" Margin="10,-10,0,0">
        <TextBlock Text="{Binding Title}" .../>
        <TextBlock Text="{Binding Subtitle}" .../>
        <TextBlock Text="{Binding Description}" .../>
    </StackPanel>
</Grid>
</DataTemplate>
```

A classe *Customer* tem uma propriedade *Title*, mas não tem as propriedades *Image*, *Subtitle* ou *Description*. Remova o controle *Border* e seu controle *Image* associado, e mude as vinculações de dados dos controles *TextBlock Subtitle* e *Description* para mostrar as propriedades *FirstName* e *LastName* em seu lugar:

```
<DataTemplate>
<Grid Width="294" Margin="6">
    ...
    <StackPanel Grid.Column="1" Margin="10,-10,0,0">
        <TextBlock Text="{Binding Title}" .../>
        <TextBlock Text="{Binding FirstName}" .../>
        <TextBlock Text="{Binding LastName}" .../>
    </StackPanel>
</Grid>
</DataTemplate>
```

O próximo passo é registrar o fato de que agora o aplicativo *Customers* implementa o contrato *Search* com o sistema operacional. Ao fazer isso, você também pode mudar os ícones que aparecem para o aplicativo na tela Iniciar do Windows e adicionar uma tela de abertura, que aparece quando o aplicativo começa a ser executado.

Registre o aplicativo Customers no Windows Search

1. No Solution Explorer, clique duas vezes no arquivo Package.appxmanifest para exibir o manifesto do aplicativo no editor de manifesto.

2. No editor de manifesto, clique na guia Declarations.

3. Na lista Available Declarations, selecione Search e clique em Add.

Essa ação registra o aplicativo Customers como provedor de pesquisa no Windows Search.

4. No Visual Studio, abra o arquivo App.xaml.cs na janela Code and Text Editor. No final da classe App, adicione o método *OnSearchActivated* mostrado aqui.

```
protected override void OnSearchActivated(
    Windows.ApplicationModel.Activation.SearchActivatedEventArgs args)
{
    var previousContent = Window.Current.Content;
    var frame = previousContent as Frame;

    frame.Navigate(typeof(SearchResultsPage), args.QueryText);
    Window.Current.Content = frame;

    Window.Current.Activate();
}
```

O evento *SearchActivated* é disparado quando o usuário procura dados no aplicativo utilizando a charm Pesquisar. O método *OnSearchActivated* do objeto *App* é executado quando esse evento é disparado. O código desse método obtém uma referência para o objeto *Frame* do Windows que atualmente exibe o aplicativo (o objeto *Frame* representa a área da tela que atualmente exibe o aplicativo) e, então, gera uma instância da classe *SearchResultsPage*, a qual exibe nesse quadro (frame), passando os critérios da consulta (disponíveis na propriedade *QueryText* do parâmetro *args*) digitados pelo usuário na charm Pesquisar dessa página.

5. No Solution Explorer, expanda a pasta Assets.

Essa pasta contém várias imagens gráficas: as imagens padrão exibidas pela tela de abertura quando o aplicativo começa a executar, o ícone que aparece se você fixa o aplicativo na tela Iniciar e o pequeno ícone que aparece na lista de aplicativos instalados no menu Iniciar ou quando o usuário ativa a charm Pesquisar (há também um arquivo de imagens que você pode anexar ao aplicativo, quando ele for carregado no Windows Store). Cada imagem tem um tamanho específico: a imagem do logotipo que aparece na tela Iniciar deve ter 150 × 150 pixels, a pequena imagem de logotipo que aparece com a charm Pesquisar deve ter 30 × 30 pixels e o logotipo usado pela tela de abertura deve ter 620 × 300 pixels.

6. Clique com o botão direito do mouse na pasta Assets, aponte para Add e então clique em Existing Item.

7. Na caixa de diálogo Add Existing Item – Customers, acesse a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 26\Resources na sua pasta Documentos, selecione todos os três arquivos dessa pasta e clique em Add.

Esses novos arquivos gráficos contêm imagens mais coloridas do que as cinza e branco comuns, fornecidas pelo template Blank App.

8. No Solution Explorer, clique duas vezes no arquivo Package.appxmanifest e, então, clique na guia Visual Assets.

Usando essa guia, você pode especificar a maneira como seu aplicativo se apresenta ao usuário, incluindo os logotipos que são exibidos.

9. Na caixa All Image Assets, clique em Square 150x150 Logo. Na caixa de texto Square 150x150 logo, digite **Assets\AdventureWorksLogo150x150.png**.

10. Na caixa All Image Assets, clique em Square 30x30 Logo. Na caixa de texto Square 30x30 logo, digite **Assets\AdventureWorksLogo30x30.png**.

11. Na caixa All Image Assets, clique em Splash Screen. Na caixa de texto Splash screen, digite **Assets\AdventureWorksLogo620x300.png**.

12. No menu Build, clique em Build Solution.

Agora você pode testar o contrato Search e verificar se funciona conforme o esperado.

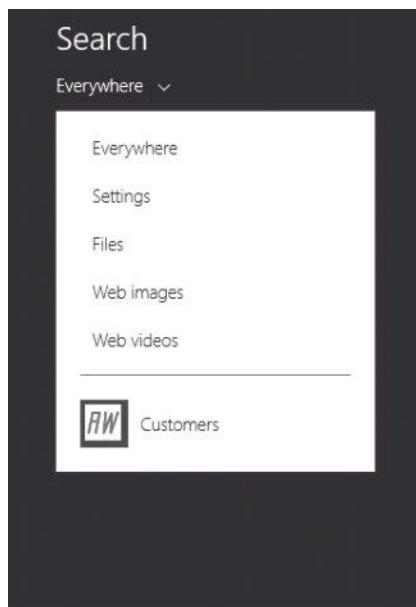
Teste o contrato Search

1. No menu Debug, clique em Start Debugging para executar o aplicativo. Observe que agora a tela de abertura aparece brevemente quando o aplicativo inicia, antes da janela Adventure Works Customers aparecer.

Quando o aplicativo aparecer, deverá exibir os detalhes do cliente 1, Orlando Gee.

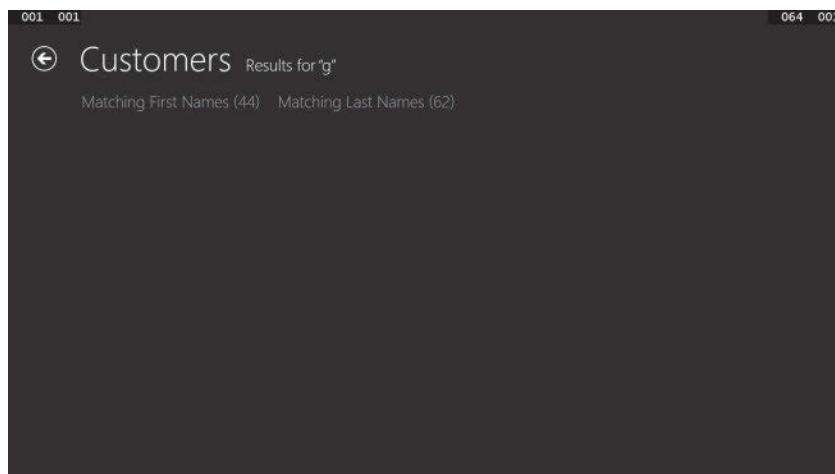
2. Pressione a tecla Windows+C para exibir a barra Charms e, então, clique em Pesquisar.

3. O painel Pesquisar deverá aparecer. Clique na lista suspensa e o aplicativo Customers deverá aparecer (junto com o logotipo pequeno), como mostra a imagem a seguir:



4. Clique no aplicativo Customers, digite **G** na caixa de texto e, então, clique no ícone Pesquisar.

A página de resultados da busca deve aparecer, exibindo o número de correspondências para clientes cujos nomes ou sobrenomes contenham a letra G. Observe que o nome especificado quando você adicionou o filtro à coleção *filterList* aparece no topo da página, junto com o número de correspondências.



5. Clique em Matching First Names.

Os dados identificados pelo filtro *Matching First Names* aparecem.

The screenshot shows a Windows Store app window titled 'Customers'. At the top, there are two buttons: 'Matching First Names (44)' and 'Matching Last Names (62)'. The main area displays a grid of 16 customer entries, each consisting of a title (Mr/Ms), first name, and last name. The entries are arranged in four columns and four rows. Some names in the list begin with the letter 'G', such as 'Gabriel', 'Greg', 'Gustavo', 'George', 'Gloria', 'Gordon', 'Gretchen', and 'Gwen'. The status bar at the bottom indicates the device is connected to '009 005' and has a battery level of '056 001'.

Mr	Ms	Mr	Mr
Douglas	Gabriele	Gabriel	Greg
Gronke	Dickmann	Beckeramp	Chapman
Ms	Mr	Ms	Ms
Margaret	Yer-Qiang	Megan	Angela
Kruska	Cheng	Davis	Barberard
Mr	Mr	Ms	Mr
Eugene	Roger	Virginia	Gustavo
Kogan	Lengel	Miller	Comargo
Mr	Ms	Mr	Mr
George	Peggy	Abigail	Douglas
Li	Justice	Gonzalez	Bateman
Ms	Mr	Ms	Ms
Yihong	Guo	Ingrid	Megan
Li	Gilbert	Burkhardt	Burke
Mr	Mr	Mr	Ms
Brigid	Grant	Gary	Jeanina Barreiro Gamarra
Grendish	Culbertson	Verges	Busino

6. Clique em Matching Last Names.

A lista de clientes cujos sobrenomes contêm a letra G deve aparecer.

7. Retorne ao Visual Studio e interrompa a depuração.

Navegue até um item selecionado

Adicionar funcionalidade de pesquisa básica é relativamente simples, mas você pode adicionar vários recursos para tornar essa funcionalidade mais útil. A mais importante é a capacidade de clicar no nome de um cliente na página de resultados da busca e ir diretamente para esse cliente no aplicativo *Customers*. É isso que você fará no último exercício deste capítulo.

Exiba o cliente selecionado a partir da página de resultados da busca

1. No Visual Studio, abra o arquivo *ViewModel.cs* na janela Code and Text Editor e, então, entre a propriedade *Current* e o método *Next*, adicione à classe *ViewModel* o método *GoTo* mostrado em negrito no código a seguir:

```
public Customer Current
{
    ...
}
```

```

public void GoTo(Customer customer)
{
    this.currentCustomer = this.customers.IndexOf(customer);
    this.OnPropertyChanged("Current");
    this.IsAtStart = (this.currentCustomer == 0);
    this.IsAtEnd = (this.customers.Count - 1 == this.currentCustomer);
}

private void Next()
{
    ...
}

```

O método *GoTo* recebe um objeto *Customer* como parâmetro e utiliza o método *IndexOf* para descobrir qual cliente está na coleção de clientes. Então, define esse cliente como o que está atualmente exibido.

2. Abra o arquivo SearchResultsPage.xaml na janela Design View.
3. No painel XAML, localize a marcação do controle *GridView resultsGridView* e adicione a propriedade *ItemClick* entre as propriedades *IsItemClickEnabled* e *ItemsSource*, como mostrado aqui em negrito:

```

<GridView
    x:Name="resultsGridView"
    ...
    IsItemClickEnabled="True"
    ItemClick="OnItemClick"
    ItemsSource="{Binding Source={...}}"
    ...
/>

```

A propriedade *ItemClick* especifica o nome do método de tratamento de evento a ser executado quando o usuário clicar em um item no controle *GridView*. Você vai escrever esse método em breve.

4. Abra o arquivo SearchPageResults.xaml.cs na janela Code and Text Editor e, então, após o construtor, adicione à classe *SearchResultsPage* o método *OnItemClick* mostrado no código a seguir:

```

private void OnItemClick(object sender, ItemClickEventArgs e)
{
    this.Frame.Navigate(typeof(MainPage), e.ClickedItem);
}

```

5. Adicione à classe *theSearchResultsPage* o método *OnItemClick* mostrado no código a seguir:

```

public sealed partial class SearchResultsPage : ...
{
    ...
    public SearchResultsPage()
    {
        this.InitializeComponent();
        this.navigationHelper = new NavigationHelper(this);
        this.navigationHelper.LoadState += navigationHelper_LoadState;
    }
}

```

```
private void OnItemClick(object sender, ItemClickEventArgs e)
{
    this.Frame.Navigate(typeof(MainPage), e.ClickedItem);
}
...
```

O método *OnItemClick* utiliza o método *Frame.Navigate* para exibir o formulário *MainPage*. O valor que está em *e.ClickedItem*, passado como parâmetro para o método *Navigate*, é uma referência para o cliente em cujo nome o usuário clicou na página de resultados da busca. O método *Navigate* causa a execução do método *OnNavigatedTo* na página de destino (neste caso, o formulário *MainPage*) e o item passado como parâmetro para o método *Navigate* é encaminhado para o método *OnNavigatedTo*.

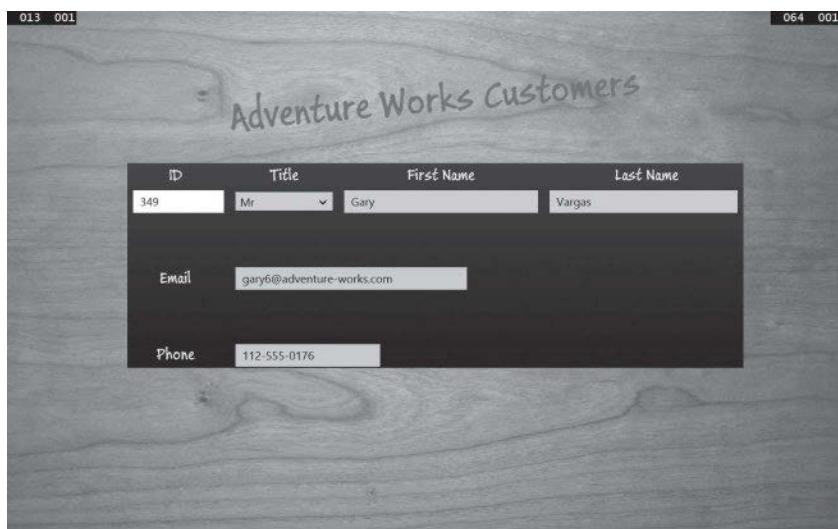
6. Abra o arquivo *MainPage.xaml.cs* na janela Code and Text Editor e, então, após o método *WindowSizeChanged*, adicione o método *OnNavigatedTo*, mostrado aqui, ao final da classe *MainPage*:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    Customer selectedCustomer = e.Parameter as Customer;

    // Se o Customer passado como parâmetro não é nulo
    // vai para esse cliente
    if (selectedCustomer != null)
    {
        ViewModel viewModel =
            (Application.Current as App).MainViewModel;
        viewModel.GoTo(selectedCustomer);
    }
    this.WindowSizeChanged(this, null);
}
```

Esse código utiliza o método *GoTo* da classe *ViewModel* para navegar até o cliente especificado, o qual será então exibido no formulário *MainPage*.

7. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
8. Quando o aplicativo aparecer, pressione a tecla Windows+C para exibir as charms e, então, clique em Search.
9. Digite **G** na caixa de texto e, então, clique no ícone do aplicativo Customers.
10. Na página de resultados da busca, clique em um dos clientes, como Gary Vargas. O formulário *MainPage* deve reaparecer, exibindo os detalhes do cliente:



- 11.** Retorne ao Visual Studio e interrompa a depuração.

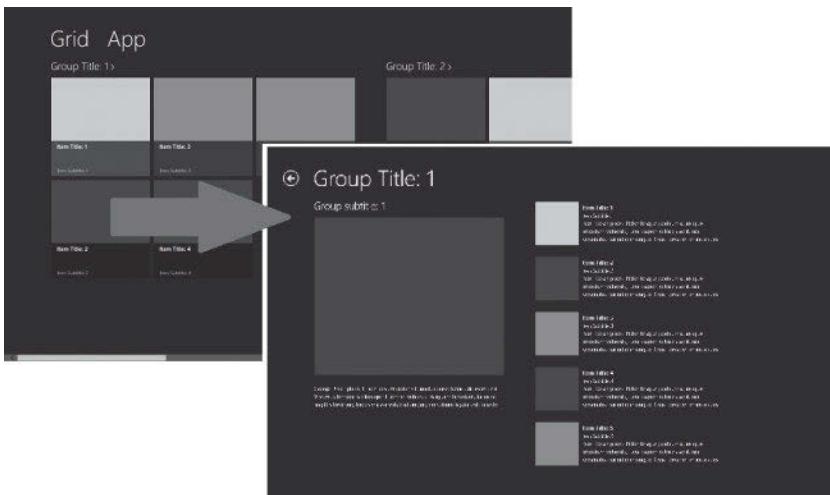
Resumo

Neste capítulo, você aprendeu a exibir dados em um formulário utilizando vinculação de dados. Viu como definir o contexto dos dados para um formulário e como criar uma origem de dados que suporta vinculação de dados, implementando a interface *INotifyPropertyChanged*. Aprendeu a utilizar o padrão Model-View-ViewModel para criar um aplicativo Windows Store. Viu como criar um ViewModel com o qual uma visualização pode interagir com uma origem de dados utilizando comandos. Por fim, você aprendeu a implementar o contrato Search para que um aplicativo Windows Store possa integrar funcionalidade de pesquisa nos recursos fornecidos pelo Windows 8.1.



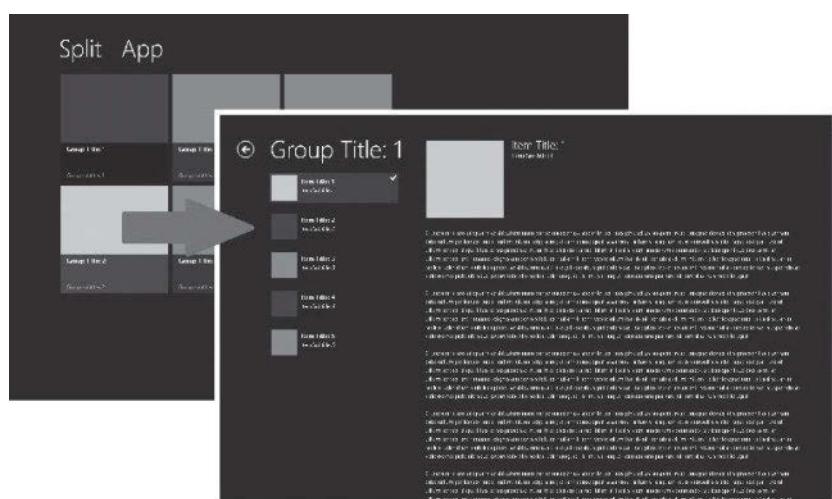
Nota Este capítulo e o anterior utilizaram o template Blank App como meio para mostrar como você cria um aplicativo Windows Store. O Visual Studio 2013 fornece três outros templates que oferecem um ponto de partida mais abrangente para compilar aplicativos Windows Store: o template Grid App, o template Split App e o template Hub App.

O template Grid App pode ser utilizado para exibir e editar dados hierárquicos que estejam organizados em grupos. Esse template gera um aplicativo que possui três páginas: uma página de nível superior, chamada Grouped Items, exibe uma lista de grupos; uma página de segundo nível, denominada Group Detail, exibe as informações detalhadas de um grupo; e uma página de terceiro nível, conhecida como Item Detail, exibe os itens de um grupo.

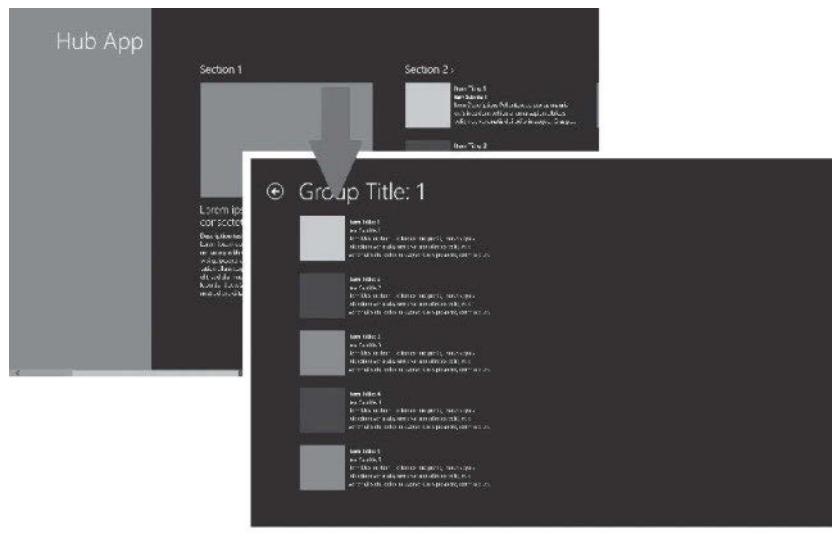


O template usa controles *GridView* e *ListView* para exibir informações utilizando vinculação de dados, e as páginas se adaptam às diferentes visualizações utilizando o Visual State Manager. Os dados são formatados e estilizados com estilos e templates de dados, da mesma maneira que você empregou neste capítulo e no anterior. O template também contém uma origem de dados de exemplo e um ViewModel simples. Você pode substituir a origem de dados de exemplo por seus dados de negócio e modificar o ViewModel para tratar de suas estruturas de dados. O objetivo é usar as páginas e o ViewModel como ponto de partida para seu aplicativo e ampliá-los com quaisquer páginas e lógica adicionais exigidas por ele. Você pode empregar as mesmas técnicas e estratégias que aprendeu neste capítulo.

O template Split App é conceitual e estruturalmente semelhante ao template Grid App, exceto que gera apenas duas páginas: uma página de nível superior, chamada Items, que exibe uma lista de grupos, e uma página de segundo nível, chamada Split, que exibe uma lista de itens em um grupo no lado esquerdo e os detalhes de um item selecionado no lado direito.



Por fim, o template Hub App fornece navegação no estilo hub. Uma página de nível superior possibilita ao usuário rolar entre diferentes grupos de itens de dados. Ele pode selecionar um grupo para exibir uma lista de itens desse grupo, e pode clicar em um item para ver os detalhes. Como os templates Grid App e Split App, o template Hub App ilustra as melhores práticas para estruturar esse tipo de aplicativo, e você pode personalizar as várias visualizações e modelos de visualização de acordo com requisitos específicos do negócio.



- Se quiser continuar no próximo capítulo, mantenha o Visual Studio 2013 executando e vá para o Capítulo 27, "Acesso a um banco de dados remoto em um aplicativo Windows Store".
- Se quiser encerrar o Visual Studio 2013 agora, no menu File, clique em Exit. Se vir uma caixa de diálogo Save, clique em Yes e salve o projeto.

Referência rápida

Para	Faça isto
Vincular a propriedade de um controle à propriedade de um objeto	Utilize uma expressão de vinculação de dados na marcação XAML do controle. Por exemplo: <code><TextBox ... Text="{Binding FirstName}" ...></code>
Permitir que um objeto notifique uma vinculação a respeito de uma mudança no valor de um dado	Implemente a interface <i>INotifyPropertyChanged</i> na classe que define o objeto e dispare o evento <i>PropertyChanged</i> sempre que o valor de uma propriedade mudar. Por exemplo: <code>class Customer : INotifyPropertyChanged { ... public event PropertyChangedEventHandler PropertyChanged; protected virtual void OnPropertyChanged(string propertyName) { if (PropertyChanged != null) { PropertyChanged(this, new PropertyChangedEventArgs(propertyName)); } } }</code>
Permitir que um controle que utiliza vinculação de dados atualize o valor da propriedade à qual está vinculado	Configure a vinculação de dados como bilateral. Por exemplo: <code><TextBox ... Text="{Binding FirstName, Mode=TwoWay}" ...></code>
Separar a lógica do negócio executada quando o usuário clica em um controle Button, da interface do usuário que contém o controle Button	Utilize um ViewModel que forneça comandos implementados com a interface <i>ICommand</i> e vincule o controle <i>Button</i> a um desses comandos. Por exemplo: <code><Button x:Name="nextCustomer" ... Command="{Binding Path=NextCustomer}"></code>

Para	Faça isto
Permitir que um aplicativo ofereça suporte para pesquisa utilizando a charm Pesquisar	Implemente o contrato Search com o template Search Contract. No método <i>navigationHelper_LoadState</i> da página de busca, adicione código para localizar todos os dados correspondentes ao termo de pesquisa digitado pelo usuário e crie filtros de pesquisa que contenham esses dados. No método <i>Filter_Checked</i> , troque para o filtro de pesquisa especificado pelo usuário. Trate o evento <i>ItemClick</i> na página de busca para navegar até o item selecionado pelo usuário nessa página e exibi-lo em seu aplicativo.

CAPÍTULO 27

Acesso a um banco de dados remoto em um aplicativo Windows Store

Neste capítulo, você vai aprender a:

- Utilizar o Entity Framework para criar um modelo de entidades que pode recuperar e modificar informações armazenadas em um banco de dados.
- Criar um web service Representational State Transfer (REST) que forneça acesso remoto a um banco de dados por meio de um modelo de entidades.
- Buscar dados de um banco de dados remoto utilizando um web service REST.
- Inserir, atualizar e excluir dados em um banco de dados remoto utilizando um web service REST.

O Capítulo 26, "Exibição e busca de dados em um aplicativo Windows Store", mostra como implementar o padrão Model-View-ViewModel (MVVM). Explica também como separar a lógica de negócio de um aplicativo da interface do usuário, utilizando uma classe *ViewModel* que dá acesso aos dados no modelo e que implementa comandos que a interface pode usar para ativar a lógica de negócio do aplicativo. O Capítulo 26 também ilustra como utilizar vinculação de dados para exibir os dados apresentados pelo *ViewModel* na interface do usuário e como a interface pode atualizar esses dados. Tudo isso resulta em um aplicativo Windows Store totalmente funcional.

Neste capítulo, voltaremos nossa atenção ao aspecto do modelo do padrão MVVM. Em especial, vamos ver como implementar um modelo que um aplicativo Microsoft Windows Store pode utilizar para recuperar e atualizar dados em um banco de dados remoto.

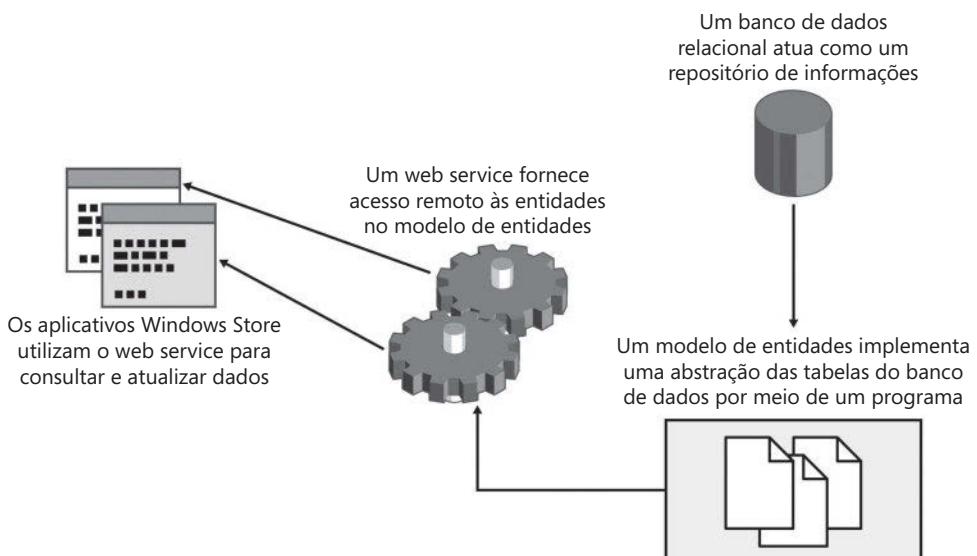
Recupere dados de um banco de dados

Até agora, os dados que utilizamos estavam confinados a uma coleção simples, incorporada ao *ViewModel* do aplicativo. No mundo real, os dados exibidos e mantidos por um aplicativo provavelmente estarão armazenados em uma origem de dados, como um banco de dados relacional.

Os aplicativos Windows Store não podem acessar um banco de dados relacional diretamente por meio das tecnologias fornecidas pela Microsoft (embora alguns outros fornecedores de banco de dados possam implementar suas próprias soluções). Isso pode parecer uma séria restrição, mas existem razões lógicas para essa limitação. Primeiramente, isso elimina qualquer dependência que um aplicativo Windows Sto-

re possa ter em relação a recursos externos, tornando-o um item independente que pode ser empacotado e baixado com facilidade da Windows Store, sem exigir que os usuários instalem e configurem um sistema de gerenciamento de banco de dados em seus computadores. Mas muitos aplicativos comerciais ainda precisarão acessar um banco de dados. Para lidar com essa situação, você pode utilizar um *web service*.

Os web services podem implementar diversas funções, mas um cenário comum é fornecer uma interface com a qual um aplicativo possa se conectar a uma origem de dados remota para recuperar e atualizar dados. Um web service pode estar localizado praticamente em qualquer lugar, desde o mesmo computador no qual o aplicativo está sendo executado até um servidor web instalado em um computador em outro continente. Desde que possa conectá-lo, você pode utilizar o web service para acessar o repositório de informações. O Microsoft Visual Studio fornece templates e ferramentas com as quais é possível construir um web service de forma muito rápida e fácil. A estratégia mais simples é basear o web service em um modelo de entidades gerado com o Entity Framework, como mostrado no diagrama a seguir:



O Entity Framework é um recurso poderoso com o qual é possível conectar-se a um banco de dados relacional. Ele pode reduzir o volume de código que a maioria dos desenvolvedores precisa escrever para adicionar recursos de acesso a dados a um aplicativo. É aí que vamos começar, mas primeiro você precisa configurar o banco de dados *AdventureWorks*; esse é o banco de dados que contém os detalhes dos clientes da *Adventure Works*.



Nota Não há espaço suficiente neste livro para dar muitos detalhes sobre como utilizar o Entity Framework, e os exercícios desta seção o conduzem apenas pelos passos mais básicos para você começar. Se quiser mais informações, investigue a página “Entity Framework” no site da Microsoft em <http://msdn.microsoft.com/data/aa937723>.

Instale o banco de dados AdventureWorks

1. Inicie o Visual Studio 2013, se ele ainda não estiver em execução.
2. No menu File, aponte para Open e então clique em File.
3. Na caixa de diálogo Open File, acesse a pasta Microsoft Press\Visual CSharp Step By Step\Chapter 27\AdventureWorks na sua pasta Documentos, clique em AttachDatabase.sql e então clique em Open.
4. Na janela Transact-SQL Editor que exibe o comando Create Database, altere o texto <YourName> para seu nome de usuário. Esse texto ocorre duas vezes: uma na segunda linha do comando e outra na terceira.
5. No menu SQL, clique em Execute para executar o comando.
6. Na caixa de diálogo Connect To Server, digite **(localdb)\v11.0** na caixa de texto Server Name, certifique-se de que a caixa Authentication esteja definida com Windows Authentication e, então, clique em Connect. Verifique se o comando termina sem erros.



Nota (localdb)\V11.0 é a string de conexão que identifica a versão do SQL Server instalada com o Visual Studio 2013. Às vezes, essa instância do SQL Server pode levar alguns segundos para iniciar, e você poderá receber um erro de tempo esgotado, após clicar em Connect. Se isso acontecer, execute o comando novamente e especifique os mesmos valores na caixa de diálogo Connect To Server; agora o SQL Server deve estar em execução e o comando deverá ser bem sucedido.

7. Feche a janela AttachDatabase.sql. Na caixa de mensagem que aparece, perguntando se você quer salvar as alterações feitas ao script, clique em No.

Crie um modelo de entidades

Agora que você já instalou o banco de dados AdventureWorks, pode utilizar o Entity Framework para criar um modelo de entidades, que um aplicativo pode usar para consultar e atualizar as informações desse banco de dados. Se você já trabalhou com bancos de dados, talvez conheça tecnologias como o ADO.NET, que fornecem uma biblioteca de classes que podem ser usadas para conectar a um banco de dados e executar comandos SQL. O ADO.NET é útil, mas exige que você tenha conhecimento satisfatório de SQL – e, se você não tomar cuidado, ele pode obrigá-lo a estruturar seu código em torno da lógica necessária para executar comandos SQL, em vez de se concentrar nas operações de negócio de seu aplicativo. O Entity Framework oferece um nível de abstração que reduz as dependências que seus aplicativos têm em relação à SQL.

Basicamente, o Entity Framework implementa uma camada de mapeamento entre um banco de dados relacional e seu aplicativo; ele gera um modelo de entidades que consiste em coleções de objetos que seu aplicativo pode utilizar, exatamente como qualquer outra coleção. Em geral, uma coleção corresponde a uma tabela no banco de dados, e cada linha de uma tabela corresponde a um item na coleção. Você realiza consultas iterando pelos itens de uma coleção, normalmente usando Language-Integrated Query (LINQ). Nos bastidores, o modelo de entidades converte suas consultas em comandos SELECT de SQL que buscam os dados. Você pode modificar os dados na coleção e, então, providenciar para que o modelo de entidades gere e execute os comandos SQL INSERT, UPDATE e DELETE apropriados para efetuar as operações equivalentes no banco de dados. Em resumo, o Entity Framework é um veículo excelente para se conectar a um banco de dados, recuperar e gerenciar dados, sem exigir a incorporação de comandos SQL em seu código.

No exercício a seguir, você vai criar um modelo de entidades muito simples para a tabela Customer do banco de dados *AdventureWorks*. Você vai seguir o que é conhecida como estratégia do *banco de dados-primeiro* na modelagem de entidades. Nessa estratégia, o Entity Framework gera classes com base nas definições das tabelas do banco de dados. O Entity Framework também torna possível empregar uma estratégia *código-primeiro*; essa estratégia pode gerar um conjunto de tabelas em um banco de dados, com base nas classes implementadas em seu aplicativo.



Nota Se quiser obter mais informações sobre a estratégia código-primeiro para criar um modelo de entidades, consulte a página "Code First to an Existing Database" no site da Microsoft em <http://msdn.microsoft.com/data/jj200620>.

Crie o modelo de entidades AdventureWorks

1. No Visual Studio, abra o projeto *Customers*, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 27\Web Service na sua pasta Documentos.

Esse projeto contém uma versão modificada do aplicativo *Customers* do conjunto de exercícios do Capítulo 26. O *ViewModel* implementa comandos adicionais para pular para o primeiro e para o último cliente na coleção de clientes, e a barra de aplicativo contém botões *First* e *Last* que ativam esses comandos.

Além disso, essa versão do projeto não implementa o contrato *Search*. O motivo por trás dessa omissão é que, fazendo isso, você pode se concentrar nos elementos básicos deste exercício, sem que os arquivos e outros elementos que constituem o contrato *Search* atrapalhem.

2. No Solution Explorer, clique com o botão direito do mouse na solução *Customers* (não no projeto *Customers*), aponte para *Add* e clique em *New Project*.

3. Na caixa de diálogo Add New Project, no painel da esquerda, clique na guia Web. No painel central, clique no template ASP.NET Web Application. Na caixa Name, digite **AdventureWorksService** e clique em OK.
4. Na caixa de diálogo New ASP.NET Project, clique em Web API e depois em OK.

Como mencionado no início desta seção, você não pode acessar um banco de dados relacional diretamente a partir de um aplicativo Windows Store, e isso inclui utilizar o Entity Framework. Em vez disso, você criou um aplicativo web (esse não é um aplicativo Windows Store), e vai hospedar o modelo de entidades que criou nesse aplicativo. O template Web API fornece assistentes e ferramentas com as quais você pode implementar rapidamente um web service, que é o que fará no próximo exercício. Esse web service fornecerá acesso remoto ao modelo de entidades para o aplicativo Windows Store Customers.

5. No Solution Explorer, clique novamente com o botão direito do mouse na solução Customers e, então, clique em Set Startup Projects.
6. Na caixa de diálogo Solution 'Customers' Property Pages, clique em Multiple Startup Projects. Defina a Action para o projeto AdventureWorksService como Start without debugging, defina a Action para o projeto Customers como Start e clique em OK.

Essa configuração garante que o aplicativo web AdventureWorksService seja executado sempre que você iniciar o projeto a partir do menu Debug.

7. Clique com o botão direito do mouse no projeto AdventureWorksService e, então, clique em Properties.
8. Na página de propriedades, clique na guia Web na coluna da esquerda.
9. Na página Web, clique em "Don't open a page. Wait for a request from an external application".

Normalmente, quando um aplicativo web é executado a partir do Visual Studio, o navegador web (Internet Explorer) se abre e tenta exibir a home page do aplicativo. O aplicativo AdventureWorksService não tem uma home page; o objetivo desse aplicativo é conter o web service ao qual os aplicativos clientes podem se conectar e recuperar dados do banco de dados AdventureWorks.

10. Na caixa Project Url, mude o endereço do aplicativo web para **http://localhost:50000/** e clique em Create Virtual Directory. Na caixa de mensagem do Microsoft Visual Studio que aparece, verifique se o diretório virtual foi criado com êxito e, então, clique em OK.

Por padrão, o template de projeto ASP.NET cria um aplicativo web hospedado pelo IIS Express e seleciona uma porta aleatória para o URL. Essa configuração define a porta como 50000 para que os passos subsequentes dos exercícios deste capítulo possam ser descritos mais facilmente.

11. No menu File, clique em Save All e, depois, feche a página de propriedades. No Solution Explorer, clique com o botão direito do mouse no projeto AdventureWorksService, aponte para Add e clique em New Item.

- 12.** Na caixa de diálogo Add New Item – AdventureWorksService, na coluna da esquerda, clique na guia Data. No painel central, clique no template ADO.NET Entity Data Model. Na caixa Name, digite **AdventureWorksModel.edmx** e clique em Add.

O Entity Data Model Wizard começa. Esse assistente pode ser usado para gerar um modelo de entidades a partir de um banco de dados existente.

- 13.** Na página Choose Model Contents do assistente, clique em Generate From Database e depois clique em Next.

- 14.** Na página Choose Your Data Connection, clique em New Connection.

- 15.** Na caixa de diálogo Connection Properties, na caixa Server Name, digite **(local-db)\v11.0**. Verifique se Use Windows Authentication está selecionado. Na caixa Select Or Enter A Database Name, digite **AdventureWorks** e clique em OK.

Essa ação cria uma conexão com o banco de dados AdventureWorks que você configurou no exercício anterior.

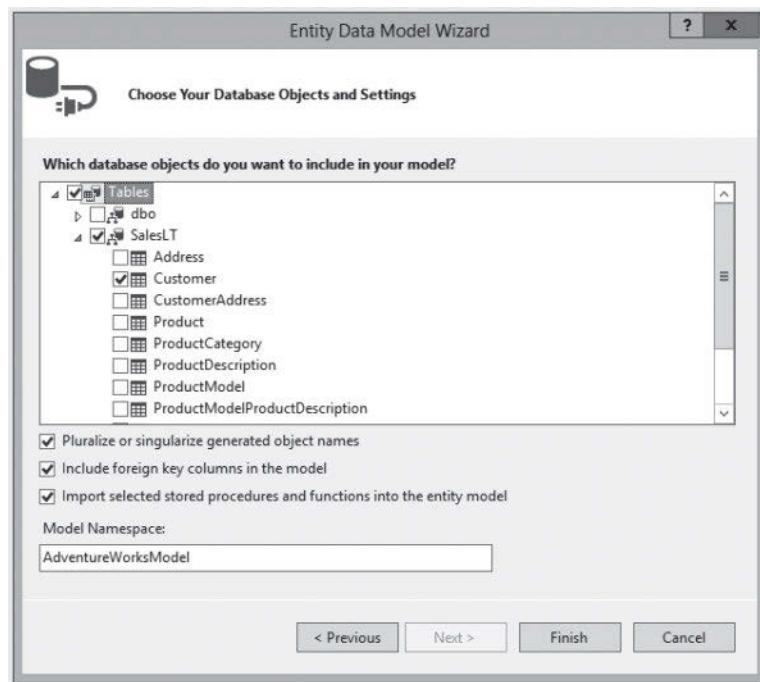
- 16.** Na página Choose Your Data Connection, verifique se Save Entity Connection Settings In Web.Config As está selecionado, confirme se o nome da string de conexão é AdventureWorksEntities e clique em Next.

- 17.** Na página Choose Your Version, aceite a versão padrão do Entity Framework e clique em Next.

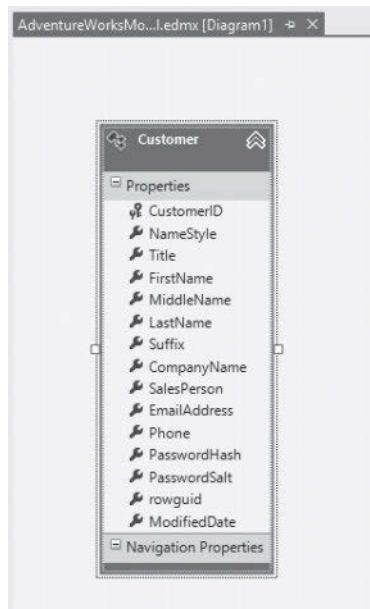


Nota Se estiver usando a edição Technical Preview do Visual Studio 2013, a versão padrão do Entity Framework referenciada pelo template ASP.NET Web Application é a 5.0. Na versão de lançamento, está previsto que isso mudará para a versão 6.0. Contudo, este aplicativo não depende de qualquer recurso específico da versão 6.0, e você pode compilar o aplicativo utilizando a versão 5.0.

- 18.** Na página Choose Your Database Objects And Settings, expanda Tables, expanda SalesLT e, em seguida, selecione Customer. Verifique se a caixa de seleção Pluralize Or Singularize Generated Object Names está selecionada (as duas outras opções dessa página também serão selecionadas por padrão), observe que o Entity Framework gera as classes para o modelo de entidades no namespace *AdventureWorksModel* e, então, clique em Finish.



O Entity Data Model Wizard gera um modelo de entidades para a tabela Customers e exibe na tela uma representação gráfica no editor Entity Model, como esta:



Se a seguinte caixa de mensagem Security Warning aparecer, marque a caixa de seleção Do Not Show This Message Again e clique em OK. Esse aviso de segurança aparece porque o Entity Framework utiliza uma tecnologia conhecida como templates T4 para gerar o código de seu modelo de entidades, e baixou esses templates da web utilizando NuGet. Os templates do Entity Framework foram verificados pela Microsoft e podem ser utilizados com segurança.



19. No editor Entity Model, clique com o botão direito do mouse na coluna NameStyle e, então, clique em Delete From Model. Utilizando a mesma técnica, exclua do modelo de entidades as colunas MiddleName, Suffix, CompanyName, SalesPerson, PasswordHash e PasswordSalt.

O aplicativo Customers não utiliza essas colunas, e não há necessidade de recuperá-las do banco de dados. Mas você não deve remover as colunas rowguid e ModifiedDate. Elas são usadas pelo banco de dados para identificar linhas na tabela Customers e monitorar as alterações feitas nessas linhas em um ambiente multiusuário. Se você remover essas colunas, não poderá salvar dados no banco de dados corretamente.

20. No menu Build, clique em Build Solution.
21. No Solution Explorer, expanda AdventureWorksModel.edmx, expanda AdventureWorksModel.tt e clique duas vezes em Customers.cs.

Esse arquivo contém a classe gerada pelo Entity Data Model Wizard para representar um cliente. Essa classe contém propriedades automáticas para cada uma das colunas da tabela Customer que você incluiu no modelo de entidades:

```
public partial class Customer
{
    public int CustomerID { get; set; }
    public string Title { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
    public string Phone { get; set; }
    public System.Guid rowguid { get; set; }
    public System.DateTime ModifiedDate { get; set; }
}
```

22. No Solution Explorer, sob AdventureWorksModel.edmx, expanda AdventureWorksModel.Context.tt e clique duas vezes em AdventureWorksModel.Context.cs.

Esse arquivo contém a definição de uma classe chamada *AdventureWorksEntities* (ela tem o mesmo nome que você utilizou quando gerou a conexão com o banco de dados no Entity Data Model Wizard):

```
public partial class AdventureWorksEntities : DbContext
{
    public AdventureWorksEntities()
        : base("name=AdventureWorksEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public DbSet<Customer> Customers { get; set; }
}
```

A classe *AdventureWorksEntities* é descendente da classe *DbContext*, e esta classe fornece a funcionalidade que um aplicativo usa para se conectar ao banco de dados. O construtor passa um parâmetro para o construtor da classe base, especificando o nome de uma string de conexão a ser utilizada para conectar o banco de dados. Se você examinar o arquivo Web.config, encontrará essa string na seção *<ConnectionStrings>*. Ele contém os parâmetros (dentre outras coisas) que você especificou quando executou o Entity Data Model Wizard.

Você pode ignorar o método *OnModelCreating* da classe *AdventureWorksEntities*. O único outro item é a coleção *Customers*. Essa coleção tem o tipo *DbSet<Customer>*. O tipo genérico *DbSet* fornece métodos com os quais é possível adicionar, inserir, excluir e consultar objetos em um banco de dados; ele funciona em conjunto com a classe *DbContext* para gerar os comandos SQL SELECT apropriados, necessários para buscar informações de cliente do banco de dados e preencher a coleção, assim como para criar os comandos SQL INSERT, UPDATE e DELETE que são executados se objetos *Customer* forem adicionados, modificados ou removidos da coleção. Uma coleção *DbSet* é frequentemente chamada de *conjunto de entidades* (ou *entity set*).

Crie e utilize um web service REST

Você criou um modelo de entidades que fornece operações para recuperar e manter informações de clientes, e o próximo passo é implementar um web service para que um aplicativo Windows Store possa acessar o modelo de entidades.

Com o Visual Studio 2013 é possível criar rapidamente um web service em um aplicativo web ASP.NET, baseado diretamente em um modelo de entidade gerado com o Entity Framework. O web service utiliza o modelo de entidades para recuperar dados de um banco de dados e atualizar o banco de dados. Crie um web service com o assistente Add Scaffold. Esse assistente pode gerar um web service que implementa o modelo REST. O modelo REST utiliza um esquema de navegação para representar objetos e serviços de negócios sobre uma rede, e o protocolo HTTP para transmitir requisições de acesso a esses objetos e serviços. Um aplicativo cliente que acessa um recurso envia uma requisição na forma de um URL, que o web service analisa (via par-

sing) e processa. Por exemplo, a Adventure Works poderia publicar as informações dos clientes, expondo os detalhes de cada cliente como um recurso único, por meio de um esquema semelhante ao seguinte:

<http://Adventure-Works.com/DataService/Customers/1>

O acesso a esse URL faz o web service recuperar os dados do cliente 1. Esses dados podem ser retornados em diversos formatos, mas, por questões de portabilidade, a maioria dos formatos comuns envolve XML e JavaScript Object Notation (JSON). Uma resposta JSON típica, gerada por uma requisição web service REST que executa a consulta anterior, é como a seguinte:

```
{  
    "CustomerID":1,  
    "Title":"Mr",  
    "FirstName":"Orlando",  
    "LastName":"Gee",  
    "EmailAddress":"orlando0@adventure-works.com",  
    "Phone":"245-555-0173"  
}
```

O modelo REST depende do aplicativo que acessa os dados e que envia o verbo HTTP adequado como parte da solicitação utilizada para acessar os dados. Por exemplo, a solicitação simples mostrada anteriormente deve enviar uma solicitação HTTP GET para o web service. No HTTP, também há suporte para outros verbos, como POST, PUT e DELETE, que você pode utilizar para criar, modificar e remover recursos, respectivamente. Escrever o código para gerar as requisições HTTP adequadas e realizar parsing sobre as respostas retornadas por um web service REST parece muito complicado. Felizmente, o Add Scaffold Wizard pode gerar a maior parte desse código para você, deixando-o livre para se concentrar na lógica do negócio de seu aplicativo.

No exercício a seguir, você vai criar um web service REST simples para o modelo de entidades AdventureWorks. Esse web service tornará possível a um aplicativo cliente consultar e manter informações dos clientes.

Crie o web service AdventureWorks

1. No Visual Studio, no projeto AdventureWorksService, clique com o botão direito do mouse na pasta Controllers, aponte para Add e clique em New Scaffold Item.
2. No Add Scaffold Wizard, no painel central, faça uma rolagem para baixo e clique no template Web API 2 Controller with read/write actions, using Entity Framework, e clique em Add.

Na caixa de diálogo Add Controller, na caixa Controller Name, digite **CustomersController**. Selecione Customer (AdventureWorksService) na lista Model Class, selecione AdventureWorksEntities (AdventureWorksService) na lista Data Context Class e clique em Add.

Em um web service criado com o template ASP.NET Web API, todas as requisições web recebidas são tratadas por uma ou mais classes controladoras, e cada classe controladora expõe métodos associados a cada um dos diferentes tipos de requisições REST para cada um dos recursos expostos. Por exemplo, CustomersController se parece com:

```
public class CustomersController : ApiController
{
    private AdventureWorksEntities db = new AdventureWorksEntities();

    // GET api/Customers
    public IEnumerable<Customer> GetCustomer()
    {
        return db.Customers.AsEnumerable();
    }

    // GET api/Customers/5
    public Customer GetCustomer(Int32 id)
    {
        Customer customer = db.Customers.Find(id);
        if (customer == null)
        {
            throw new HttpResponseException(
                Request.CreateResponse(HttpStatusCode.NotFound));
        }

        return customer;
    }

    // PUT api/Customers/5
    public HttpResponseMessage PutCustomer(Int32 id, Customer customer)
    {
        ...
    }

    // POST api/Customers
    public HttpResponseMessage PostCustomer(Customer customer)
    {
        ...
    }

    // DELETE api/Customers/5
    public HttpResponseMessage DeleteCustomer(Int32 id)
    {
        ...
    }

    ...
}
```

O primeiro método *GetCustomer* lida com requisições para recuperar todos os clientes, e atende a esse pedido simplesmente retornando a coleção *Customers* inteira do modelo de dados Entity Framework que você criou antes. Em segundo plano, o Entity Framework busca todos os clientes do banco de dados e utiliza essa informação para preencher a coleção *Customers*. Esse método é chamado se um aplicativo envia uma requisição HTTP GET para o URL *api/Customers* desse web service.

O método *GetCustomer* é sobre carregado, e a segunda versão recebe um parâmetro inteiro. Esse parâmetro especifica a *CustomerID* de um cliente determinado e esse método utiliza o Entity Framework para localizar os detalhes desse cliente, antes de retorná-lo. Esse método é executado quando um aplicativo envia uma requisição HTTP GET para o URL *api/Customers/n*, onde n é a ID do cliente a ser recuperado.

O método *PutCustomer* é executado quando um aplicativo envia uma requisição HTTP PUT para o web service. A requisição especifica a ID e os detalhes de um cliente, e o código nesse método utiliza o Entity Framework para atualizar o cliente especificado, usando as informações de detalhe (o funcionamento interno desse método não está mostrado no exemplo de código anterior).

O método *PostCustomer* responde às requisições HTTP POST e recebe os detalhes de um cliente como parâmetro. Esse método adiciona um novo cliente com esses detalhes no banco de dados.

Por fim, o método *DeleteCustomer* lida com requisições HTTP DELETE e remove o cliente com a ID especificada.

O template ASP.NET Web API gera automaticamente código que direciona as requisições para o método apropriado nas classes controladoras, e você pode adicionar mais classes controladoras, caso precise gerenciar outros recursos, como Products ou Orders.



Nota Para obter informações detalhadas sobre a implementação de web services REST com o template ASP.NET Web API, visite a página "Web API" em <http://www.asp.net/web-api>.

Você também pode criar classes controladoras manualmente, utilizando o mesmo padrão mostrado pela classe *CustomersController* — não é preciso buscar e armazenar dados em um banco de dados utilizando o Entity Framework. O template ASP.NET Web API contém um exemplo de controladora no arquivo *ValuesController.cs*, que você pode copiar e ampliar com seu próprio código.

3. Na pasta Controllers, clique com o botão direito do mouse no arquivo *ValuesController.cs* e, então, clique em Delete. Na caixa de mensagem, clique em OK para confirmar que você deseja excluir esse arquivo.

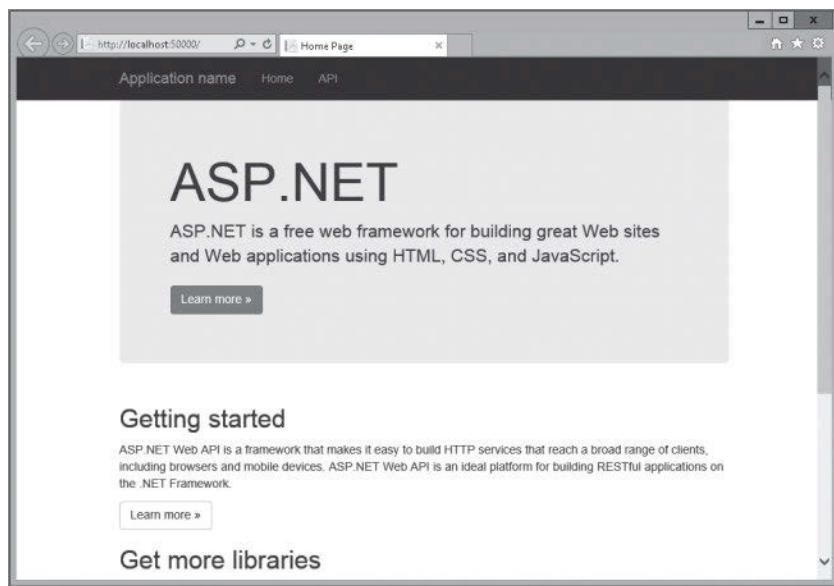
Não vamos usar a classe de exemplo *ValuesController* neste exercício.



Nota Não exclua a classe *HomeController*. Essa classe controladora atua como ponto de entrada para o aplicativo web que contém o web service.

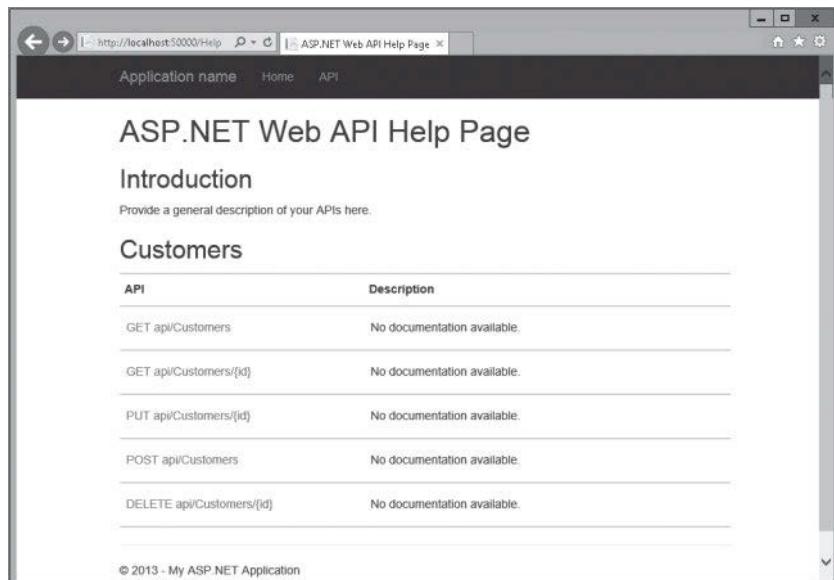
4. No Solution Explorer, verifique se você configurou o web service corretamente. Clique com o botão direito do mouse no projeto *AdventureWorksService* e, então, no menu de atalho que aparece, aponte para View e clique em View In Browser.

Seu navegador web iniciará e deverá exibir a seguinte página:



5. Na barra de título, clique em API.

Aparece outra página, que resume as requisições REST que um aplicativo pode enviar para o web service:



6. Na barra de endereço, digite **http://localhost:50000/api/Customers/1** e, então, pressione Enter. Quando a mensagem "Do you want to open or save 1.json (223 bytes) from localhost?" aparecer, clique em Open.

Essa requisição é direcionada para o método *GetCustomer* sobrecarregado da classe *CustomersController*, e o valor 1 é passado como parâmetro para esse método. Observe que a API Web utiliza rotas que começam com o caminho "api" após o endereço do servidor.

O método *GetCustomer* recupera do banco de dados os detalhes do cliente 1 e os retorna como um objeto formatado em JSON. O arquivo 1.json contém esses dados, o qual pode ser aberto no Bloco de Notas. Os dados se parecem com:

```
{"CustomerID":1,"Title":"Mr","FirstName":"Orlando","LastName":"Gee","EmailAddress":"orlando0@adventure-works.com","Phone":"245-555-0173","rowguid":"3f5ae95e-b87d-4aed-95b4-c3797afcb74f","ModifiedDate":"2001-08-01T00:00:00"}
```

7. Feche o Bloco de Notas, feche o navegador web e retorne ao Visual Studio.

A próxima etapa desta jornada é conectar-se ao web service a partir do aplicativo Windows Store Customers e, então, utilizar o web service para buscar alguns dados.

O .NET Framework fornece a classe *HttpClient*, que pode ser usada por um aplicativo para formular e enviar requisições HTTP REST para um web service, e a classe *HttpResponseMessage*, que um aplicativo pode usar para processar o resultado do web service. Essas classes abstraem os detalhes do protocolo HTTP do código de seu aplicativo. Assim, você pode se concentrar na lógica de negócios que exibe e manipula os objetos publicados por meio do web service. Você vai utilizar essas classes no próximo exercício.

Instalando o pacote ASP.NET Web API Client Libraries

As classes *HttpClient* e *HttpResponseMessage* fazem parte do pacote ASP.NET Web API Client Libraries. Talvez seja necessário baixar e instalar a última versão desse pacote e adicioná-la ao projeto Customers antes de continuar. Para instalar o pacote, execute os passos a seguir:

1. No Solution Explorer, clique com o botão direito do mouse no projeto Customers e, então, no menu de atalho que aparece, clique em Manage NuGet Packages.
2. Na janela Customers – Manage NuGet Packages, no painel da esquerda, clique em Online. Acima do painel central, na caixa de lista à esquerda, selecione Include Prerelease. Na caixa de busca acima do painel da direita, digite **Web API**.
3. No painel central, selecione Microsoft ASP.NET Web API 2 Client e clique em Install.
4. Na caixa de diálogo License Acceptance, caso queira aceitar os termos de licenciamento, clique em I Accept.
5. Feche a janela Customers – Manage NuGet Packages.

Busque dados do web service AdventureWorks

1. No Solution Explorer, no projeto Customers, clique com o botão direito do mouse no arquivo `DataSource.cs` e, então, no menu de atalho, clique em `Delete`. Na caixa de mensagem, clique em `OK` para confirmar que você deseja excluir o arquivo.

Esse arquivo continha os dados de exemplo utilizados pelo aplicativo `Customers`. Você vai modificar a classe `ViewModel` para buscar esses dados do web service, de modo que esse arquivo não é mais necessário.

2. No Solution Explorer, clique duas vezes em `ViewModel.cs` para exibir o arquivo na janela `Code and Text Editor`.
3. Adicione as seguintes diretivas `using` à lista localizada no início do arquivo:

```
using System.Net.Http;
using System.Net.Http.Headers;
```

4. Na classe `ViewModel`, adicione as seguintes variáveis mostradas em negrito, antes do construtor de `ViewModel`:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    private const string ServerUrl = "http://localhost:50000/";
    private HttpClient client = null;

    public ViewModel()
    {
        ...
    }
    ...
}
```

A variável `ServerUrl` contém o endereço de base do web service. Você vai usar a variável `client` para se conectar ao web service.

5. No construtor de `ViewModel`, inicialize a lista de clientes com `null` e adicione as seguintes instruções, que configuram a variável `client`:

```
public ViewModel()
{
    ...
    this.customers = null;
    this.client = new HttpClient();
    this.client.BaseAddress = new Uri(ServerUrl);
    this.client.DefaultRequestHeaders.Accept =
        Add(new MediaTypeWithQualityHeaderValue("application/json"));
}
```

A lista de clientes contém os clientes exibidos pelo aplicativo; ela já foi preenchida com os dados do arquivo `DataSource.cs` que você removeu.

A variável `client` é inicializada com o endereço do servidor web ao qual enviará requisições. Um web service REST pode receber requisições e enviar respostas em diversos formatos, mas o aplicativo `Customers` utilizará JSON. A última instrução no código anterior configura a variável `client` para enviar requisições nesse formato.

6. No Solution Explorer, clique duas vezes no arquivo `Customer.cs` na pasta-raiz do projeto `Customers`, para exibi-lo na janela Code and Text Editor.
7. Imediatamente após a propriedade `Phone`, adicione à classe `Customer` as propriedades públicas mostradas em negrito no código a seguir:

```
public class Customer : INotifyPropertyChanged
{
    ...
    public string Phone
    {
        ...
    }

    public System.Guid rowguid { get; set; }
    public System.DateTime ModifiedDate { get; set; }

    ...
}
```

O web service recupera esses campos do banco de dados, e o aplicativo `Customers` deve estar preparado para tratar deles; caso contrário, eles serão perdidos se o usuário modificar os detalhes de um cliente (você vai acrescentar essa capacidade ao aplicativo `Customers` mais adiante neste capítulo).

8. Retorne à classe `ViewModel`. Após o construtor, adicione o método público `GetDataAsync` mostrado aqui:

```
public async Task GetDataAsync()
{
    try
    {
        var response = await this.client.GetAsync("api/customers");
        if (response.IsSuccessStatusCode)
        {
            var customerData =
                await response.Content.ReadAsAsync<IEnumerable<Customer>>();
            this.customers = customerData as List<Customer>;
            this.currentCustomer = 0;
            this.OnPropertyChanged("Current");
            this.IsAtStart = true;
            this.IsAtEnd = (this.customers.Count == 0);
        }
        else
        {
            // TODO: tratar falha de GET
        }
    }
    catch (Exception e)
    {
        // TODO: tratar exceções
    }
}
```

```
    }
```

Esse método é assíncrono; ele utiliza o método `GetAsync` do objeto `HttpClient` para ativar a operação `api/customers` do web service. Essa operação busca os detalhes dos clientes no banco de dados AdventureWorks. O próprio método `GetAsync` é assíncrono e retorna um objeto `HttpResponseMessage` empacotado em uma `Task`. O `HttpResponseObject` contém um código de status que indica se a requisição foi bem-sucedida e, se foi, o aplicativo utiliza o método `ReadAsStringAsync` da propriedade `Content` do objeto `HttpResponseMessage` para recuperar os dados retornados pelo web service. Esses dados são atribuídos à coleção de clientes, a propriedade `currentCustomer` do `ViewModel` é definida de modo a apontar para o primeiro cliente dessa coleção e as propriedades `IsAtStart` e `IsAtEnd` são inicializadas de modo a indicar o estado do `ViewModel`.



Nota O pacote ASP.NET Web API Client Libraries presume que todas as requisições de web service podem levar um tempo indeterminado para executar e, consequentemente, classes como `HttpClient` e `HttpResponseMessage` são projetadas para aceitar operações assíncronas, a fim de evitar o bloqueio de um aplicativo enquanto espera por uma resposta. Na verdade, essas classes aceitam apenas operações assíncronas; não existem versões síncronas de `GetAsync` ou `ReadAsStringAsync`.

Observe que, atualmente, o método `GetDataAsync` não trata exceções ou falhas, a não ser por consumi-las silenciosamente. Veremos uma técnica para informar exceções em um aplicativo Windows Store mais adiante neste capítulo.



Nota Em um aplicativo de produção, você não deve buscar dados pela rede desnecessariamente. Em vez disso, deve ser seletivo quanto aos dados que recupera. Mas, neste aplicativo, o banco de dados AdventureWorks contém apenas algumas centenas de clientes, de modo que todos são recuperados e armazenados em cache na lista de clientes.

- 9.** Modifique o método de acesso *get* da propriedade *Current* como mostrado a seguir:

```
public Customer Current
{
    get
    {
        if (this.customers != null)
        {
            return this.customers[currentCustomer];
        }
        else
        {
            return null;
        }
    }
}
```

O método *GetDataAsync* é assíncrono; portanto, existe a possibilidade de que a coleção de clientes possa não ser preenchida quando os controles do formulário *MainPage* tentarem se vincular a um cliente. Nessa situação, essa modificação impede que as vinculações de dados gerem uma exceção de referência nula ao acessarem a coleção de clientes.

- 10.** No construtor de *ViewModel*, atualize as condições que tornam possível a execução de cada um dos comandos, como mostrado em negrito no código a seguir:

```
public ViewModel()
{
    ...
    this.NextCustomer = new Command(this.Next,
        () => { return this.customers != null &&
            this.customers.Count > 0 && !this.IsAtEnd; });
    this.PreviousCustomer = new Command(this.Previous,
        () => { return this.customers != null &&
            this.customers.Count > 0 && !this.IsAtStart; });
    this.FirstCustomer = new Command(this.First,
        () => { return this.customers != null &&
            this.customers.Count > 0 && !this.IsAtStart; });
    this.LastCustomer = new Command(this.Last,
        () => { return this.customers != null &&
            this.customers.Count > 0 && !this.IsAtEnd; });
}
```

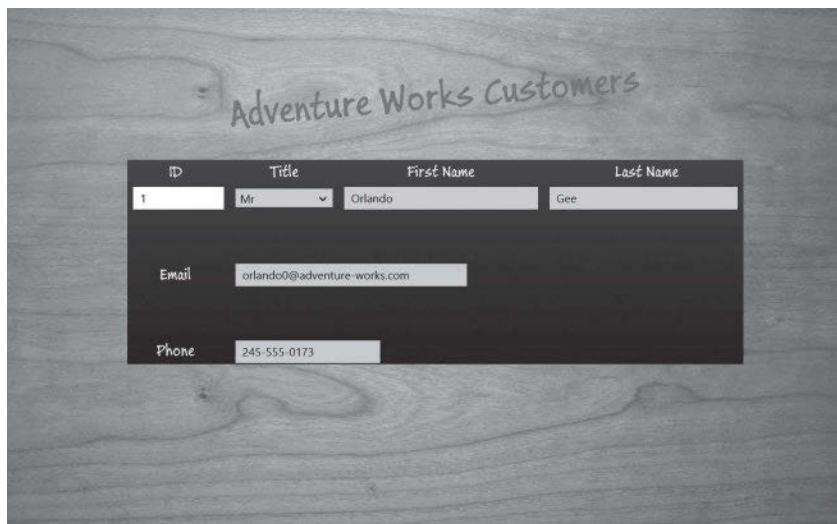
Essas alterações garantem que os botões da barra do aplicativo não sejam habilitados até que exista algum dado para exibir.

- 11.** No Solution Explorer, expanda *MainPage.xaml* e clique duas vezes em *MainPage.xaml.cs* para abri-lo na janela Code and Text Editor.
- 12.** Adicione a instrução a seguir, mostrada em negrito, ao construtor de *MainPage*:

```
public MainPage()
{
    ...
    ViewModel viewModel = new ViewModel();
    viewModel.GetDataAsync();
    this.DataContext = viewModel;
}
```

Essa instrução preenche o ViewModel.

13. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.
14. Inicialmente, o formulário aparecerá vazio, enquanto o método *GetDataAsync* executar, mas após alguns segundos, os detalhes do primeiro cliente, Orlando Gee, deverão aparecer:



15. Clique com o botão direito do mouse no formulário para exibir a barra do aplicativo. Utilize os botões de navegação para se mover pela lista de clientes, a fim de verificar se o formulário funciona conforme o esperado.
16. Retorne ao Visual Studio e interrompa a depuração.

Como um último floreio para esta seção, quando o formulário for exibido inicialmente, seria interessante permitir que o usuário soubesse que, embora o formulário pareça estar vazio, o aplicativo está no processo de busca de dados. Em um aplicativo Windows Store, você pode utilizar um controle *ProgressRing* para dar esse retorno. Esse controle deve ser exibido quando o ViewModel estiver ocupado, se comunicando com o web service, mas de resto inativo.

Adicione um indicador de ocupado ao formulário Customers

1. Exiba o arquivo ViewModel.cs na janela Code and Text Editor. Após o método `GetDataAsync`, adicione o campo privado `_isBusy` e a propriedade pública `IsBusy` à classe `ViewModel`, como ilustrado aqui:

```
private bool _isBusy;
public bool IsBusy
{
    get { return this._isBusy; }
    set
    {
        this._isBusy = value;
        this.OnPropertyChanged("IsBusy");
    }
}
```

2. No método `GetDataAsync`, adicione as seguintes instruções mostradas em negrito:

```
public async Task GetDataAsync()
{
    try
    {
        this.IsBusy = true;
        var response = await this.client.GetAsync("api/customers");
        ...
    }
    catch (Exception e)
    {
        // TODO: tratar exceções
    }
    finally
    {
        this.IsBusy = false;
    }
}
```

O método `GetData` define a propriedade `IsBusy` com *true* antes de executar a consulta para buscar as informações do cliente. O bloco `finally` garante que a propriedade `IsBusy` seja definida novamente com *false*, mesmo que ocorra uma exceção.

3. Abra o arquivo `MainPage.xaml` na janela Design View.
4. No painel XAML, adicione como primeiro item no controle `Grid` de nível superior o controle `ProgressRing` mostrado em negrito no código a seguir:

```
<Grid Style="{StaticResource GridStyle}">
    <ProgressRing HorizontalAlignment="Center"
    VerticalAlignment="Center" Foreground="AntiqueWhite"
    Height="100" Width="100" IsActive="{Binding IsBusy}"
    Canvas.ZIndex="1"/>
    <Grid x:Name="customersTabularView" Margin="40,104,0,0" ...>
        ...
    </Grid>
</Grid>
```

Definir a propriedade *Canvas.ZIndex* com “1” garante que o *ProgressRing* apareça na frente dos outros controles exibidos pelo controle *Grid*.

5. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.

Observe que, quando o aplicativo inicia, o anel de progresso aparece brevemente antes que o primeiro cliente seja exibido. Se você verificar que o primeiro cliente aparece rápido demais, pode introduzir um pequeno atraso no método *GetData*, apenas para ter certeza de que o anel de progresso está funcionando. Adicione a seguinte instrução, a qual faz uma pausa de 5 segundos no método:

```
public async Task GetData()
{
    try
    {
        this.IsBusy = true;
        await Task.Delay(5000);
        this.connection = new AdventureWorksEntities(...);
        ...
    }
    ...
}
```

Certifique-se de remover essa instrução quando terminar de testar o anel de progresso.

6. Retorne ao Visual Studio e interrompa a depuração.

Insira, atualize e exclua dados por meio de um web service REST

Além de oferecer aos usuários a capacidade de consultar e exibir dados, muitos aplicativos terão o requisito de também permitir que eles insiram, atualizem e exclam informações. A API Web do ASP.NET implementa um modelo que oferece suporte para essas operações, utilizando requisições HTTP PUT, POST e DELETE. Por convenção, uma requisição PUT modifica um recurso existente em um web service e uma requisição POST cria uma nova instância de um recurso. Uma requisição DELETE remove um recurso. O código gerado pelo Add Scaffold Wizard do template ASP.NET Web API segue essas convenções.

Idempotência em web services REST

Em um web service REST, as requisições PUT devem ser idempotentes. Isso significa que, se você fizer a mesma atualização repetidamente, o resultado deverá ser sempre o mesmo. No exemplo AdventureWorksService, se você modificar um cliente e configurar o número do telefone como "888-888-8888", não importará quantas vezes executar essa operação, pois o efeito é constante. Isso poderia parecer óbvio, mas você deve projetar um web service REST tendo esse requisito em mente. Com essa estratégia de projeto, um web service pode ser robusto em face de requisições simultâneas, ou mesmo em caso de falhas de rede (se um aplicativo cliente perder a conexão com o web service, pode simplesmente tentar se reconectar e executar a mesma requisição outra vez, sem se preocupar se a requisição anterior foi bem-sucedida). Portanto, considere um web service REST como uma maneira de armazenar e recuperar dados, e não como um conjunto de operações específicas de negócio.

Por exemplo, se estivesse construindo um sistema bancário, você poderia ficar tentado a fornecer um método *CreditAccount* (crédito em conta) que adicionasse um valor ao saldo na conta de um cliente, e expor esse método como uma operação PUT. Mas, sempre que executasse essa operação, o resultado seria um crédito progressivo na conta. Portanto, torna-se necessário monitorar se as chamadas à operação são bem-sucedidas. Seu aplicativo não poderá executar essa operação repetidamente se considerar que uma chamada anterior falhou ou atingiu o tempo limite, pois poderia resultar em vários créditos duplicados na mesma conta.

No próximo exercício, você vai ampliar o aplicativo Customers e adicionar recursos com os quais os usuários podem adicionar novos clientes e modificar os detalhes de clientes já existentes, construindo as requisições REST adequadas e enviando-as para o web service AdventureWorksService. Você não fornecerá nenhuma funcionalidade para excluir clientes. Essa restrição garante que você tenha um registro de todos os clientes que fizeram negócios com a organização Adventure Works, o qual pode ser necessário em caso de auditoria. Além disso, mesmo que um cliente não esteja ativo há muito tempo, existe uma chance de que possa fazer um pedido futuramente.

Implemente funcionalidade de adição e edição na classe *ViewModel*

1. No Visual Studio, abra o projeto Customers, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 27\Updatable ViewModel na sua pasta Documentos.

O código do arquivo *ViewModel.cs* está ficando extenso, de modo que foi reorganizado em regiões para ficar mais fácil de gerenciar. A classe *ViewModel* também foi estendida com as seguintes propriedades booleanas, que indicam o "modo" no qual o *ViewModel* está operando: *Browsing*, *Adding* ou *Editing*. Essas propriedades são definidas na região chamada *Properties For Managing The Edit Mode* (Propriedades para gerenciar o modo de edição):

- **IsBrowsing** Essa propriedade indica se o ViewModel está no modo Browsing. Quando o ViewModel está no modo Browsing, os comandos *FirstCustomer*, *LastCustomer*, *PreviousCustomer* e *NextCustomer* são habilitados e uma visualização pode chamá-los para percorrer os dados.
- **IsAdding** Essa propriedade indica se o ViewModel está no modo Adding. Nesse modo, os comandos *FirstCustomer*, *LastCustomer*, *PreviousCustomer* e *NextCustomer* são desabilitados. Você vai definir um comando *AddCustomer*, um comando *SaveChanges* e um comando *DiscardChanges*, que serão habilitados nesse modo.
- **IsEditing** Essa propriedade indica se o ViewModel está no modo Editing. Como no modo Adding, nesse modo, os comandos *FirstCustomer*, *LastCustomer*, *PreviousCustomer* e *NextCustomer* são desabilitados. Você vai definir um comando *EditCustomer*, que será habilitado nesse modo. O comando *SaveChanges* e o comando *DiscardChanges* também serão habilitados, mas o comando *AddCustomer* será desabilitado. O comando *EditCustomer* será desabilitado no modo Adding.
- **IsAddingOrEditing** Essa propriedade indica se o ViewModel está no modo Adding ou Editing. Você vai usar essa propriedade nos métodos que definirá neste exercício.
- **CanBrowse** Essa propriedade retorna *true* se o ViewModel está no modo Browsing e há uma conexão aberta para o web service. O código no construtor que cria os comandos *FirstCustomer*, *LastCustomer*, *PreviousCustomer* e *NextCustomer* foi atualizado para utilizar essa propriedade, a fim de determinar se esses comandos devem ser habilitados ou desabilitados, como segue:

```
public ViewModel()
{
    ...
    this.NextCustomer = new Command(this.Next,
        () => { return this.CanBrowse &&
            this.customers.Count > 0 && !this.IsAtEnd; });
    this.PreviousCustomer = new Command(this.Previous,
        () => { return this.CanBrowse &&
            this.customers.Count > 0 && !this.IsAtStart; });
    this.FirstCustomer = new Command(this.First,
        () => { return this.CanBrowse &&
            this.customers.Count > 0 && !this.IsAtStart; });
    this.LastCustomer = new Command(this.Last,
        () => { return this.CanBrowse &&
            this.customers.Count > 0 && !this.IsAtEnd; });
}
```

- **CanSaveOrDiscardChanges** Essa propriedade retorna *true* se o ViewModel está no modo Adding ou Editing e tem uma conexão aberta para o web service.

A região Methods For Fetching And Updating Data (Métodos para buscar e atualizar dados) contém os seguintes métodos:

- **GetData** Este é o mesmo método que você criou anteriormente neste capítulo. Ele se conecta ao web service e recupera os detalhes de cada cliente.
- **ValidateCustomer** Este método recebe um objeto *Customer* e examina as propriedades *FirstName* e *LastName* para garantir que não estejam vazias. Inspecciona também as propriedades *EmailAddress* e *Phone* para verificar se elas contêm informações que estão em um formato válido. O método retorna *true* se os dados são válidos e *false*, caso contrário. Você vai usar esse método quando criar o comando *SaveChanges*, mais adiante neste exercício.



Nota O código que valida as propriedades *EmailAddress* e *Phone* utiliza expressões regulares, por meio da classe *Regex* definida no namespace *System.Text.RegularExpressions*. Para usar essa classe, defina uma expressão regular em um objeto *Regex* que especifique o padrão a que os dados devem corresponder e, então, chame o método *IsMatch* do objeto *Regex* com os dados que você precisa validar. Para obter mais informações sobre expressões regulares e sobre a classe *Regex*, visite a página "Regular Expression Object Model" no site da Microsoft em <http://msdn.microsoft.com/library/30wbz966>.

- **CopyCustomer** O objetivo desse método é criar uma cópia rasa de um objeto *Customer*. Você vai utilizá-lo quando criar o comando *EditCustomer*, para fazer uma cópia dos dados originais de um cliente, antes de serem alterados. Se o usuário optar por descartar as alterações, elas podem simplesmente ser copiadas de volta, a partir da cópia feita por esse método.
2. No Solution Explorer, expanda o projeto *Customers* e clique duas vezes no arquivo *ViewModel.cs* para exibi-lo na janela Code and Text Editor.
 3. No arquivo *ViewModel.cs*, expanda a região Methods For Fetching And Updating Data. Nessa região, acima do método *ValidateCustomer*, crie o método *Add* mostrado aqui:

```
// Cria um novo cliente (vazio)
// e coloca o formulário no modo de inclusão (Adding)
private void Add()
{
    Customer newCustomer = new Customer { CustomerID = 0 };
    this.customers.Insert(currentCustomer, newCustomer);
    this.IsAdding = true;
    this.OnPropertyChanged("Current");
}
```

Esse método cria um novo objeto *Customer*. Ele está vazio, a não ser pela propriedade *CustomerID*, que está temporariamente definida com 0 para propósitos de exibição; o verdadeiro valor dessa propriedade é gerado quando o cliente é salvo no banco de dados, conforme já descrito. O cliente é adicionado à lista de clientes (a visualização utiliza vinculação de dados para exibir os dados dessa lista), o *ViewModel* é colocado no modo Adding e o evento *PropertyChanged* é disparado para indicar que o cliente *Current* foi alterado.

- 4.** Adicione a seguinte variável *Command*, mostrada em negrito, ao início da classe *ViewModel*:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command LastCustomer { get; private set; }
public Command AddCustomer { get; private set; }
    ...
}
```

- 5.** No construtor de *ViewModel*, instancie o comando *AddCustomer*, como mostrado aqui em negrito:

```
public ViewModel()
{
    ...
    this.LastCustomer = new Command(this.Last, ...);
this.AddCustomer = new Command(this.Add,
    O => { return this.CanBrowse; });
}
```

Esse código referencia o método *Add* que você acabou de criar. O comando é habilitado se o *ViewModel* tem uma conexão com o web service e está no modo Browsing (o comando *AddCustomer* não será habilitado se o *ViewModel* já estiver no modo Adding).

- 6.** Após o método *Add* na região Methods For Fetching And Updating Data, crie uma variável privada *Customer* chamada *oldCustomer* e defina outro método chamado *Edit*:

```
// Edita o cliente atual
// - salva os detalhes existentes do cliente
// e coloca o formulário no modo de edição (Editing)
private Customer oldCustomer;

private void Edit ()
{
    this.oldCustomer = new Customer();
    this.CopyCustomer(this.Current, this.oldCustomer);
    this.IsEditing = true;
}
```

Esse método copia os detalhes do cliente atual para a variável *oldCustomer* e coloca o *ViewModel* no modo Editing. Nesse modo, o usuário pode alterar os detalhes do cliente atual. Se, subsequentemente, o usuário optar por descartar as alterações, os dados originais poderão ser copiados de volta a partir da variável *oldCustomer*.

- 7.** Adicione a seguinte variável *Command*, mostrada em negrito, ao início da classe *ViewModel*:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command AddCustomer { get; private set; }
public Command EditCustomer { get; private set; }
    ...
}
```

8. No construtor de *ViewModel*, instancie o comando *EditCustomer*, como mostrado em negrito no código a seguir:

```
public ViewModel()
{
    ...
    this.AddCustomer = new Command(this.Add, ...);
    this.EditCustomer = new Command(this.Edit,
        O => { return this.CanBrowse; });
}
```

Esse código é semelhante à instrução do comando *AddCustomer*, exceto que referencia o método *Edit*.

9. Após o método *Edit* na região Methods For Fetching And Updating Data, adicione o seguinte método chamado *Discard* à classe *ViewModel*:

```
// Descarta as alterações feitas no modo Adding ou Editing
// e retorna o formulário para o modo Browsing
private void Discard ()
{
    // Se o usuário estava adicionando um novo cliente, remove-o
    if (this.IsAdding)
    {
        this.customers.Remove(this.Current);
        this.OnPropertyChanged("Current");
    }

    // Se o usuário estava editando um cliente existente,
    // restaura os detalhes salvos
    if (this.IsEditing)
    {
        this.CopyCustomer(this.oldCustomer, this.Current);
    }

    this.IsBrowsing = true;
}
```

O objetivo desse método é tornar possível ao usuário descartar qualquer alteração feita quando o *ViewModel* está no modo Adding ou Editing. Se o *ViewModel* está no modo Adding, o cliente atual é removido da lista (esse é o novo cliente criado pelo método *Add*) e o evento *PropertyChanged* é lançado para indicar que o cliente atual na lista de clientes mudou. Se o *ViewModel* está no modo Editing, os detalhes originais da variável *oldCustomer* são copiados de volta para o cliente atualmente exibido. Por fim, o *ViewModel* volta para o modo Browsing.

10. Adicione a variável *Command DiscardChanges* à lista no início da classe *ViewModel* e atualize o construtor para instanciar esse comando, como mostrado aqui em negrito:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command EditCustomer { get; private set; }
    public Command DiscardChanges { get; private set; }
    ...
    public ViewModel()
    {
        ...
        this.EditCustomer = new Command(this.Edit, ...);
        this.DiscardChanges = new Command(this.Discard,
            () => { return this.CanSaveOrDiscardChanges; });
    }
    ...
}
```

Observe que o comando *DiscardChanges* só é habilitado se a propriedade *CanSaveOrDiscardChanges* for *true*; o ViewModel tem uma conexão com o web service e está no modo *Adding* ou *Editing*.

11. Na região Methods For Fetching And Updating Data, após o método *Discard*, adicione mais um método, chamado *SaveAsync*, como mostrado no código a seguir. Esse método deve ser marcado como *async*.

```
// Salva o cliente novo ou atualizado de volta no web service
// e retorna o formulário para o modo Browsing
private async void SaveAsync()
{
    // Valida os detalhes do cliente
    if (this.ValidateCustomer(this.Current))
    {
        // Só continua se os detalhes do cliente são válidos
        this.IsBusy = true;
        try
        {
            // Se o usuário está adicionando um novo cliente,
            // envia ao web service uma requisição HTTP POST com os detalhes
            if (this.IsAdding)
            {
                var response =
                    await client.PostAsJsonAsync("api/customers", this.Current);
                if (response.IsSuccessStatusCode)
                {
                    // TODO: Exibir os detalhes do novo cliente
                }
                // TODO: Tratar da falha de POST
            }
            // O usuário deve estar editando um cliente já existente;
            // portanto, envia os detalhes utilizando uma requisição PUT
        else
```

```
{  
    string path = string.Format("api/customers/{0}",  
        this.Current.CustomerID);  
    var response = await client.PutAsJsonAsync(path, this.Current);  
    if (response.IsSuccessStatusCode)  
    {  
        this.IsEditing = false;  
        this.IsBrowsing = true;  
    }  
    // TODO: Tratar falha de PUT  
}  
}  
catch (Exception e)  
{  
    // TODO: Tratar exceções  
}  
finally  
{  
    this.IsBusy = false;  
}  
}  
}
```

Esse método ainda não está completo. O código inicial que você acabou de digitar verifica se os detalhes do cliente são válidos. Se forem, então podem ser salvos, e a propriedade *IsBusy* do ViewModel é definida com *true* para indicar que isso pode levar algum tempo, enquanto as informações são enviadas pela rede para o web service (lembre-se de que a propriedade *IsActive* do controle *ProgressRing* do formulário *Customers* está vinculada a essa propriedade, e o anel de progresso será exibido enquanto os dados estiverem sendo salvos).

O código do bloco *try* determina se o usuário está adicionando um novo cliente ou editando os detalhes de um cliente já existente. Se o usuário está adicionando um novo cliente, o código utiliza o método *PostAsJsonAsync* do objeto *HttpClient* para enviar uma mensagem POST para o web service. Lembre-se de que a requisição POST é enviada para o método *PostCustomer* da classe *CustomersController* do web service, e esse método espera um objeto *Customer* como seu parâmetro. Os detalhes são transmitidos no formato JSON.

Se o usuário está editando um cliente já existente, o aplicativo chama o método *PutAsJsonAsync* do objeto *HttpClient*. Esse método gera uma requisição PUT, a qual é passada para o método *PutCustomer* da classe *CustomersController* do web service. O método *PutCustomer* atualiza os detalhes do cliente no banco de dados e espera como parâmetros a ID e os detalhes do cliente. Novamente, esses dados são transmitidos para o web service no formato JSON.

Uma vez enviados os dados, a propriedade *isBusy* é definida como *false*, o que faz o controle *ProgressRing* desaparecer.

12. No método `SaveAsync`, substitua o comentário `// TODO: Displays the details of the new customer` pelo código mostrado em negrito a seguir:

```
if (response.IsSuccessStatusCode)
{
    // Obter a ID do cliente recentemente criado e exibi-la
    Uri customerUri = response.Headers.Location;
    var newCust = await this.client.GetAsync(customerUri);
    if (newCust.IsSuccessStatusCode)
    {
        var customerData = await newCust.Content.ReadAsAsync<Customer>();
        this.CopyCustomer(customerData, this.Current);
        this.OnPropertyChanged("Current");
        this.IsAdding = false;
        this.IsBrowsing = true;
    }
    else
    {
        // TODO: Tratar falha de GET
    }
}
```

A coluna `CustomerID` da tabela `Customer` no banco de dados `AdventureWorks` contém valores gerados automaticamente. O usuário não fornece um valor para esses dados ao criar um cliente; em vez disso, o próprio banco de dados gera o valor, quando um cliente é adicionado a ele. Assim, o banco de dados pode garantir que cada cliente tenha uma ID exclusiva. Portanto, após ter enviado a requisição POST para o web service, você deve enviar uma requisição GET a fim de obter a ID do cliente. Felizmente, o objeto `HttpResponseMessage`, passado de volta pelo web service como resultado da requisição POST, contém um URL que pode ser usado por um aplicativo para consultar os novos dados. Esse URL está disponível na propriedade `Headers.Location` da resposta e terá a forma `api/Customers/n`, onde `n` é a ID do cliente. O código que você acabou de adicionar envia um pedido GET para esse URL usando o método `GetAsync` do objeto `HttpClient`, e lê os dados do novo cliente de volta, usando o método `ReadAsAsync` da resposta. Então, o código atualiza com esses dados os detalhes do cliente armazenados na coleção de clientes.



Nota Pode parecer que esse código está fazendo uma viagem de ida e volta desnecessária para o web service, a fim de buscar a ID do cliente, a qual está disponível na propriedade `Headers.Location` da mensagem de resposta do pedido POST. Contudo, esse passo verifica se os dados foram salvos corretamente, e pode haver outros campos que são transformados pelo web service quando os dados são salvos; portanto, esse processo garante que o aplicativo exiba os dados conforme aparecem no banco de dados.

13. No Solution Explorer, na pasta Controllers, abra o arquivo `CustomersController.cs` e exiba-o na janela Code and Text Editor.

14. No método *PostCustomer*, antes das instruções que salvam o novo cliente no banco de dados, adicione o seguinte código mostrado em negrito.

```
// POST api/Customers
public HttpResponseMessage PostCustomer(Customer customer)
{
    if (ModelState.IsValid)
    {
        customer.ModifiedDate = DateTime.Now;
        customer.rowguid = Guid.NewGuid();
        db.Customers.Add(customer);
        db.SaveChanges();
        ...
    }
}
```

A tabela Customer no banco de dados *AdventureWorks* tem alguns requisitos adicionais; especificamente, se você estiver adicionando ou editando um cliente, deve definir a propriedade *ModifiedDate* do cliente de modo a refletir a data em que a alteração foi feita. Além disso, se estiver criando um novo cliente, deve preencher a propriedade *rowguid* do objeto *Customer* com um novo GUID, antes de poder salvá-lo (essa é uma coluna obrigatória da tabela Customer; outros aplicativos dentro da organização Adventure Works utilizam essa coluna para monitorar informações sobre os clientes).



Nota GUID significa Globally Unique Identifier (identificador globalmente exclusivo). Um GUID é uma string gerada pelo Windows que, quase com certeza, é única (existe uma possibilidade muito pequena de que o Windows possa gerar um GUID não exclusivo, mas a possibilidade é tão pequena que pode ser desconsiderada). Os GUIDs são frequentemente utilizados pelos bancos de dados como valores de chave para identificar linhas individuais, como no caso da tabela Customer no banco de dados *AdventureWorks*.

15. No método *PutCustomer*, atualize a propriedade *ModifiedDate* do cliente, antes da instrução que indica que o cliente foi modificado, como mostrado aqui em negrito:

```
// PUT api/Customers/{id}
public HttpResponseMessage PutCustomer(Int32 id, Customer customer)
{
    ...
    customer.ModifiedDate = DateTime.Now;
    db.Entry(customer).State = EntityState.Modified;
    ...
}
```

16. Retorne à classe *ViewModel* na janela Code and Text Editor.
17. Adicione a variável *Command SaveChanges* à lista no início da classe *ViewModel* e atualize o construtor para instanciar esse comando, como mostrado a seguir:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command DiscardChanges { get; private set; }
    public Command SaveChanges { get; private set; }
    ...
    public ViewModel()
    {
        ...
        this.DiscardChanges = new Command(this.Discard, ...);
        this.SaveChanges = new Command(this.SaveAsync, () =>
            { return this.CanSaveOrDiscardChanges; });
    }
    ...
}
```

18. No menu Build, clique em Build Solution e verifique se seu aplicativo compila sem erros.

Relate erros e atualize a interface do usuário

Você adicionou os comandos por meio dos quais o usuário pode recuperar, adicionar, editar e salvar informações de clientes. Mas, se algo der errado e ocorrer um erro, o usuário não vai saber o que aconteceu. Isso porque a classe *ViewModel* não inclui um recurso de relato de erros. Uma maneira de adicionar esse recurso é capturar as mensagens de exceção que ocorrem e expô-las como uma propriedade da classe *ViewModel*. Uma visualização pode usar vinculação de dados para conectar essa propriedade e exibir as mensagens de erro.

Adicione relato de erros à classe *ViewModel*

1. No arquivo *ViewModel.cs*, expanda a região chamada Properties For "Busy" And Error Message Handling (Propriedades para tratamento de "ocupado" e mensagem de erro).
2. Após a propriedade *IsBusy*, adicione a variável de string privada *_lastError* e a propriedade de string pública *LastErrorMessage* mostradas aqui:

```
private string _lastError = null;
public string LastErrorMessage
{
    get { return this._lastError; }
    private set
    {
        this._lastError = value;
        this.OnPropertyChanged("LastErrorMessage");
    }
}
```

3. Na região Methods For Fetching And Updating Data, localize o método *GetDataTableAsync*. Esse método contém a seguinte rotina de tratamento de exceções:

```
catch (Exception e)
{
    // TODO: Tratar exceções
}
```

4. Após o comentário *// TODO: Tratar erros*, adicione o seguinte código mostrado em negrito:

```
catch (Exception e)
{
    // TODO: Tratar exceções
    this.LastError = e.Message;
}
```

5. No bloco *else*, imediatamente antes da rotina de tratamento de exceções, após o comentário *// TODO: Tratar falha de GET*, adicione o seguinte código mostrado em negrito:

```
else
{
    // TODO: Tratar falha de GET
    this.LastError = response.ReasonPhrase;
}
```

A propriedade *ReasonPhrase* do objeto *HttpResponseMessage* contém uma string indicando o motivo da falha informada pelo web service.

6. No final do bloco *if*, imediatamente antes do bloco *else*, adicione a seguinte instrução mostrada em negrito:

```
if
{
    ...
    this.IsAtEnd = (this.customers.Count == 0);
    this.LastError = String.Empty;
}
else
{
    this.LastError = response.ReasonPhrase;
}
```

Essa instrução remove qualquer mensagem de erro da propriedade *LastError*.

7. Localize o método *ValidateCustomer* e adicione a seguinte instrução mostrada em negrito, imediatamente antes da instrução *return*:

```
private bool ValidateCustomer(Customer customer)
{
    ...
    this.LastError = validationErrors;
    return !hasErrors;
}
```

O método *ValidateCustomer* preenche a variável *validationErrors* com informações sobre qualquer propriedade no objeto *Customer* que contenha dados inválidos. A instrução que você acabou de adicionar copia essas informações para a propriedade *LastError*.

8. Localize o método *SaveAsync*. Nesse método, adicione o seguinte código mostrado em negrito, para capturar qualquer erro e falha HTTP do web service:

```
private async void SaveAsync()
{
    // Valida os detalhes do cliente
    if (this.ValidateCustomer(this.Current))
    {
        ...
        try
        {
            // Se o usuário está adicionando um novo cliente,
            // envia uma requisição HTTP POST para o web service com os detalhes
            if (this.IsAdding)
            {
                ...
                if (response.IsSuccessStatusCode)
                {
                    ...
                    if (newCust.IsSuccessStatusCode)
                    {
                        ...
                        this.IsBrowsing = true;
                        this.LastError = String.Empty;
                    }
                    else
                    {
                        // TODO: Tratar falha de GET
                        this.LastError = response.ReasonPhrase;
                    }
                }
                // TODO: Tratar falha de POST
                else
                {
                    this.LastError = response.ReasonPhrase;
                }
            }
            // O usuário deve estar editando um cliente já existente,
            // para enviar os detalhes usando uma requisição PUT
            else
            {
                ...
                if (response.IsSuccessStatusCode)
                {
                    this.IsEditing = false;
                    this.IsBrowsing = true;
                    this.LastError = String.Empty;
                }
                // TODO: Tratar falha de PUT
                else
                {

```

```
        this.LastError = response.ReasonPhrase;
    }
}
catch (Exception e)
{
    // TODO: Tratar exceções
    this.LastError = e.Message;
}
finally
{
    this.IsBusy = false;
}
}
```

9. Localize o método *Discard* e, ao final desse método, adicione a instrução mostrada aqui em negrito:

```
private void Discard()
{
    ...
    this.LastError = String.Empty;
}
```

- 10.** No menu Build, clique em Build Solution e verifique se o aplicativo compila sem erros.

Agora o ViewModel está completo. O estágio final é incorporar os novos comandos, informações de estado e recursos de relato de erros na visualização fornecida pelo formulário Customers.

Integre funcionalidades de adição e edição no formulário Customers

1. Abra o arquivo MainPage.xaml na janela Design View.

A marcação XAML do formulário *MainPage* já foi modificada, e os controles *TextBlock* a seguir foram adicionados aos controles *Grid* que exibem os dados:

```
<Page
  x:Class="Customers.MainPage"
  ...>

<Grid Style="{StaticResource GridStyle}">
  ...
  <Grid x:Name="customersTabularView" ...>
    ...
    <Grid Grid.Row="2">
      ...
      <TextBlock Grid.Row="3" Grid.RowSpan="4"
Grid.Column="7" Style="{StaticResource ErrorMessageStyle}"/>
      </Grid>
    </Grid>
    <Grid x:Name="customersColumnarView" Margin="20,10,20,110" ...>
      ...
      <Grid Grid.Row="1">
        ...
        <TextBlock Grid.Row="6" Grid.Column="0"
Grid.ColumnSpan="2" Style="{StaticResource ErrorMessageStyle}"/>
        </Grid>
```

```
</Grid>
...
</Grid>
...
</Page>
```

O *ErrorMessageStyle* referenciado por esses controles *TextBlock* está definido no arquivo AppStyles.xaml.

2. Defina a propriedade *Text* de ambos os controles *TextBlock* de modo a se vincularem à propriedade *LastError* do ViewModel, como mostrado aqui em negrito:

```
...
<TextBlock Grid.Row="3" Grid.RowSpan="4" Grid.Column="7"
Style="{StaticResource ErrorMessageStyle}" Text="{Binding LastError}"/>
...
<TextBlock Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
Style="{StaticResource ErrorMessageStyle}" Text="{Binding LastError}"/>
```

3. Os controles *TextBox* e *ComboBox* do formulário que exibem dados dos clientes só devem permitir que o usuário modifique esses dados se o ViewModel estiver no modo Adding ou Editing; caso contrário, devem ser desabilitados. Adicione a propriedade *IsEnabled* em cada um desses controles e vincule-a à propriedade *IsAddingOrEditing* do ViewModel, como segue:

```
...
<TextBox Grid.Row="1" Grid.Column="1" x:Name="id"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
<ComboBox Grid.Row="1" Grid.Column="3" x:Name="title"
.IsEnabled="{Binding IsAddingOrEditing}" ...>
</ComboBox>
<TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
<TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="3" Grid.Column="3" ... x:Name="email"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="5" Grid.Column="3" ... x:Name="phone"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
...
<TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" />
.IsEnabled="{Binding IsAddingOrEditing}" .../>
<ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle"
.IsEnabled="{Binding IsAddingOrEditing}" ...>
</ComboBox>
<TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
<TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
```

4. Adicione uma barra de aplicativo na parte inferior da página, imediatamente após a barra de aplicativo superior, utilizando o elemento `<Page.BottomAppBar>`. Essa barra de aplicativo deve conter botões para os comandos `AddCustomer`, `EditCustomer`, `SaveChanges` e `DiscardChanges`, como segue:

```
<Page ...>
  ...
  <Page.TopAppBar >
    ...
  </Page.TopAppBar>
  <Page.BottomAppBar>
    <AppBar IsSticky="True">
      <Grid>
        <StackPanel Orientation="Horizontal"
HorizontalContentAlignment="Right">
          <AppBarButton x:Name="addCustomer"
Icon="Add" Command="{Binding Path=AddCustomer}"/>
          <AppBarButton x:Name="editCustomer"
Icon="Edit" Command="{Binding Path>EditCustomer}"/>
          <AppBarButton x:Name="saveChanges"
Icon="Save" Command="{Binding Path=SaveChanges}"/>
          <AppBarButton x:Name="discardChanges"
Icon="Undo" Command="{Binding Path=DiscardChanges}"/>
        </StackPanel>
      </Grid>
    </AppBar>
  </Page.BottomAppBar>
</Page>
```

Observe que a convenção para comandos na barra de aplicativo inferior é agrupá-los, começando no lado direito. Os ícones referenciados pelos botões são as imagens padrão fornecidas com o template Windows Store App.

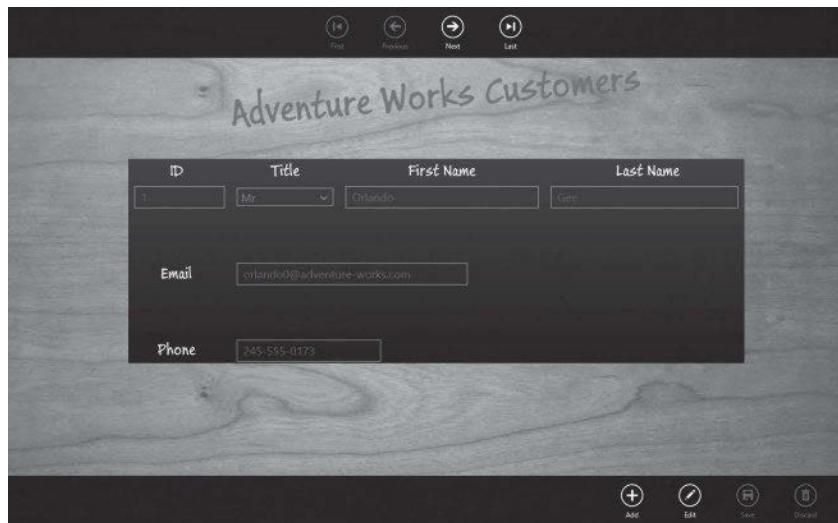
Teste o aplicativo Customers

1. No menu Debug, clique em Start Debugging para compilar e executar o aplicativo.

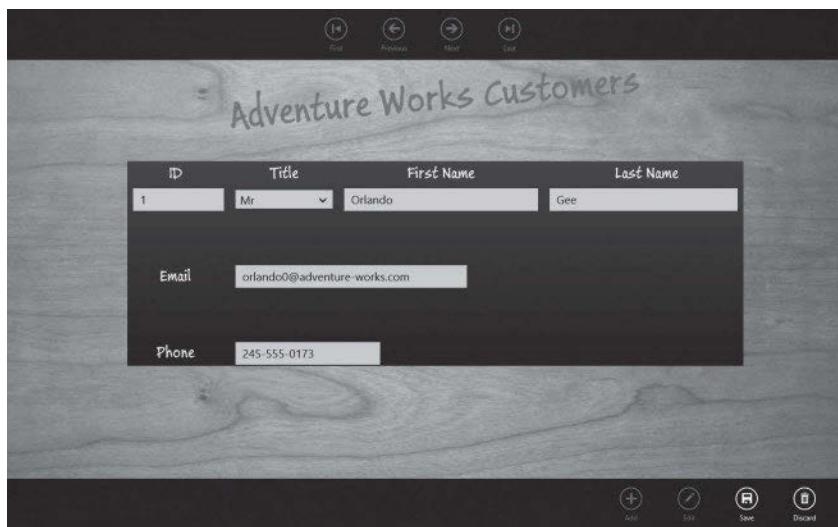
Quando o formulário *Customers* aparecer, observe que os controles *TextBox* e *ComboBox* estarão desabilitados. Isso porque a visualização está no modo *Browsing*.

2. Clique com o botão direito do mouse no formulário e verifique que as barras de aplicativo superior e inferior aparecem.

Você pode usar os botões First, Next, Previous e Last da barra de aplicativo superior como antes (lembre-se de que os botões First e Previous não serão habilitados até que você saia do primeiro cliente). Na barra de aplicativo inferior, os botões Add e Edit deverão estar habilitados, mas os botões Save e Discard devem estar desabilitados. Isso porque os comandos *AddCustomer* e *EditCustomer* são habilitados quando o ViewModel está no modo Browsing, e os comandos *SaveChanges* e *DiscardChanges* só são habilitados quando o ViewModel está no modo Adding ou Editing.



3. Na barra de aplicativo inferior, clique em Edit.
4. Os botões da barra de aplicativo superior se tornam desabilitados, pois agora o ViewModel está no modo Editing. Além disso, os botões Add e Edit também estão desabilitados, mas os botões Save e Discard devem estar habilitados. Além disso, agora os campos de entrada de dados do formulário estão habilitados, e o usuário pode modificar os detalhes do cliente.



5. Modifique os detalhes do cliente: apague o nome, digite **Test** para o endereço de e-mail, digite **Test 2** para o número de telefone e clique em Save.

Essas alterações violam as regras de validação implementadas pelo método *ValidateCustomer*. O método *ValidateCustomer* preenche a propriedade *LastError* do ViewModel com mensagens de validação, e elas são exibidas no formulário, no *TextBlock* vinculado à propriedade *LastError*:



6. Clique em Discard e verifique se os dados originais são restabelecidos no formulário, as mensagens de validação desaparecem e o ViewModel volta para o modo Browsing.
7. Clique em Add. Os campos do formulário devem ser esvaziados (fora o campo ID, que exibe o valor 0). Insira os detalhes de um novo cliente. Certifique-se de fornecer nome e sobrenome, um endereço de e-mail válido, da forma *nome@empresa.com*, e um número de telefone (você também pode incluir parênteses, hífens e espaços).
8. Clique em Save. Se os dados forem válidos (não houver erros de validação), eles deverão ser salvos no banco de dados. Você verá a ID gerada para o novo cliente no campo ID, e o ViewModel deverá voltar para o modo Browsing.
9. Experimente o aplicativo e tente adicionar e editar mais clientes. Observe que é possível redimensionar a visualização para exibir o layout em colunas e o formulário ainda deverá funcionar.
10. Quando tiver terminado, retorne ao Visual Studio e interrompa a depuração.

Resumo

Neste capítulo, você aprendeu a utilizar o Entity Framework para criar um modelo de entidades que pode utilizar para se conectar a um banco de dados SQL Server. Viu também como criar um web service REST, que um aplicativo Windows Store pode utilizar para consultar e atualizar dados no banco de dados por meio do modelo de entidades, e aprendeu a integrar código que chama o web service em um ViewModel.

Você finalizou todos os exercícios deste livro. Esperamos que já esteja se entendendo muito bem com a linguagem C# e saiba utilizar o Visual Studio 2013 para compilar aplicativos profissionais para Windows 7 e para Windows 8.1. Entretanto, a história ainda não acabou. Você saltou apenas o primeiro obstáculo, mas os melhores programadores para a linguagem C# aprendem com a experiência contínua, e você só poderá obter essa experiência ao construir aplicativos em C#. Ao fazer isso, descobrirá novas maneiras de utilizar a linguagem C# e os diversos recursos disponíveis no Visual Studio 2013 para os quais não tive espaço suficiente neste livro. Além disso, lembre-se de que a linguagem C# é uma linguagem em evolução. Em 2001, quando escrevi a primeira edição deste livro, a linguagem C# lançava a sintaxe e a semântica necessárias para compilar aplicativos que utilizavam o Microsoft .NET Framework 1.0. Em 2003, foram implementados alguns aprimoramentos no Visual Studio e no .NET Framework 1.1 e, mais tarde, em 2005, surgiu o C# 2.0 com suporte para genéricos e para o .NET Framework 2.0. O C# 3.0 acrescentou diversos recursos, como os tipos anônimos, as expressões lambda e, o mais importante, a LINQ. O C# 4.0 estendeu a linguagem ainda mais, com o suporte para argumentos nomeados, parâmetros opcionais, interfaces contravariantes e covariantes, e a integração com linguagens dinâmicas. O C# 5.0 adicionou suporte completo para processamento assíncrono por meio da palavra-chave *async* e do operador *await*.

Paralelamente à evolução da linguagem de programação C#, o sistema operacional Windows mudou consideravelmente desde a primeira edição deste livro. É indiscutível que as alterações fomentadas pelo Windows 8 e ampliadas no Windows 8.1 foram as mais radicais nesse período, e os desenvolvedores familiarizados com as edições anteriores do Windows têm agora novos e excitantes desafios para compi-

lar aplicativos para a moderna plataforma móvel, baseada em toques, fornecida pelo Windows 8.1. Sem dúvida, o Visual Studio 2013 e a linguagem C# contribuirão para ajudá-lo a enfrentar esses desafios.

O que a próxima versão da linguagem C# e do Visual Studio nos oferecerão? Fique antenado!

Referência rápida

Para	Faça isto
Criar um modelo de entidades utilizando a Entity Framework	Adicione um novo item ao seu projeto, utilizando o template ADO.NET Entity Data Model. Use o Entity Data Model para conectar-se ao banco de dados que contém as tabelas que você quer modificar e selecione as tabelas exigidas por seu aplicativo.
Criar um web service REST que forneça acesso remoto ao banco de dados por meio de um modelo de entidades	Crie um projeto ASP.NET usando o template Web API. Execute o assistente Add Scaffold e selecione Web API 5 Controller with read/write actions, using Entity Framework. Especifique o nome da classe de entidade apropriada no modelo de entidades como a classe <i>Model</i> , e a classe de contexto de dados do modelo de entidades como classe de contexto <i>Data</i> .
Consumir um web service REST em um aplicativo Windows Store	Adicione o pacote ASP.NET Web Client Libraries ao projeto e use um objeto <i>HttpClient</i> para conectar o banco de dados. Defina a propriedade <i>BaseAddress</i> do objeto <i>HttpClient</i> de forma a referenciar o endereço do web service. Por exemplo:
	<pre>string ServerUrl = "http://localhost:50000/"; HttpClient client = new HttpClient(); client.BaseAddress = new Uri(ServerUrl);</pre>
Recuperar dados de um web service REST em um aplicativo Windows Store	Chame o método <i>GetAsync</i> do objeto <i>HttpClient</i> e especifique o URL do recurso a acessar. Se o método <i>GetAsync</i> for bem-sucedido, busque os dados utilizando o método <i>ReadAsStringAsync</i> do objeto <i>HttpResponseMessage</i> retornado pelo método <i>GetAsync</i> . Por exemplo:
	<pre>HttpClient client = ...; var response = await client.GetAsync("api/customers"); if (response.IsSuccessStatusCode) { var customerData = await response.Content. ReadAsStringAsync<IEnumerable<Customer>>() ... } else { // GET failou }</pre>

Para	Faça isto
Adicionar um novo item de dado em um web service REST a partir de um aplicativo Windows Store	<p>Use o método <i>PostAsJsonAsync</i> do objeto <i>HttpClient</i> e especifique como parâmetros o novo item a ser criado e o URL da coleção que vai armazenar esse item. Examine o status do objeto <i>HttpResponseMessage</i> retornado por <i>PostAsJsonAsync</i> para verificar se a operação POST foi bem-sucedida. Por exemplo:</p> <pre>HttpClient client = ...; Customer newCustomer =; var response = await client.PostAsJsonAsync("api/customers", newCustomer); if (!response.IsSuccessStatusCode) { // POST falhou }</pre>
Atualizar um item existente em um web service REST a partir de um aplicativo Windows Store	<p>Use o método <i>PutAsJsonAsync</i> do objeto <i>HttpClient</i> e especifique como parâmetros o item a ser atualizado e o URL desse item. Examine o status do objeto <i>HttpResponseMessage</i> retornado por <i>PostAsJsonAsync</i> para verificar se a operação PUT foi bem-sucedida. Por exemplo:</p> <pre>HttpClient client = ...; Customer updatedCustomer =; string path = string.Format("api/customers/{0}", updatedCustomer.CustomerID); var response = await client.PutAsJsonAsync(path, updatedCustomer); if (!response.IsSuccessStatusCode) { // PUT falhou }</pre>

Índice

Símbolos

- & (E comercial)
 - operador && (AND lógico), 95, 97, 111
 - operador address-of, 202
 - operador AND bit a bit, 361
- < > (sinais de menor e maior)
 - operador < (menor que), 95, 97, 111
 - operador < e operador >, aos pares, 510
 - operador << (deslocamento à esquerda), 360
 - operador <= (menor ou igual a), 95, 97, 111
 - operador > (maior que), 95, 97, 111
 - operador >= (maior ou igual a), 95, 97, 111
 - operador de >> deslocamento à direita, 360
 - operadores <= e >=, aos pares, 510
- * (asterisco)
 - em ponteiros, 203
 - operador *= (multiplicação e atribuição), 114
 - operador de multiplicação, 52, 58, 96
 - associatividade, 60
 - sobrecregando, 513
- \ (barra invertida), caractere de escape em C#, 110
- { } (chaves), 59
 - bloco de código incluído em, duração de variáveis definidas em, 196
 - em expressões lambda, 420
 - incluindo blocos de código, 98
 - uso na inicialização de elementos de array com valores específicos, 228
- ^ (acento circunflexo), operador XOR bit a bit, 361, 364
- , (vírgula), separando várias inicializações e atualizações em loops for, 122
- . (notação de ponto), 279, 314
- = (sinal de igual)
 - confundindo = e == em instrução if, 98
 - implementando versão sobrecregada, 514
 - aos pares com o operador !=, 509
 - operador =>, 420
 - operador == (igual a), 94, 97, 111
 - operador de atribuição, 60, 97, 113
 - usando propriedades para simular, 503
 - usando com cláusula on de expressão LINQ, 489
- ! (ponto de exclamação)
 - operador != (desigualdade), 94, 97, 111
 - aos pares com operador ==, 510
 - implementando versão sobrecregada, 514
 - operador NOT lógico, 96, 191
- (sinal de menos)
 - operador -- (decremento), 61, 96, 114
 - declarando sua própria versão, 508
 - operador -= (subtração e atribuição), 114, 133
 - cancelando a inscrição em um evento, 465
- removendo métodos de delegates, 449
- operador de negação, 96
- operador de subtração, 52, 54, 57, 97
 - sobrecregando, 512
- \n (caractere de nova linha), 119
- () (parênteses)
 - em chamadas de método, 70
 - incluindo parâmetros em expressões lambda, 420
 - usando para anular precedência de operador, 59, 96
- % (sinal de porcentagem)
 - operador %= (módulo e atribuição), 114
 - operador módulo, 53, 55, 58, 96, 126
- | (barra vertical)
 - operador || (OR lógico), 95, 97, 111
 - operador OR bit a bit, 360
- + (sinal de mais)
 - operador ++ (incremento), 61, 96, 114
 - declarando sua própria versão, 508
 - em classes versus estruturas, 509
 - operador += (adição e atribuição), 114, 127, 133
 - avaliação, entendendo, 507
 - usando com delegates, 448, 449
 - usando com eventos, 464
- operador de adição, 52, 57, 97, 113, 504–507
 - sobrecregando, 512
- operador de concatenação de strings, 52
- operador unário +, 96
- ? (ponto de interrogação)
 - indicando que enumeração é nullable, 207
 - indicando que estrutura é nullable, 216
 - indicando que tipo-valor é nullable, 190, 204
- " (aspas, duplas), 110
- ' (aspas, simples), 110
- ; (ponto e vírgula)
 - em expressões lambda, 420
 - finalizando instruções, 39
 - declarações de variável, 42
 - separando inicialização, expressão booleana e variável de controle de atualização em loops for, 122
 - substituindo corpo de método em declarações de método de interface, 285
- / (barra normal)
 - /* e */ circundando comentários de várias linhas, 11
 - ///, iniciando comentários em arquivos XAML, 33
 - operador /= (divisão e atribuição), 114, 126
 - operador de divisão, 52, 55, 58, 96
 - associatividade, 60
 - sobrecregando, 513
 - // precedendo comentários, 11

[] (colchetes), 59
 em indexadores, 362
 significando variáveis de array, 226
 tamanho de array em, 227
 usando indexadores para simular operador [], 503
 ~ (til)
 em sintaxe de destrutor, 315
 operador NOT, 360
`_`(sublinhado), iniciando nomes de campos privados, 165

A

ADO.NET, 711
 agregando dados, 484, 488
 algoritmo hill-climbing, 531
 ambiente de execução gerenciado, 223
 animações em aplicativos Windows Store, 612
 transições de estado visual, 648
 aparelhos Windows Phone, largura dos, 628
 APIs Win32, 223
 aplicativo de console Hello World, 8–14
 aplicativo gráfico, criando, 18–37
 adicionando código, 34–37
 C# e arquivos XAML criados pelo Visual Studio, 20
 No Windows 7 ou 8, 21
 aplicativos, 18. *Consulte também* aplicativos Windows Store
 aplicativos de console, 3
 criando usando Visual Studio 2013, 3, 38
 passos no processo, 5
 escrevendo o aplicativo Hello World, 8–14
 aplicativos não gerenciados, 223
 classes consumidas por, por meio de WinRT, 310
 aplicativos web, criando, 713
 aplicativos Windows Store, 18, 611–660
 acessando banco de dados remotos a partir de, 709–761
 inserção, atualização e exclusão de dados por meio de web service REST, 729–747
 recuperando dados de um banco de dados, 709–729
 construindo usando o template Blank App, 616–659
 aplicando estilos a uma interface do usuário, 650–659
 criando o aplicativo Adventure Works Customers (exercício), 616–618
 implementando interface de usuário escalonável, 618–650
 criando, 660
 criando para Windows 8.1 usando o Visual Studio 2013, 38
 definidos, 612–615
 examinando o código gerado pelo Visual Studio para, 30–33
 exibindo e buscando dados, 661–708
 contratos do Windows 8.1, 689–708
 implementando o padrão MVVM, 661–689
 fechando, 29
 no Windows 8 e 8.1, executando usando WinRT, 309
 templates para, 705
 usando Simulator para testar, 630–633
 aplicativos WPF
 criando para Windows 7 ou 8 usando o Visual Studio 2013, 38
 examinando arquivos de código gerados pelo Visual Studio, 33
 argumentos
 escrevendo método que modifica, usando palavras-chave ref e out, 192–195
 incapacidade de modificar argumento original por meio de alterações em parâmetro, 192
 passando argumentos nomeados, 85
 argumentos nomeados, 85
 resolvendo ambiguidades com, 86
 aritmética, efetuando em enumerações, 211
 aritmética de ponto flutuante, palavras-chave checked e unchecked e, 148
 arquivo App.config, 8
 arquivo App.xaml, 31
 examinando descrição XAML de aplicativo WPF, 34
 arquivo App.xaml.cs, examinando no Solution Explorer, 31–33
 arquivo AppStyles.xaml, 651
 referenciando em dicionário de recursos global, 652
 arquivo MainPage.xaml, 20
 examinando para o aplicativo Windows Store, 30
 arquivo MainWindow.xaml, 21
 arquivo MainWindow.xaml.cs, examinando o conteúdo do, 33
 arquivo Program.cs, 8
 arquivo Solution 'TestHello', 7
 arquivos .csproj, 44
 arquivos .sln, 44
 arquivos de projeto (sufixo .csproj), 44
 arquivos de solução (sufixo .sln), 44
 array implicitamente tipados, 229–230
 arrays, 226–248
 classe List <T> versus, 408
 comparação com coleções, 421–426
 comparação com indexadores, 364–366
 propriedades, arrays e indexadores, 365
 copiando, 233–235
 declarando e criando, 226–228
 criando instância de array, 227, 247
 declarando variáveis de array, 226, 247
 implementando interface IEnumerable, 428
 limitações dos, 406
 listando elementos com instrução foreach, 428
 na classe Dictionary < TKey, TValue >, 414
 parâmetro. *Consulte* arrays de parâmetro
 preenchendo e usando, 228–233, 247
 acessando elemento individual de array, 230
 criando array implicitamente tipado, 229
 inicializando elementos de array com valores específicos, 228
 iterando por um array, 230
 passando arrays como parâmetros de método e valores de retorno, 232

- usando arrays multidimensionais, 235–246, 248
 - criando um array irregular, 236, 248
 - usando arrays para implementar um jogo de cartas, 237–246
 - arrays associativos, 414
 - arrays de parâmetro, 249–260
 - comparando com parâmetros opcionais, 257–259
 - sobrecarga, recapitação da, 249
 - usando argumentos de array, 250–257
 - declarando um array params, 251
 - params object[], 253
 - pontos importantes sobre arrays params, 252
 - usando um array params (exercício), 254–257
 - arrays irregulares, 236, 248
 - arrays multidimensionais, 235–246
 - criando arrays irregulares, 236
 - palavra-chave params, incapacidade de usar com, 252
 - usando para implementar um jogo de cartas, 237–246
 - arrays retangulares, 236
 - árvores binárias, 383
 - construindo classe de árvore binária usando genéricos, 386–392
 - consultando dados em objetos Tree<TItem>, 489–495
 - criando enumerador para, 430–434
 - definindo enumerador para classe Tree<TItem>
 - usando um iterador, 439–441
 - definindo um método genérico para construir, 396–398
 - testando a classe Tree<TItem>, 392–395
 - ASP.NET Web API, modelo que suporta inserção, atualização e exclusão de dados, 729
 - ASP.NET Web API Client Libraries, instalando, 722
 - assemblies
 - namespaces e, 17
 - para namespaces System.Windows e Windows.UI, 309
 - assembly System.Windows.Controls, 34
 - assembly Windows.UI.Xaml.Controls, 34
 - assinando eventos, 464, 475
 - assinatura, método, 267
 - assistente Add Scaffold, 717
 - associatividade, 60, 502
 - operador de atribuição e, 60
 - resumo da associatividade dos operadores, 96
 - atribuição de valores a variáveis, 42, 43
 - atributo Grid.Row, 636
 - avaliação colocada em cache (LINQ), 496
 - avaliação postergada, LINQ e, 495–499
- B**
- banco de dados, remoto, acessando a partir de aplicativo Windows Store, 709–761
 - criando e usando um web service REST, 717–729
 - criando um modelo de entidades, 711–717
 - inserção, atualização e exclusão de dados por meio do web service REST, 729–747
 - implementando funcionalidade de adição e edição no ViewModel, 730–738
 - informando erros e atualizando a interface do usuário, 739–741
 - integrando funcionalidade de adição e edição ao formulário Customers, 742–744
 - testando o aplicativo Customers, 744–746
 - instalando o banco de dados AdventureWorks, 711
 - recuperando dados de um banco de dados, 709–729
 - bancos de dados relacionais
 - tabelas em, versus coleções na memória, 487
 - barra de aplicativo, 686–689
 - adicionando botões Next e Previous, 686–689
 - bibliotecas
 - referências a, na pasta References, 7
 - blocos de código
 - agrupando instruções if em, 98
 - duração de variáveis definidas em, 196
 - usando com instruções for, 122
 - usando com instruções while, 116
 - blocos de instrução try/catch/finally, 156
 - bloco de instrução try/catch, escrevendo, 141–144
 - bloco try/finally, método Finalize em, 316
 - blocos try, 135
 - chamando método de descarte em bloco finally de try/finally, 320
 - operador await e, 575
 - blocos finally, 155–156
 - bloqueando dados, 593
 - usando coleção concorrente e bloqueio para acesso seguro aos dados, 598–608
 - usando objeto ReaderWriterLockSlim, 610
 - botão Event Handlers for the Selected Element, 35
 - boxing e unboxing, 199–202, 205
 - desempenho e, 201, 379
 - busca
 - habilitando aplicativo Windows Store para suportar busca, 707
 - navegando até item selecionado, 701–704

C

C#

 - arquivo-fonte, Program.cs, 8
 - código para página em aplicativo Windows Store, exibindo, 31
 - como uma linguagem em evolução, 747
 - diferenciação de letras maiúsculas e minúsculas, 8
 - espaço em branco em, 40
 - palavras-chave, 40

C/C++

 - comportamento de cópia de variáveis de estrutura em C++, 221
 - declarações de array, 227
 - estruturas em C++, sem funções membro, 224
 - expressões de inteiro em instruções if, 98
 - fall-through de switch, diferenças no C#, 108
 - funções ou sub-rotinas, semelhança de métodos com, 65

- herança em C++, 263
- métodos globais, 66
- operação new em C++, 314
- operador delete em C++, inexistente em C#, 314
- operador resto, 53
- palavras-chave public e private em C++, 164
- parâmetro this oculto para operadores em C++, 504
- params como equivalente type-safe das macros varargs, 252
- ponteiros de função em C++, semelhança com delegates, 445
- ponteiros e código inseguro, 202
- sobrecargas de operador em C++, 506
- variáveis locais não atribuídas, 43
- variáveis não atribuídas, 94
- caixa
 - diferenciação de letras maiúsculas e minúsculas em C#, 8
 - em nomes de variável, 42
- caixa de diálogo Add New Project, templates exibidos em, 435
- caixa de diálogo Connect To Server, 711
- caixa de diálogo Connection Properties, 714
- caixa de diálogo New ASP.NET Project, 713
- caixa de diálogo New Project, 5
- caixa de diálogo Open, 117
- caixa de diálogo Open Project, 44
- caixa de diálogo Quick Find, 48
- campos, 163
 - convenções de atribuição de nomes, 165
 - estáticos, 175–182
 - criando um campo compartilhado, 176
 - estrutura, 213
- implementando encapsulamento usando métodos, 337–339
- implementando propriedades para acessar.
 - Consulte* propriedades
- inicialização, 164
- não definidos em interfaces, 290
- nomes de, aviso sobre nomes de propriedade e, 341
- privados ou públicos, 164
- readonly, 240
- campos readonly, 240
- cancelando a inscrição em eventos, 465, 476
- caractere de nova linha (\n), 119
- casting
 - casting de dados seguro, 201–203, 205
 - operador as, 202
 - operador is, 201
 - casting explícito e tipo de objeto, 379
 - convertendo entre objetos e strings, 399
 - usando em unboxing, 200
- ciclo de vida de aplicativos Windows Store, 614
- CIL (Common Intermediate Language), 223
- classe AggregateException, 565, 269
- classe ArgumentOutOfRangeException, 151
- classe Array, 230
 - método Clone, 235
 - método Copy, 234
 - método CopyTo, 234
- classe Barrier, 596
- classe base, 262
 - chamando construtores de classe base, 264
 - chamando implementação original de método em, 269
 - membros protegidos, acesso a, 272
 - métodos, 267
- classe BasicCollection<T>, 437–439
- classe Button, 466
- classe Circle, 162, 164
 - copiando tipos-referência e privacidade de dados, 185
- classe ConcurrentBag<T>, 597
- classe ConcurrentDictionary< TKey, TValue >, 597
- classe ConcurrentQueue<T>, 597
- classe ConcurrentStack<T>, 598
- classe Console, 9
- classe CountdownEvent, 595
- classe DbContext, 717
- classe Dictionary< TKey, TValue >, 407, 414–415
- classe DispatcherTimer, 681
- classe Enumerable, 480–486
 - Count, Max, Min e outros métodos de resumo, 484
 - método GroupBy, 484
 - método Join, 485
 - método OrderBy, 483
 - método OrderByDescending, 484
 - método Select, 481–482
 - definição do, 481
 - método ThenBy ou ThenByDescending, 484
 - método Where, 482
- classe EventArgs, 467
- classe Exception, 138
 - rotina de tratamento catch para todas as exceções, 157
 - rotina de tratamento genérica para capturar exceções Exception, 154
- classe FileInfo, 118
- classe FileOpenPicker, processamento assíncrono na, 582
- classe GC, 329
- classe HashSet<T>, 407, 417
 - SortedSet<T> e, 418
- classe HttpClient, 722
 - método GetAsync, 725
- classe HttpResponseMessage, 722, 725
- classe LinkedList<T>, 407, 410–412
 - classe List<T>, 407, 408–410
 - criando, manipulando e iterando pela, 409
 - determinando o número de elementos na, 410
 - métodos usando delegates, 445–447
 - recursos que evitam as limitações de arrays, 408
- classe ManualResetEventSlim, 594, 609
- classe Math, 162, 175

classe MessageBox, 36
 classe Parallel
 abstraindo tarefas usando, 546–550
 método Parallel.For, 546
 método Parallel.ForEach<T>, 546
 método Parallel.Invoke, 547
 parallelizando operações no aplicativo
 GraphData, 547–550
 cancelando loop Parallel.For ou ForEach, 559
 método Parallel.For, 592
 quando não usar, 550–552
 classe Program, 8
 classe Random, 228
 classe ReaderWriterLockSlim, 595, 610
 classe Regex, 732
 classe SearchResultsPage, 693
 método OnItemClick, 702
 classe SemaphoreSlim, 595, 609
 classe SortedDictionary< TKey, TValue >, 415
 classe SortedList< TKey, TValue >, 407, 415
 classe SortedSet< T >, 418
 classe Stack< T >, 406, 407, 413–414
 classe StorageFile, 582
 operações assíncronas, 582
 classe System.Array, 230, 428. *Consulte também* classe Array; arrays
 classe System.Int32, 211
 implementando IComparable e IComparable< T >, 394
 classe System.Int64, 211
 classe System.Object, 198, 264, 378
 método ToString, 268
 substituindo métodos Equals ou GetHashCode, 510
 classe System.Random, 228
 classe System.Single, 211
 classe System.String, 184
 implementação de IComparable e
 IComparable< T >, 394
 classe System.ValueType, 264
 substituindo métodos Equals ou GetHashCode, 510
 classe System.Windows.MessageBox, 36
 classe SystemException, 138
 classe Task, 530
 classe Task< TResult >, 580
 classe TextReader, 118
 método Close, 319
 classe Thread, 530
 classe ThreadPool, 530
 classe Util, 279
 classe ValueType, 264
 classes, 161
 abstratas, 301–303
 implementando e usando, 304–308
 anônimas, 180
 arrays de, 227
 atribuindo em hierarquia de herança, 265–267
 consumidas por aplicativos não gerenciados por
 meio de WinRT, 310
 controlando a acessibilidade, 164–175
 atribuição de nomes e acessibilidade, 165
 classes parciais, 168
 construtores, 165–166
 sobrecregendo construtores, 167
 declarando, 182
 definindo e usando, 162–163
 definindo em namespaces, 15
 definindo o escopo, 73
 encapsulamento, 162
 entendendo a classificação, 161
 escrevendo construtores e criando objetos, 169–173
 estáticas, 177
 estruturas *versus*, 214
 comparando comportamento de cópia de classe
 e estrutura, 221–223
 herança. *Consulte* herança
 ícone IntelliSense para, 11
 implementando propriedade de interface em, 357
 métodos de instância, 173–175
 objetos *versus*, 163
 operadores em, 509
 palavras-chave usadas em definição de métodos,
 310
 referenciando uma classe por meio de sua
 interface, 287
 seladas, 303–310
 tipo-valor, 183
 classes abstratas, 301–303, 312
 implementando e usando (exercício), 304–308
 métodos abstratos, 303
 classes anônimas, 180
 classes controladoras, web service REST
 criando adicionais, 720
 requisições web recebidas, manipuladas por, 718
 classes derivadas, 262
 criando a partir de classe base, 282
 membros protegidos, 272
 métodos mascarando métodos de classe base, 267
 classes parciais, 168
 classes seladas, 303–310, 312
 cláusula de expressões LINQ, 489
 CLR (Common Language Runtime), 83, 223
 gerenciando threads para implementar tarefas
 simultâneas, 531
 WinRT e, 309
 código gerenciado, 223
 código nativo, 223
 coleções, 406–427
 adicionando ou removendo itens, 426–427
 classes de coleção, 406–418
 Dictionary< TKey, TValue >, 414–415
 HashSet< T >, 417
 LinkedList< T >, 410–412
 List< T >, 408–410
 mais usadas, 407
 não genéricas, em System.Collections, 407
 Queue< T >, 412–413
 SortedList< TKey, TValue >, 415
 Stack< T >, 413

- classes de coleção concorrentes, 597–598
- comparação com arrays, 421–426
 - usando classes de coleção para jogar cartas, 421–426
- criando, 426–427
- enumerando, 428–442, 479
 - elementos em uma coleção, 428–436
 - implementando enumerador usando um iterador, 437–441
- iterando por, 427
- localizando número de elementos em, 426–427
- localizando um item em, 426–427
- melhorando o desempenho ao iterar por, usando PLINQ, 585–590
- métodos *Find*, predicados e expressões lambda, 419–421
- na memória, *versus* tabelas em banco de dados relacional, 487
- usando inicializadores de coleção, 418
- coleções baseadas em dicionário
 - adicionando ou removendo um item, 426–427
 - classe *ConcurrentDictionary<TKey, TValue>*, 597
 - localizando e acessando um valor, 419
 - localizando um item em, 426–427
- coleções concorrentes
 - classes, 597
 - usando com bloqueio para acesso seguro aos dados, 598–608
- coleções enumeráveis, 428, 479
 - implementando a interface *IEnumerable*, 434–436
 - projetando campos especificados de, 500
 - tornando enumerável uma classe de coleção, 442
- coleta de lixo, 314–319
 - chamando o coletor de lixo, 317
 - classe GC fornecendo acesso à, 329
 - como o coletor de lixo funciona, 318
 - escrevendo destrutores, 314–316
 - forçando, 333
 - impedindo de chamar destrutor em um objeto, 324
 - motivos para usar o coletor de lixo, 316–318
 - permitindo ao CLR gerenciar, 329
- comandos, 707
 - adicionando a um ViewModel, 678
 - implementando a classe *Command*, 679–682
 - vínculo de botões com comandos no ViewModel, 687
- comentários
 - /** e **/* em torno de comentários de várias linhas, 11
 - ///*, iniciando comentários em arquivos XAML, 33
 - //* (barras normais) precedendo, 11
 - comentários TODO, uso pelos desenvolvedores, 169
 - comentários TODO, 169, 186
- Common Intermediate Language (CIL), 223
- Common Language Runtime. Consulte CLR
- Common WPF Controls, 23
- Common XAML Controls, 23
- Component Object Model (COM), 83
 - operadores compostos, controlar usando as palavras-chave *checked* e *unchecked*, 149
- concorrência
 - questões a considerar na implementação, 530
 - sincronizando o acesso simultâneo aos dados, 591–608
- configuração *Minimum Width*, 628
- configurações do ambiente de desenvolvimento, 4
- conjunto de caracteres, 126
- conjuntos de entidades (entity set), 717
- construtor padrão, 166
 - escrevendo seu próprio construtor e, 167
- construtores, 165
 - chamando construtor de classe base a partir de construtor de classe derivada, 264, 282
 - declarando e chamando, 182
 - escrevendo, 170–173
 - inicializadores de objeto e, 354
 - não definidos em interfaces, 291
 - padrão, 166
 - para classes estáticas, 177
 - para estruturas, 214, 216
 - públicos e privados, 166
 - sobrecrevendo, 167
- consultando dados na memória, 477–501
- LINQ (Language-Integrated Query), 477
 - usando LINQ em um aplicativo C#, 478–499
 - filtrando dados, 482
 - junção de dados com *join*, 485
 - LINQ e avaliação postergada, 495–499
 - operadores de consulta, 487–489
 - ordenando, agrupando e agregando dados, 483
 - recuperando dados de *BinaryTree* usando métodos de extensão, 489–493
 - recuperando dados de *BinaryTree* usando operadores de consulta, 494
 - selecionando dados, 480–482
- continuações, 533
 - especificadas pelo operador *await*, 575, 576
 - usando com tarefas canceladas e falhas, 566
- contrato File Open Picker, 689
- contrato Search, 690
 - implementando, 690–701, 707
 - registrando aplicativo *Customers* na Pesquisa do Windows, 698–700
 - testando o contrato Search, 699
- contrato Share Target, 689
- contratos, 614, 689–704
 - implementando o contrato Search, 690–701
- controle AppBar, 687
- controle Grid
 - implementando layout tabular usando, 633–643
 - modificando o layout para mudar de escala para diferentes tamanhos e orientações, 635–643
 - para aplicativo Windows Store, 619
- controle TextBlocks
 - adicionando a formulários usando a janela Design View, 23
 - adicionando à página em aplicativo Windows Store, 619

- configurando propriedades, 620
 - rótulos para TextBoxes, 625
 - rótulos usados para identificar dados na página, 621
 - controles**
 - aplicando estilos a, em aplicativo Windows Store, 650, 660
 - arrastando da Toolbox para formulários, 621
 - hierarquia de, em formulários complexos, 55
 - propriedades anexadas, 636
 - vinculando comandos a, 681
 - controles AppBarButton, 687
 - controles Button
 - adicionando a formulários, 26
 - adicionando a visualizações em aplicativos Windows Store, 686–689
 - vinculando a comandos fornecidos pelo ViewModel, 707
 - controles ComboBox
 - adicionando a aplicativo Windows Store, 623–625
 - adicionando ComboBoxItem, 624
 - implementando vinculação de dados para controles ComboBox de título, 673
 - usando vinculação de dados com, 672
 - controles ComboBoxItem, 625
 - controles TextBox
 - adicionando a aplicativo Windows Store
 - exibindo texto ID First Name e Last Name, 622–624
 - para endereço de e-mail e número de telefone, 625
 - adicionando a formulários, 25
 - convenções de atribuição de nomes
 - interfaces, 286
 - para campos e métodos, 165
 - para classes, 165
 - propriedades e nomes de campo, aviso sobre, 341
 - conversões de alargamento, 518
 - conversões de estreitamento, 518
 - conversões explícitas, 518
 - conversões implícitas, 518
 - cópia profunda, 185, 235
 - cópia rasa, 185, 235
 - Count, Max, Min e outros métodos de resumo, 484
 - chamando Count, 488
 - curto-círcito, 96
- D**
- datas e hora
 - comparando datas em aplicativos reais, 105
 - comparando datas em método usando instrução if em cascata, 100–105
 - criando e usando estrutura para representar uma data, 217–220
 - declarações, variável, 42, 62
 - implicitamente tipada, 63
 - decrementando variáveis, 61
 - prefixo e sufixo, 61
 - delegates, 443–462
 - cancelando a inscrição a eventos, 465
 - chamando, 475
 - declarando e usando (exercício), 451–461
 - criando o componente CheckoutController, 456–460
 - examinando o aplicativo Wide World Importers, 451–456
 - testando o aplicativo, 460
 - declarando um tipo delegate, 475
 - entendendo, 444
 - exemplos na biblioteca de classes do .NET Framework, 445–447
 - expressões lambda, 461–463
 - fazendo a inscrição a eventos, 465
 - ícone IntelliSense para, 11
 - implementando sistema de controle de fábrica usando, 448–450
 - chamando um delegate, 449
 - delegate que referencia vários métodos, 449
 - removendo métodos de delegate, 449
 - métodos adaptadores, 461
 - modificador async com, 577
 - para eventos, 466
 - registrando delegate Action com token de cancelamento de tarefa, 554
 - sistema de controle de fábrica implementado sem, 447
 - tipo Action e Func, usando com a classe Command, 679
 - usando para imitar chamada de função como operador, 503
 - uso com objeto Dispatcher, método Invoke, 574
 - depurador
 - percorrendo métodos com o depurador do Visual Studio 2013, 80–83
 - tratamento de exceções e depurador do Visual Studio, 150
 - depurando aplicativos Windows Store, 27
 - desempenho
 - arrays multidimensionais e, 235
 - impacto de disparar o evento CanExecuteChanged com muita frequência, 688
 - destrutores
 - chamando método Dispose a partir de, 322–334, 327
 - criando classe simples que usa, 324–326
 - escrevendo, 314–316, 333
 - restrições sobre destrutores, 315
 - não definidos em interfaces, 291
 - não executando até que seja feita a coleta de lixo de objetos, 317
 - Device Window (no menu Design), 27
 - dicionário de recursos, 651, 660
 - adicionando referência para arquivo AppStyles.xaml, 652
 - diretiva using, 15
 - instrução using versus, 320

dispositivos baseados em toque e interfaces de usuário, 223, 612
D DivideByZeroException, 565
 duplicatas, eliminando de resultados de consulta, 485, 488

E

editor Entity Model, 716
 elementos Setter (XAML), 652
 elementos Style (XAML), 652
 encapsulamento, 162
 implementando usando métodos, 337–339
 Entity Data Model Wizard, 714
 Entity Framework, 710
 mapeamento de camadas entre banco de dados relacional e seu aplicativo, 712
 templates T4, aviso de alerta sobre, 716
 versões, 714
 entrada/saída (E/S) de arquivo, fonte de operações lentas, 582
 enumeração TaskContinuationOptions, 566
 enumeração TaskCreationOptions, 532
 enumeração TaskStatus, 558
 enumerações, 206–211
 arrays de, 227
 casting para int, 241
 declarando, 207, 224
 escolhendo valores de enumeração literais, 208
 escolhendo valores de enumeração subjacentes, 209–211
 usando, 207–208
 enumerações, ícone IntelliSense para, 11
 enumeradores, 429
 definindo para a classe Tree<TItem> usando um iterador, 439–441
 definindo usando um iterador, 442
 implementando manualmente, 430–434
 implementando sem usar um iterador, 442
 implementando usando um iterador, 437–441
 equi-joins, suporte da LINQ para, 489
 erros e exceções, 134–157
 exceções, 76
 exceções de tarefa
 continuações com tarefas canceladas ou falhas, 567
 tarefas canceladas, 561
 tratamento com a classe AggregateException, 565, 269
 garantindo descarte seguro quanto a exceções, 320, 333
 implementando descarte seguro quanto a exceções (exercício), 324–332
 lançando exceções, 151–155
 lidando com erros, 134–135
 testando código e capturando exceções, 135–146
 capturando várias exceções, 138–144
 exceções não tratadas, 136
 propagando exceções, 144–146
 usando várias rotinas de tratamento catch, 137

tratamento de exceções e o depurador do Visual Studio, 150
 usando aritmética de inteiros verificada e não verificada, 146–150
 escrevendo expressões verificadas, 148
 escrevendo instruções verificadas, 147
 usando um bloco finally, 155
 escopo, 72–74
 definindo escopo de classe, 73
 definindo escopo local, 72
 em instruções for, 123
 iniciando novo escopo em blocos de código, 99
 sobrecregando métodos, 74
 variáveis declaradas em, instrução using, 321
 escopo local, definindo, 72
 espaço em branco em C#, 40
 esquema de nomes camelCase, 165
 esquema de nomes PascalCase, 165
 estado ColumnarLayout, 648
 estado Running, tarefas, 558
 estado TabularLayout, 648
 estilo ColumnarHeaderStyle, 655
 estilo FontStyle, 654
 estilo GridStyle, 653
 estilo HeaderStyle, 654
 modificando com elementos adicionais da propriedade Setter, 656
 estilo LabelStyle, 657
 estilo TabularHeaderStyle, 655
 estilos, aplicando na interface do usuário de aplicativo Windows Store, 650–659, 660
 criando estilos personalizados, 660
 definindo estilos para o formulário Customers, 651–659
 estrutura KeyValuePair< TKey, TValue >, 415
 estruturas, 211–224
 arrays de, 227
 classes versus, 214
 compatibilidade com runtime no Windows 8 e 8.1, 223
 copiando variáveis de estrutura, 220
 comparando comportamento de cópia de estrutura e classe, 221–223
 criando e usando estrutura para representar uma data, 217–220
 declarando, 213, 225
 declarando variáveis de estrutura, 215, 225
 herança não aplicável às, 264
 ícone IntelliSense para, 11
 implementando propriedade de interface em, 357
 initialização, 216
 operadores em, 509
 palavras-chave usadas na definição de métodos, 310
 tipos de estrutura comuns, 211
 usadas como parâmetros para o método Console.WriteLine, 250
 evento CanExecuteChanged, 679
 adicionando a classe Command ao, 681
 adicionando cronômetro para disparar o evento, 681

disparando com muita frequência, 688
 evento PropertyChanged, 707
 eventos, 443
 cancelando a inscrição a, 476
 declarando, 475
 eventos de interface do usuário (IU), 466–474
 adicinando evento à classe CheckoutController (exercício), 467–474
 fazendo a inscrição, 475
 habilitando notificações usando, 463–466
 declarando um evento, 464
 disparando um evento, 465
 fazendo a inscrição em um evento, 464
 ícone IntelliSense para, 11
 recurso de segurança, 466
 tarefas esperando por, 594
 exceções. *Consulte* erros e exceções
 exceções não tratadas, 136
 capturando, 154
 informe do Windows de, 139–141
 experiência do usuário, 612
 expressões booleanas
 criando, 111
 em instruções for, 121
 em instruções if, 98
 em instruções while, 115
 expressões lambda, 419–421
 e métodos anônimos, 463
 formas de, 461–476
 sintaxe, 420
 usando em método Select, 480
 usando em método Where, 483
 expressões verificadas, escrevendo, 148
 Extensible Application Markup Language. *Consulte* XAML
 extensões de nome de arquivo, arquivos de solução e projeto, 44

F

F (sufixo de tipo), 50
 filas
 classe ConcurrentQueue<T>, 597
 classe Queue baseada em objetos, 376–379
 classe Queue versus classe Queue<T>, 382
 classe Queue<T> genérica, 380–382, 406, 407
 exemplo de classe Queue<int> e operações, 412–413
 filtrando dados, 482, 500
 usando operador where, 488
 fluxos (stream), escrevendo em, 582
 formulários, 22, 443
 localizando e selecionando controles com a janela Document Outline, 55
 redimensionando de formulários WPF, 27
 Windows Forms e aplicativos WPF, 18
 funções
 método, 461
 no Visual Basic, 67
 semelhança dos métodos com, 65

G

Generate Method Stub Wizard, 75
 genéricos, 376–405
 árvore binárias, teoria das, 383–386
 classes de coleção, 406–418
 construindo classe de árvore binária usando genéricos, 386–392
 testando a classe Tree<TItem>, 393–395
 criando o método genérico, 396–398
 definindo método para construir uma árvore binária, 396–398
 criando uma classe genérica, 383–395
 método genérico, Select<TSource, TResult>, 481
 problemas do tipo objeto, 376–379
 solução para os problemas do tipo objeto, 380–382
 usando restrições, 382
 variância e interfaces genéricas, 398–404
 versus classes generalizadas, 382
 gerenciamento de recursos, 319–324
 chamando método Dispose a partir de um destrutor, 322–334
 descarte seguro quanto a exceções, 320
 implementando descarte seguro quanto a exceções (exercício), 324–332
 criando classe que usa um destrutor, 324–326
 impedindo múltiplos descartes de objeto, 328–330
 implementando a interface IDisposable, 326–328
 segurança de thread e método Dispose, 330
 verificando descarte de objeto após uma exceção, 331
 métodos de descarte, 319
 gerenciamento de recursos, destrutores e, 314
 gestos, 613
 gestos baseados em toque, interagindo com aplicativos, 18
 GUIDs (identificadores globalmente exclusivos), 738

H

heap, 196
 arrays em, 227
 cópia automática de itens da pilha (boxing), 199
 custo de boxing e unboxing, 201
 diversas variáveis referenciando o mesmo objeto, 198
 instâncias de classe no, 215
 organização pelo runtime, 197
 usando, 197
 herança, 261–283
 acesso protegido, 272
 aplicável somente às classes, não às estruturas, 264
 atribuindo classes, 265–267
 chamando construtores de classe base, 264
 classe System.Object como classe-raiz, 264
 classes implementando interfaces, 286
 herdando de outra classe, 287
 criando uma hierarquia de classes (exercício), 272–277

- da classe System.Object, 198
- declarando métodos new, 267
- declarando métodos override, 269–271
- declarando métodos virtuais, 268
- definindo métodos, 261
- entendendo os métodos de extensão, 278–282
- impedindo com classes seladas, 303
- interface herdando de outra interface, 287
- interfaces, 291
- métodos virtuais e polimorfismo, 270
- tipos de exceção, 138
- HttpResponseObject, 725

- I**
- idempotência em web services REST, 730
- identificadores, 40–41
 - para variáveis, 42
- identificadores globalmente exclusivos (GUIDs), 738
- imagem de fundo para página ou controle, 654
- Implement Interface Wizard, 287
- implementação de interface explícita, 286, 289
 - implementando um indexador, 367
- incrementando e decrementando variáveis, 61
 - controlando operadores ++ e -- com as palavras-chave checked/unchecked, 148
 - prefixo e sufixo, 61
 - usando operadores ++ e -- em lugar de operadores de atribuição compostos, 114
- indexadores, 359–375
 - comparação com arrays, 364–366
 - propriedades, arrays e indexadores, 365
 - criando para classe ou estrutura, 379
 - definição, 359
 - em interfaces, 366–367
 - exemplo sem usar indexadores, 360–361
 - exemplo usando indexadores, 362
 - métodos de acesso, entendendo, 364
 - pontos importantes sobre, 363
 - usando em um aplicativo Windows, 367–374
 - escrevendo os indexadores, 370–373
 - examinando o aplicativo de agenda de telefones, 368–370
 - testando o aplicativo, 373
 - usando para simular o operador [], 503
- IndexOutOfRangeException, 230, 566
- indicador de ocupado, adicionando no formulário Customer, 728
- índices
 - array, 230
 - arrays multidimensionais, 235
 - tipos inteiros como, 241
 - classe Dictionary<TKey, TValue>, 414
- initializadores, coleção, 418
- initializadores de objeto, 354, 355
- instrução lock, 330
- instrução return, 67
- instrução using, 333
 - e interface IDisposable, 320–322
 - o objetivo da, 331
- instruções, 39–40
 - sintaxe e semântica, 39
- instruções break, 108, 124
 - obrigatórias para cada case em instrução switch, 108
- instruções continue, 124
- instruções de bloco não verificadas, 147
- instruções de decisão, 93–112
 - declarando variáveis booleanas, 93
 - instruções if, 97–105
 - instruções switch, 105–111
 - usando operadores booleanos, 94–97
- instruções do, 123–132, 133
 - escrevendo, 125–127
 - percorrendo, 127–132
- instruções else, 97. *Consulte também* instruções if
- instruções for, 121–123, 133
 - escopo, 123
- instrução continue em, 124
- iterando por coleções, 427
- iterando por um array, 230, 247
 - omitindo a inicialização, expressão booleana ou variável de controle de atualização, 122
 - várias inicializações e atualizações em, 122
- instruções foreach
 - iterando por coleções, 427
 - iterando por coleções List<T>, 409
 - iterando por um array, 231, 247, 428
- instruções if, 97–105, 111
 - em cascata, 99–105
 - expressões booleanas em, 98
 - sintaxe, 97
 - tipos de dados usados em, 107
 - usando blocos para agrupar, 98
- instruções switch, 105–111, 112
 - escrevendo, 108–111
 - regras de fall-through, 108
 - regras para, 107
 - sintaxe, 106
- instruções verificadas, escrevendo, 147
- instruções while, 115–121, 133
 - escrevendo, 116–121
 - sintaxe, 115
- inteiros
 - tipos inteiros como índices em um array, 241
 - valores inteiros associados a elementos de enumeração, 208, 209
- IntelliSense no Visual Studio 2013, 9
- ícones para membros de classe, 11
- Interface de usuário (IU)
 - criando para aplicativos Windows Store
 - aplicando estilos a uma interface de usuário, 650–659
 - aplicativo Adventure Works Customers (exercício), 616–618
 - implementando interface de usuário escalonável, 618–650
 - eventos, 466–474

interface de usuário escalonável para aplicativo Windows Store, implementando, 618–650, 660 adaptando o layout usando Visual State Manager, 643–650 implementando layout tabular usando controle Grid, 633–643 modificando o layout para diferentes tamanhos e orientações, 635–643 organizando a página do aplicativo Customers, 618–629 testando o aplicativo no Visual Studio 2013 Simulator, 630–633 interface ICommand, 678, 679 interface IComparable<T>, 490 interface IComparer<T>, 402–404 interface IDisposable, 321 implementação (exemplo), 322 implementando (exercício), 326–328 implementando, 333 interface IEnumerable, 428 estruturas de dados que implementam, 479 método GetEnumerator, 429 interface IEnumerator, 429 classe que implementa, 437 estruturas de dados que implementam, 479 implementando, 434–436 interface IEnumerator, 429 interface IEnumerator<T>, 429 interface INotifyPropertyChanged, implementando, 670–673 interface System.Collections.IEnumerable, 428 interface System.Collections_IEnumerator, 429 interfaces, 284–301 convenção de atribuição de nomes, 286 declarando e implementando propriedades de interface, 345–351, 357 definindo, 285, 311 definindo e usando (exercício), 291–301 criando classes que implementam a interface, 293–298 testando implementação de classes, 298–301 entendendo, 284 genéricas, 382 variância e, 398–404, 405 herdando de outra interface, 287 ícone IntelliSense para, 11 implementando, 286–287, 311 implementando explicitamente, 289 indexadores em, 366–367 palavras-chave usadas na definição de métodos, 310 referenciando a classe por meio de sua interface, 287 restrições, 290 trabalhando com várias interfaces, 288 interfaces contravariantes, 402–404 interfaces covariantes, 400, 405 Interoperabilidade de linguagem, operadores e, 507 InvalidCastException, 200

iteradores definindo enumerador usando, 442 implementando enumerador sem usar um iterador, 442 implementando um enumerador usando, 437–441 definindo enumerador para a classe Tree<TItem>, 439–441

J

janela Code and Text Editor, 6, 47 arquivo Program.cs na, 8 localizando e substituindo código, 48 janela Design View, 20 alterações na descrição de formulário XAML e, 24 ampliando e reduzindo, 47 expandindo/recolhendo elementos no painel XAML, 644 usando para adicionar controles de formulário, 22 janela Document Outline, 55 fixando com o botão Auto Hide, 56 janela Error List, 12, 515 janela Object Collection Editor, 624 janela Output, 12 janela Properties botão Event Handlers for the Selected Element, 35 especificando propriedades de controles de formulário, 24 janela Task List, localizando comentários TODO, 186 janela Transact-SQL Editor, 711 Java arrays de arrays, 237 colchetes antes do nome de variável de array, 227 herança, 263 métodos virtuais, 269 recurso varargs operando como a palavra-chave params em C#, 252 JavaScript Object Notation. Consulte JSON jogo de cartas usando arrays para implementar, 237–246 usando classes de coleção, 421–426 JSON (JavaScript Object Notation), 718 resultado retornado de web service REST, 722

L

lançando exceções, 151–155, 157 capturando a exceção, 153 capturando exceções não tratadas, 154 Language-Integrated Query. Consulte LINQ larguras de aplicativos Windows Store definindo o layout para visualização estreita, 644–647 mínima padrão, 628 testando no Simulator, 642 layout tabular, implementando usando o controle Grid, 633–643 modificando o layout para mudar de escala para diferentes tamanhos e orientações, 635–643

- legibilidade, 612
- Lei de Moore, 529
- Linguagem de programação Haskell, 419
- linguagens de programação funcionais, 419
- LINQ (Language-Integrated Query), 477, 570, 712
 - operador await, incapacidade de usar em consultas LINQ, 575
 - parallelizando uma consulta LINQ, 609
 - PLINQ (Parallel LINQ), 571
 - cANCELAMENTO de consulta PLINQ, 609
 - usando para parallelizar acesso declarativo aos dados, 584–590
 - usando em um aplicativo C#, 478–499
 - consultando dados em objetos Tree<TItem>, 489–495
 - filtrando dados, 482
 - LINQ e avaliação postergada, 495–499
 - ordenando, agrupando e agregando dados, 483–485
 - selecionando dados, 480–482
 - unindo dados com join, 485
 - usando operadores de consulta, 487–489
- M**
- membros de classe
 - ícones IntelliSense para, 11
 - lista suspensa em Code and Text Editor, 49
- memória
 - alocação e reivindicação para variáveis e objetos, 313
 - alocação para instância de array, 228
 - como a memória do computador é organizada, 195–198
 - pilha e o heap, 196
 - usando a pilha e o heap, 197
 - não gerenciada, para objetos criados em código inseguro, 203
 - uso por arrays multidimensionais, 235
- memória não gerenciada, 203
- menu Debug
 - Start Debugging, 14
 - Start Without Debugging, 13
- método CanExecute, 678
 - classe Command, implementando, 680
- método Clone, 185
 - classe Array, 235
 - usando para copiar arrays, 365
- método Close, 319
- método Compare, 402
- método Console.WriteLine, 9
 - chamando método ToString automaticamente, 212
 - exemplo clássico de sobrecarga em C#, 249
 - sobrecarga para suportar argumento de string de formato contendo espaços reservados, 254
- método ContinueWith, objeto Task, 533, 566
 - parâmetros especificando itens adicionais, 533
- método Copy, classe Array, 234
- método CopyTo, classe Array, 234
 - método de extensão AsParallel, 585
- método Dispose, 321, 327
 - chamando a partir de um destrutor, 322–334, 327
 - segurança de thread e, 330
- método Distinct, 485, 488, 493
- método Equals
 - estruturas, 213
 - substituindo em System.Object ou System. ValueType, 510
 - substituindo na classe Complex, 515
- método Execute, 679
- método Finalize, 316
- método Find, coleções genéricas, 419–421
 - classe List<T>, 445
- método For, classe Parallel, 546, 592
 - cancelando, 559
 - regra geral para usar, 552
- método ForEach<T>, classe Parallel, 546
 - cancelando, 559
 - regra geral para usar, 552
- método GC.Collect, 317
- método GC.SuppressFinalize, 324, 329
- método GetAwaiter, objetos que podem esperar, 575
- método GetHashCode, 402
 - substituindo em System.Object ou System. ValueType, 510
 - substituindo na classe Complex, 515
- método GroupBy, 484, 493, 500
- método InitializeComponent, classe MainWindow, 34
- método Int32.Parse, 52
- método Invoke
 - classe Parallel, 547
 - reservando para operações que utilizam bastante poder de computação, 550
 - objeto Dispatcher, 574
- método Join, 485, 501
- método Main, 8
 - exemplo de aplicativo de console, 9
 - parâmetros de array e, 233
- método MoveNext, enumeradores, 429
- método OrderBy, 483, 500
- método OrderByDescending, 484
- método Pop, 406
- método Push, 406
- método Run, classe Task, 532, 568
- método Select, 480–482, 492, 500
- método ShowBoolValue, 51
- método ShowDoubleValue, 51
- método ShowFloatValue, 49
- método ShowIntValue, 50
- método Start, objeto Task, 532
- método ThenBy ou ThenByDescending, 484
- método ToArray, 498
 - classes de coleção, 421
- métodoToList, 496, 498
- método ToString, 50
 - classe System.Object, 268
- método typeSelectionChanged, 49
- método Wait, objeto Task, 534, 565, 568
- método Where, 482, 493, 500

- métodos, 163
 abstratos, 303
 adaptadores, 461
 anônimos, 463
 assinatura de, 267
 assíncronos. *Consulte* operações assíncronas chamando, 69, 444
 comprimento dos, 69
 construtor. *Consulte* construtores convenções de atribuição de nomes, 165 criando um método genérico, 396–398 declarando, 66 declarando métodos new para classe em hierarquia de herança, 267 declarando métodos override, 269–271 declarando métodos virtuais, 268 definindo e chamando um método genérico, 404 delegates. *Consulte* delegates elementos dos, 419 escopo, 72–74 escrevendo, 74–83 parâmetros, 78 percorrendo com o depurador, 80 testando o programa, 79 usando Generate Method Stub Wizard, 75 estáticos, 175–181, 182 ícone IntelliSense para, 11 implementando encapsulamento usando, 337–339 indexadores *versus*, 363 instância, 173–175 interface, 285, 291 implementação de, 286 memória exigida para parâmetros e variáveis locais, 196 métodos de extensão, entendendo, 278–282 métodos virtuais e polimorfismo, 270 não params, prioridade sobre métodos params, 253 palavras-chave usadas na definição de métodos para interfaces, classes e estruturas, 310 passando arrays como parâmetros e valores de retorno, 232 privados ou públicos, 164 retornados por expressões lambda, 419–421 retornando dados de, 67 selados, 303 sobrecarregados, 10 sobrecarregando, 74, 249 substituindo por propriedades (exercício), 347–351 usando parâmetros opcionais e argumentos nomeados, 83–91 definindo e chamando método com parâmetros opcionais, 87–91 definindo parâmetros opcionais, 85 passando argumentos nomeados, 85 resolvendo ambiguidades, 86 usando parâmetros por valor e por referência, 186–189 usando parâmetros ref e out, 192–195 métodos adaptadores, 461 métodos anônimos, expressões lambda e, 463 métodos de acesso, 338. *Consulte também* propriedades métodos de acesso get e set. *Consulte também* propriedades para indexadores, 363 entendendo, 364 para propriedades, 338 restrições sobre, 344 métodos de classe. *Consulte* métodos e dados static métodos de descarte, 319, 333 definidos, 319 descarte seguro quanto a exceções, 320 métodos de extensão, 278–282 criando (exercício), 279–282 definindo para um tipo, 283 Enumerable.Select, 481 ícone IntelliSense para, 11 métodos de instância definição, 173 escrevendo e chamando, 173–175 métodos de superconjunto ou subconjunto, classe HashSet<T>, 417 métodos e dados static, 175–182 classes static, 177 criando campos static usando a palavra-chave const, 177 criando um campo compartilhado, 176 escrevendo membros static e chamando métodos static, 178–180 método static aceitando parâmetro de valor, 186 métodos IntersectWith, UnionWith e ExceptWith, classe HashSet<T>, 417 métodos modificadores, 338. *Consulte também* propriedades métodos sobrecarregados, 10 métodos virtuais, 269, 283 e polimorfismo, 270 exibição do IntelliSense de métodos disponíveis, 276 métodos de acesso indexadores implementados em uma classe, 367 regras importantes para, 270 métodos WaitAll e WaitAny, classe Task, 534, 565 método WaitAll, 568 Microsoft Blend for Visual Studio 2013 definindo estilos complexos para integrar em um aplicativo, 659 Microsoft SQL Server, 478 mobilidade como requisito importante dos aplicativos modernos, 613 modelo de entidades, criando, 711–717, 748 modelo de entidades AdventureWorks, 712–729 modelo first-in, first-out (FIFO), 376, 407 modelo last-in, first-out (LIFO), 406 modificador abstract não aplicável a operadores, 504 palavra-chave abstract, 302, 311

- modificador `async`
 - conceito errado sobre, 575
 - implementando método assíncrono, 575
 - nenhuma instrução `await` em método `async`, 577
 - prefixando delegates, 577
- modificador `private`, 272
 - construtores, 166
 - copiando tipos-referência e privacidade de dados, 185
 - em nível de classe *versus* em nível de objeto, 174
 - identificadores, esquema de nomes para, 165
 - propriedades, 343
 - uso com a palavra-chave `static`, 178
- modificador `protected`, 272
 - propriedades, 343
- modificador `public`, 272
 - campos de estrutura `e`, 213
 - construtores, 166
 - exigido para operadores, 504
 - identificadores, esquema de nomes para, 165
 - propriedades, 343
- modificador `static`
 - exigido para operadores, 504
 - propriedades, 342
 - uso com a palavra-chave `private`, 178
- modificadores. Consulte também palavras-chave e
 - modificadores listados por toda parte
 - aplicáveis e não aplicável a operadores, 504
- modificadores de acesso
 - não aplicáveis para destrutores, 315
 - não definidos para métodos de interface, 291
 - para propriedades, 343
- modo Debug, 14
 - aplicativo Windows Store no Windows 8.1, 28
- modo retrato, vendo aplicativo no, 27
- multitarefa
 - fazendo por meio de processamento paralelo, 527–529
 - implementando com o .NET Framework, 530–552
- N**
- namespace `System`, 15
 - assemblies implementando classes no, 17
- namespace `System.Collections`, 407
- namespace `System.Collections.Concurrent`, 597
- namespace `System.Collections.Generic`
 - classe `SortedDictionary<TKey, TValue>`, 415
 - classe `SortedSet<T>`, 418
 - classes de coleção, 406
 - interface `IComparer`, 402
- namespace `System.Collections.Generic.Concurrent`, 408
- namespace `System.Linq`, classe `Enumerable`, 481–487
- namespace `System.Text.RegularExpressions`, 732
- namespace `System.Threading`, 530
 - primitivas de sincronização, 594
- namespace `System.Threading.Tasks`
- classe `Parallel`, 546
- enumeração `TaskStatus`, 558
- namespace `Systems.Collections.Generics`, 381
- namespace `Windows.UI.Popups`, 36
- namespace `Windows.UI.Xaml`
 - classe `DispatcherTimer`, 681
- namespace `Windows.UI.Xaml.Media.Imaging`, 535
- namespaces
 - e assemblies, 17
 - em aplicativos WPF *versus* aplicativos Windows Store, 34
 - ícone IntelliSense para, 11
 - usando, 14–17
- namespaces `System.Windows`, 34
- namespaces `Windows.UI`, 34
- `NaN` (não é um número), 53
- .NET Framework
 - biblioteca de classes, divisão em assemblies, 17
 - bibliotecas ou assemblies, 7
 - classe `FileInfo`, 118
 - classe `HttpClient`, 722
 - classe `HttpResponseMessage`, 722
 - classe `TextReader`, 118
 - classes de coleção, 406–418
 - no namespace `System.Collections`, 407
 - no namespace `System.Collections.Generic`, 406
 - no namespace `System.Collections.Generic.Concurrent`, 408
 - classes de coleção concorrentes, 597
 - classes de exceção, 151
 - classes e controles de GUI, eventos, 466
 - CLR (Common Language Runtime), 223
 - delegates, exemplos em biblioteca de classes, 445–447
 - e compatibilidade com WinRT no Windows 8 e 8.1, 308–310
 - eventos, 463
 - extensões da PLINQ, 585
 - `IEnumerable` e interface `IEnumerators`, 429
 - implementando multitarefa usando, 530–552
 - interfaces exibindo covariância, 401
 - interfaces `IEnumerable<T>` e `IEnumerator<T>`, 429
 - namespace `System.Collections.Generics`, 381
 - padrão de projeto `IAsyncResult` em versões anteriores, 583
 - primitivas de sincronização, 594
 - sincronizando acesso a dados entre tarefas, 571
 - tipos de exceção, 138
 - tipos primitivos e equivalentes em C#, 212
 - notação camelCase, 42
 - notação de array
 - referenciando elemento existente em `List<T>`, 408
 - usando com `Dictionary<TKey, TValue>`, 415
 - usando para acessar valores em coleções de dicionário, 419
 - notação de ponto `(.)`, 279, 314
 - notação húngara, 42, 286
 - `NotImplementedException`, 76

números complexos, desenvolvendo classe que simula, 511–517
 adicionando operadores de conversão, 520–523
 criando a classe Complex e implementando operadores aritméticos, 511–514
 implementando operadores de igualdade, 514–517

O

objeto App, definido no arquivo App.xaml de aplicativo WPF, 34
 objeto CancellationToken, 553, 590, 610
 método ThrowIfCancellationRequested, 561, 269
 objeto CancellationTokenSource, 553, 269, 590, 610
 método Cancel, 554
 objeto CommandManager, 681
 objeto Dispatcher, 574
 objeto MessageDialog, 36
 método ShowAsync, 581
 objeto ParallelLoopState, 546, 559
 objeto ParallelQuery, 585
 método WithCancellation, 590, 609
 objeto RoutedEventArgs, 466
 objeto stopwatch, 536
 objeto System.Diagnostics.Stopwatch, 536
 objeto System.Threading.CancellationTokenSource, 553
 objeto TaskScheduler, 532
 objeto WriteableBitmap, 535, 583
 objetos
 alocação e reivindicação de memória para, 313
 array de parâmetros de tipo object, 253
 classe System.Object, 198, 264
 método Finalize, 316
 substituindo métodos Equals e GetHashCode, 510
 classes versus, 163
 convertendo em strings, 50
 criando, 170
 criando usando a palavra-chave new, 163
 inicializando usando propriedades, 353–356, 358
 memória exigida para, 196
 modificador private e, 174
 verificando o tipo com o operador is, 201
 vida dos, 313–319
 objetos que podem esperar, 575
 opção Allow Unsafe Code, 203
 operações assíncronas, 570–610
 cancelando uma consulta PLINQ, 590
 implementando métodos assíncronos, 571–584, 608
 definindo métodos assíncronos, problema, 571–574
 definindo métodos assíncronos, solução, 574–580
 definindo métodos assíncronos que retornam valores, 580–581
 métodos assíncronos e APIs do Windows Runtime, 581–585
 usando métodos assíncronos no aplicativo GraphDemo, 577–580
 melhorando o desempenho ao iterar sobre coleções, 585–590

padrão de projeto IAsyncResult nas versões anteriores do .NET Framework, 583
 requisições de web service, 725
 sincronizando o acesso simultâneo aos dados, 591–608
 bloqueando dados, 593
 cancelando a sincronização, 596
 classes de coleção concorrentes, 597
 primitivas de sincronização para coordenar tarefas, 594–596
 usando coleção concorrente e bloqueio para acesso seguro aos dados, 598–608
 usando PLINQ para parallelizar acesso declarativo aos dados, 584–590
 operador address-of (&), 202
 operador as, 202
 fazendo casting e testando se o resultado é null, 205
 operador await, 574, 576
 chamando método assíncrono que retorna um valor, 581
 Pontos importantes sobre, 575
 operador de adição. Consulte + (sinal de mais), sob Símbolos
 operador de atribuição. Consulte = (sinal de igual), sob Símbolos; Consulte + (sinal de mais), sob Símbolos
 operador de concatenação de strings (+), 52
 operador de decreto. Consulte - (sinal de menos), sob Símbolos
 operador de decreto pós-fixado (--), 96
 operador de decreto prefixado (--), 96
 operador de desigualdade. Consulte ! (ponto de exclamação), sob Símbolos
 operador de divisão. Consulte / (barra normal), sob Símbolos
 operador de incremento. Consulte + (sinal de mais), sob Símbolos
 operador de incremento prefixado (++), 96
 operador de incremento prefixado (++), 96
 operador de multiplicação. Consulte * (asterisco), sob Símbolos
 operador de seleção, 487, 494, 500
 operador de subtração. Consulte - (sinal de menos), sob Símbolos
 operador from, 487, 494, 500
 operador group, 488
 operador group by, 500
 operador is, 201
 determinando se um objeto é do tipo especificado, 288
 testando se o casting é válido, 205
 operador join, 489, 501
 operador maior ou igual a. Consulte > = e igual (sinais de maior e igual), sob Símbolos
 operador maior que. Consulte > (sinal de maior), sob Símbolos
 operador menor ou igual a. Consulte < = (sinais de menor e igual), sob Símbolos

- operador menor que. *Consulte <* (sinal de menor), sob Símbolos
- operador módulo. *Consulte %* (sinal de porcentagem), sob Símbolos
- operador NOT (!), 94
- operador NOT (~), 360
- operador orderby, 488, 500
- operador resto (ou módulo) (%), 53
- operador where, 488, 500
- operadores, 52
 - associatividade, 60
 - bit a bit, 360
 - booleanos, usando, 94
 - e tipos de dados, 52
 - entendendo, 502–507
 - restrições de operadores, 503
 - incremento e decremento, 61
 - sobrecregando
 - avaliação de atribuição composta, 507–508
 - comparando operadores em estruturas e classes, 509
 - criando operadores simétricos, 505–507
 - declarando operadores de incremento e decremento, 508
 - definindo pares de operador, 509–510
 - e interoperabilidade de linguagem, 507
 - implementando operadores, 511–517
 - operadores de conversão, 517–523
 - precedência e associatividade, 96
 - uso em seus próprios tipos de estrutura, 213
 - uso em variáveis de enumeração, 208
 - operadores aritméticos, 52–61
 - controlando a precedência, 59
 - implementando versões sobrecregadas na classe Complex, 512
 - tipos de dados usados com, 52
 - usando em valores int, 53–61
 - operadores binários, 503
 - argumentos explícitos, 504
 - operadores bit a bit, 360
 - operadores booleanos, 94–97
 - operadores de igualdade e relacionais, 94
 - operadores de lógica condicional, 95
 - operadores compostos, controlar usando as palavras-chave checked e unchecked, 148
 - operadores de atribuição, compostos, 113–114, 133
 - entendendo avaliação de atribuição composta, 507–508
 - operadores de conversão, 517–523
 - conversões predefinidas para tipos predefinidos, 518
 - criando operadores simétricos, 520
 - definidos pelo usuário, implementando, 519
 - definindo, 524
 - escrevendo, 520–523
 - operadores de igualdade, 94, 111
 - associatividade e precedência, 97
 - implementando versões sobrecregadas, 514–517
 - substituindo operadores == e !=, 509
- operadores de lógica condicional, 95
 - curto-círcuito, 96
- operadores lógicos, 95
 - associatividade e precedência, 97
 - em expressões booleanas, 111
- operadores relacionais, 94, 111
 - associatividade e precedência, 97
- operadores unários, 503
 - argumento explícito, 504
- operandos, 52
- OperationCanceledException, 561, 566, 269
 - tratando, 562
- orientação paisagem, 628
- orientação retrato, 628
- orientações
 - aplicativo para tablet em, paisagem ou retrato, 628
 - testando no Simulator, 632
 - testando para aplicativo Windows Store no Simulator, 640
- OverflowException, lançada por instruções verificadas, 147

P

- padrão de projeto IAsyncResult, 583
- padrão MVVM (Model-View-ViewModel), 661–689
 - adicionando botões Next e Previous à View, 686–689
 - adicionando comandos a um ViewModel, 678–681
 - adicionando comandos NextCustomer e PreviousCustomer ao ViewModel, 682–686
 - criando um ViewModel, 674–678
 - definido, 662
 - exibindo dados via vinculação de dados, 662–668
 - modificando dados via vinculação de dados, 668
- páginas, 22
 - adaptando a diferentes resoluções de tela e orientação de dispositivo, 27
- palavra-chave await, fora de métodos async, 575
- palavra-chave case, 106
- palavra-chave checked, 147, 157
- palavra-chave const, criando um campo static com, 177, 182
- palavra-chave default, 106
 - inicializando variável definida com um parâmetro de tipo, 434
- palavra-chave enum, 207
- palavra-chave interface, 285
- palavra-chave namespace, 15
- palavra-chave new, 163, 311
 - chamando construtores, 182
 - criação de objeto, 313
 - criando instância de array, 227, 247
- palavra-chave object, 198
- palavra-chave out, 192
 - incapacidade de usar com arrays params, 253
- palavra-chave override, 269, 271, 311
 - não aplicável aos operadores, 504
 - usando com palavra-chave virtual, 270

- palavra-chave params, 251
 - pontos importantes sobre arrays params, 252
- palavra-chave private, 164, 311
- palavra-chave protected, 311
- palavra-chave public, 164, 311
- palavra-chave ref, 192
 - incapacidade de usar com arrays params, 253
- palavra-chave sealed, 311
 - não aplicável a operadores, 504
- palavra-chave string, 184
- palavra-chave switch, 106
- palavra-chave this
 - uso com indexadores, 362
- palavra-chave unchecked, 147
- palavra-chave unsafe, tornando código inseguro, 203
- palavra-chave var, 180
 - usando no lugar de um tipo, 63
 - uso na definição de tipo de coleção enumerável, 482
- palavra-chave virtual, 311
 - declarando implementações de propriedade como virtuais, 346
 - não aplicável aos operadores, 504
- palavras-chave, 40
 - definindo métodos para interfaces, classes e estruturas, 310
- palavras-chave explicit e implicit, 519
- palavras-chave implicit e explicit, 519
- parallelismo
 - classe Parallel determinando o grau de, 547
 - usando a classe Task para implementar, 534–545
- paralelização
 - dividindo método em séries de operações paralelas, 576
 - grau de *versus* unidades de, 530
 - usando PLINQ para parallelizar acesso declarativo aos dados, 584–590
- parâmetro de tipo (<T>) para genéricos, 380
- parâmetros
 - expressão lambda, 420
 - memória exigida para, 196
 - operador, 504
 - parâmetros opcionais para métodos, 83–92
 - definindo, 85
 - definindo e chamando método com, 87–91
 - resolvendo ambiguidades com, 86
 - passando arrays como parâmetros, 232
 - parâmetros de array e método Main, 233
 - tipos genéricos como, 396
 - usando parâmetros por valor e por referência, 186–189
 - usando parâmetros ref e out, 192–195, 204
 - variáveis nullable como, 191
- parâmetros opcionais
 - comparando com arrays de parâmetro, 257–259
 - definindo, 85
 - definindo e chamando um método com, 87–91
 - resolvendo ambiguidades com, 86
- parâmetros out
 - criando, 193, 205
 - indexadores *versus* arrays, 365
- parâmetros ref
 - criando, 193
 - indexadores *versus* arrays, 365
 - passando um argumento para, 204
 - usando, 194
- pasta Debug, 14
- pasta Documents, 6
- pasta MainPage.xaml, 20
- pasta obj, 14
- pasta Properties, 7
- pasta References, adicionando/removendo referências para assemblies, 17
- pasta TestHello, 7
- pastas bin e obj, 14
- pilha, 196
 - classe ConcurrentStack<T>, 598
 - cópia automática de item da pilha para o heap (boxing), 199
 - instâncias de estrutura na, 215
 - organização pelo runtime, 196
 - usando, 197
- pixels, tamanho de fonte medido em, 25
- PLINQ (Parallel LINQ), 571
 - cancelamento de consulta PLINQ, 609
 - usando para parallelizar acesso declarativo aos dados, 584–590
 - cancelando uma consulta PLINQ, 590
 - melhorando o desempenho ao iterar por coleções, 585–590
- polimorfismo
 - definido, 271
 - métodos virtuais e, 270
- ponteiros e código inseguro, 202
- precedência, 502
 - controlando com operadores aritméticos, 59
 - expressões contendo operadores com a mesma precedência, 60
 - precedência de operadores, resumo da, 96
- predicados, 419–421
 - delegates e, 445
- prefixo e sufixo, operadores de incremento (++) e decremento (--) , 61, 508
- processadores multinúcleo, surgimento dos, 528
- processamento paralelo, usando para fazer multitarefa, 527–529
- propagando exceções, 144–146
- propriedade Content, alterando para controles Button, 26
- propriedade Count
 - classe List<T>, 410
 - localizando número de elementos em coleções, 426–427
- propriedade Current, interface IEnumerator, 429
- propriedade DataContext, objeto MainPage, 677
- propriedade FontSize, alterando para controle TextBlock, 24

- propriedade HasValue, tipos nullable, 191
 propriedade Length
 classe Array, 230, 247
 classe List<T>, 410
 propriedade Name para todos os controles, 623
 propriedade RenderTransform, 656
 propriedade StartupUri, objeto App, aplicativo WPF, 34
 propriedade Status, objeto Task, 558
 propriedade Title, alterando para formulários, 26
 propriedade Value, tipos nullable, 191
 propriedades, 337–358, 359
 acessibilidade, 343
 arrays e indexadores, 365
 declarando e implementando em uma interface, 357
 declarando propriedades de interface, 345–351
 definido, 339
 gerando propriedades automáticas, 351–352, 358
 ícone IntelliSense para, 11
 inicializando objetos usando, 353–356, 358
 leitura/esrita, 342, 357
 nomes de, aviso sobre nomes de campo e, 341
 restrições em, entendendo, 344–345
 somente escrita, 342, 357
 somente leitura, 342, 357
 substituindo métodos por (exercício), 347–351
 usando, 341–342
 usando adequadamente, 345
 usando para simular operador de atribuição (=), 503
 propriedades anexas, 636
 propriedades leitura/esrita, 342, 357
 propriedades somente escrita, 342, 357
 propriedades somente leitura, 342, 357
- R**
- recurso ImageBrush, 652
 recurso Peek Definition, 88
 recursos não gerenciados, destrutores e, 314
 refatorando código, 79
 referências
 armazenamento na pilha, 197
 criando várias referências a um objeto, 316
 definindo com null para descarte imediato de, 314
 referenciando uma classe por meio de sua interface, 287
 reivindicação pelo heap, 198
 referências de objeto. Consulte também referências; tipos-referência
 armazenadas por classes no namespace System. Collections, 407
 região Methods For Fetching And Updating Data, ViewModel, 732
 regra de atribuição definitiva, 43
 relato de erro, adicionando à classe ViewModel, 739–741
 requisições DELETE, 729
 requisições e respostas HTTP, 718
 requisições PUT, POST e DELETE, 729
 web service REST, 720
- requisições POST, 729
 requisições PUT, 729
 resoluções de tela
 páginas em aplicativos Windows Store ajustando-se a, 27
 Simulator no Visual Studio 2013, 631
 testando para aplicativo Windows Store no Simulator, 641
 rotinas de tratamento catch, 135, 157
 capturando exceções não tratadas, 154
 capturando uma exceção, 153
 usando várias, 137
 rótulos case em instruções switch, 107
- S**
- seção All WPF Controls (Windows 7 e 8), 23
 seção All XAML Controls (Windows 8.1), 23
 selecionador de arquivos Open, 116
 semântica, 39
 Simulator, usando para testar aplicativos Windows Store, 630–633, 640–643
 sincronizando acesso simultâneo aos dados, 591–608, 609
 bloqueando dados, 593
 cancelando a sincronização, 596–597
 classes de coleção concorrentes, 597–598
 primitivas de sincronização para coordenar tarefas, 594–596
 usando coleção concorrente e bloqueio para acesso seguro aos dados, 598–608
 sintaxe, 39
 sistema de controle de fábrica, 447–450
 implementando sem usar delegates, 447
 implementando usando delegate, 448
 sistemas de gerenciamento de banco de dados relacional, 478
 sobrecarregando métodos, 74, 249
 construtores, 167
 Solution Explorer, 6
 arquivos de projeto no, 7
 spinning, 586
 SQL (Structured Query Language), 478
 Entity Framework reduzindo a dependência de, 711
 geração de comandos SQL por DbContext e DbSet, 717
 modelo de entidades convertendo consultas LINQ em comandos SQL, 712
 SQL Server, 478
 StackOverflowException, 341
 status Created, tarefas, 558
 status RanToCompletion, tarefas, 558
 status WaitingToRun, tarefas, 558
 strings
 arrays de, 229
 como objetos, 398
 convertendo em valores inteiros, 52
 convertendo objetos em, 50
 convertendo variáveis de enumeração em, 208

- implementação de `IComparable` e `IComparable<T>`, 394
- operador `+=` usado em, 114
- tipo `string` como classe, `System.String`, 184
- Structured Query Language.** Consulte `SQL`
- substituindo métodos, 283
 - declarando métodos override, 269–271
- sufixos de tipo, 50
- suspendendo e retomando aplicativos, 614

- T**
- tarefas, 527–269, 571–574
 - cancelando tarefas e tratando exceções, 552–567, 269
 - adicionando cancelamento no aplicativo `GraphDemo`, 554–558
 - cancelando loop `Parallel For` ou `ForEach`, 559
 - continuações com tarefas canceladas ou falhas, 566
 - exibindo o status de cada tarefa, 559–561
 - reconhecendo cancelamento e tratando exceção, 562–564
 - tratando exceções com a classe `AggregateException`, 565
 - implementando multitarefa usando .NET Framework, 530–552
 - abstraindo tarefas usando a classe `Parallel`, 546–550
 - criando, executando e controlando tarefas, 531–534
 - quando não usar a classe `Parallel`, 550–552
 - usando a classe `Task` para implementar paralelismo, 534–545
 - motivos para fazer multitarefa usando processamento paralelo, 527–529
 - PLINQ baseada em. Consulte `PLINQ`
 - sincronizando acesso simultâneo aos dados, 591–608
 - primitivas de sincronização para coordenar tarefas, 594–596
 - tarefas canceladas, 558
 - usando continuações com, 566
 - tarefas falhas, 533, 558
 - usando continuações com, 566
 - Task Manager, fechando aplicativos Windows Store, 29
 - `TaskCanceledException`, 566
 - taxa de redesenho, monitorando para aplicativos Windows Store, 28
 - template ADO.NET Entity Data Model, 714
 - template ASP.NET Web API, 713
 - versão do Entity Framework referenciada, 714
 - web service criado com, 718
 - template Blank App, 20, 616. Consulte também aplicativos Windows Store
 - template Grid App, 616, 705
 - template Hub App, 706
 - template Split App, 616, 705
 - template Web API, 713
 - template Windows Forms Application, 18
 - template WPF Application, 18, 21
 - templates
 - aplicativo Windows Store, 616, 705
 - escolhendo template para aplicativo de console, 5
 - para aplicativos gráficos, 18
 - tempo de resposta, problemas com, 570
 - threads
 - interrompendo quando o coletor de lixo é executado, 318
 - segurança de threads e o método `Dispose`, 330
 - tarefas, threads e o `ThreadPool`, 530
 - tipo `bool`, 43
 - tipo `char`, 43
 - tipo de retorno para métodos, 66
 - tipos genéricos como, 396
 - tipo `decimal`, 43
 - tipo `delegate RoutedEventHandler`, 466
 - tipo `double`, 43
 - tipo `float`, 43
 - tipo genérico `DbSet`, 717
 - tipo `int`, 43, 360–361
 - implementação de `IComparable` e `IComparable<T>`, 394
 - operadores usados para manipular bits individuais, 360
 - tamanho, 146
 - tipo `long`, 43
 - tipo `object`, 78
 - problemas do, 376–379
 - tipo `string`, 43
 - tipo `TaskContinuationOptions`, 533
 - tipos anônimos, arrays de, 229
 - tipos construídos, 382
 - tipos de dados
 - `casting`, usando o operador `as`, 202
 - conversões, 517–523
 - conversões predefinidas para tipos predefinidos, 518
 - implementando operadores de conversão definidos pelo usuário, 519–523
 - declarando variáveis locais implicitamente tipadas, 62
 - elementos de array, 227, 229
 - especificando para variáveis, 42
 - método `ToString`, 50
 - operadores `e`, 52
 - primitivos, 43–51
 - tipos numéricos e valores infinitos, 53
 - tipos primitivos em C# e equivalentes no .NET Framework, 212
 - uso de instrução `switch em`, 107
 - variáveis criadas com a palavra-chave `var`, 180
 - verificando o tipo de objeto com o operador `is`, 201
 - tipos de dados primitivos, 43–51
 - como tipos-valor, 183

- em C# e tipos equivalentes no .NET Framework, 212
 exibindo valores, 44–51
 tamanho fixo, 146
- tipos delegate Action**, 447
 delegate Action<T, ...>, 446
 registrando com token de cancelamento de tarefa, 554
 usando com tarefas, 531
- tipos delegate Func**, 446–447
 delegate Func<T, ...>, 446
- tipos enum**. Consulte enumerações
- tipos nullable**, 190
 como parâmetros de método, 192
 criação em memória heap, 196
 entendendo as propriedades dos, 191
 enumeração, 207
 estrutura, 216
- tipos numéricos**
 e valores infinitos, 53
 usando operador resto com, 53
- tipos-referência**, 183, 378
 alocação e reivindicação de memória para, 313
 arrays, 227
 classes, 215
 cópia, privacidade de dados e, 185
 copiando uma variável de tipo-referência, 204
 copiando variáveis de tipo-referência, 213
 covariância, 401
 inicializando usando valores null, 189
 método GetHashCode, 402
 modificadores ref e out em parâmetros de referência, 195
 usando para parâmetros de método, 188, 192
 variáveis de tipo objeto referindo-se a, 198
- tipos-valor**, 183
 alocação e reivindicação de memória para, 313
 copiando variáveis de tipo-valor, 213
 copiando variáveis e classes, 183, 204
 declarando variáveis e classes como, 184
 criação na memória de pilha, 196
 criação na memória heap, 196
 criando com enumerações e estruturas, 206–225
 estruturas, 211–224
 trabalhando com enumerações, 206–211
 estruturas, 216. Consulte também estruturas
 modificadores ref e out em parâmetros de valor, 195
 tipos nullable, 190
 usando parâmetros de valor, 186–188, 192
 usando tipos nullable, 190–192
 variáveis de tipo object referindo-se a, 199
- Toolbox**
 arrastando um controle para um formulário, 25
 exibindo ou ocultando, 23
 transformações, 656
- U**
- unboxing, 199–201
- V**
- valor null**
 atribuindo a variáveis de referência, 190
 declarando uma variável que pode armazenar, 204
 entendendo, 189
 incapacidade de atribuir a tipos-valor, 190
 verificando se variável nullable contém, 191
 valores de retorno, passando arrays como, 232
 valores infinitos, 53
 variância, interfaces genéricas e, 398–404, 405
 interfaces contravariantes, 402–404
 interfaces covariantes, 400
 interfaces invariantes, 400
variáveis, 41–43
 alocação e reivindicação de memória para, 313
 armazenando informações sobre um único item, 226
 array, 226, 247
 atribuição de nomes, 41
 declarando, 42
 declarando variáveis locais implicitamente tipadas, 62
 definidas em bloco de código, duração de, 196
 enumeração, 207, 224
 escopo, 72
 estrutura, 215, 225
 inicializando variável definida com um parâmetro de tipo, 434
 múltiplas, referindo-se ao mesmo objeto, 198
 tipo-valor, 183
 variáveis locais não atribuídas, 43
 variáveis booleanas, declarando, 93, 111
 variáveis implicitamente tipadas, 63
 variáveis locais
 memória exigida por, 196
 não atribuídas, 43
 variáveis locais não atribuídas, 43
 variável sentinela, 116
 verificação de overflow aritmético, 146, 157
 ativando e desativando no Visual Studio 2013, 147
- versão executável de um programa**, 14
- ViewModel**
 adicionando comandos ao, 678–682
 adicionando comandos NextCustomer e PreviousCustomer ao, 682–686
 adicionando informe de erros ao, 739–741
 criando, 674–678
 implementando funcionalidade de adição e edição no, 730–738
 método GoTo, 701
vinculação de dados, 707
 associando os mesmos dados a vários controles, 647
 exibindo dados usando, 662–668
 implementando para controles ComboBox de título, 673
 modificando dados usando, 668
 usando com controle ComboBox, 672

- V**
- Visual Basic
 - atribuição de nomes membros de classe, caixa e, 165
 - código gerenciado, 223
 - colchetes em declarações de array do C#, 227
 - funções, procedimentos e sub-rotinas, 67
 - funções ou sub-rotinas, semelhança dos métodos com, 65
 - métodos globais, 66
 - operadores e interoperabilidade de linguagem com C#, 507
 - Visual State Manager, adaptando layout de aplicativo
 - Windows Store com, 643–650
 - Visual Studio 2013
 - criando aplicativo de console, 38
 - criando aplicativo Windows Store para Windows 8.1, 38
 - criando aplicativo WPF para Windows 7 ou 8, 38
 - criando um aplicativo gráfico, 18–37
 - escrevendo seu primeiro programa, 8–14
 - iniciando a programação com, 3–8
 - arquivos de projeto no Solution Explorer, 7
 - configurações de desenvolvimento padrão, 4
 - criando aplicativo de console, 5
 - página Iniciar, 4
 - Microsoft Blend, 659
 - Retornando para, após depurar aplicativo Windows Store, 28
 - Simulator, 630–633, 640–643
 - Technical Preview Edition, versão padrão do Entity Framework, 714
 - templates e ferramentas para construir web services, 710
 - templates para aplicativos Windows Store, 616
 - visualização Snapped para aplicativos, 27
 - void, palavra-chave, tipo de retorno para métodos, 67
 - W**
 - web services, 710
 - criando e usando web service REST, 717–729
 - buscando dados do web service AdventureWorks, 723–729
 - criando o web service AdventureWorks, 718–722
 - inserção, atualização e exclusão de dados por meio de web service REST, 729–747
 - implementando funcionalidade de adição e edição no ViewModel, 730–738
 - informando erros e atualizando a interface do usuário, 739–742
 - integrando funcionalidade de adição e edição no formulário Customers, 742
 - testando o aplicativo Customers, 744–747
 - web services REST, 748
 - criando e usando, 717–729
 - idempotência em, 730
 - inserção, atualização e exclusão de dados por meio de, 729–747
 - Windows 7
 - caixa de diálogo Open, 117
 - exercícios neste livro, 19
 - iniciando o Visual Studio 2013, 4
 - templates para aplicativos gráficos, 18
 - Windows 8 e 8.1
 - assincronicidade no Windows 8.1, 570
 - compatibilidade com Windows Runtime (WinRT) no, 308–310
 - contratos no Windows 8.1, 689–704
 - criando aplicativo de console no Visual Studio 2013, 3
 - criando aplicativo gráfico no Visual Studio 2013 (Windows 8.1), 19
 - estruturas e compatibilidade com Windows Runtime, 223
 - executando aplicativo Windows Store no modo Debug no Windows 8.1, 28
 - exercícios neste livro, 19
 - gestos de interação com o Windows 8.1, 613
 - ícones e barras de ferramentas para Windows 8.1, exibindo, 635
 - interface de usuário estilo Windows Store (Windows 8.1), 18
 - selecionador de arquivo Open File Picker (Windows 8.1), 116
 - templates para aplicativos Windows Store (Windows 8), 18
 - Windows 8.1 executando em uma ampla variedade de dispositivos, 613
 - Windows Runtime (WinRT), 223
 - assincronicidade, 570
 - compatibilidade com, 308–310
 - métodos assíncronos e APIs do Windows Runtime, 581–584
 - WPF (Windows Presentation Foundation), objeto CommandManager, 681
 - X**
 - XAML (Extensible Application Markup Language), 19
 - arquivo App.xaml para aplicativo WPF, examinando, 34
 - controle TextBlock para um formulário, 24
 - definida, 20
 - XML, dados de web service REST, 718