

Microsoft®

Visual C#® 2010

John Sharp



eBook + exercícios
(em inglês)

Passo a Passo



COLEÇÃO MICROSOFT

HALVORSON, M.

Microsoft Visual Basic 2008 Passo a Passo

HALVORSON, M.

Microsoft Visual Basic 2010 Passo a Passo

HOTEK, M.

Microsoft SQL Server 2008 Passo a Passo

JACOBSON, MISNER & HITACHI CONSULTING

Microsoft SQL Server 2005 Analysis Services Passo a Passo

KIRIATY & COLS.

Introdução ao Windows 7 para Desenvolvedores

MISNER & HITACHI CONSULTING

Microsoft SQL Server 2005 Reporting Services Passo a Passo

SHARP, J.

Microsoft Visual C# 2008 Passo a Passo

SHEPHERD, G.

Microsoft ASP.NET 3.5 Passo a Passo

SOLID QUALITY LEARNING

Microsoft SQL Server 2005 Fundamentos de Bancos de Dados

Passo a Passo

SOLID QUALITY LEARNING

Microsoft SQL Server 2005 Técnicas Aplicadas Passo a Passo

STANEK, W.

Microsoft Exchange Server 2010: Guia de Bolso do Administrador

STANEK, W.

Microsoft SQL Server 2008: Guia de Bolso do Administrador

STANEK, W.

Microsoft Windows Server 2003: Guia de Bolso do Administrador

STANEK, W.

Microsoft Windows XP Professional: Guia de Bolso do Administrador – 2.ed

STANEK, W.

Windows 7: Guia de Bolso do Administrador

STANEK, W.

Windows Server 2008: Guia Completo

STANEK, W.

Windows Server 2008: Guia de Bolso do Administrador

STANEK, W.

Windows Vista: Guia de Bolso do Administrador

Microsoft®

Visual C#® 2010

Passo a Passo



Este é o seu guia para aprender a programar com o Microsoft Visual Studio® 2010.



Microsoft Press® é uma marca registrada da Microsoft Corporation.

O autor

John Sharp é um importante tecnólogo na Content Master, empresa de consultoria e criação técnica integrante do CM Group Ltd. Especialista em desenvolvimento de aplicativos no Microsoft .NET Framework e em outras tecnologias, John já produziu inúmeros tutoriais, publicações técnicas e apresentações sobre sistemas distribuídos, SOA e Web services, linguagem C# e questões de interoperabilidade. Ele tem contribuído para o desempenho de vários cursos para o Microsoft Training (foi coautor do primeiro curso de programação em C# para a Microsoft) e também é autor de diversos livros populares, como o *Microsoft Visual C# 2008 Passo a Passo*.



S531m Sharp, John.

Microsoft Visual C# 2010 : passo a passo / John Sharp ;
tradução: Teresa Cristina Felix de Sousa, Edson Furmankiewicz ;
revisão técnica: Daniel Antonio Callegari. – Porto Alegre :
Bookman, 2011.

780 p. : il. ; 25 cm.

ISBN 978-85-7780-849-6

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

CDU 004.413Visual C#

Microsoft

John Sharp

Microsoft®

Visual C#® 2010 Passo a Passo

Revisão técnica:

Daniel Antonio Callegari
Doutor em Ciência da Computação
Professor da PUC-RS e profissional certificado Microsoft



2011

Obra originalmente publicada sob o título
Microsoft Visual C# 2010 Step by Step, autoria de John Sharp

ISBN 9780735626706

Original English language copyright © 2010, O'Reilly Media, Inc.

Tradução em língua portuguesa © 2011 Bookman Companhia Editora Ltda., uma divisão da Artmed Editora SA. Esta tradução é vendida e publicada com permissão da O'Reilly Media, Inc., proprietária ou controladora dos direitos de publicação e venda da obra.

Portuguese language translation copyright © 2011 Bookman Companhia Editora Ltda., a Division of Artmed Editora S.A. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls of all rights to publish and sell the same.

Tradução (*Microsoft Visual C# 2008 Passo a Passo*): Edson Furmaniakiewicz

Atualização de conteúdo (*Microsoft Visual C# 2010 Passo a Passo*): Teresa Cristina Felix de Sousa

Capa: *VS Digital*, arte sobre capa original

Leitura final: Aline Grodt

Editora sênior – Bookman: Arysinha Affonso

Editora responsável por esta obra: Elisa Viali

Projeto e editoração: Techbooks

Microsoft, Microsoft Press, Excel, IntelliSense, Internet Explorer, Jscript, MS, MSDN, SQL Server, Visual Basic, Visual C#, Visual C++, Visual Studio, Win32, Windows e Windows Vista são exemplos comerciais notórios ou marcas comerciais registradas da Microsoft Corporation nos Estados Unidos e/ou em outros países. Outros nomes de produto e de empresa mencionados aqui podem ser marcas comerciais de seus respectivos proprietários.

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

Este livro expressa visões e opiniões do autor. As informações contidas neste livro são fornecidas sem qualquer garantia legal, expressa ou implícita. Os autores, a Microsoft Corporation, e seus revendedores ou distribuidores não serão responsáveis por quaisquer danos causados ou alegadamente causados direta ou indiretamente por este livro.

Reservados todos os direitos de publicação, em língua portuguesa, à

ARTMED® EDITORA S.A.

(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA 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

IMPRESSO NO BRASIL

PRINTED IN BRAZIL

Agradecimentos

Diz um ditado popular que os pintores da Forth Railway Bridge, uma grande ponte da era Vitoriana que cruza a Foz do Rio Forth, ao norte de Edinburgo, têm emprego vitalício. Segundo a lenda, são necessários muitos anos para pintá-la de uma extremidade à outra e, ao final do serviço, já é hora de começar tudo de novo. Não tenho certeza se a demora se deve às agruras do clima da Escócia ou à durabilidade da tinta aplicada, embora minha filha insista em dizer que, na verdade, os membros da Câmara Municipal de Edinburgo ainda não decidiram qual é a melhor paleta de cores para a ponte. Às vezes, acho este livro parecido com a empreitada escocesa. Quando finalmente concluo a obra, a Microsoft anuncia outra atualização excelente do Visual Studio e do C#, e meus amigos da Microsoft Press me perguntam: “quais são seus planos para a próxima edição?”. No entanto, ao contrário do trabalho na Forth Railway Bridge, escrever uma nova versão deste texto é sempre agradável, com muito mais possibilidades criativas do que tentar descobrir novas maneiras de segurar um pincel. Há sempre novidades, além de uma tecnologia inovadora para conhecer. Nesta edição, abordarei os novos recursos do C# 4.0 e do .NET Framework 4.0, extremamente úteis para construir aplicativos para hardware cada vez mais potentes. Mesmo que pareça interminável, esta atividade é sempre compensadora e gratificante.

Em um projeto como este, boa parte da satisfação vem de trabalhar com um grupo de pessoas talentosas e motivadas da Microsoft Press, com os desenvolvedores da Microsoft dedicados ao Visual Studio 2010 e com quem faz a revisão de todos os capítulos e apresenta sugestões de melhorias. Gostaria de citar principalmente Rosemary Caperton e Stephen Sagman, que se empenharam incansavelmente para manter o projeto nos trilhos; Per Blomqvist, que revisou (e corrigiu) cada capítulo; e Roger Leblanc, que teve a árdua tarefa de editar o manuscrito e converter minha prosa em um texto comprehensível. Uma menção especial para Michael Blome, que me deu acesso ao software e respondeu à minha avalanche de perguntas sobre a Task Parallel Library (TPL). Vários colaboradores da Content Master dedicaram um bom tempo revisando e testando o código dos exercícios — agradeço a Mike Sumption, Chris Cully, James Millar e Louisa Perry. Também agradeço a Jon Jagger, coautor da primeira edição deste livro, nos idos de 2001.

Por último, mas não menos importante, agradeço à minha família. Minha esposa Diana é uma maravilhosa fonte de inspiração. Ao escrever o Capítulo 28 sobre a Task Parallel Library (TPL), tive um branco e pedi a ela que me explicasse, em suas próprias palavras, os métodos de barreira. Ela me lançou um olhar de desboche e deu uma resposta que, embora anatomicamente correta se eu estivesse sendo submetido a uma cirurgia, indicava que ou eu elaborara mal a minha pergunta, ou ela interpretara mal o que eu perguntara! James já está grande e logo saberá o quanto vai ter de trabalhar se quiser manter o estilo de vida que Diana e eu gostaríamos de ter em nossa velhice. Francesca também cresceu, e parece ter aprimorado sua estratégia para conseguir tudo o que quer; basta, para isso, olhar para mim com seus olhos lânguidos e brilhantes, e sorrir.

Por fim, dá-lhe Gills!

—John Sharp

Sumário geral

Parte I Apresentando o Microsoft Visual C# e o Microsoft Visual Studio 2010

1	Bem-vindo ao C#	35
2	Trabalhando com variáveis, operadores e expressões	59
3	Escrevendo métodos e aplicando escopo	79
4	Utilizando instruções de decisão.	105
5	Utilizando atribuição composta e instruções de iteração.	123
6	Gerenciando erros e exceções	141

Parte II Entendendo a linguagem C#

7	Criando e gerenciando classes e objetos	161
8	Entendendo valores e referências	183
9	Criando tipos-valor com enumerações e estruturas	205
10	Utilizando arrays e coleções	223
11	Entendendo arrays de parâmetros	251
12	Trabalhando com herança	263
13	Criando interfaces e definindo classes abstratas	285
14	Utilizando a coleta de lixo e o gerenciamento de recursos	311

Parte III Criando componentes

15	Implementando propriedades para acessar campos	327
16	Utilizando indexadores	347

17	Interrompendo o fluxo do programa e tratando eventos	361
18	Apresentando genéricos	385
19	Enumerando coleções	413
20	Consultando dados na memória utilizando expressões de consulta	427
21	Sobrecarga de operadores	451
Parte IV Construindo Aplicativos WPF		
22	Apresentando o Windows Presentation Foundation	475
23	Obtendo a entrada do usuário	509
24	Realizando validações	541
Parte V Gerenciando dados		
25	Consultando informações em um banco de dados	567
26	Exibindo e editando dados com o Entity Framework e vinculação de dados	597
Parte VI Construindo soluções profissionais com o Visual Studio 2010		
27	Introdução à Task Parallel Library	631
28	Realizando acesso a dados em paralelo	681
29	Criando e utilizando um Web service	715
Apêndice		
	Interoperabilidade com linguagens dinâmicas	749
Índice		
		757

Sumário

Parte I Apresentando o Microsoft Visual C# e o Microsoft Visual Studio 2010

1	Bem-vindo ao C#	35
	Começando a programar com o ambiente do Visual Studio 2010	35
	Escrevendo seu primeiro programa	40
	Utilizando namespaces	46
	Criando um aplicativo gráfico	49
	Referência rápida do Capítulo 1	58
2	Trabalhando com variáveis, operadores e expressões	59
	Entendendo instruções	59
	Utilizando identificadores	60
	Identificando palavras-chave	60
	Utilizando variáveis	61
	Nomeando variáveis	62
	Declarando variáveis	62
	Trabalhando com tipos de dados primitivos	63
	Variáveis locais não atribuídas	64
	Exibindo valores de tipos de dados primitivos	64
	Utilizando operadores aritméticos	68
	Operadores e tipos	69
	Examinando operadores aritméticos	70
	Controlando a precedência	73
	Utilizando a associatividade para avaliar expressões	74
	A associatividade e o operador de atribuição	74
	Incrementando e decrementando variáveis	75
	Prefixo e sufixo	76
	Declarando variáveis locais implicitamente tipadas	76
	Referência rápida do Capítulo 2	78
3	Escrevendo métodos e aplicando escopo	79
	Criando métodos	79
	Declarando um método	80
	Retornando dados de um método	81
	Chamando métodos	83
	Especificando a sintaxe de chamada de método	83

Aplicando escopo	85
Definindo o escopo local	86
Definindo o escopo de classe.	86
Sobrecarregando métodos	87
Escrevendo métodos	88
Utilizando parâmetros opcionais e argumentos nomeados	96
Definindo parâmetros opcionais	97
Passando argumentos nomeados	98
Resolvendo ambiguidades com parâmetros opcionais e argumentos nomeados.	98
Referência rápida do Capítulo 3	104
4 Utilizando instruções de decisão.	105
Declarando variáveis booleanas.	105
Utilizando operadores booleanos	106
Entendendo operadores de igualdade e relacionais.	106
Entendendo operadores lógicos condicionais	107
Curto-circuito	108
Resumindo a precedência e a associatividade dos operadores.	108
Utilizando instruções <i>if</i> para tomar decisões	109
Entendendo a sintaxe da instrução <i>if</i>	109
Utilizando blocos para agrupar instruções.	110
Instruções <i>if</i> em cascata.	111
Utilizando instruções <i>switch</i>	116
Entendendo a sintaxe da instrução <i>switch</i>	117
Seguindo as regras da instrução <i>switch</i>	118
Referência rápida do Capítulo 4	122
5 Utilizando atribuição composta e instruções de iteração.	123
Utilizando operadores de atribuição composta	123
Escrevendo instruções <i>while</i>	124
Escrevendo instruções <i>for</i>	129
Entendendo o escopo da instrução <i>for</i>	130
Escrevendo instruções <i>do</i>	131
Referência rápida do Capítulo 5	140

6	Gerenciando erros e exceções	141
	Lidando com erros	141
	Testando o código e capturando as exceções	142
	Exceções não tratadas	143
	Utilizando múltiplas rotinas de tratamento <i>catch</i>	144
	Capturando múltiplas exceções	145
	Utilizando aritmética verificada e não verificada de números inteiros	150
	Escrevendo instruções verificadas	150
	Escrevendo expressões verificadas	151
	Lançando exceções	153
	Utilizando um bloco <i>finally</i>	156
	Referência rápida do Capítulo 6	158
 Parte II Entendendo a linguagem C#		
7	Criando e gerenciando classes e objetos	161
	Entendendo a classificação	161
	O objetivo do encapsulamento	162
	Definindo e utilizando uma classe	162
	Controlando a acessibilidade	164
	Trabalhando com construtores	165
	Sobrecregendo construtores	166
	Entendendo dados e métodos <i>static</i>	174
	Criando um campo compartilhado	175
	Criando um campo <i>static</i> utilizando a palavra-chave <i>const</i>	176
	Classes estáticas	176
	Classes anônimas	179
	Referência rápida do Capítulo 7	181
8	Entendendo valores e referências	183
	Copiando variáveis de tipo-valor e classes	183
	Entendendo valores nulos e tipos nullable	188
	Utilizando tipos nullable	189
	Entendendo as propriedades dos tipos nullable	190
	Utilizando parâmetros <i>ref</i> e <i>out</i>	191
	Criando parâmetros <i>ref</i>	191
	Criando parâmetros <i>out</i>	192

Como a memória do computador é organizada	194
Utilizando a pilha e o heap	196
A classe <i>System.Object</i>	197
Boxing	197
Unboxing	198
Casting de dados seguro	200
O operador <i>is</i>	200
O operador <i>as</i>	200
Referência rápida do Capítulo 8	203
9 Criando tipos-valor com enumerações e estruturas	205
Trabalhando com enumerações	205
Declarando uma enumeração	205
Utilizando uma enumeração	206
Escolhendo valores literais de enumeração	207
Escolhendo o tipo subjacente de uma enumeração	208
Trabalhando com estruturas	210
Declarando uma estrutura	212
Entendendo as diferenças entre estrutura e classe	213
Declarando variáveis de estrutura	214
Entendendo a inicialização de estruturas	215
Copiando variáveis de estrutura	219
Referência rápida do Capítulo 9	222
10 Utilizando arrays e coleções	223
O que é um array?	223
Declarando variáveis de array	223
Criando uma instância de array	224
Inicializando variáveis de array	225
Criando um array implicitamente tipado	226
Acessando um elemento individual de um array	227
Iterando por um array	227
Copiando arrays	229
Utilizando arrays multidimensionais	230
Utilizando arrays para jogar cartas	231
O que são classes de coleção?	238
A classe de coleção <i>ArrayList</i>	240
A classe de coleção <i>Queue</i>	242
A classe de coleção <i>Stack</i>	242
A classe de coleção <i>Hashtable</i>	243

A classe de coleção <i>SortedList</i>	245
Utilizando inicializadores de coleção	246
Comparando arrays e coleções	246
Utilizando classes de coleção para jogar cartas	246
Referência rápida do Capítulo 10	250
11 Entendendo arrays de parâmetros	251
Utilizando argumentos de arrays	252
Declarando um array <i>params</i>	253
Utilizando <i>params object[]</i>	255
Utilizando um array <i>params</i>	256
Comparando arrays de parâmetros e parâmetros opcionais	258
Referência rápida do Capítulo 11	261
12 Trabalhando com herança	263
O que é herança?	263
Utilizando a herança	264
Chamando construtores da classe base	266
Atribuindo classes	267
Declarando métodos <i>new</i>	269
Declarando métodos virtuais	270
Declarando métodos <i>override</i>	271
Entendendo o acesso <i>protected</i>	274
Entendendo métodos de extensão	279
Referência rápida do Capítulo 12	283
13 Criando interfaces e definindo classes abstratas	285
Entendendo interfaces	285
Definindo uma interface	286
Implementando uma interface	287
Referenciando uma classe por meio de sua interface	288
Trabalhando com múltiplas interfaces	289
Implementando explicitamente uma interface	289
Restrições das interfaces	291
Definindo e utilizando interfaces	291
Classes abstratas	301
Métodos abstratos	302
Classes seladas	303
Métodos selados	303
Implementando e utilizando uma classe abstrata	304
Referência rápida do Capítulo 13	309

14 Utilizando a coleta de lixo e o gerenciamento de recursos	311
O tempo de vida de um objeto	311
Escrevendo destrutores	312
Por que utilizar o coletor de lixo?	314
Como funciona o coletor de lixo?	315
Recomendações.	316
Gerenciamento de recursos	316
Métodos de descarte	317
Descarte seguro quanto a exceções	317
A instrução <i>using</i>	318
Chamando o método <i>Dispose</i> a partir de um destrutor.	320
Implementando descarte seguro quanto a exceções	321
Referência rápida do Capítulo 14	324

Parte III Criando componentes

15 Implementando propriedades para acessar campos	327
Implementando encapsulamento com métodos	328
O que são propriedades?	329
Utilizando propriedades	331
Propriedades somente-leitura	332
Propriedades somente-gravação	332
Acessibilidade de propriedades	333
Entendendo as restrições de uma propriedade	334
Declarando propriedades de interface	336
Utilizando propriedades em um aplicativo Windows.	337
Gerando propriedades automáticas	338
Inicializando objetos com propriedades	340
Referência rápida do Capítulo 15	345
16 Utilizando indexadores	347
O que é um indexador?	347
Um exemplo que não utiliza indexadores	347
O mesmo exemplo utilizando indexadores	349
Entendendo os métodos de acesso do indexador	351
Comparando indexadores e arrays.	352
Indexadores em interfaces	354
Utilizando indexadores em um aplicativo Windows.	355
Referência rápida do Capítulo 16	360

17	Interrompendo o fluxo do programa e tratando eventos	361
	Declarando e utilizando delegates	361
	O cenário da fábrica automatizada	362
	Implementando a fábrica sem utilizar delegates	362
	Implementando a fábrica utilizando um delegate	363
	Utilizando delegates	365
	Expressões lambda e delegates	370
	Criando um método adaptador	371
	Utilizando uma expressão lambda como um adaptador	371
	A forma das expressões lambda	372
	Ativando notificações por meio de eventos	374
	Declarando um evento	374
	Fazendo a inscrição em um evento	375
	Entendendo eventos de interface WPF	377
	Utilizando eventos	378
	Referência rápida do Capítulo 17	383
18	Apresentando genéricos	385
	O problema com <i>objects</i>	385
	A solução dos genéricos	387
	Classes genéricas <i>versus</i> generalizadas	389
	Genéricos e restrições	390
	Criando uma classe genérica	390
	A teoria das árvores binárias	390
	Construindo uma classe de árvore binária com genéricos	393
	Criando um método genérico	402
	Definindo um método genérico para criar uma árvore binária	403
	Variância e interfaces genéricas	405
	Interfaces covariantes	407
	Interfaces contravariantes	409
	Referência rápida do Capítulo 18	412
19	Enumerando coleções	413
	Enumerando os elementos em uma coleção	413
	Implementando manualmente um enumerador	415
	Implementando a interface <i>IEnumerable</i>	419
	Implemente um enumerador utilizando um iterador	421
	Um iterador simples	421
	Definindo um enumerador para a classe <i>Tree<TItem></i> por meio de um iterador	423
	Referência rápida do Capítulo 19	426

20 Consultando dados na memória utilizando expressões de consulta	427
O que é a Language Integrated Query?	427
Utilizando a LINQ em um aplicativo C#	428
Selecionando dados	430
Filtrando dados	432
Ordenando, agrupando e agregando dados	433
Junção de dados	436
Utilizando operadores de consulta	437
Consultando dados em objetos <i>Tree<TItem></i>	439
LINQ e avaliação postergada	444
Referência rápida do Capítulo 20	449
21 Sobrecarga de operadores	451
Entendendo os operadores	451
Restrições dos operadores	452
Operadores sobrecarregados	452
Criando operadores simétricos	454
Entendendo a avaliação da atribuição composta	456
Declarando operadores de incremento e decremento	457
Comparando operadores em estruturas e classes	458
Definindo pares de operadores	458
Implementando operadores	459
Entendendo os operadores de conversão	466
Fornecendo conversões predefinidas	466
Implementando operadores de conversão definidos pelo usuário	467
Criando operadores simétricos, uma retomada do assunto	468
Escrevendo operadores de conversão	469
Referência rápida do Capítulo 21	472
Parte IV Construindo Aplicativos WPF	
22 Apresentando o Windows Presentation Foundation	475
Criando um aplicativo WPF	475
Construindo o aplicativo WPF	476
Adicionando controles ao formulário	490
Utilizando controles WPF	490
Alterando as propriedades dinamicamente	498

Tratando eventos em um formulário WPF	502
Processando eventos no Windows Forms	503
Referência rápida do Capítulo 22	508
23 Obtendo a entrada do usuário	509
Diretrizes e estilos de menu.....	509
Menus e eventos de menu.....	510
Criando um menu	510
Tratando eventos de menu	516
Menus de atalho	523
Criando menus de atalho.....	523
Caixas de diálogo comuns do Windows.....	527
Utilizando a classe <i>SaveFileDialog</i>	527
Aprimorando a capacidade de resposta de um aplicativo WPF	530
Referência rápida do Capítulo 23	540
24 Realizando validações	541
Validando os dados.....	541
Estratégias para validar a entrada do usuário	541
Um exemplo – order tickets (solicitar ingressos) para eventos	542
Realizando a validação com vinculação de dados	543
Alterando o ponto em que a validação ocorre.....	559
Referência rápida do Capítulo 24	564
Parte V Gerenciando dados	
25 Consultando informações em um banco de dados	567
Consultando um banco de dados por meio do ADO.NET.....	567
O banco de dados Northwind	568
Criando o banco de dados.....	568
Utilizando ADO.NET para consultar informações de pedidos.....	570
Consultando um banco de dados usando LINQ to SQL	581
Definindo uma classe de entidade.....	581
Criando e executando uma consulta LINQ to SQL	583
Busca adiada e busca imediata	585
Fazendo junção de tabelas e criando relações	586
Busca adiada e imediata, uma retomada do assunto.....	590
Definindo uma classe <i>DataContext</i> personalizada	591
Utilizando a LINQ to SQL para consultar informações de pedidos ..	591
Referência rápida do Capítulo 25	596

26	Exibindo e editando dados com o Entity Framework e vinculação de dados	597
	Utilizando vinculação de dados com Entity Framework	598
	Utilizando vinculação de dados para modificar dados.	615
	Atualizando dados existentes	615
	Tratando atualizações conflitantes.	616
	Adicionando e excluindo dados.	619
	Referência rápida do Capítulo 26	628
Parte VI	Construindo soluções profissionais com o Visual Studio 2010	
27	Introdução à Task Parallel Library	631
	Por que fazer multitarefa por meio de processamento paralelo.	632
	O surgimento do processador multinúcleo	633
	Implementando multitarefa em um aplicativo	634
	Tarefas, threads e <i>ThreadPool</i>	635
	Criando, executando e controlando tarefas.	636
	Utilizando a classe Task para implementar paralelismo	640
	Abstraindo tarefas com a classe Parallel	649
	Retornando um valor de uma tarefa	656
	Utilizando tarefas e threads	
	de interface do usuário em conjunto	660
	Cancelando tarefas e tratando exceções	664
	Mecânica do cancelamento cooperativo	665
	Tratando exceções de tarefas com a classe <i>AggregateException</i>	673
	Utilizando continuações com tarefas canceladas e com falhas	677
	Referência rápida do Capítulo 27	678
28	Realizando acesso a dados em paralelo	681
	Utilizando a PLINQ para parallelizar o acesso declarativo a dados	682
	Utilizando a PLINQ para melhorar o desempenho ao iterar sobre uma coleção	682
	Especificando opções para uma consulta PLINQ	687
	Cancelando uma consulta PLINQ	688

Sincronizando acessos imperativos e simultâneos a dados	688
Bloqueando dados	691
Primitivas de sincronização na Task Parallel Library	693
Primitivas de cancelamento e sincronização	700
Classes de coleção concorrentes	700
Utilizando uma coleção concorrente e um bloqueio para implementar o acesso a dados thread-safe	702
Referência rápida do Capítulo 28	713
29 Criando e utilizando um Web service	715
O que é um Web service?	716
O papel do Windows Communication Foundation	716
Arquiteturas de Web Service	716
Web Services SOAP	717
Web Services REST	719
Construindo Web services	720
Criando o Web Service SOAP ProductInformation	721
Web Services SOAP, clientes e proxies	729
Consumindo o Web Service SOAP ProductInformation	730
Criando o Web service REST ProductDetails	736
Consumindo o Web service REST ProductDetails	743
Referência rápida do Capítulo 29	748
Apêndice	
Interoperabilidade com linguagens dinâmicas	749
O que é o Dynamic Language Runtime?	750
A palavra-chave <i>dynamic</i>	751
Exemplo: IronPython	752
Exemplo: IronRuby	754
Resumo	756
Índice	757

Introdução

O 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# com o Visual Studio 2005 introduziu vários recursos novos importantes na linguagem, como iteradores genéricos 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). A incorporação mais recente da linguagem, o C# 4.0, oferece aprimoramentos que melhoram sua interoperabilidade com outras linguagens e tecnologias. Esses recursos abrangem o suporte para argumentos nomeados e opcionais; o tipo *dynamic* (dinâmico), o qual indica que o tempo de execução da linguagem deve implementar a ligação tardia para um objeto; e a variância, que resolve algumas questões relacionadas ao modo como as interfaces genéricas são definidas. O C# 4.0 tira proveito da versão mais recente do .NET Framework, também na versão 4.0. As inclusões no .NET Framework mais importantes são as classes e os tipos que constituem a Task Parallel Library (TPL). Com a TPL, agora você pode construir, de modo rápido e fácil, aplicativos altamente escalonáveis para processadores multinúcleo. O suporte para Web services e Windows Communication Foundation também foi ampliado; atualmente, é possível construir serviços que seguem o modelo REST, assim como o esquema SOAP mais tradicional.

O ambiente de desenvolvimento do Microsoft Visual Studio 2010 facilita o uso de todos esses recursos poderosos, e os diversos novos assistentes e melhorias incluídos na versão 2010 podem aumentar consideravelmente a sua produtividade como desenvolvedor.

Para quem é este livro?

Este livro é destinado a desenvolvedores que desejam aprender os conceitos básicos da programação com o C# utilizando o Visual Studio 2010 e o .NET Framework versão 4.0. Nele, você aprenderá os recursos da linguagem C# e os utilizará para criar aplicativos que são executados no sistema operacional Microsoft Windows. Ao concluir esta obra, você terá um entendimento completo do C# e o terá utilizado para produzir aplicativos do Windows Presentation Foundation, acessar bancos de dados do Microsoft SQL Server pelo ADO.NET e pelo LINQ, construir aplicativos ágeis e escalonáveis por meio da TPL, e criar Web services REST e SOAP no WCF.

Encontrando 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">Instale os arquivos de exercícios conforme descrito na próxima seção, "Exemplos de código".Siga os capítulos nas Partes I, II e III sequencialmente.Complete as Partes IV, V e VI de acordo com seu nível de experiência e interesse.
Familiarizado com linguagens de programação procedurais como C, mas é iniciante em C#	<ol style="list-style-type: none">Instale os arquivos de exercícios conforme descrito na próxima seção, "Exemplos de código". Folheie os cinco primeiros capítulos para obter uma visão geral do C# e Visual Studio 2010 e, em seguida, concentre-se nos Capítulos 6 a 21.Complete as Partes IV, V e VI conforme seu nível de experiência e interesse.
Migrando de uma linguagem orientada a objetos como C++ ou Java	<ol style="list-style-type: none">Instale os arquivos de exercícios conforme descrito na próxima seção, "Exemplos de código".Folheie os sete primeiros capítulos para obter uma visão geral do C# e Visual Studio 2010 e, em seguida, concentre-se nos Capítulos 8 a 21.Leia as Partes IV e V para obter informações sobre como criar aplicativos Windows e utilizar um banco de dados.Leia a Parte VI para obter informações sobre como criar aplicativos escalonáveis e Web services.

Se você está	Siga estes passos
Migrando do Visual Basic 6	<ol style="list-style-type: none"> 1. Instale os arquivos de exercícios conforme descrito na próxima seção, “Exemplos de código”. 2. Siga os capítulos nas Partes I, II e III sequencialmente. 3. Leia a Parte IV para obter informações sobre como criar aplicativos Windows. 4. Leia a Parte V para obter informações sobre como acessar um banco de dados. 5. Leia a Parte IV para obter informações sobre como criar aplicativos escalonáveis e Web services. 6. Leia as seções de Referência rápida no final dos capítulos para obter informações sobre questões específicas do C# e construções do Visual Studio 2010.
Consultando o livro depois de fazer os exercícios	<ol style="list-style-type: none"> 1. Utilize o índice ou o sumário para localizar as informações sobre assuntos específicos. 2. Leia as seções de Referência rápida no final de cada capítulo para encontrar uma revisão sucinta da sintaxe e das técnicas apresentadas no capítulo.

Convenções e recursos deste livro

Este livro usa algumas convenções para tornar as informações legíveis e fáceis de compreender. Antes de começar, leia a lista a seguir, que explica as convenções que você verá ao longo da obra e indica recursos úteis que você talvez queira utilizar.

Convenções

- Cada exercício é uma série de tarefas e cada tarefa é apresentada como uma sequência de etapas numeradas (1, 2 e assim por diante). Um marcador redondo (•) indica um exercício que tem apenas uma etapa.
- As notas marcadas como “dica” fornecem informações adicionais ou métodos alternativos para completar uma etapa com sucesso.
- As notas marcadas como “importante” alertam para informações que precisam ser verificadas antes de continuar.
- Textos que você deve digitar 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.

Outros recursos

- Quadros ao longo do livro fornecem informações mais abrangentes sobre o exercício. Os quadros podem conter informações básicas, dicas de projeto ou recursos relacionados ao que está sendo discutido.
- Cada capítulo termina com uma seção chamada Referência rápida, que contém lembretes sobre como executar as tarefas aprendidas no capítulo.

Software de pré-lançamento

Este livro foi escrito e testado com o Visual Studio 2010 Beta 2. Revisamos e testamos nossos exemplos com a última versão do software. Entretanto, é possível que você detecte pequenas diferenças entre a sua versão e os exemplos, texto e capturas de tela deste livro.

Requisitos de hardware e software

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

- Microsoft Windows 7 Home Premium, Windows 7 Professional, Windows 7 Enterprise ou Windows 7 Ultimate Edition. Os exercícios também funcionarão no Microsoft Windows Vista com o Service Pack 2 ou posterior.
- Microsoft Visual Studio 2010 Standard, Visual Studio 2010 Professional ou Microsoft Visual C# 2010 Express e Microsoft Visual Web Developer 2010 Express.
- Microsoft SQL Server 2008 Express (fornecido com todas as edições do Visual Studio 2010, Visual C# 2010 Express e Visual Web Developer 2010 Express).
- Processador de 1,6 GHz ou mais rápido. Os Capítulos 27 e 28 exigem um processador dual-core ou superior.
- 1 GB de RAM física disponível para um processador X32, 2 GB para um processador X64.
- Monitor de vídeo (com resolução de 1024 × 768 ou mais alta) com pelo menos 256 cores.
- Unidade de CD-ROM ou DVD-ROM.
- Mouse Microsoft ou dispositivo indicador compatível.

Você também precisará ter acesso de administrador ao seu computador para configurar o SQL Server 2008 Express Edition.

Exemplos de código

O CD que acompanha este livro contém exemplos de código que você deve usar para realizar os exercícios. Assim, você não desperdiçará tempo para criar arquivos que não são relevantes à execução do exercício. Os arquivos e as instruções passo a passo também permitem aprender exercitando, de um modo fácil e eficaz, adquirindo e lembrando novas habilidades.

Instalando os exemplos de código

Siga estes passos para instalar os exemplos de código e o software necessário no computador a fim de usá-los com os exercícios.

1. Remova o CD localizado no final do livro e o insira na unidade de CD-ROM.

Nota Um contrato de licença para o usuário final abrirá automaticamente. Se esse texto não aparecer, abra *Meu Computador* na área de trabalho ou o menu *Iniciar*, dê um clique duplo no ícone da unidade de CD-ROM e, em seguida, dê um clique duplo em *StartCD.exe*.

2. Leia o contrato de licença para o usuário final. Se concordar com os termos, selecione a opção para aceitar e clique em *Next*.

Um menu com opções relacionadas ao livro aparece.

3. Clique em *Instalar Exemplos de Código*.

4. Siga as instruções que aparecem. Os exemplos de código são instalados no seu computador no seguinte local:

Documentos\Microsoft Press\Visual CSharp Step by Step

Utilizando 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 2010 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 projetos concluídos são armazenados em pastas com o sufixo “- Completed”.

Projeto	Descrição
Capítulo 1	
TextHello	Guia você passo a passo ao longo do processo de criação de um programa simples exibindo uma saudação baseada em texto.
WPFHello	Exibe a saudação em uma janela, utilizando o Windows Presentation Foundation.
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 2010.
DailyRate usando parâmetros opcionais	Mostra como definir um método que aceita parâmetros opcionais e como chamá-lo por meio de argumentos nomeados.
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.

Projeto	Descrição
Capítulo 6	
MathsOperators	Revisita o projeto MathsOperators do Capítulo 2, "Trabalhando com variáveis, operadores e expressões" 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 e demonstra como utilizar as palavras-chave <i>ref</i> e <i>out</i> .
Capítulo 9	
StructsAndEnums	Define um tipo de estrutura para representar uma data de calendário.
Capítulo 10	
Cards Using Arrays	Mostra como utilizar arrays para modelar mãos de cartas em um jogo de cartas.
Cards Using Collections	Ilustra como reestruturar o programa de jogo de cartas para utilizar coleções em vez de arrays.
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> .
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.

(Continua)

Projeto	Descrição
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	Estende o projeto Drawing Using Interfaces para fatorar a funcionalidade comum de objetos de forma em classes.
Capítulo 14	
UsingStatement	Revê uma parte do código do Capítulo 5, "Utilizando atribuição composta e instruções de iteração", revelando que o código não é seguro quanto a exceções e mostrando como torná-lo seguro com uma instrução <i>using</i> .
Capítulo 15	
WindowProperties	Apresenta um aplicativo Windows simples que utiliza diversas propriedades para exibir o tamanho da sua janela principal. As atualizações são exibidas automaticamente enquanto o usuário redimensiona a janela.
AutomaticProperties	Mostra como criar propriedades automáticas para uma classe e as utiliza 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	
Clock Using Delegates	Exibe um Relógio Internacional (World Clock) que indica a hora local, assim como as horas em Londres, Nova York e Tóquio. O aplicativo utiliza delegates para iniciar e interromper as exibições do relógio.
Clock Using Event	Esta versão do aplicativo World Clock usa eventos para iniciar e parar as exibições do relógio.
Capítulo 18	
BinaryTree	Mostra como empregar genéricos para criar uma estrutura segura para tipos 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.
BinaryTreeTest	Agente de teste que cria instâncias do tipo <i>Tree</i> definido no projeto BinaryTree.

Projeto	Descrição
Capítulo 19	
BinaryTree	Mostra como implementar a interface genérica <code>IEnumerator<T></code> para criar um enumerador para a classe genérica <code>Tree</code> .
IteratorBinaryTree	Utiliza um iterador para gerar um enumerador para a classe genérica <code>Tree</code> .
EnumeratorTest	Agente de teste que testa o enumerador e o iterador da classe <code>Tree</code> .
Capítulo 20	
QueryBinaryTree	Mostra como utilizar consultas LINQ para recuperar dados de um objeto de árvore binária.
Capítulo 21	
ComplexNumbers	Define um novo tipo que modela números complexos e implementa operadores comuns para esse tipo.
Capítulo 22	
BellRingers	Aplicativo Windows Presentation Foundation que demonstra como definir estilos e utilizar controles WPF básicos.
Capítulo 23	
BellRingers	Extensão do aplicativo criado no Capítulo 22, "Apresentando o Windows Presentation Foundation", mas com menus suspensos e menus pop-up adicionados à interface do usuário.
Capítulo 24	
OrderTickets	Demonstra como implementar regras de negócio para validar a entrada do usuário em um aplicativo WPF utilizando, por exemplo, informações de clientes.
Capítulo 25	
ReportOrders	Mostra como acessar um banco de dados utilizando código ADO.NET. O aplicativo recupera as informações da tabela Orders no banco de dados Northwind.
LINQOrders	Ilustra como utilizar o LINQ para SQL, para acessar um banco de dados e recuperar informações da tabela Orders no banco de dados Northwind.

(Continua)

Projeto	Descrição
Capítulo 26	
Suppliers	Demonstra como utilizar a vinculação de dados com um aplicativo WPF para exibir e formatar dados recuperados de um banco de dados em controles em um formulário WPF. O aplicativo também permite ao usuário modificar informações na tabela Products no banco de dados Northwind.
Capítulo 27	
GraphDemo	Gera e exibe um gráfico complexo em uma forma WPF. Utiliza um único thread para efetuar os cálculos.
GraphDemo Using Tasks	Versão do projeto GraphDemo que cria várias tarefas para efetuar os cálculos do gráfico simultaneamente.
GraphDemo Using Tasks that Return Results	Versão estendida do projeto GraphDemo Using Tasks que demonstra como retornar dados de uma tarefa.
GraphDemo Using the Parallel Class	Versão do projeto GraphDemo que usa a classe Parallel para abstrair o processo de criação e gerenciamento de tarefas.
GraphDemo Canceling Tasks	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 Parallel para criar e executar tarefas.
Capítulo 28	
CalculatePI	Utiliza um algoritmo de amostragem estatística para calcular uma aproximação de PI. Usa tarefas paralelas.
PLINQ	Apresenta alguns exemplos de como utilizar o PLINQ para consultar dados por meio de tarefas paralelas.
Capítulo 29	
ProductInformationService	Implementa um Web service SOAP construído por meio do WCF. O Web service expõe um método que retorna informações de preços de produtos do banco de dados Northwind.
ProductDetailsService	Implementa um Web service REST construído por meio do WCF. O Web service fornece um método que retorna os detalhes de um produto especificado do banco de dados Northwind.
ProductDetailsContract	Contém os contratos de serviços e dados implementados pelo Web service ProductDetailsService.
ProductClient	Ensina a criar um aplicativo WPF que consome um Web service, e mostra como chamar os métodos Web nos Web services ProductInformationService e ProductDetailsService.

Desinstalando os exemplos de código

Siga estes passos para remover os exemplos de código do computador.

1. No *Painel de Controle*, em *Programas e Recursos*, clique em *Desinstalar um programa*.
2. A partir da lista programas atualmente instalados, selecione Microsoft Visual Basic 2010 Passo a Passo.
3. Clique em *Desinstalar*.
4. Siga as instruções que aparecem para remover os exemplos de código.

Conteúdo adicional online

Materiais novos ou atualizados que complementam este livro serão publicados no site Microsoft Press Online Developer Tools. O tipo de material inclui atualizações no conteúdo do livro, artigos, links para conteúdo suplementar, errata, exemplos de capítulo e muito mais. Este site está disponível em www.microsoft.com/learning/books/online/developer, e é atualizado periodicamente.

Suporte para este livro*

Todo esforço foi feito para garantir a precisão deste livro e do conteúdo do CD que o acompanha. À medida que correções ou alterações forem detectadas, elas serão adicionadas a um artigo do Microsoft Knowledge Base.

A Microsoft Press fornece suporte aos seus livros e CDs neste site:

<http://www.microsoft.com/learning/support/books/>.

Perguntas e comentários

Se você tem comentários, perguntas ou ideias referentes ao livro ou ao CD, ou perguntas que não foram respondidas ao visitar os sites acima, envie-as por e-mail (em inglês) para a Microsoft Press:

mspininput@microsoft.com

Suporte a produtos Microsoft não é oferecido pelo endereço citado acima.

* N. de E.: Comentários sobre a edição brasileira desta obra podem ser encaminhados para secretariaeditorial@grupoaeditoras.com.br.

Parte I

Apresentando o Microsoft Visual C# e o Microsoft Visual Studio 2010

Capítulo 1: Bem-vindo ao C#	35
Capítulo 2: Trabalhando com variáveis, operadores e expressões.....	59
Capítulo 3: Escrevendo métodos e aplicando escopo	79
Capítulo 4: Utilizando instruções de decisão	105
Capítulo 5: Utilizando atribuição composta e instruções de iteração ...	123
Capítulo 6: Gerenciando erros e exceções	141

Capítulo 1

Bem-vindo ao C#

Neste capítulo, você vai aprender a:

- Utilizar o ambiente de programação do Microsoft Visual Studio 2010.
- Criar um aplicativo console em C#.
- Explicar o objetivo dos namespaces.
- Criar um aplicativo gráfico simples em C#.

O Microsoft Visual C# é a mais poderosa linguagem orientada para componentes da Microsoft. O C# desempenha um papel importante na arquitetura do Microsoft .NET Framework, sendo comparado, algumas vezes, à função que o C desempenhou no desenvolvimento do UNIX. Se você já conhece uma linguagem como C, C++ ou Java, notará que a sintaxe do C# é, para a sua tranquilidade, muito familiar. Se está acostumado a programar em outras linguagens, deve conseguir assimilar rapidamente a sintaxe e o modo de trabalhar no C#; basta aprender a colocar as chaves e os ponto e vírgulas no lugar. Este é o livro certo para ajudá-lo!

Na Parte I, você aprenderá os fundamentos do C#. Descobrirá como declarar variáveis e como utilizar operadores aritméticos como o sinal de adição (+) e o sinal de subtração (-) para manipular os valores em variáveis. Verá como escrever métodos e passar argumentos para eles. Além disso, aprenderá a utilizar as instruções de seleção como *if* e as instruções de iteração como *while*. Por fim, entenderá como o C# utiliza as exceções para manipular os erros de maneira simples e fácil. Esses tópicos são uma introdução ao C# e, a partir dela, você vai progredir para recursos mais avançados, da Parte II até a Parte VI.

Começando a programar com o ambiente do Visual Studio 2010

O Visual Studio 2010 é um ambiente de programação rico em recursos que contém a funcionalidade necessária para criar projetos grandes ou pequenos em C#. Você pode inclusive construir projetos que combinem módulos de diferentes linguagens, como C++, Visual Basic e F#. No primeiro exercício, você irá abrir o ambiente de programação do Visual Studio 2010 e aprender a criar um aplicativo de console.



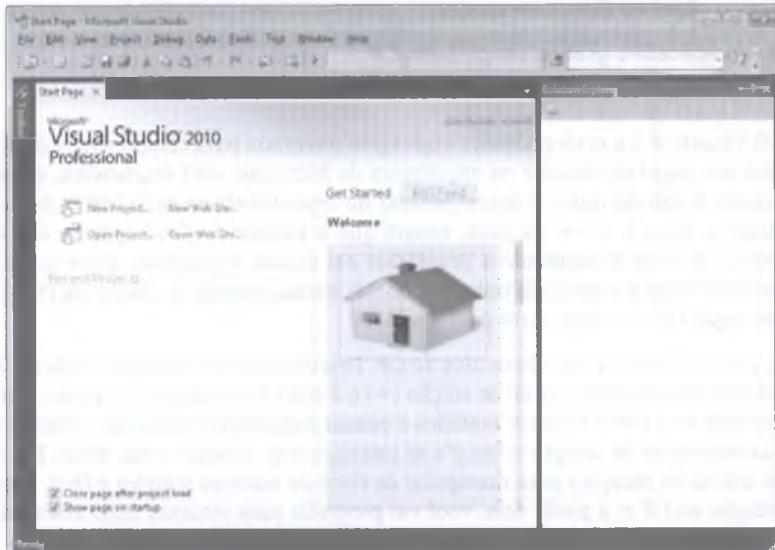
Nota Um aplicativo de console é executado em uma janela de prompt de comando, em vez de fornecer uma interface gráfica com o usuário.

Crie um aplicativo de console no Visual Studio 2010

- Se você estiver utilizando o Visual Studio 2010 Standard ou o Visual Studio 2010 Professional, siga estes passos para iniciar o Visual Studio 2010:

1. No Microsoft Windows, clique no botão *Iniciar*, aponte para *Todos os Programas* e, em seguida, aponte para o grupo de programas *Microsoft Visual Studio 2010*.
2. No grupo de programas Microsoft Visual Studio 2010, clique em *Microsoft Visual Studio 2010*.

O Visual Studio 2010 inicia, como mostrado a seguir:



Nota Se você estiver usando o Visual Studio 2010 pela primeira vez, talvez apareça uma caixa de diálogo solicitando que você escolha as configurações de ambiente de desenvolvimento padrão. O Visual Studio 2010 pode ser personalizado de acordo com a sua linguagem de desenvolvimento preferida. Todas as caixas de diálogo e ferramentas do ambiente de desenvolvimento integrado (*Integrated Development Environment – IDE*) terão o conjunto de seleções padrão referente à linguagem que você escolher. Selecione *Visual C# Development Settings* na lista e clique no botão *Start Visual Studio*. Após alguns instantes, o IDE do Visual Studio 2010 aparece.

- Se você estiver utilizando o Visual C# 2010 Express, na barra de tarefas Microsoft Windows, clique no botão *Iniciar*, aponte para *Todos os Programas* e clique em *Microsoft Visual C# 2010 Express*.

O Visual C# 2010 Express inicia, como mostrado a seguir:

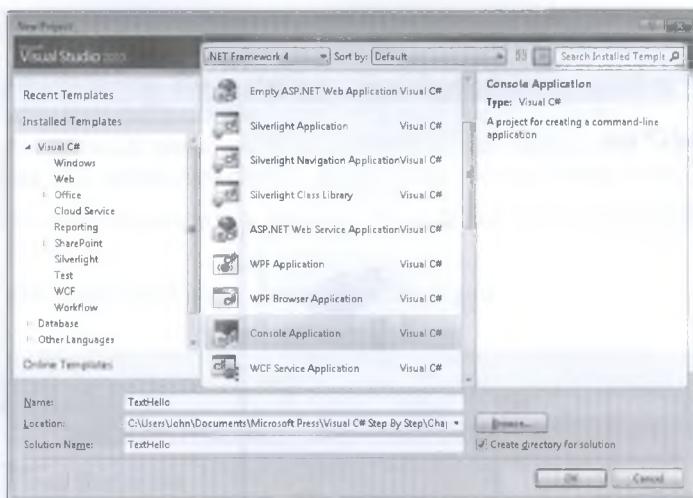


Nota Na primeira execução do Visual C# 2010, será exibida uma caixa de diálogo solicitando que você escolha as configurações padrão do ambiente de desenvolvimento. Selecione *Expert Settings* na lista e clique no botão *Start Visual Studio*. Após alguns instantes, o IDE do Visual C# 2010 é exibido.

Nota Para não ser repetitivo, eu apenas escrevo “Inicie o Visual Studio” quando você precisa abrir o Visual Studio 2010 Standard, o Visual Studio 2010 Professional ou o Visual C# 2010 Express. Além disso, a menos que dito explicitamente, todas as referências ao Visual Studio 2010 se aplicam ao Visual Studio 2010 Standard, ao Visual Studio 2010 Professional e ao Visual C# 2010 Express.

- Se você estiver utilizando o Visual Studio 2010 Standard ou o Visual Studio 2010 Professional, siga estes passos para criar um novo aplicativo de console.
 1. No menu *File*, aponte para *New* e clique em *Project*.

A caixa de diálogo *New Project* 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.
 2. No painel da esquerda, em *Installed Templates*, clique em *Visual C#*. No painel central, verifique se a caixa de combinação posicionada no início do painel exibe o texto *.NET Framework 4.0* e depois clique no ícone *Console Application*. Talvez seja necessário fazer uma rolagem no painel central para visualizar esse ícone.



3. No campo *Name*, se você estiver usando o Windows Vista, digite **C:\Users\SeuNome\Documentos\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. Se estiver utilizando o Windows 7, digite **C:\Users\SeuNome\Meus Documentos\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. Substitua o texto *SeuNome* nesses caminhos pelo seu nome de usuário do Windows.



Nota Para economizar espaço no livro, escreverei simplesmente “C:\Users\SeuNome\Documentos” ou “C:\Documents and Settings\SeuNome\Meus Documentos” para me referir à sua pasta Documentos.



Dica Se a pasta que você especificar não existir, o Visual Studio 2010 criará uma nova para você.

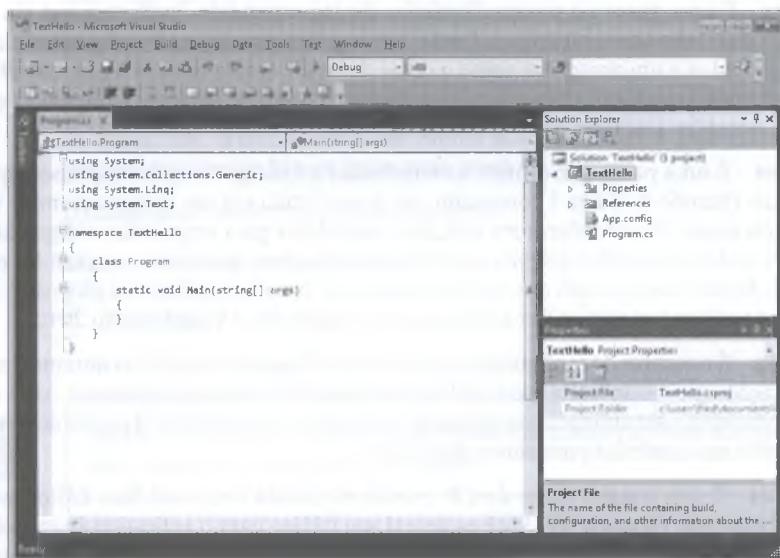
4. No campo *Name*, digite **TextHello**.
 5. Certifique-se de que a caixa de seleção *Create directory for solution* está selecionada e clique em **OK**.
- Se você estiver utilizando o Visual C# 2010 Express, siga estes passos para criar um novo aplicativo de console.
1. No menu *File*, clique em *New Project*.
 2. Na caixa de diálogo *New Project*, no painel central, clique no ícone *Console Application*.
 3. No campo *Name*, digite **TextHello**.

- Clique em *OK*.

Por padrão, o Visual C# 2010 Express salva as soluções na pasta C:\User\SeuNome\AppData\Local\Temporary Projects. Ao salvar uma solução, você pode especificar outro local.

- No menu *File*, clique em *Save TextHello As*.
- Na caixa de diálogo *Save Project*, no campo *Location*, informe a pasta **Microsoft Press\Visual CSharp Step By Step\Chapter 1**, na pasta Documentos.
- Clique em *Save*.

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



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 e oferece botões de atalho para executar os comandos utilizados com mais frequência.

O painel *Code and Text Editor*, que ocupa a parte principal do IDE, exibe o conteúdo dos arquivos-fonte. Em um projeto multiarquivo, quando você edita mais de um arquivo, 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* (no lado direito da caixa de diálogo) 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 2010 como parte do seu projeto:

- **Solution 'TextHello'** É o arquivo de solução de nível superior, e há apenas um por aplicativo. Se utilizar o Windows Explorer para examinar a pasta Documentos\Microsoft\VisualCSharp Step By Step\Chapter 1\TextHello, você verá que o nome real desse arquivo é TextHello.sln. Cada arquivo de solução contém referências a um ou mais arquivos de projeto.
- **TextHello** 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 itens do projeto. Todos os códigos-fonte de um mesmo projeto devem ser escritos na mesma linguagem de programação. No Windows Explorer, esse arquivo se chama TextHello.csproj e está armazenado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 1\TextHello\TextHello.
- **Properties** É uma pasta do projeto TextHello. Se for expandida, 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. Aprender a utilizar esses atributos está além do escopo deste livro.
- **References** É uma pasta que contém as referências ao código compilado que seu aplicativo pode utilizar. Quando o código é compilado, ele é convertido em um código assembly e recebe um nome exclusivo. Desenvolvedores utilizam assemblies para empacotar códigos úteis que escreveram, podendo distribuí-los para outros desenvolvedores que queiram utilizá-los nos seus aplicativos. Muitos dos recursos que você utilizará ao escrever os aplicativos propostos por este livro usam os códigos assembly fornecidos pela Microsoft com o Visual Studio 2010.
- **App.config** É o arquivo de configuração de aplicativo. 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 conhecerá outros detalhes sobre este aplicativo 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 pela primeira vez. Você escreverá seu código para o aplicativo de console nesse arquivo. Ele contém um código que o Visual Studio 2010 fornece automaticamente, o qual será examinado a seguir.

Escrevendo seu primeiro programa

O arquivo Program.cs define uma classe chamada *Program* que contém um método chamado *Main*. Todos os métodos devem ser definidos dentro de uma classe. Você aprenderá mais sobre classes no Capítulo 7, “Criando e gerenciando classes e objetos”. O método *Main* é especial – ele designa o ponto de entrada do programa e deve ser um método estático. (Veremos métodos em detalhes no Capítulo 3, “Escrevendo métodos e aplicando escopo”, e o Capítulo 7 descreve os métodos estáticos.)



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

Nos exercícios a seguir, você irá escrever um código para exibir a mensagem “Hello World” no console; compilar e executar seu aplicativo de console Hello World; e 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. Nessa nova linha, digite a palavra **Console**, que é o nome de uma classe predefinida. 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 *Con*, a lista IntelliSense voltará automaticamente para o item *Console* e você poderá pressionar as teclas Tab ou Enter para selecioná-la.

O código *Main* deve ser semelhante a este:

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



Nota *Console* é uma classe predefinida que contém os métodos para exibir mensagens na tela e obter entradas a partir do teclado.

2. Digite um ponto logo após *Console*. Outra lista IntelliSense aparece exibindo os métodos, propriedades e campos da classe *Console*.
3. 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
}
```

4. 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 sobre-carregados 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*.

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

Main deve se parecer com isto:

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

6. 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的习惯 de digitar pares de caracteres correspondentes, como (e) e { 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, “Implementando propriedades para acessar campos”)

(continua)

Ícone	Significado
	classe (Capítulo 7)
	estrutura (Capítulo 9, "Criando tipos-valor com enumerações e estruturas")
	enumeração (Capítulo 9)
	interface (Capítulo 13, "Criando interfaces e definindo classes abstratas")
	delegate (Capítulo 17, "Interrompendo o fluxo do programa e tratando eventos")
	método de extensão (Capítulo 12, "Trabalhando com herança")

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



Nota Você verá frequentemente linhas de código contendo duas barras seguidas por um texto comum. Esses são os comentários. Eles são ignorados pelo compilador, mas são muito úteis para os desenvolvedores porque ajudam a documentar o que um programa está fazendo. Por 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 multilinha 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*.

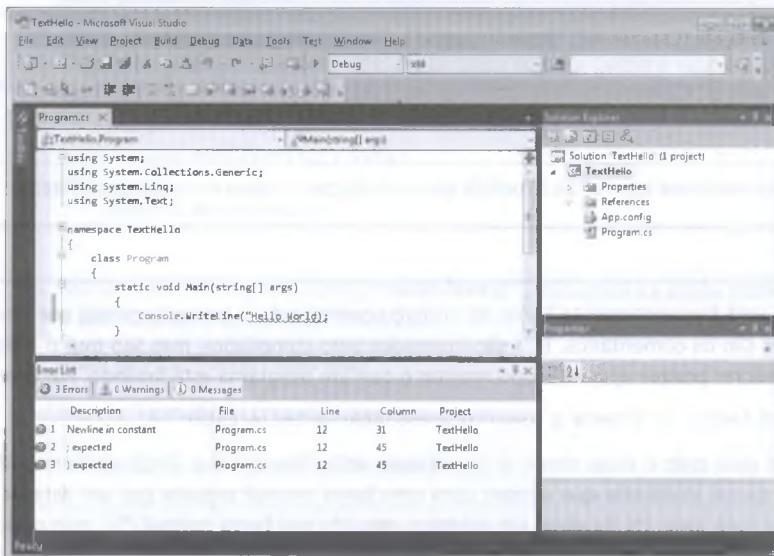


Dica Se a janela *Output* não aparecer no menu *View*, clique em *Output* para exibi-la.

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

```
-----Build started: Project: TextHello, Configuration: Debug x86 -----
CopyFileToOutputDirectory:
TextHello -> C:\Users\John\Meus Documentos\Microsoft Press\Visual CSharp Step By Step
\Chapter 1\TextHello\TextHello\bin\Debug\TextHello.exe
===== Build: 1 succeeded or up-to-date, 0 failed, 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.



Dica Você pode clicar duas vezes em um item na janela *Error List*, e o cursor será posicionado na linha que causou o erro. 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 exatamente antes de compilá-lo porque o comando *Build Solution* faz o salvamento automático.

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.

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

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

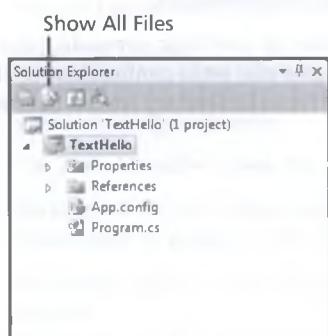


Nota O prompt "Press any key to continue . . ." é gerado pelo Visual Studio sem que você tenha escrito um código para fazer isso. Se executar o programa utilizando o comando *Start Debugging* no menu *Debug*, o aplicativo será executado, mas a janela de comando fechará imediatamente sem esperar que você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

- Verifique se a janela de comandos que exibe a saída do programa tem o foco, e, em seguida, pressione Enter.

A janela de comandos é fechada, e você retorna ao ambiente de programação do Visual Studio 2010.

- No *Solution Explorer*, clique no projeto *TextHello* (não na solução) e depois no botão da barra de ferramentas *Show All Files*, na barra de ferramentas *Solution Explorer* – esse é o botão posicionado na extremidade esquerda na barra de ferramentas da janela *Solution Explorer*.



As entradas nomeadas *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\TextHello\TextHello*). O Visual Studio as cria quando você compila seu aplicativo, e estas contêm a versão executável do programa e alguns outros arquivos utilizados para compilar e depurar o aplicativo.

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

Outra pasta chamada *Debug* é exibida.



Nota Você também verá uma pasta chamada *Release*.

6. No *Solution Explorer*, expanda a pasta *Debug*.

Quatro outros itens chamados *TextHello.exe*, *TextHello.pdb*, *TextHello.vshost.exe* e *TextHello.vshost.exe.manifest* aparecem.

O programa compilado é o arquivo *TextHello.exe*, que é executado quando você clica em *Start Without Debugging* no menu *Debug*. Os outros dois arquivos contêm informações que são utilizadas pelo Visual Studio 2010, se você executar o programa no modo *Debug* (quando você clica em *Start Debugging* no menu *Debug*).

Utilizando 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 nomes, mais métodos e mais classes. Conforme o número de nomes aumenta, também aumenta a probabilidade de a compilação do projeto falhar porque dois ou mais nomes entram em conflito (especialmente quando um programa também usa bibliotecas escritas por outros desenvolvedores, os quais 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 solução não é boa porque 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 incompreensíveis.

Os namespaces ajudam a resolver esse problema criando um contêiner nomeado, para outros identificadores, 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 *TextHello*, como mostrado a seguir:

```
namespace TextHello
{
    class Greeting
    {
        ***
    }
}
```

Você pode então referenciar a classe *Greeting* como *TextHello.Greeting* nos seus próprios programas. Se outro desenvolvedor também criar uma classe *Greeting* em um namespace diferente, como *NewNamespace*, e instalá-la no seu computador, seus programas ainda funcionarão conforme o esperado, pois usarão a classe *TextHello.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 2010 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 simplesmente atribuir à classe um nome globalmente único como *SystemConsole*, sem se incomodar com um namespace. Felizmente, é possível resolver esse problema com uma diretiva *using* nos seus programas. Se você retornar ao programa *TextHello* no Visual Studio 2010 e examinar o arquivo *Program.cs* na janela *Code and Text Editor*, notará as seguintes instruções no início do arquivo:

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

Uma instrução *using* adiciona um namespace ao escopo. No código subsequente, no mesmo arquivo, você não tem mais que qualificar explicitamente os objetos com o namespace ao qual eles pertencem. Os quatro namespaces mostrados contêm classes utilizadas com tanta frequência que o Visual Studio 2010 adiciona essas instruções *using* automaticamente toda vez que você cria um novo projeto. Você pode adicionar outras diretivas *using* na parte superior de um arquivo-fonte.

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

Experimente os nomes longos

- Na janela *Code and Text Editor* que exibir o arquivo *Program.cs*, comente a primeira diretiva *using* na parte superior do arquivo, desta maneira:

```
// using System;
```

- 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.

- Na janela *Error List*, clique duas vezes na mensagem de erro. O identificador que causou o erro é destacado no arquivo-fonte *Program.cs*.

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

O código *Main* deve ser semelhante a este:

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



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

5. No menu *Build*, clique em *Build Solution*.

A compilação 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 novamente.

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

Namespaces e assemblies

Uma instrução *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 normalmente tem 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 que o .NET Framework Class Library abrange, como *System.Console*, são fornecidas nos assemblies instalados no seu computador 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, o .NET Framework Class Library é dividido em alguns assemblies, agrupados de acordo com a área funcional a que as classes estão relacionadas. Por exemplo, há um assembly "básico" que contém todas as classes comuns, como *System.Console*, e há outros assemblies que contêm classes para manipular bancos de dados, acessar Web services, compilar interfaces gráficas com o usuário e assim por diante. Se quiser utilizar uma classe em um assembly, você deverá adicionar ao seu projeto uma referência a este. 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 para múltiplos namespaces; e um único namespace pode abranger múltiplos assemblies. 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 *TextHello*, expanda a pasta *References*. Você verá que um aplicativo Console automaticamente inclui referências aos assemblies chamados *Microsoft.CSharp*, *System*, *System.Core*, *System.Data*, *System.Data.DataExtensions*, *System.Xml* e *System.Xml.Linq*. Você pode adicionar referências para assemblies adicionais a um projeto clicando com o botão direito do mouse na pasta *References* e em *Add Reference* – você executará essa tarefa nos próximos exercícios.

Criando um aplicativo gráfico

Até aqui, você usou o Visual Studio 2010 para criar e executar um aplicativo Console básico. O ambiente de programação do Visual Studio 2010 também contém tudo que você precisa para criar aplicativos gráficos baseados no Windows. Você pode projetar a interface de usuário baseada em formulários de um aplicativo baseado em Windows de modo interativo. O Visual Studio 2010 então gera as instruções do programa para implementar a interface com o usuário que você projetou.

O Visual Studio 2010 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 para um aplicativo gráfico, e a janela *Design View* para organizar sua interface com o 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 2010. Esse programa exibirá 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 em uma caixa de mensagem.

Nota O Visual Studio 2010 fornece dois templates para compilar aplicativos gráficos – o template *Windows Form 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.

Crie um aplicativo gráfico no Visual Studio 2010

- Se você estiver utilizando o Visual Studio 2010 Standard ou o Visual Studio 2010 Professional, siga estes passos para criar um novo aplicativo gráfico:

1. No menu *File*, aponte para *New* e então clique em *Project*. A caixa de diálogo *New Project* abre.

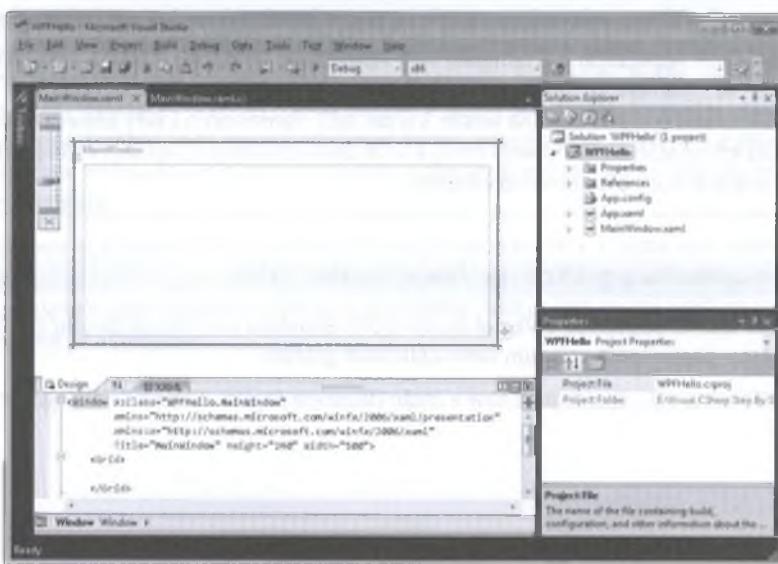
2. No painel à esquerda, em *Installed Templates*, clique em *Visual C#*.
3. No painel central, clique no ícone *WPF Application*.
4. Certifique-se de que o campo *Location* refere-se à pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 1`, na pasta *Documentos*.
5. No campo *Name*, digite **WPFHello**.
6. No campo *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 *TextHello*.

7. Clique em *OK*.
- Se você estiver utilizando o Visual C# 2010 Express, siga estes passos para criar um novo aplicativo gráfico.

1. No menu *File*, clique em *New Project*.
2. Se a caixa de mensagem *New Project* aparecer, clique em *Save* para salvar suas alterações no projeto *TextHello*. Na caixa de diálogo *Save Project*, verifique se o campo *Location* está configurado como `Microsoft Press\Visual CSharp Step By Step\Chapter 1` em sua pasta *Documentos* e então clique em *Save*.
3. Na caixa de diálogo *New Project*, clique no ícone *WPF Application*.
4. No campo *Name*, digite **WPFHello**.
5. Clique em *OK*.

O Visual Studio 2010 fecha seu aplicativo atual e cria o novo aplicativo WPF. Um formulário WPF vazio é exibido na janela *Design View*, com outra contendo uma descrição XAML do formulário, como mostrado na ilustração a seguir:





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

XAML significa Extensible Application Markup Language e é uma linguagem tipo XML utilizada por aplicativos WPF para definir o layout de um formulário e seu conteúdo. Se você conhece XML, a XAML deverá lhe parecer familiar. Na realidade, você pode definir completamente um formulário WPF escrevendo uma descrição XAML, se não quiser utilizar a janela de visualização Design View do Visual Studio ou se não tiver acesso ao Visual Studio; a Microsoft fornece um editor de XAML chamado XAMLPad que é instalado com o Windows Software Development Kit (SDK).

No próximo exercício, você vai utilizar a janela Design View para adicionar três controles ao formulário Windows e examinar alguns dos códigos C# gerados automaticamente pelo Visual Studio 2010 para implementar esses controles.

Crie a interface do usuário

1. Clique na guia *Toolbox* que é 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 Windows. Expanda a seção Common WPF Controle. Esta seção exibe uma lista de controles utilizados pela maioria dos aplicativos WPF. A seção All Controls exibe uma lista mais extensa de controles.

2. Na seção Common WPF Controls, clique em Label e arraste o controle do rótulo para a parte visível do formulário.

Um controle *label* (rótulo) é adicionado ao formulário (você o moverá para a localização correta mais adiante), e a *Toolbox* é oculta.



Dica Se você quiser que a *Toolbox* permaneça visível, mas não oculte qualquer parte do formulário, clique no botão *Auto Hide* à direita da barra de título da *Toolbox*. (Ele se parece com um pino.) A *Toolbox* aparece permanentemente no lado esquerdo da janela do Visual Studio 2010, 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.

3. O controle *label* no formulário provavelmente não está exatamente onde você quer. Você pode clicar e arrastar os controles que adicionou a um formulário para reposicioná-los. Utilizando essa técnica, mova o controle *label* para posicioná-lo próximo ao canto superior esquerdo do formulário. (O local exato não é importante para esse aplicativo.)



Nota A descrição XAML do formulário na parte inferior do painel agora inclui o controle *label*, além de propriedades como sua localização no formulário, coordenado pela propriedade *Margin*. A propriedade *Margin* consiste em quatro números que indicam a distância de cada borda do rótulo a partir das bordas do formulário. Se você mover o controle dentro do formulário, o valor da propriedade *Margin* muda. Se o formulário for redimensionado, os controles ancorados nas bordas que se movem do formulário são redimensionados para preservar os valores de margem. Você pode evitar isso configurando os valores de *Margin* como zero. Você aprenderá mais sobre propriedades *Margin* e também sobre as propriedades *Height* e *Width* dos controles WPF no Capítulo 22, "Apresentando o Windows Presentation Foundation".

4. No menu *View*, clique em *Properties Window*.

Se já estava aberta, a janela *Properties* aparece no canto inferior direito da tela, sob *Solution Explorer*. Para especificar as propriedades dos controles, use o painel XAML, abaixo da janela *Design View*. Entretanto, a janela *Properties* é uma maneira mais prática de modificar as propriedades dos itens em um formulário assim como de outros itens em um projeto. Ela diferencia letras maiúsculas de minúsculas e exibe as propriedades do item selecionado. Se clicar na barra de título do formulário exibido na janela *Design View*, você verá que a janela *Properties* exibe as propriedades do próprio formulário. Se você clicar no controle *label*, a janela exibirá as propriedades do rótulo. Se clicar em outro local no formulário, a janela *Properties* exibirá as propriedades de um item misterioso chamado *grade* (*grid*). Uma grade é quase como um contêiner para itens em um formulário WPF, e você pode utilizá-la, entre outras coisas, para indicar como os itens no formulário devem ser alinhados e agrupados.

5. Clique no controle *label* no formulário. Na janela *Properties*, localize a propriedade *FontSize*. Mude a propriedade *FontSize* para 20 e então, na janela *Design View*, clique na barra de título do formulário.

O tamanho do texto no rótulo muda.

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

Sobrescreva o valor da propriedade *FontSize*, no painel XAML, e altere-a novamente para 12. O tamanho do texto no rótulo da janela *Design View* voltará ao anterior.

7. No painel XAML, examine as outras propriedades do controle *label*.

As propriedades listadas no painel XAML são as únicas que não têm valores padrão. Se você modificar quaisquer valores de propriedade na janela *Properties*, eles aparecerão como parte da definição do rótulo no painel.

8. Altere o valor da propriedade *Content*, de **Label** para **Please enter your name**.

Observe que o texto exibido no rótulo sobre o formulário muda, embora o rótulo continue muito pequeno para mostrá-lo corretamente.

9. Na janela *Design View*, clique no controle *label*. Posicione o mouse sobre a borda direita desse controle. O perfil do mouse deve mudar para uma seta de duas pontas, indicando que você pode utilizar o mouse para redimensionar o controle. Clique o mouse e arraste a borda direita do controle *label* mais para a direita, até você conseguir ver o texto completo do rótulo.
10. Clique no formulário na janela *Design View* e exiba a *Toolbox* novamente.
11. Na *Toolbox*, clique e arraste o controle *TextBox* para o formulário. Mova o controle da caixa de texto para posicioná-lo imediatamente abaixo do controle *label*.



Dica Ao 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 corretamente.

12. Com o controle de caixa de texto ainda selecionado, na janela *Properties*, altere o valor da propriedade *Name* exibida na parte superior da janela para **userName**.



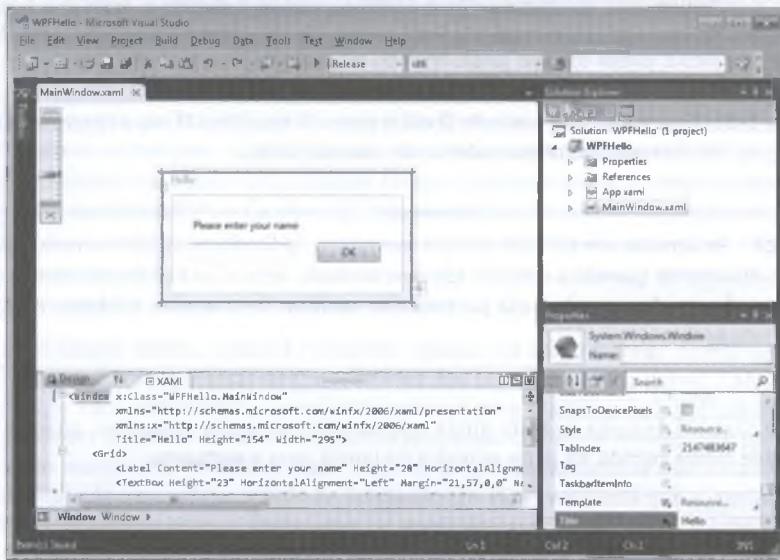
Nota Falaremos mais sobre as convenções de nomes para controles e variáveis no Capítulo 2, “Trabalhando com variáveis, operadores e expressões”.

13. Exiba a *Toolbox* novamente, depois clique e arraste um controle *Button* para o formulário. Posicione o controle *button* à direita da caixa do controle caixa de texto no formulário, de modo que a parte inferior do botão fique alinhada horizontalmente com a parte inferior da caixa de texto.
14. Na janela *Properties*, altere a propriedade *Name* do controle *button* para **ok**. E altere a propriedade *Content* do *Button* para **OK**. Verifique se a legenda do controle *button* no formulário muda.
15. Clique na barra de título do formulário *MainWindow.xaml* na janela *Design View*. Na janela *Properties*, mude a propriedade *Title* para **Hello**.
16. Na janela *Design View*, observe que uma alça de redimensionamento (um pequeno quadrado) aparece no canto inferior direito do formulário quando selecionada. Mova o ponteiro do mouse sobre a alça de redimensionamento. Quando o ponteiro virar uma seta de duas pontas diagonal, clique e 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 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 deve ficar parecido com a figura a seguir:



- No menu *Build*, clique em *Build Solution* e verifique se a compilação do projeto foi bem-sucedida.

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

O aplicativo deve ser executado, exibindo seu formulário. Você pode digitar seu nome na caixa de texto e clicar em *OK*, mas nada acontece ainda. É necessário adicionar algum código para processar o evento *Click* para o botão *OK*, o que faremos em seguida.

- Clique no botão *Close* (o *X* no canto superior direito do formulário) para fechar o formulário e retornar ao Visual Studio.

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 gera uma grande quantidade de código que trata das tarefas de rotina que todos os aplicativos gráficos devem realizar, como abrir e exibir um formulário. Antes de adicionar seu próprio código ao aplicativo, é importante entender o que Visual Studio gerou.

No *Solution Explorer*, expanda o nó de *MainWindow.xaml*. O arquivo *MainWindow.xaml.cs* aparece. Clique duas vezes no arquivo *MainWindow.xaml.cs*, e o código do formulário é exibido na janela *Code and Text Editor*. Ele se parece com este:

```
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 WPFHello
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>

    public partial class MainWindow : Window
    {

        public MainWindow()
        {
            InitializeComponent();
        }

    }
}
```

Além de muitas instruções *using* colocar em escopo alguns namespaces que a maioria dos aplicativos WPF utiliza, o arquivo contém apenas a definição de uma classe chamada *MainWindow*. Há um pouco de código para a classe *MainWindow* conhecido como construtor que chama um método denominado *InitializeComponent*, mas isso é tudo. (Um *construtor* é um método especial com o mesmo nome da classe. Ele é executado quando é criada uma instância da classe que pode conter um código para inicializar a instância. Discutiremos construtores no Capítulo 7.) Na realidade, o aplicativo contém muito mais código, mas a maior 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.

O objetivo desse código, que você *pode* ver nessa classe, é adicionar seus próprios métodos para tratar a lógica do seu aplicativo, como determinar o que acontece quando o usuário clica no botão *OK*.

Dica Você também pode exibir o arquivo do código C# para um formulário WPF clicando com o botão direito do mouse em qualquer lugar na janela *Design View* e depois em *View Code*.

Você deve estar querendo saber onde está o método *Main* e como o formulário será exibido quando o aplicativo for executado; lembre-se de que o *Main* define o ponto em que o programa inicia. Na *Solution Explorer*, deve aparecer um outro arquivo-fonte chamado *App.xaml*. Se você clicar duas vezes nesse arquivo, será exibida a descrição XAML desse item.

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="WPFHello.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>
```

Se você clicar na guia *Design*, na parte inferior do painel XAML, a janela *Design View* para o App.xaml será exibida e apresentará o texto “Intentionally left blank. The document root element is not supported by the visual designer”. Isso ocorre porque não é possível utilizar a janela *Design View* para modificar o arquivo App.xaml. Clique na guia *XAML* para retornar ao painel XAML. Se você expandir o nó App.xaml no *Solution Explorer*, verá que há também um arquivo Application.xaml.cs. Ao clicar duas vezes nesse arquivo, você descobrirá que ele contém este código:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Windows;

namespace WPFHello
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>

    public partial class App : Application
    {
    }
}
```

Mais uma vez, há algumas instruções *using*, mas não muito além disso, nem mesmo um método *Main*. De fato, *Main* está aí, mas também está oculto. O código para *Main* é gerado com base nas configurações no arquivo App.xaml file; em particular, *Main* criará e exibirá o formulário especificado pela propriedade *StartupUri*. Se quiser exibir um formulário diferente, edite o arquivo App.xaml.

É o momento de você mesmo escrever algum código!

Escreva o código para o botão OK

1. Clique na guia *MainWindow.xaml* acima da janela *Code and Text Editor* para exibir MainWindow na janela *Design View*.
2. Clique duas vezes no botão *OK* no formulário.

O arquivo MainWindow.xaml.cs aparece na janela *Code and Text Editor*, mas um novo método chamado *ok_Click* foi adicionado. O Visual Studio gera automaticamente o código para chamar esse método sempre que o usuário clica no botão *OK*. Esse é um exemplo de evento, e você aprenderá muito mais sobre como os eventos funcionam à medida que avançar no livro.

3. Adicione o seguinte código mostrado em negrito ao método *ok_Click*:

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

Esse é o código que será executado quando o usuário clicar no botão *OK*. Não se preocupe com a sintaxe desse código ainda (simplesmente certifique-se de copiá-lo exatamente como mostrado), pois você aprenderá tudo sobre métodos no Capítulo 3. A parte interessante é a instrução *MessageBox.Show*. Essa instrução exibe uma caixa de mensagem contendo o texto "Hello" com qualquer nome digitado pelo usuário na caixa de texto de nome de usuário no formulário anexado.

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

```
<Button Height="23" ... Click="ok_Click"/>
```

6. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
7. Quando o formulário aparecer, digite seu nome na caixa de texto e então clique em *OK*. Uma caixa de mensagens de boas-vindas com seu nome aparece.



8. Clique em *OK* na caixa de mensagem, e ela será fechada.
9. Feche o formulário.

Neste capítulo, você viu como é possível utilizar o Visual Studio 2010 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 simples interface gráfica do usuário.

- Se você quiser seguir o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 2.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 1

Para	Faça isto
Criar um novo aplicativo de console utilizando o Visual Studio 2010 Standard ou Professional	No menu <i>File</i> , aponte para <i>New</i> e, em seguida, clique em <i>Project</i> para abrir a caixa de diálogo <i>New Project</i> . No painel à esquerda, em <i>Installed Templates</i> , clique em <i>Visual C#</i> . No painel central clique em <i>Console Application</i> . Especifique um diretório para os arquivos do projeto na caixa <i>Location</i> . Digite um nome para o projeto. Clique em <i>OK</i> .
Criar um novo aplicativo de console utilizando o Visual C# 2010 Express	No menu <i>File</i> , clique em <i>New Project</i> para abrir a caixa de diálogo <i>New Project</i> . Para o modelo, selecione <i>Console Application</i> . Escolha um nome para o projeto. Clique em <i>OK</i> .
Criar um novo aplicativo gráfico utilizando o Visual Studio 2010 Express ou Professional	No menu <i>File</i> , aponte para <i>New</i> e clique em <i>Project</i> para abrir a caixa de diálogo <i>New Project</i> . No painel à esquerda, em <i>Installed Templates</i> , clique em <i>Visual C#</i> . No painel central, Clique em <i>WPF Application</i> . Especifique um diretório para os arquivos de projeto. Na caixa <i>Location</i> , digite um nome para o projeto. Clique em <i>OK</i> .
Criar um novo aplicativo gráfico utilizando o Visual C# 2010 Express	No menu <i>File</i> , clique em <i>New Project</i> para abrir a caixa de diálogo <i>New Project</i> . Para o modelo, selecione <i>WPF Application</i> . Escolha um nome para o projeto. Clique em <i>OK</i> .
Compilar o aplicativo	No menu <i>Build</i> , clique em <i>Build Solution</i> .
Executar o aplicativo	No menu <i>Debug</i> , clique em <i>Start Without Debugging</i> .

Capítulo 2

Trabalhando com 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.

No Capítulo 1, “Bem-vindo ao C#”, você aprendeu a utilizar o ambiente de programação do Microsoft Visual Studio 2010 para compilar e executar um programa Console e um aplicativo Windows Presentation Foundation (WPF). Este capítulo apresenta os elementos de sintaxe e semântica do Microsoft Visual C#, incluindo instruções, palavras-chave e identificadores. Você estudará os tipos primitivos que são compilados na linguagem C# e as características dos valores que cada tipo armazena. Também verá como declarar e utilizar as variáveis locais (que só existem dentro de uma função ou outra pequena seção do código), entenderá os operadores aritméticos que o C# fornece, descobrirá como utilizar operadores para manipular valores e aprenderá a controlar expressões que contêm dois ou mais operadores.

Entendendo instruções

Uma *instrução* é um comando que executa uma ação. Você combina instruções para criar métodos (para aprender mais sobre métodos, ver Capítulo 3, “Escrevendo métodos e aplicando o escopo”). Imagine um método como uma sequência nomeada de instruções. *Main*, que foi apresentado no capítulo anterior, é um exemplo de método. 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, sem seu ponto e vírgula de terminação, a instrução a seguir não será compilada:

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



Dica O 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 e ater-se a ele 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 abordagem facilita a manutenção dos seus programas. Nos capítulos deste livro, você verá exemplos das instruções mais importantes do C#.

Utilizando identificadores

Identificadores são os nomes utilizados para identificar os elementos nos seus programas, como namespaces, classes, métodos e variáveis. (Discutiremos 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 identificadores diferentes.

Identificando 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. Elas estão listadas na tabela a seguir.

<i>abstract</i>	<i>do</i>	<i>in</i>	<i>protected</i>	<i>true</i>
<i>as</i>	<i>double</i>	<i>int</i>	<i>public</i>	<i>try</i>
<i>base</i>	<i>else</i>	<i>interface</i>	<i>readonly</i>	<i>typeof</i>
<i>bool</i>	<i>enum</i>	<i>internal</i>	<i>ref</i>	<i>uint</i>
<i>break</i>	<i>event</i>	<i>is</i>	<i>return</i>	<i>ulong</i>

<i>byte</i>	<i>explicit</i>	<i>lock</i>	<i>sbyte</i>	<i>unchecked</i>
<i>case</i>	<i>extern</i>	<i>long</i>	<i>sealed</i>	<i>unsafe</i>
<i>catch</i>	<i>false</i>	<i>namespace</i>	<i>short</i>	<i>ushort</i>
<i>char</i>	<i>finally</i>	<i>new</i>	<i>sizeof</i>	<i>using</i>
<i>checked</i>	<i>fixed</i>	<i>null</i>	<i>stackalloc</i>	<i>virtual</i>
<i>class</i>	<i>float</i>	<i>object</i>	<i>static</i>	<i>void</i>
<i>const</i>	<i>for</i>	<i>operator</i>	<i>string</i>	<i>volatile</i>
<i>continue</i>	<i>foreach</i>	<i>out</i>	<i>struct</i>	<i>while</i>
<i>decimal</i>	<i>goto</i>	<i>override</i>	<i>switch</i>	
<i>default</i>	<i>if</i>	<i>params</i>	<i>this</i>	
<i>delegate</i>	<i>implicit</i>	<i>private</i>	<i>throw</i>	

 **Dica** Na janela *Code and Text Editor* do Visual Studio 2010, as palavras-chave são pintadas de azul quando digitadas.

O C# também utiliza os identificadores relacionados abaixo. 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.

<i>dynamic</i>	<i>join</i>	<i>set</i>
<i>from</i>	<i>let</i>	<i>value</i>
<i>get</i>	<i>orderby</i>	<i>var</i>
<i>group</i>	<i>partial</i>	<i>where</i>
<i>into</i>	<i>select</i>	<i>yield</i>

Utilizando variáveis

Uma *variável* é uma localização da memória que armazena um valor, ou seja, 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 identifica 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.

Nomeando variáveis

Adote uma convenção de nomes que torne claras as variáveis definidas. A lista a seguir contém algumas recomendações gerais:

- Não inicie um identificador com um sublinhado.
- 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.



Nota A utilização de identificadores cuja única diferença seja a distinção entre maiúsculas e minúsculas pode limitar a reutilização das classes nos aplicativos desenvolvidos usando outras linguagens que não diferem maiúsculas e minúsculas, como o Microsoft 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 a notação húngara (programadores em Microsoft Visual C++ provavelmente já conhecem a notação húngara. Se você não souber o que é, não se preocupe!).



Importante Você deve considerar as duas primeiras recomendações anteriores obrigatorias porque estão relacionadas para conformidade com a Common Language Specification (CLS). Se você deseja escrever programas que possam interoperar com outras linguagens, como o Microsoft Visual Basic .NET, deve obedecer a essas recomendações.

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

Declarando 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* (idade) 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 Os programadores que utilizam o Microsoft Visual Basic devem 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*. Observe que o ponto e vírgula é requerido novamente.

```
age = 42;
```

O sinal de igual (=) é o operador de *atribuição*, que atribui o valor que está à sua direita à variável que está à 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 ponteiro do mouse sobre uma variável na janela Visual Studio 2010 *Code and Text Editor*, uma dica de tela será exibida informando o tipo de variável.

Trabalhando com 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 (em bits)	Intervalo	Exemplo de uso
<i>int</i>	Números inteiros	32	-2^{31} a $2^{31} - 1$	<code>int count; count = 42;</code>
<i>long</i>	Números inteiros (intervalo maior)	64	-2^{63} a $2^{63} - 1$	<code>long wait; wait = 42L;</code>
<i>float</i>	Números de ponto flutuante	32	$\pm 1.5 \times 10^{45}$ a $\pm 3.4 \times 10^{38}$	<code>float away; away = 0.42F;</code>
<i>double</i>	Números de ponto flutuante de precisão dupla (maior precisão)	64	$\pm 5.0 \times 10^{-324}$ a $\pm 1.7 \times 10^{308}$	<code>double trouble; trouble = 0.42;</code>
<i>decimal</i>	Valores monetários	128	28 números significativos	<code>decimal coin; coin = 0.42M;</code>

Tipo de dado	Descrição	Tamanho (em bits)	Intervalo	Exemplo de uso
<i>string</i>	Sequência de caracteres	16 bits por caractere	Não aplicável	<code>string vest; vest = "fortytwo";</code>
<i>char</i>	Caractere único	16	0 a 216 – 1	<code>char grill; grill = 'x';</code>
<i>bool</i>	Booleano	8	Verdadeiro ou falso	<code>bool teeth; teeth = false;</code>

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 um erro de tempo de compilação porque a variável *age* não foi atribuída:

```
int age;
Console.WriteLine(age); // erro de tempo de compilação
```

Exibindo 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. Inicialize o Visual Studio 2010 se ainda não estiver em execução.
2. Se estiver utilizando o Visual Studio 2010 Standard ou o Visual Studio 2010 Professional, no menu *File*, aponte o cursor para *Open* e então clique em *Project/Solution*.

Se você estiver utilizando o Visual C# 2010 Express, no menu *File*, clique em *Open Project*.

A caixa de diálogo *Open Project* aparece.

3. Vá para a pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 2\PrimitiveDataTypes* na sua pasta *Documentos*. Selecione o arquivo de solução *PrimitiveDataTypes* e clique em *Open*.

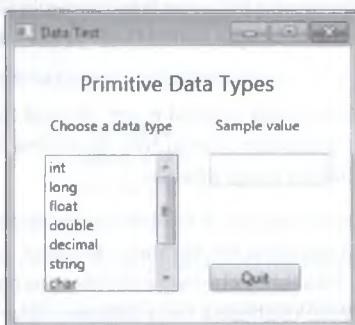
A solução é carregada, e o *Solution Explorer* exibe o projeto *PrimitiveDataTypes*.



Nota Os nomes dos arquivos de solução têm o sufixo .sln, como em PrimitiveDataTypes.sln. Uma solução pode conter um ou mais projetos, cujos arquivos têm o sufixo .csproj. Se um projeto for aberto em vez de uma solução, o Visual Studio 2010 criará para ele, automaticamente, um novo arquivo de solução. Se a solução for compilada, o Visual Studio 2010 salvará automaticamente todos os arquivos novos ou atualizados, e você será solicitado a fornecer um nome e um local para o novo arquivo de solução.

4. No menu *Debug*, clique em *Start Without Debugging*.

Talvez sejam exibidos alguns avisos no Visual Studio que certamente podem ser ignorados. (Você os corrigirá no próximo exercício.) A seguinte janela de aplicativo será exibida:



5. Na lista *Choose a data type*, clique no tipo *string*.

O valor "forty two" aparece na caixa *Sample value*.

6. Clique no tipo *int* na lista.

O valor "to do" ("a fazer") aparece na caixa *Sample value*, indicando que as instruções exibindo um valor *int* ainda precisam ser escritas.

7. Clique em cada tipo de dado na lista. Confirme que o código para os tipos *double* e *bool* ainda não está implementado.
8. Clique em *Quit* para fechar a janela e parar o programa. O controle retorna ao ambiente de trabalho do Visual Studio 2010.

Utilize tipos de dados primitivos no código

1. No *Solution Explorer*, clique duas vezes em *MainWindow.xaml*.

O formulário WPF para o aplicativo aparece na janela *Design View*.

2. Clique com o botão direito do mouse em qualquer lugar na janela *Design View* para exibir o formulário *MainWindow.xaml* e então clique em *View Code*.

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



Nota Lembre-se de que você também pode utilizar o *Solution Explorer* para acessar o código; clique no sinal de adição, +, à esquerda do arquivo *MainWindow.xaml*, e então dê um clique duplo em *MainWindow.xaml.cs*.

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



Dica Para localizar um item no seu projeto, no menu *Edit*, aponte para *Find and Replace* e clique em *Quick Find*. Uma caixa de diálogo é aberta e pergunta o que você deseja pesquisar. Digite o nome do item que está procurando e clique em *Find Next*. Por padrão, a pesquisa não diferencia maiúsculas de minúsculas. Se você quiser executar uma pesquisa que diferencie letras maiúsculas de minúsculas, clique no botão de adição, +, ao lado do rótulo *Find Options* para exibir as opções adicionais e selecione a caixa de seleção *Match Case*. Se tiver tempo, você pode experimentar as outras opções.

Você também pode pressionar Ctrl+F (pressione a tecla Control e, em seguida, pressione 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 ao utilizar a caixa de lista suspensa de membros da classe, posicionada acima da janela *Code and Text Editor*, à direita. Essa lista exibe todos os métodos na classe e as variáveis e outros itens que a classe contém. (Você conhecerá mais detalhes sobre esses itens nos capítulos posteriores.) Na caixa da lista suspensa, clique em *showFloatValue()*, e o cursor saltará imediatamente para o método *showFloatValue()* na classe.

O método *showFloatValue* é executado quando você clica no tipo *float* na caixa de listagem. Esse método contém as três instruções a seguir:

```
float variable;  
variable=0.42F;  
value.Text = "0.42F";
```

A primeira instrução declara uma variável chamada *variable* do tipo *float*.

A segunda instrução atribui o valor 0.42F à *variable* (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 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 instrução exige um pouco mais de atenção. A maneira como você exibe um item em uma caixa de texto é configurando a propriedade *Text*. Observe que a propriedade de um objeto é acessada utilizando a mesma notação de “ponto” que vimos para executar um método (lembre-se de *Console.WriteLine* no Capítulo 1?). Os dados adicionados à propriedade *Text* devem ser uma string (uma string incluída entre aspas duplas) e não um número. Se você tentar atribuir um número à propriedade *Text*, seu programa não compilará. Nesse programa, a instrução simplesmente

exibe o texto “0.42F” na caixa de texto. Em um aplicativo do mundo real, você adicionaria instruções que convertem o valor da variável *variable* em uma string e então o adicionaria à propriedade *Text*. No entanto, é necessário entender um pouco mais sobre o C# e o Microsoft .NET Framework antes de fazer isso (o Capítulo 11, “Entendendo arrays de parâmetros”, e o Capítulo 21, “Sobrecarga do operador”, abrangem as conversões de tipos de dados).

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

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

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

5. 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 variable;
    variable = 42;
    value.Text = "to do"
}
```

6. A instrução original nesse método altera a string “*to do*” para “*42*”.

O método agora deve estar exatamente como este:

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

 **Nota** Se já tiver experiência em programação, talvez você fique tentado a alterar a terceira instrução para

```
value.Text = variable;
```

Aparentemente o valor de *variable* deve ser exibido na caixa de texto *value* no formulário. Entretanto, o C# realiza uma verificação estrita dos tipos; caixas de texto só podem exibir valores *string*, e *variable* é um *int*, portanto, essa instrução não será compilada. Veremos algumas técnicas simples de converter entre valores numéricos e *string* mais adiante neste capítulo.

7. No menu *Debug*, clique em *Start Without Debugging*.

O formulário aparece novamente.

8. Selecione o tipo *int* na lista *Choose a data type*. Confirme se o valor 42 está sendo exibido na caixa de texto *Sample value*.

9. Clique em *Quit* para fechar a janela e retornar para o Visual Studio.
10. Na janela *Code and Text Editor*, localize o método *showDoubleValue*.
11. Edite o método *showDoubleValue* exatamente como mostrado em negrito no seguinte código:

```
private void showDoubleValue()
{
    double variable;
    variable = 0.42;
    value.Text = "0.42";
}
```

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

13. Edite o método *showBoolValue* exatamente como a seguir:

```
private void showBoolValue()
{
    bool variable;
    variable = false;
    value.Text = "false";
}
```

14. No menu *Debug*, clique em *Start Without Debugging*.
15. 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*.
16. Clique em *Quit* para parar o programa.

Utilizando operadores aritméticos

O C# suporta as operações aritméticas que você aprendeu no colégio: o sinal de mais (+) para adição, o sinal de menos (-) para subtração, o asterisco (*) para multiplicação e a barra (/) para divisão. Esses 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 taxa diária) e de 20 (o número de dias que o consultor trabalhou):

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



Nota Os valores em que um operador opera 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 utilizar todos os operadores aritméticos em valores do tipo *char*, *int*, *long*, *float*, *double* ou *decimal*, entretanto, com exceção do operador de adição, *+*, os operadores aritméticos em valores do tipo *string* ou *bool* não podem ser utilizados. Portanto, a instrução a seguir não é permitida porque o tipo *string* não suporta o operador de subtração (por isso, não há sentido em subtrair uma string de outra):

```
// erro de tempo de compilação  
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

Você pode utilizar o operador *+* para concatenar valores de *string*, mas aja com cautela para não obter resultados inesperados. Por exemplo, a seguinte instrução escreve “431” (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 realizar cálculos aritméticos em valores armazenados em *strings*.

Você deve estar ciente de que o tipo de resultado de uma operação aritmética depende do tipo de 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 os valores para baixo 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 incompatibilidade e gera um código que converte o *int* em um *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 familiar: o operador resto ou módulo, que é representado pelo sinal de porcentagem (%). O resultado de $x \% y$ é o resto da divisão de x por y . Por exemplo, $9 \% 2$ é 1 , porque 9 dividido por 2 é 4 , resto 1 .

Tipos numéricos e valores infinitos

Há um ou dois outros recursos 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 na verdade 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 $0.0/0.0$. Normalmente, 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 um outro valor especial para essa situação chamado *NaN*, que significa “not a number”. 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 avaliar $10 + Infinity$, o resultado será *Infinity*. A única exceção a essa regra é quando você multiplica *Infinity* por 0, que resulta em 0, enquanto o resultado da expressão *NaN * 0* é *NaN*.



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

Examinando operadores aritméticos

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

Trabalhe com operadores aritméticos

1. Abra o projeto MathsOperators, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 2\MathsOperators na sua pasta *Documentos*.
2. No menu *Debug*, clique em *Start Without Debugging*.
Um formulário aparece na tela.
3. Digite 54 na caixa de texto do operando esquerdo.
4. Digite 13 na caixa de texto do operando direito.
Agora você pode aplicar qualquer um dos operadores aos valores das caixas de texto.
5. Clique no botão – *Subtraction* e, em seguida, clique em *Calculate*.

O texto na caixa *Expression* muda para $54 - 13$, e o resultado 41 aparece na caixa *Result*, como mostrado na figura a seguir:



6. Clique no botão */ Division* e, em seguida, clique em *Calculate*.

O texto na caixa *Expression* muda para $54 / 13$, e o número 4 aparece na caixa *Result*. Em uma situação real, $54 / 13$ é uma dízima periódica; no entanto, aqui o C# está realizando uma divisão de inteiro, e quando um inteiro é dividido por outro inteiro, a resposta que você obtém é um inteiro, como explicado anteriormente.

7. Clique no botão *% Remainder* e então em *Calculate*.

O texto na caixa de texto *Expression* muda para $54 \% 13$, e o número 2 aparece na caixa *Result*. Isso acontece porque o resto, após a divisão de 54 por 13, é 2. ($54 - ((54 / 13) * 13)$) é 2 se você arredondar para baixo em cada etapa – meus antigos professores de matemática do colégio devem estar horrorizados por eu estar dizendo que $(54 / 13) * 13$ não é igual a 54!

8. Teste as outras combinações de números e operadores. Quando terminar, clique em *Quit* para retornar ao ambiente de programação do Visual Studio 2010.

No próximo exercício, você examinará o código do programa *MathsOperators*.

Examine o código do programa *MathsOperators*

1. Exiba o formulário *MainWindow.xaml* na janela *Design View*. (Clique duas vezes no arquivo *MainWindow.xaml* na *Solution Explorer*.)
2. 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 WPF complexo. Os controles são organizados hierarquicamente, começando pela *Window* que constitui o formulário WPF. Como mencionado no capítulo anterior, um formulá-

rio WPF realmente contém um controle *Grid*, e os outros controles são colocados nesse *Grid*. Quando você clica em cada controle no formulário, o nome do controle é destacado na janela *Document Outline*. De maneira semelhante, se selecionar um controle na janela *Document Outline*, o controle correspondente é selecionado na janela *Design View*. Se você posicionar o mouse sobre um controle na janela *Document Outline*, será exibida uma imagem do controle (e todos os controles filhos nele contidos).

3. No formulário, clique nos dois controles *TextBox* em que o usuário digita os números. Na janela *Document Outline*, verifique se eles estão nomeados como *lhsOperand* e *rhsOperand* (é possível ver o nome de um controle nos parênteses à direita do controle).

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

4. Na parte inferior do formulário, verifique se o controle *TextBox* utilizado para exibir a expressão que está sendo avaliada tem o nome *expression* e se o controle *TextBox* utilizado para exibir o resultado do cálculo tem o nome *result*.
5. Feche a janela *Document Outline*.
6. Exiba o código de *MainWindow.xaml.cs* na janela *Code and Text Editor*.
7. Na janela *Code and Text Editor*, localize o método *subtractValues*. Ele se parece a este:

```
private void subtractValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome;
    outcome = lhs - rhs;
    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 de texto *lhsOperand*. Lembre-se de que a propriedade *Text* de um controle de caixa de texto contém uma string, que precisa ser convertida em um inteiro antes de ser atribuída a uma variável *int*. O tipo de dados *int* fornece o método *int.Parse*, que faz precisamente isso.

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

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

A quarta instrução subtrai o valor da variável *rhs* do valor da variável *lhs*, e o resultado é atribuído a *outcome*.

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

A sexta instrução exibe o resultado do cálculo atribuindo-o à propriedade *Text* da caixa de texto *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-la à propriedade *Text*. É isso que o método *ToString* do tipo *int* faz.

O método *ToString*

Cada classe no .NET Framework tem um método *ToString*. A finalidade de *ToString* é converter um objeto na sua representação de string. No exemplo anterior, o método *ToString* de um objeto inteiro, *outcome*, é utilizado para converter o valor inteiro de *outcome* no valor string equivalente. Essa conversão é necessária porque o valor é exibido na propriedade *Text* da caixa de texto *result* – a propriedade *Text* só pode conter strings. Ao criar suas próprias 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, “Criando e gerenciando classes e objetos”.)

Controlando 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 realizada primeiro, a adição ou a multiplicação? A ordem das operações importa porque muda o resultado:

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

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

Os parênteses podem ser utilizados para sobrescrever 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 * para produzir 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 [].

Utilizando a associatividade para avaliar expressões

A precedência de operadores é apenas uma questão a ser considerada. O que acontece quando uma expressão contém operadores diferentes que têm a mesma precedência? Aqui, a associatividade se torna importante já que *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

Essa expressão é potencialmente ambígua. Qual deve ser realizada primeiro, a divisão ou a multiplicação? A precedência dos dois operadores é a mesma (são ambos multiplicativos), mas a ordem na qual a expressão é calculada é importante porque dois resultados diferentes podem ser obtidos:

- Se realizar 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 realizar 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.



Nota À medida que cada novo operador for descrito nos capítulos subsequentes, sua associatividade também será abordada.

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, aceita dois operandos; o operando à sua direita é avaliado e então é armazenado no operando à sua 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 provavelmente está 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 com uma 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 muito útil se você quiser inicializar diferentes variáveis com o mesmo valor. Torna-se claro a 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 sendo atribuído.

Entretanto, trate essa construção com um pouco de cautela. Um erro frequentemente cometido por novos programadores C# é tentar combinar esse uso do operador de atribuição com declarações de variáveis, como esta:

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

Este é um código válido do C# (porque ele é compilado). Na realidade, ele declara as variáveis *myInt*, *myInt2* e *myInt3*, e inicializa *myInt3* com o valor 10. Contudo, ele não inicializa *myInt* ou *myInt2*. Se você tentar utilizar *myInt* ou *myInt2* em expressões como:

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

o compilador gerará os seguintes erros:

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

Incrementando e decrementando variáveis

Se quiser adicionar 1 a uma variável, o operador + pode ser utilizado:

```
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 operadores *unários*, ou seja, eles têm um único operando. Eles compartilham a mesma precedência e associatividade à esquerda que o operador unário!, que será discutido no Capítulo 4, “Utilizando instruções de decisão”.

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 sufixado
++count; // incremento prefixado
count--; // decremento sufixado
--count; // decremento prefixado
```

Utilizar a forma prefixa ou sufixa do operador `++` ou `--` não faz a menor diferença para a variável que está sendo incrementada ou decrementada. Por exemplo, se você escrever `count++`, o valor de `count` aumenta por 1, e se escrever `++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 da incrementação, enquanto o valor retornado por `++count` é o valor de `count` depois que a incrementação ocorre. Veja um exemplo:

```
int x;
x = 42;
Console.WriteLine(x++); // x agora é 43, 42 é escrito no console
x = 42;
Console.WriteLine(++x); // x agora é 43, 43 é escrito no console
```

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 incrementado. 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 são apresentadas no Capítulo 5, “Utilizando atribuição composta e instruções de iteração”. Se você estiver utilizando os operadores de incremento e decremento isoladamente, mantenha a forma sufixada e seja consistente.

Declarando 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 tentar utilizá-la. Você pode declarar e inicializar uma variável na mesma instrução, desta maneira:

```
int myInt = 99;
```

Ou assim, supondo que *myOtherInt* é uma variável do tipo inteiro já inicializada:

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

Agora, lembre-se de que o valor que você atribui a uma variável deve ser do mesmo tipo que a 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 informar se este não corresponde ao tipo da variável. Também pode instruir o compilador C# a deduzir o tipo de uma variável a partir de uma expressão e utilizá-lo ao declarar a variável usando a palavra-chave *var* no lugar do tipo, da seguinte maneira:

```
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 utilizados para inicializá-las. Nesses exemplos, *myVariable* é um *int*, e *myOtherVariable* é uma *string*. 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 – 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 enfatizar que você deve esquecer tudo que já aprendeu ao programar no Visual Basic variáveis *Variant*. 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 um purista, provavelmente esteja rangendo os dentes e perguntando-se por que os projetistas de uma linguagem perfeita como o C# permitiram que um recurso como *var* fosse utilizado. 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 facilmente novos bugs no seu código). Mas confie no fato de que *var* tem um lugar válido no C#, como veremos ao trabalhar nos capítulos a seguir. Por enquanto, iremos nos ater ao uso de variáveis explicitamente tipadas, exceto quando a tipagem implícita tornar-se uma necessidade.

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

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 3.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir um caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 2

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>
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>
Converter uma <i>string</i> em um <i>int</i>	Chame o método <i>System.Int32.Parse</i> . Por exemplo: <code>System.Int32.Parse("42");</code>
Sobrescrever 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 diferentes variáveis	Use uma instrução de atribuição que lista 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

Escrevendo métodos e aplicando 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, “Trabalhando com variáveis, operadores e expressões”, você aprendeu a declarar variáveis, a criar expressões utilizando operadores, e entendeu de que modo a precedência e a associatividade controlam a maneira como as expressões que contêm múltiplos operadores são avaliadas. Neste capítulo, você aprenderá sobre os métodos. Aprenderá também a utilizar os argumentos e parâmetros para passar informações para um método e a retorná-las empregando as instruções de retorno. Por fim, verá como entrar e sair dos métodos usando o depurador integrado do Microsoft Visual Studio 2010. Essas informações são úteis quando você precisa rastrear a execução dos seus métodos se eles não funcionam conforme o esperado.

Criando métodos

Um *método* é uma sequência nomeada de instruções. Se você já utilizou linguagens como 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 é chamado. Além disso, os métodos podem receber alguns dados para serem processados e retornar informações, que normalmente são o resultado do processamento, caracterizando-se como um mecanismo poderoso e fundamental.

Declarando um método

A sintaxe para declarar um método C# é:

```
tipoDeRetorno nomeDoMétodo (listaDeParâmetros)
```

```
{  
    // instruções do corpo do método entram aqui  
}
```

- 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 identificadoras 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 os 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 as chaves de abertura e de fechamento { }.



Importante Os programadores C, C++ e Microsoft Basic devem notar que o C# não suporta métodos globais. Você deve escrever todos seus métodos dentro de uma classe ou seu código não compilará.

Eis 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 entram aqui  
    // ...  
}
```



Nota Você deve especificar explicitamente os tipos de qualquer parâmetro 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 Os programadores em Visual Basic devem notar 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*.

Retornando 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 de retorno no final do processamento do método. Uma instrução de retorno consiste em uma palavra-chave *return* seguida por uma expressão que especifica 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 de retorno deve retornar um *int*; caso contrário, o programa não compilará. Eis um exemplo de um método com uma instrução *return*:

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

A instrução *return* geralmente está no final do método porque o faz terminar e controla os retornos para a instrução que chamou o método, como descrito posteriormente neste capítulo. Todas 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 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 uma outra versão do projeto MathsOperators do Capítulo 2. Essa versão foi aprimorada pela utilização cuidadosa de alguns pequenos métodos.

Examine as definições de método

1. Inicialize o Visual Studio 2010 se ainda não estiver em execução.
2. Abra o projeto *Methods* na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 3\Methods na sua pasta *Documentos*.
3. No menu *Debug*, clique em *Start Without Debugging*.
O Visual Studio 2010 compila e executa o aplicativo.
4. Explore o aplicativo e o modo como ele funciona, e clique em *Quit*.
5. Exiba o código de MainWindow.xaml.cs na janela *Code and Text Editor*.
6. Na janela *Code and Text Editor*, localize o método *addValues*.

O método é semelhante a este:

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

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

A segunda utiliza o 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*.

O método *showResult* é semelhante a 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 de texto *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 ajudar 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 que uma tela, considere a possibilidade de dividi-lo em métodos menores para torná-lo mais legível.

Chamando métodos

Os métodos existem para ser 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.

Especificando a sintaxe de chamada de método

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

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

- 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* = o método retornar um valor, o método será executado, mas o valor de retorno será descartado.
- A *listaDeArgumentos* oferece informações opcionais 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, sem parênteses
addValues();         // erro de tempo de compilação, sem argumentos suficientes
addValues(39);       // erro de tempo de compilação, sem argumentos suficientes
addValues("39", "3"); // erro de tempo de compilação, tipos 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 a analisar 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 2010, se você estiver continuando do exercício anterior. Se não, abra-o na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 3\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. (Abordaremos a finalidade das instruções *try* no Capítulo 6, “Gerenciando erros e exceções”.)

As instruções são:

```
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*. Entretanto, as partes interessantes são como as variáveis são inicializadas. Em ambos os casos, o método *Parse* da classe *System.Int32* é chamado. (*System* é um namespace, e *Int32* é o nome da classe 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 de caixa de texto *lhsOperand* e *rhsOperand* no 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 seus 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 seu 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 do 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 chamado *ToString* pertencente ao objeto chamado *answer*.

Aplicando escopo

Em alguns exemplos, você pode ver que é possível criar variáveis dentro de um método. Essas variáveis passam a existir a partir do ponto em que elas são definidas, e as instruções subsequentes no mesmo método podem então utilizá-las; uma variável só pode ser explorada depois de ser criada. Quando o método termina, essas variáveis desaparecem.

Se uma variável pode ser empregada em um local específico em um programa, dizemos que ela está no *escopo* desse local. 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 verá agora.

Definindo o escopo local

As chaves de abertura e fechamento que formam o corpo de um método definem um escopo. 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 outro método. Essa estrutura significa que você não pode utilizar as variáveis locais para compartilhar informações entre os 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*, ocorrendo depois que a linha do código é declarada como *myVar*.

Definindo o escopo de classe

As chaves de abertura e fechamento que formam o corpo de uma classe também criam um escopo. Todas as variáveis que você declara dentro do corpo de uma classe (mas não dentro de um método) estão no escopo dela. O nome apropriado do C# para as variáveis definidas por uma classe é *field* (campo). Ao contrário das variáveis locais, os campos podem ser utilizados para compartilhar informações entre métodos. A seguir, 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 na 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 diferente. Um método pode utilizar um campo antes da instrução que define o campo – o compilador resolve os detalhes para você!

Sobrecregando métodos

Se dois identificadores têm o mesmo nome e são declarados no mesmo escopo, dizemos que eles estão *sobrecregados* (*overloaded*). Um identificador sobrecregido costuma ser um erro capturado como um erro de tempo de compilação. Por exemplo, se 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, você também receberá um erro de tempo de compilação. Não vale a pena mencioná-lo, uma vez que tudo que vimos até aqui tem resultado em um erro de tempo de compilação. Mas há uma maneira útil e importante pela qual você pode sobrecregar um identificador.

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#, você notará 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*); e 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 chama a versão do método que tem o conjunto de parâmetros correspondente. Segue um exemplo:

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

A sobrecrença é útil principalmente quando você precisa executar a mesma operação em diferentes tipos de dados. Você pode sobrecregar 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. Esse recurso é permitido para que, quando você chamar um método, possa fornecer uma lista de argumentos separados por vírgula; e o número e o tipo dos argumentos sejam utilizados pelo compilador para selecionar um dos métodos sobrecarregados. Mas observe que, embora possa sobreregar os parâmetros de um método, você não pode sobreregar 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).

Escrevendo métodos

Nos exercícios a seguir, você criará 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 Console para ter uma ideia do programa. Por fim, você irá explorar o depurador do Visual Studio 2010 para entrar e sair das chamadas de método à medida que elas são executadas.

Desenvolva a lógica do aplicativo

1. Utilizando o Visual Studio 2010, abra o projeto DailyRate na pasta \Microsoft Press\VisualCSharp Step By Step\Chapter 3\DailyRate na sua pasta Documentos.
2. No *Solution Explorer*, clique duas vezes no arquivo *Program.cs* para exibir o programa na janela *Code and Text Editor*.
3. Adicione as seguintes instruções ao corpo do método *run*, entre as chaves de abertura e de fechamento:

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

O método *run* é chamado pelo método *Main* quando o aplicativo inicia. (A maneira como ele é chamado requer um entendimento das classes, o que examinaremos no Capítulo 7, “Criando e gerenciando classes e objetos”.)

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 para o 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.

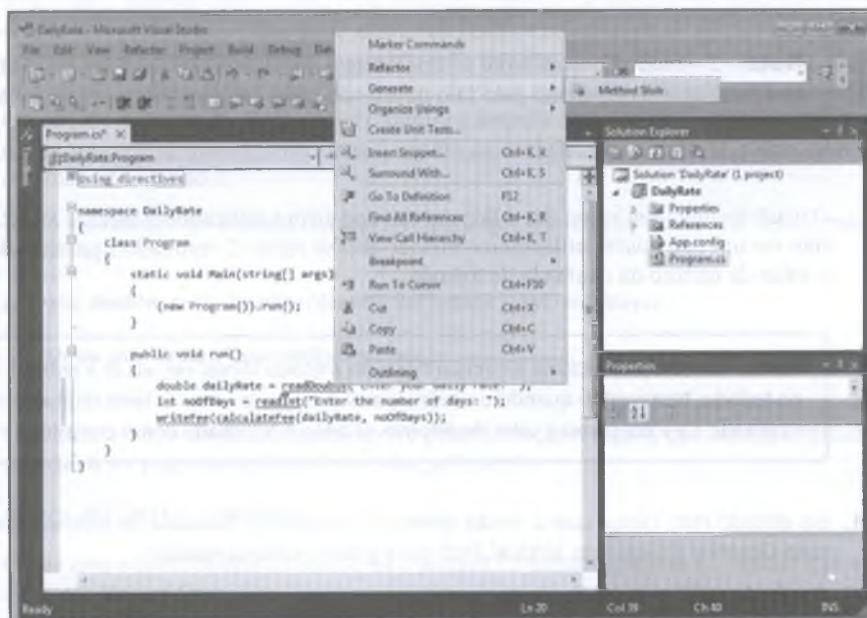


Nota Você ainda não escreveu os métodos `readDouble`, `readInt`, `writeFee` ou `calculateFee`, portanto, o IntelliSense não exibirá esses métodos quando você digitar esse código. Não tente compilar o aplicativo ainda, pois ele falhará.

Escreva os métodos utilizando o assistente Generate Method Stub

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

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



- 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 *NotImplementedException* (exceções são descritas no Capítulo 6). Você substituirá o corpo pelo seu próprio código na próxima etapa.

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

```
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 e contém uma mensagem solicitando que o usuário digite a taxa diária.



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

O usuário digita um valor, que 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 a seguir:

```
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 o nome dos argumentos passados para gerar os nomes dos parâmetros (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. Discutiremos o tipo *object* em mais detalhes no Capítulo 7.

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 a seguir, que calcula e retorna a remuneração a ser paga multiplicando os dois parâmetros:

```
return dailyRate * noOfDays;
```

9. Clique com o botão direito do mouse na chamada do método *writeFee* no método *run* e clique em *Generate Method Stub*.

Observe que o Visual Studio utiliza a definição do método *calculateFee* para concluir que seu parâmetro deve ser um *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*.

- Digite as instruções a seguir dentro do método *writeFee*:

```
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 para converter o valor da expressão *p * 1.1* em uma string e utilizar o operador + para concatená-la à mensagem.

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

Refatorando o código

Um recurso muito útil do Visual Studio 2010 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, realce o bloco de código que você acabou de digitar e, no menu *Refactor*, clique em *Extract Method*. A caixa de diálogo *Extract Method* é exibida, 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. O *Extract Method* também é inteligente o bastante para descobrir se o método deve ter algum parâmetro e retornar um valor.

Teste o programa

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

O Visual Studio 2010 compila o programa e o executa. Uma janela de console é exibida.

- No prompt *Enter your daily rate*, digite **525** e pressione Enter.
- 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

- Pressione a tecla Enter para fechar o aplicativo e retornar ao ambiente de programação do Visual Studio 2010.

No próximo exercício, você vai utilizar o depurador do Visual Studio 2010 para executar seu programa lentamente. Você verá quando cada método é chamado (o que é citado como *stepping into of the method* ou “entrar no método”) e como cada instrução de retorno 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 vai 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 *Debug*.

Inspecione os métodos passo a passo utilizando o depurador do Visual Studio 2010

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

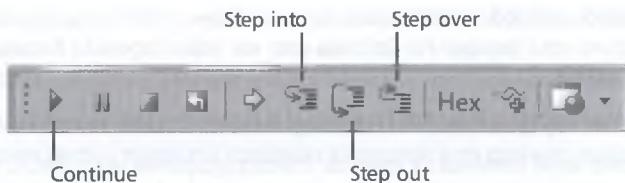
```
double dailyRate = readDouble("Enter your daily rate: ");
```

3. Clique com o botão direito do mouse em qualquer lugar dessa linha e, no menu de atalho, 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, que também é realçada por um segundo plano amarelo.

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* se parece a esta (embora a barra de ferramentas seja um pouco diferente entre o Visual Studio 2010 e o Microsoft Visual C# 2010 Express, ela não contém o botão Breakpoint no lado direito):



Dica Para fazer a barra de ferramentas *Debug* ser exibida na sua própria janela, utilize a alça na extremidade esquerda da barra de ferramentas e arraste-a sobre a janela *Code and Text Editor*.

5. Na barra de ferramentas *Debug*, clique no botão *Step Into* (é o sexto botão à esquerda).

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 novamente em *Step Into*. O cursor avança para 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 sétimo botão à esquerda).

Essa ação faz o método executar a próxima instrução sem depurá-la (sem entrar nele). O cursor amarelo se move para a segunda instrução do método, e o programa exibe o prompt *Enter your daily rate* em uma janela Console antes de retornar ao Visual Studio 2010 (a janela Console talvez esteja 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 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 Console e pressione Enter.

O controle retorna ao Visual Studio 2010. 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 oitavo botão à esquerda).

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



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 escolheu executar o método sem fazer a inspeção passo a passo. A janela Console aparece novamente solicitando o número de dias.

14. Na janela Console, digite **17** e pressione Enter.

O controle retorna ao Visual Studio 2010. 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*.

O cursor amarelo pula de volta 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 a variável *p* na definição do método.

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

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

A mensagem *The consultant's fee is: 9817.5* é exibida na janela Console (você pode precisar abrir a janela Console no primeiro plano para exibi-la se ela estiver atrás do Visual Studio 2010). O cursor amarelo retorna à terceira instrução do método *run*.

20. Na barra de ferramentas *Debug*, clique em *Continue* (o primeiro botão na barra de ferramentas) para fazer o programa continuar a executar sem parar em cada instrução.



Dica Você também pode pressionar F5 para continuar a execução no depurador.

O aplicativo termina e para de executar.

Utilizando 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 ele 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, existem outras linguagens e tecnologias que os desenvolvedores podem utilizar para construir aplicativos Windows e componentes que não seguem essas regras. Um recurso importante do C# e de outras linguagens elaboradas para o .NET Framework é a possibilidade de interoperar com os aplicativos e componentes escritos em outras tecnologias. Uma das principais tecnologias utilizadas pelo Microsoft Windows é o Component Object Model ou COM, que não oferece suporte para métodos sobrecarregados, mas utiliza métodos que aceitam 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 os parâmetros opcionais.

Os parâmetros opcionais também são úteis em outras situações. Eles representam uma solução compacta e simples, quando não é possível utilizar a 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 *ints* e um *float*. Vamos supor que você queria fornecer uma implementação do método *DoWorkWithData* que aceite apenas dois parâmetros: *intData* e *floatData*. Você pode sobrepor 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 sobreposição a ser chamada:

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

DoWorkWithData(arg1, arg2, arg3); // Chamar a sobreposição com três parâmetros
DoWorkWithData(arg1, arg2);      // Chamar a sobreposição 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)
{
    ...
}
```

A questão é que, para o compilador, essas duas sobrecargas parecem idênticas e 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 era 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.

Definindo 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 ele 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, e 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 de "Hello" quando o método é executado.

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

Passando argumentos nomeados

Por padrão, o C# utiliza a posição de cada argumento na chamada a um método para determinar os parâmetros aos quais eles são aplicáveis. 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. O C# também permite especificar os parâmetros pelo nome, e esse recurso deixa você passar os argumentos em uma sequência diferente. Para passar um argumento como um parâmetro nomeado, inclua o nome do parâmetro, 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 a seguir:

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

Esse recurso também permite 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
```

Resolvendo 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 tenha definido o método *optMethod* como um método sobre carregado, 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 )
{
    . . .
}
```

Este 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, 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, este código vai deixá-lo intrigado!

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

Dessa 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, o terceiro e o 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. O compilador determina que essa é uma chamada de método ambígua e 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 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 2010, abra o projeto DailyRate, armazenado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 3\DailyRate Using Optional Parameters, dentro da pasta Documentos.
2. No *Solution Explorer*, clique duas vezes no arquivo *Program.cs* para exibir o código do programa na janela *Code and Text Editor*.
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 ele 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 sobre carregadas 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 demonstrado 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, que chamam o *calculateFee* e exibem os resultados:

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

7. No menu *Debug*, clique em *Start Without Debugging*, para construir e executar o programa. O programa é executado em uma janela do 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. Isso acontece porque essa é a versão que mais se aproxima à 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 o *calculateFee*, conforme indicado em negrito no seguinte exemplo de código:

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

9. No menu *Debug*, clique em *Start Without Debugging*, para construir e executar o programa. O programa exibe as seguintes mensagens:

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

Dessa vez, o método *run* chamou a versão de *calculateFee* que aceita um único parâmetro opcional. Como anteriormente, 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 construir e executar o programa. 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 o *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 construir e executar o programa. O programa exibe as seguintes mensagens:

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

Como anteriormente, o método *run* chamou 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 o *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 construir e executar o programa. O programa exibe as seguintes mensagens:

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

Dessa 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. Esta é a única versão do método *calculateFee* que corresponde à 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 = 5)
{
    Console.WriteLine("calculateFee using two optional parameters");
    return theDailyRate * noOfDays;
}
```

17. No método *run*, modifique a instrução que chama o *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 construir e executar o programa. O programa exibe as seguintes mensagens:

```
calculateFee using two optional parameters
Fee is 1875
```

Quando você especificou a taxa, mas não a taxa diária (etapa 13), o método *run* chamou a versão de *calculateFee* que aceita um único parâmetro opcional. Dessa 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. Pressione qualquer tecla para fechar a janela do console e retornar ao Visual Studio.

Neste capítulo, você aprendeu a definir métodos para implementar um bloco de código nomeado. Você examinou como passar parâmetros para os métodos e como retornar dados dos métodos. Você também viu como chamar um método, passar argumentos e obter um valor de retorno. Você 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 2010 para passar pelo código ao longo de sua execução. Finalmente, você aprendeu a escrever métodos que aceitam parâmetros opcionais e a chamar métodos por meio de parâmetros nomeados.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 4.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 3

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

Capítulo 4

Utilizando 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.

No Capítulo 3, “Escrevendo métodos e aplicando escopo”, você aprendeu a agrupar as instruções relacionadas em métodos. Também aprendeu a utilizar parâmetros para passar informações para um método e a utilizar as instruções *return* para passar informações a partir de um método. Dividir um programa em um conjunto de métodos distintos, cada um deles projetado para executar uma tarefa ou cálculo específico, é uma estratégia necessária. Muitos programas precisam resolver problemas grandes e complexos. A divisão de um programa em métodos ajuda a entender esses problemas e a focar a solução de uma parte a cada vez. Você pode precisar escrever métodos que executem diferentes ações, dependendo das circunstâncias. Neste capítulo, você verá como realizar essa tarefa.

Declarando variáveis booleanas

No mundo da programação 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, “A variável *x* contém o valor 99?”, a resposta será *true*. Se você perguntar “*x* é menor que 10?”, a resposta será *false*. 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 essa são uma fonte comum de erros nos programas 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 Microsoft Visual C# fornece um tipo de dados chamado *bool*. Uma variável *bool* pode armazenar um dos dois valores: *true* ou *false*. 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
```

Utilizando 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, sendo o mais simples deles o operador NOT, que é 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 *false*.

Entendendo operadores de igualdade e relacionais

Dois operadores booleanos utilizados com frequência são os operadores de igualdade (==) e desigualdade (!=). Esses operadores binários são utilizados para descobrir se um valor é igual a outro valor de mesmo tipo. 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

Intimamente ligados a esses dois operadores estão os operadores *relacionais*. 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

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.

Entendendo operadores lógicos condicionais

O C# também fornece dois outros operadores booleanos: o operador lógico AND, que é representado pelo símbolo `&&`, e o operador lógico OR, que é representado pelo símbolo `||`. Coletivamente, eles são conhecidos como os operadores lógicos condicionais. O propósito dessa expressão é combinar duas expressões ou valores booleanos em um único resultado booleano. Esses operadores binários 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 devem ser verdadeiros ou falsos.

O resultado do operador `&&` será *true* se e somente se as duas expressões booleanas em que ele opera 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 estas duas expressões:

```
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 em que ele opera 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.

Resumindo a precedência e a 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
Primário	<code>()</code>	Substitui precedência	Esquerda
	<code>++</code>	Sufixo de incremento	
	<code>--</code>	Sufixo de decremeno	
Unário	<code>!</code>	NOT lógico	Esquerda
	<code>+</code>	Adição	
	<code>-</code>	Subtração	
	<code>++</code>	Prefixo de incremento	
	<code>--</code>	Prefixo de decremeno	

Categoría	Operadores	Descrição	Associatividade
Multiplicativo	*	Multiplicação	Esquerda
	/	Divisão	
	%	Resto da divisão	
Aditivo	+	Adição	Esquerda
	-	Subtração	
Relacional	<	Menor que	Esquerda
	<=	Menor ou igual a	
	>	Maior que	
	>=	Maior ou igual a	
Igualdade	==	Igual a	Esquerda
	!=	Diferente de	
AND condicional	&&	AND lógico	Esquerda
OR condicional		OR lógico	Esquerda
Atribuição	=		Direita

Utilizando instruções *if* para tomar decisões

Se você quiser escolher entre executar dois blocos diferentes de código com base no resultado de uma expressão booleana, utilize uma instrução *if*.

Entendendo a sintaxe da instrução *if*

A sintaxe de uma instrução *if* é a seguinte (*if* e *else* são palavras-chave do C#):

```
if ( expressãoBoolena )
    instrução-1;
else
    instrução-2;
```

Se a *expressãoBoolena* for avaliada como *true*, a *instrução-1* é executada; caso contrário, a *instrução-2* é executada. A palavra-chave *else* e a *instrução-2* subsequente são opcionais. Se não houver uma cláusula *else* e a *expressãoBoolena* for *false*, a execução continua com o código que vem depois da instrução *if*.

Por exemplo, eis 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 utilizando o 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 irá compilar seu código. Por exemplo:

```
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). O resultado é que o código após a instrução *if* é executado 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 muito usado  
***  
if (inWord) // mais comum e considerado um estilo melhor
```

Utilizando 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 realizar mais de uma instrução quando uma expressão booleana for verdadeira. As instruções poderão ser agrupadas 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. Um bloco também inicia um novo escopo. As variáveis podem ser definidas dentro de um bloco, mas elas desaparecerão no final do bloco.

No exemplo a seguir, duas instruções que redefinem a variável *seconds* como 0 e incrementam a variável *minutes* estão agrupadas em um bloco, e o bloco inteiro é 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 e informará um erro de sintaxe.

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 alcançada somente se o primeiro teste for *false*. Da mesma forma, a terceira instrução *if* só será avaliada se o primeiro e o segundo teste 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. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver em execução.
2. Abra o projeto Selection, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter4\Selection na sua pasta Documentos.
3. No menu *Debug*, clique em *Start Without Debugging*.

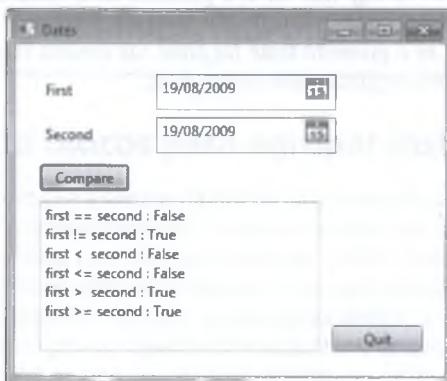
O Visual Studio 2010 compila e executa o aplicativo. O formulário contém dois controles *Date TimePicker* chamados *first* e *second* (esses controles exibem um calendário permitindo que você selecione uma data ao clicar no ícone). Esses dois controles são inicialmente configurados com a data atual.

4. Clique em *Compare*.

O texto a seguir é exibido na caixa de texto:

```
first == second : False
first != second : True
first < second : False
first <= second : False
first > second : True
first >= second : True
```

A expressão booleana *first == second* deve ser *true* porque tanto *first* quanto *second* estão configurados como a data atual. De fato, somente o operador “menor que” e o operador “maior que” ou “igual a” parecem funcionar corretamente.



5. Clique em *Quit* para retornar ao ambiente de programação do Visual Studio 2010.
6. Exiba o código de MainWindow.xaml.cs na janela *Code and Text Editor*.

7. Localize o método *compareClick*, que é semelhante a este:

```
private void compareClick(object sender, RoutedEventArgs e)
{
    int diff = dateCompare(first.SelectedDate.Value, second.SelectedDate.Value);
    info.Text = "";
    show("first == second", diff == 0);
    show("first != second", diff != 0);
    show("first < second", diff < 0);
    show("first <= second", diff <= 0);
    show("first > second", diff > 0);
    show("first >= second", diff >= 0);
}
```

Esse método é executado sempre que o usuário clica no botão *Compare* do formulário. Ele recupera os valores de datas exibidos nos controles *DateTimePicker* *first* e *second* no formulário. A data selecionada pelo usuário em cada controle *DateTimePicker* fica disponível na propriedade *SelectedDate*. Para recuperar a data, use a propriedade *Value* dessa propriedade. (Você conhecerá mais detalhes sobre propriedades no Capítulo 15, “Implementando propriedades para acessar campos”.) O tipo dessa propriedade é *DateTime*. O tipo de dado *DateTime* é apenas mais um tipo de dado, como *int* ou *float*, exceto pelo fato de que contém subelementos que permitem acessar as partes individuais de uma data, como ano, mês ou dia.

O método *compareClick* passa os dois valores de *DateTime* para o método *dateCompare*, que os compara. Examinaremos o método *dateCompare* no próximo passo.

O método *show* resume os resultados da comparação no controle de caixa de texto *info* do formulário.

8. Localize o método *dateCompare*, que se parece com este:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    // TO DO
    return 42;
}
```

Esse método atualmente retorna o mesmo valor sempre que é chamado, em vez de 0, -1 ou +1, dependendo dos valores de seus parâmetros. Isso explica por que o aplicativo não funciona conforme o esperado!

O propósito desse método é examinar os argumentos e retornar um valor inteiro com base nos valores relativos; ele deve retornar 0 se os argumentos tiverem o mesmo valor, -1 se o valor do primeiro argumento for menor que o valor do segundo argumento e +1 se o valor do primeiro argumento for maior que o valor do segundo argumento (uma data é considerada maior que outra se vier antes dela cronologicamente). Você precisa implementar a lógica nesse método para comparar duas datas corretamente.

9. Remova o comentário *// TO DO* e a instrução *return* do método *dateCompare*.*

* N. de R.T.: O comentário “TO DO”, que significa “A FAZER”, é reconhecido automaticamente pelo Visual Studio.

10. 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`. Se a expressão `leftHandSide.Year > rightHandSide.Year` for *true*, a data em `leftHandSide` deve ser posterior à data em `rightHandSide`, e 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.

11. 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 semelhante para comparar meses 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.

12. 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`.

13. No menu *Debug*, clique em *Start Without Debugging*.

O aplicativo é recompilado e reiniciado. Mais uma vez, os dois controles `DateTimePicker`, `first` e `second`, são definidos com a data de hoje.

14. Clique em *Compare*.

O texto a seguir é exibido na caixa de texto:

```

first == second : True
first != second : False
first < second : False
first <= second : True
first > second : False
first >= second : True

```

Esses são os resultados corretos para datas idênticas.

15. Clique no ícone para o segundo controle `DateTimePicker` e clique na data de amanhã no calendário exibido.

16. Clique em *Compare*.

O texto a seguir é exibido na caixa de texto:

```

first == second : False
first != second : True
first < second : True
first <= second : True
first > second : False
first >= second : False

```

Mais uma vez, esses são os resultados corretos quando a primeira data é anterior à segunda data.

17. Teste algumas outras datas e verifique se os resultados são os esperados. Clique em *Quit* depois de terminar.

Comparando 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 utilizaria para comparar datas em um aplicativo real. Na biblioteca de classes do Microsoft .NET Framework, as datas são armazenadas utilizando um tipo especial chamado *DateTime*. Se examinar o método *Compare* que você escreveu no exercício anterior, verá que os dois parâmetros, *leftHandSide* e *rightHandSide*, são valores *DateTime*. A lógica escrita só compara a parte da data dessas variáveis – também há um elemento hora. 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 predefinido chamado *Compare* para fazer exatamente isso. O método *Compare* 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.

Utilizando instruções *switch*

Algumas vezes, ao escrever uma instrução *if* em cascata, todas as instruções *if* aparecem 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.

Entendendo 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* é avaliada uma vez. O controle passa então para o bloco do código identificado pela *expressãoConstante*, cujo valor é igual ao resultado da *expressãoDeControle* (o identificador é chamado de *rótulo de caso*). A execução prossegue até a instrução *break* e, então, a instrução *switch* termina, e o programa continua a partir da 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 são executadas.

Nota Cada valor da *expressãoConstante* deve ser único, assim a *expressãoDeControle* só corresponderá a um deles. Se o valor de *expressãoDeControle* não corresponder a nenhum valor de *expressãoConstante* e não houver um rótulo *default*, a execução do programa continuará na primeira instrução após a chave de fechamento da instrução *switch*.

Por exemplo, 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;
}
```

Seguindo as regras da instrução *switch*

A instrução *switch* é muito útil, mas, infelizmente, nem sempre você poderá utilizá-la da maneira desejada. Todas as instruções *switch* que você escrever devem obedecer às seguintes regras:

- A instrução *switch* só pode ser utilizada em tipos de dados primitivos, como *int* ou *string*. Com qualquer outro tipo (incluindo *float* e *double*), você terá de utilizar uma instrução *if*.
- Os rótulos de caso devem ser expressões constantes, como 42 ou "42". Se for necessário calcular valores dos rótulos de caso em tempo de execução, utilize uma instrução *if*.
- Os rótulos de caso devem ser expressões únicas. Ou seja, dois rótulos de caso 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 para o 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, e o compilador gerará um erro. Por exemplo:

```
switch (trumps)
{
    case Hearts :
    case Diamonds :      // Fall-through permitido – nenhum código entre 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* ou *throw*. A instrução *throw* será descrita no Capítulo 6, "Gerenciando erros e exceções".

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 no texto "<" de modo 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*.

Regras de fall-through da instrução *switch*

Como você não pode passar accidentalmente de um rótulo de caso 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, normalmente é posicionado como o último rótulo, mas não é obrigatório).

Os programadores 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: é comum em programas C ou C++ instrução *break* ser esquecida, permitindo que a execução prossiga (*fall through*) para o próximo rótulo, originando erros que são 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!

Escreva instruções *switch*

1. Inicialize o Visual Studio 2010 se ainda não estiver em execução.
2. Abra a pasta SwitchStatement project, localize a pasta \Microsoft Press\Visual Step by CSharp Step by Step\Chapter 4\SwitchStatement na sua pasta Documentos.
3. No menu *Debug*, clique em *Start Without Debugging*.

O Visual Studio 2010 compila e executa o aplicativo, que 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 >.

6. Feche o formulário e retorne ao Visual Studio 2010.

7. Exiba o código para 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 para o 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 seguintes instruções à instrução *switch* depois da chave de abertura da instrução e imediatamente antes do rótulo *default*:

```
case '<' :  
    target.Text += "&lt;";  
    break;
```

Se o caractere que está sendo copiado for um <, esse código acrescentará a string "<" ao texto que está sendo gerado no lugar dele.

9. Adicione as seguintes instruções à instrução *switch* depois da instrução *break* que você recém adicionou, 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# e na XML – elas são utilizadas para delimitar constantes de caracteres e strings. As barras invertidas (\) no final dos dois rótulos de 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 Without 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) &amp; &amp; (hi >= number)
```

13. Teste outras strings e verifique se todos os caracteres especiais (<, >, &, “ e ’) são tratados corretamente.

14. Feche o formulário.

Neste capítulo, você conheceu expressões booleanas e variáveis; 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 você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 5.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto

Referência rápida do Capítulo 4

Para	Faça isto	Exemplo
Determinar se dois valores são equivalentes	Utilize o operador == ou !=.	answer == 42
Comparar o valor de duas expressões	Utilize o operador <, <=, > ou >=.	age >= 21
Declarar uma variável booleana	Utilize a palavra-chave <i>bool</i> como o tipo da variável.	bool inRange;
Criar uma expressão booleana que seja verdadeira apenas se as duas outras condições forem verdadeiras	Utilize o operador &&.	inRange = (lo <= number) && (number <= hi);
Criar uma expressão booleana que seja verdadeira se uma das duas outras condições for verdadeira	Utilize o operador .	outOfRange = (number < lo) (hi < number);
Executar uma instrução se uma condição for verdadeira	Use uma instrução <i>if</i> .	if (inRange) process();
Executar mais de uma instrução se uma condição for verdadeira	Utilize uma instrução <i>if</i> e um bloco.	if (seconds == 59) { seconds = 0; minutes++; }
Associar diferentes instruções a diferentes valores de uma expressão de controle	Use uma instrução <i>switch</i> .	switch (current) { case 0: * * * break; case 1: * * * break; default : * * * break; }

Capítulo 5

Utilizando 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 as instruções de iteração (ou repetição) *while*, *for* e *do*.
- Iinspecionar uma instrução *do* passo a passo e ver como os valores de variáveis mudam.

No Capítulo 4, “Utilizando instruções de decisão”, você aprendeu a usar as construções *if* e *switch* para executar as instruções seletivamente. Neste capítulo, você verá como utilizar várias instruções de iteração (*loop*) para executar uma ou mais instruções repetidamente. Ao escrever as instruções de iteração, normalmente você precisa controlar o número de iterações que serão executadas. Isso é feito por meio de uma variável que atualiza seu valor a cada iteração e para o processo quando a variável atinge um valor específico. Você aprenderá também sobre operadores de atribuição especiais que devem ser utilizados para atualizar o valor de uma variável nessas circunstâncias.

Utilizando 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 adicionar 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# permite executar

essa tarefa de maneira mais rápida utilizando o operador `+=`. Para adicionar 42 a `answer`, escreva esta instrução:

```
answer += 42;
```

Utilize esse atalho 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 que os operadores de atribuição simples.

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.



Nota 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++;
```

Escrevendo 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 ( expressãoBooleana )
    instrução
```

A expressão booleana 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 é repetida e então a expressão booleana é avaliada de novo. Esse processo continua até que a expressão booleana seja avaliada como *false*, momento em que a instrução *while* termina. A execução então continua com a primeira instrução depois da instrução *while*. Uma instrução *while* compartilha muitas semelhanças semânticas 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 você quiser executar duas ou mais instruções sob o controle de uma instrução *while*, deve utilizar chaves para agrupar essas instruções em um bloco.

Observe uma instrução *while* que escreve os valores de 0 a 9 no console:

```
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 esquecer de incluir uma instrução para fazer a expressão booleana ser, por fim, avaliada como *false* 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 *Sentinel*.

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 para uma caixa de texto em um formulário.

Escreva uma instrução *while*

1. Utilizando o Microsoft Visual Studio 2010, abra o projeto *WhileStatement*, localizado na pasta *\Microsoft Press\Visual CSharp Step by Step\Chapter 5\WhileStatement* na sua pasta Documentos.
2. No menu *Debug*, clique em *Start Without Debugging*.

O Visual Studio 2010 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*.

A caixa de diálogo *Open* se abre.

4. Abra a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 5\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 pequena, no formulário, mas o conteúdo do arquivo MainWindow.xaml.cs 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. Feche o formulário e retorne ao Visual Studio 2010.

7. Exiba o código para o arquivo MainWindow.xaml.cs na janela *Code and Text Editor* e localize o método *openFileDialogFileOk*.

Esse método é chamado quando o usuário clica no botão *Open* após selecionar um arquivo na caixa de diálogo *Open*. O corpo do método está atualmente implementado da seguinte maneira:

```
private void openFileDialogFileOk(object sender, System.ComponentModel.CancelEventArgs e)
{
    string fullpathname = openFileDialog.FileName;
    FileInfo src = new FileInfo(fullpathname);
    filename.Text = src.Name;

    // adicione o loop while aqui
}
```

A primeira instrução declara uma variável *string* chamada *fullpathname* e a inicializa para a propriedade *FileName* do objeto *openFileDialog*. Essa propriedade contém o nome completo (incluindo a pasta) do arquivo-fonte selecionado na caixa de diálogo *Open*.



Nota O objeto *openFileDialog* é uma instância da classe *OpenFileDialog*. Essa classe fornece métodos que podem ser utilizados para exibir a caixa de diálogo Open padrão do Windows, selecionar um arquivo e recuperar o nome e caminho do arquivo selecionado. Esta é uma das diversas classes fornecidas na Biblioteca de Classes do .NET Framework que você pode utilizar para executar tarefas comuns, que exigem a seleção de um arquivo. Essas classes são conhecidas genericamente como classes de Diálogo Comum. Você conhecerá mais detalhes sobre elas no Capítulo 23, "Obtendo a entrada do usuário".

A segunda instrução declara uma variável *FileInfo* chamada *src* e a inicializa para um objeto que representa o arquivo selecionado na caixa de diálogo *Open*. (*FileInfo* é uma classe fornecida pelo Microsoft .NET Framework que pode ser utilizada para manipular arquivos.)

A terceira instrução atribui a propriedade *Text* do controle *filename* à propriedade *Name* da variável *src*. A propriedade *Name* da variável *src* contém o nome do arquivo selecionado na caixa

de diálogo *Open*, sem o nome da pasta. Essa instrução exibe o nome do arquivo na caixa de texto do formulário.

8. Substitua o comentário *// adicione um loop while aqui (//add while loop here)* 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.

9. Digite a seguinte instrução depois da linha que você acabou de adicionar para o método *openFileDialogOk*:

```
TextReader reader = src.OpenText();
```

Essa instrução declara uma variável *TextReader* chamada *reader*. *TextReader* é outra classe, disponibilizada pelo .NET Framework, que pode ser utilizada para ler fluxos de caracteres a partir de fontes como arquivos. Ela está localizada no namespace *System.IO*. A classe *FileInfo* fornece o método *OpenText* para abrir um arquivo para leitura. Essa instrução abre o arquivo selecionado pelo usuário na caixa de diálogo *Open* de modo que a variável *reader* possa ler o conteúdo desse arquivo.

10. Adicione a seguinte instrução depois da linha anterior que você adicionou ao método *openFileDialogOk*:

```
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 ou um valor especial chamado *null* caso não haja mais linhas para ler. (Se inicialmente não houver linha alguma, o arquivo deve estar vazio.)

11. Adicione as seguintes instruções ao método *openFileDialogOk* 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 ele não for nulo, 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* então lê a próxima linha de texto antes de realizar a próxima iteração. O loop *while* termina quando não há mais texto no arquivo e o método *ReadLine* retorna um valor *null*.

12. Adicione a seguinte instrução depois da chave de fechamento no fim do loop *while*:

```
reader.Close();
```

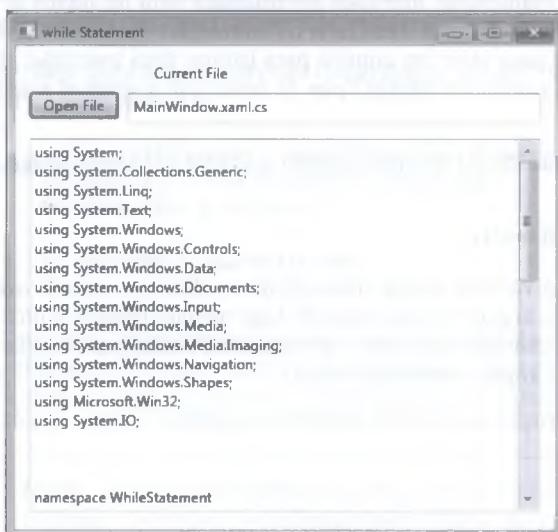
Essa instrução fecha o arquivo. É uma prática recomendada fechar os arquivos após utilizá-los; esse procedimento permite que outros aplicativos utilizem o arquivo, além de liberar memória e outros recursos necessários para a leitura do arquivo.

13. No menu *Debug*, clique em *Start Without Debugging*.

14. Quando o formulário aparecer, clique em *Open File*.

15. Na caixa de diálogo *Open File*, abra a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 5\WhileStatement na sua pasta Documentos. Selecione o arquivo MainWindow.xaml.cs e clique em *Open*.

Desta vez, o conteúdo do arquivo selecionado aparece na caixa de texto – você deve reconhecer o código que acabou de editar:



16. Role pelo texto na caixa de texto e localize o método *openFileDialogFileOk*. Verifique se esse método contém o código que você acabou de adicionar.

17. Feche o formulário e retorne ao ambiente de programação do Visual Studio 2010.

Escrevendo instruções *for*

A maioria das instruções *while* tem a seguinte estrutura geral:

```
inicialização
while (expressão booleana)
{
    instrução
    atualização da variável de controle
}
```

Com uma instrução *for*, você pode escrever uma versão mais formal desse tipo de construção combinando a inicialização, a expressão booleana e o código que atualiza a variável de controle. Você achará a instrução *for* útil porque é muito mais difícil de esquecer qualquer uma de suas três partes. Observe a sintaxe da instrução *for*:

```
for (inicialização; expressão booleana; atualização da variável de controle)
    instrução
```

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 uma vez 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 verdadeira, 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.

Você pode omitir qualquer uma das três partes de uma instrução *for*. Se você omitir a expressão booleana, ela assumirá *true* por padrão. A instrução *for* a seguir é executada continuamente:

```
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 vírgulas, mesmo quando omitidas.

Se necessário, é possível fornecer várias inicializações e várias atualizações em um loop *for*. (Você pode ter somente uma expressão booleana.) Para conseguir isso, separe as várias inicializações e atualizações por vírgula, como mostrado no exemplo a seguir:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
    ...
}
```

Como um exemplo final, 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';
}
```

 **Dica** Considera-se um bom estilo utilizar chaves para delinear explicitamente o bloco de instrução para o corpo das instruções *if*, *while* e *for*, mesmo quando o bloco contém apenas uma instrução. Escrevendo o bloco, você facilita a adição de instruções ao bloco posteriormente. Sem o bloco, para adicionar outra instrução, você teria que lembrar de adicionar a instrução extra e as chaves, e é muito fácil esquecer as chaves.

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* próximas entre si 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
{
    ...
}
```

Escrevendo 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);
```

Você deve utilizar um bloco de *instruções* se o corpo do loop incluir mais de uma instrução. Eis uma versão do exemplo que escreve os valores de 0 a 9 no console, dessa 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 a instrução *break* é utilizada para sair de uma instrução *switch*. Você também pode utilizar uma instrução *break* 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). Eis 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 está 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 da etapa anterior
enquanto dec não for 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 2010, abra o projeto DoStatement, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 5\DoStatement na sua pasta Documentos.
2. Exiba o formulário WPF, *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 é gerada. A caixa de texto inferior, 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*, que é executado quando o usuário clica no botão *Show Steps*, no formulário. Atualmente, ele está vazio.
4. Adicione as seguintes instruções, que aparecem em negrito, ao método *showStepsClick*:

```
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 armazene-o 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. Use 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 realiza 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 sendo compilada. 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, e você verá que a primeira instrução dentro do 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 dentro do 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 um valor de 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 valor de inteiro 57. O C# permite tratar um caractere como um inteiro e realizar 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 dentro do 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 caractere correspondente. Assim, por exemplo, se *digitCode* tiver o valor 54, *Convert.ToChar(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) que corresponde ao número que o usuário digitou. A próxima tarefa é prefixar esse dígito à string sendo gerada, desta maneira:

```
current = digit + current;
```

A próxima instrução dentro do 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 realiza mais uma iteração.

No exercício final, você utilizará o depurador do Visual Studio 2010 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 clique em *Run To Cursor*.
3. Quando o formulário aparecer, digite 999 na caixa de texto superior e, em seguida, clique em *Show Steps*.

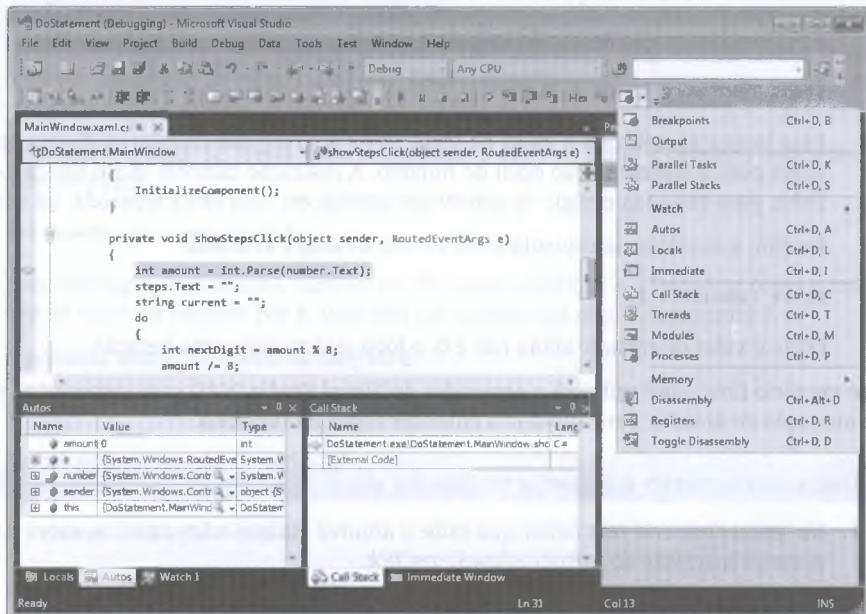
O programa para e você é colocado no Visual Studio 2010 no modo de depuração. Uma seta amarela na margem esquerda da janela *Code and Text Editor* indica a instrução atual.

4. Exiba a barra de ferramentas *Debug* se ainda não estiver visível. No menu *View*, aponte para *Toolbars* e clique em *Debug*.
5. Se estiver utilizando o Visual Studio 2010 Professional ou Standard, na barra de ferramentas *Debug*, clique na seta suspensa *Breakpoints*. Se estiver usando o Visual C# 2010 Express, na barra de ferramentas *Debug*, clique na seta suspensa *Output*.



Nota A seta suspensa *Breakpoints* ou *Output* é o ícone mais à direita na barra de ferramentas *Debug*.

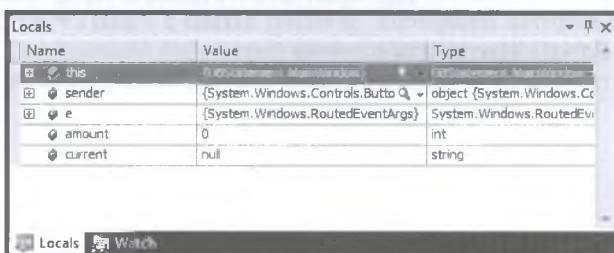
O menu a seguir é exibido:



Nota Se você estiver utilizando o Microsoft Visual C# 2010 Express, o menu suspenso *Output* conterá um subconjunto daqueles mostrados nessa imagem.

6. No menu drop-down, clique 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:



7. Na barra de ferramentas *Debug*, clique no botão *Step Into*. O depurador executa a 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 a 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:

```
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*, e que o valor das três variáveis é 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 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 a instrução:

```
Amount /= 8;
```

O valor de *amount* muda para 124, na janela *Locals*.

13. Clique em *Step Into*.

O depurador executa a instrução:

```
int digitCode = '0' + nextDigit;
```

O valor de *digitCode* na janela *Locals* muda para 55. Esse é o código de caracteres de '7' (48 + 7).

14. Clique em *Step Into*.

O depurador executa a 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 instrução:

```
current = current + digit;
```

O valor de *current* muda para "7" na janela *Locals*.

16. Clique em *Step Into*.

O depurador executa a instrução:

```
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 a instrução:

```
while (amount != 0);
```

O valor de *amount* é 124 e a expressão *124 != 0* é avaliada como *true*, portanto, o loop *do* realiza uma 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 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);
```

O valor de *amount* agora é 0, portanto, a expressão *amount != 0* será avaliada como *false* e o loop *do* terminará.

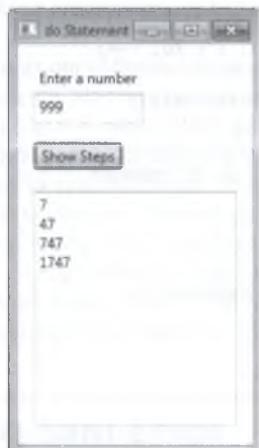
22. Clique em *Step Into*. O depurador executa a instrução:

```
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. Clique no botão *Continue* na barra de ferramentas *Debug*.

O formulário aparece exibindo as quatro etapas utilizadas para criar uma representação octal de 999: 7, 47, 747 e 1747.



24. Feche o formulário para retornar ao ambiente de programação do Visual Studio 2010.

Neste capítulo, você aprendeu a utilizar os operadores de atribuição composta para atualizar variáveis numéricas. Você viu como é possível utilizar a as instruções *while*, *for* e *do* para executar o código várias vezes, enquanto uma condição booleana *for true*.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 6.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 5

Para	Faça isto
Adicionar uma quantidade a uma variável	Utilize o operador de adição composta. Por exemplo: <code>variable += amount;</code>
Subtrair uma quantidade de uma variável	Utilize o operador de subtração composta. Por exemplo: <code>variable -= amount;</code>
Executar uma ou mais instruções zero ou mais vezes, enquanto uma condição for verdadeira	Utilize uma instrução <code>while</code> . Por exemplo: <code>int i = 0; while (i < 10) { Console.WriteLine(i); i++; }</code> Como alternativa, utilize uma instrução <code>for</code> . Por exemplo: <code>for (int i = 0; i < 10; i++) { Console.WriteLine(i); }</code>
Executar repetidamente as instruções uma ou mais vezes	Utilize a instrução <code>do</code> . Por exemplo: <code>int i = 0; do { Console.WriteLine(i); i++; } while (i < 10);</code>

Capítulo 6

Gerenciando 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*.

Você viu até agora as principais instruções do Microsoft Visual C# necessárias para escrever métodos, declarar variáveis, utilizar operadores para criar valores, escrever instruções *if* e *switch* para executar um código seletivamente e escrever as instruções *while*, *for* e *do* para executar o código repetidamente. Mas os capítulos anteriores não consideraram a possibilidade (ou probabilidade) de algo dar errado. É muito difícil garantir que o código sempre funcione conforme o esperado. As falhas podem ocorrer por vários motivos, muitos dos quais estão além do seu controle como programador. Todos os aplicativos que você escrever devem ser capazes de detectar falhas e de lidar com elas de maneira elegante. Neste capítulo final da Parte I, “Apresentando o Microsoft Visual C# e o Microsoft Visual Studio 2010”, você aprenderá como o C# utiliza exceções para sinalizar a ocorrência de um erro e como utilizar as instruções *try*, *catch* e *finally* para capturar e lidar com os erros que essas exceções representam. No final deste capítulo, você terá uma base sólida em C#, sobre a qual construirá seu conhecimento na Parte II, “Entendendo a linguagem C#”.

Lidando com erros

Faz parte da vida que coisas ruins aconteçam às vezes. 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 falham, outros aplicativos em execução no mesmo computador em que seu programa é executado consomem, de modo descontrolado, toda a memória disponível e as redes se desconectam no momento mais atribulado. Erros podem ocorrer em praticamente qualquer estágio da execução de um programa; portanto, como detectá-los e tentar se recuperar deles?

Ao longo dos anos, vários mecanismos foram criados. Uma abordagem 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 simplesmente muito trabalhoso. Por isso, elas utilizam *exceções*. Se quiser escrever programas robustos em C#, você precisa conhecer as exceções.

Testando o código e capturando 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 simplesmente propenso a erros. Você também pode se desviar do 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 o fluxo principal do programa utilizando as exceções e as rotinas de tratamento de exceção. Para escrever programas compatíveis com exceções, você precisa fazer duas coisas:

1. 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 dentro do 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 um outro fragmento de código projetado para capturar e tratar a exceção – uma rotina de tratamento *catch*.
2. 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 específica e você pode ter múltiplas 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 poderiam aparecer no bloco *try*. Se qualquer uma das instruções dentro do bloco *try* causar um erro, o runtime gera e lança uma exceção. O runtime então examina as rotinas de tratamento *catch* após o bloco *try* e transfere o controle diretamente para a primeira rotina de tratamento correspondente.

Observe um exemplo do código em um bloco *try* que tenta converter as strings que um usuário digitou em algumas caixas de texto em um formulário para valores inteiros, chamar um método para calcular um valor e gravar o resultado em outra caixa de texto. Converter uma string em um número inteiro requer que a sequência contenha um conjunto de dígitos válido e não alguma string arbitrária. Se a sequência contém caracteres inválidos, o método *Int.Parse* lança automaticamente 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:

```
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)
{
    // Trate a exceção
    . . .
}
```

Uma rotina de tratamento *catch* utiliza 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.

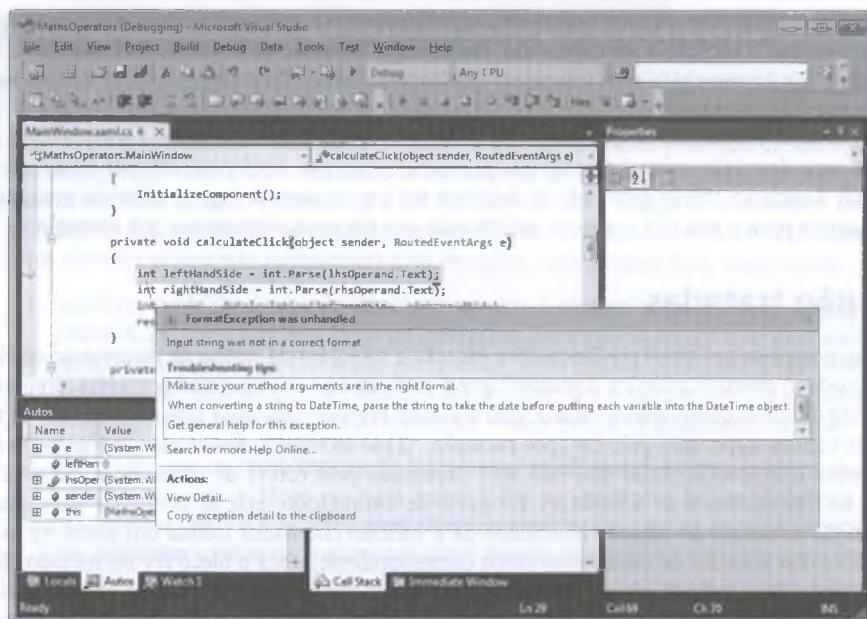
Exceções não tratadas

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 está 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 *FormatException catch*. Se isso ocorrer e se 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 a rotina de tratamento *catch* correspondente, após o bloco *try* no método chamador, e executá-la. 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 capturar uma exceção, a execução continua no método que contém o bloco *catch* que capturou a exceção. 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 2010 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 incluída 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, por meio da barra de ferramentas *Debug* e das várias janelas de depuração.

Utilizando múltiplas 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 múltiplas rotinas de tratamento *catch*, uma após a outra, assim:

```
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 adjacente de captura de *OverflowException* não será executado. Em vez disso, a exceção se propaga para o método que chamou o código, como descrito anteriormente nesta seção.

Capturando 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 pouco provável que você queira escrever rotinas de tratamento *catch* para todas as exceções possíveis que seu código pode lançar. Portanto, 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 herança no Capítulo 12, “Trabalhando com herança”.) *FormatException* e *OverflowException* pertencem a uma família chamada *SystemException*, assim como várias outras exceções. *SystemException* é, em si mesmo, um membro de uma família maior, chamada *Exception*, que é a bisavó de todas as exceções. Se você capturar *Exception*, a rotina de tratamento irá capturar 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 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 nem sempre é recomendado. O objeto exceção passado para a rotina de tratamento *catch* contém informações úteis referentes à exceção, que não são acessíveis ao utilizar essa versão da construção *catch*.

Nesse ponto, há uma pergunta final que você deve estar se fazendo: o que acontece se a mesma exceção corresponder a múltiplas 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 escutadas)?

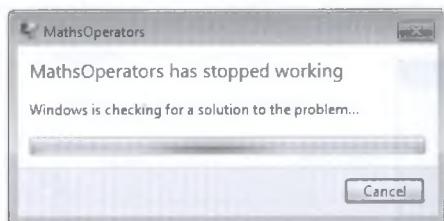
Quando ocorre uma exceção, a primeira rotina de tratamento encontrada pelo runtime que corresponde à exceção é utilizada e as outras são ignoradas. O que isso significa é que, se você colocar uma rotina de tratamento para *Exception* antes de uma rotina de tratamento para *FormatException*, a rotina 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 nenhuma das rotinas de tratamento *catch* específicas corresponder à exceção, a rotina de tratamento *catch* geral irá corresponder.

No próximo exercício, você irá escrever um bloco *try* e capturar uma exceção.

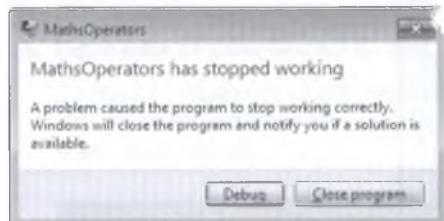
Escreva um bloco de instruções *try/catch*

1. Inicie o Visual Studio 2010 se ele ainda não estiver em execução.
2. Abra a solução MathsOperators localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 6\MathsOperators na sua pasta Documentos.
Essa é uma variação do programa que você viu no Capítulo 2, “Trabalhando com variáveis, operadores e expressões”. Ele foi utilizado para demonstrar os diversos operadores aritméticos.
3. No menu *Debug*, clique em *Start Without Debugging*.
O formulário aparece. Agora você digitará um texto que não será válido na caixa de texto do operando esquerdo. Essa operação demonstrará a falta de robustez da versão atual do programa.
4. Digite **John** na caixa de texto do operando esquerdo e clique em *Calculate*.

Essa ação aciona o tratamento de erros do Windows e a seguinte caixa de diálogo é exibida:

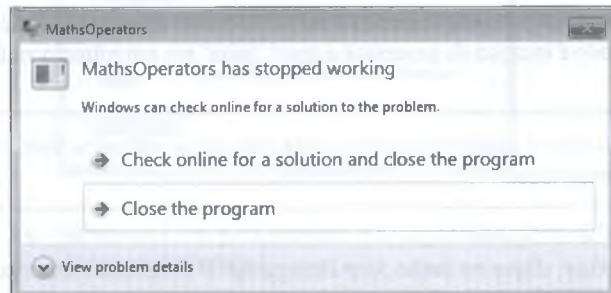


Essa caixa é seguida por outra caixa de diálogo que reporta uma exceção não tratada:



Nota No Visual C# 2010 Express, o botão *Debug* não aparece.

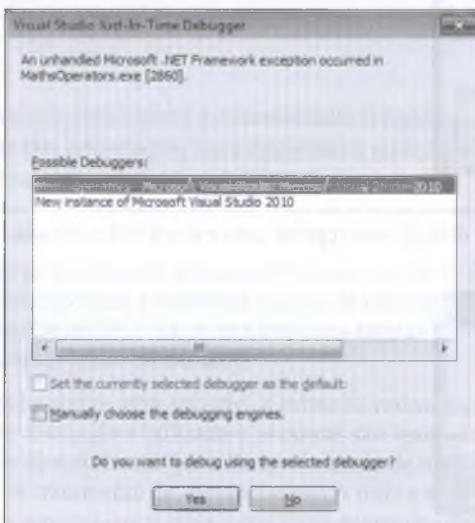
Você poderia ver uma versão diferente dessa caixa de diálogo, dependendo de como você configurou o informe de problemas no painel de controle.



Se vir essa caixa de diálogo, simplesmente clique em *Close the program* e continue na segunda sentença da etapa 6 a seguir.

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 for exibida, clique em *Cancel* e continue na segunda sentença da etapa 6.

5. Se você estiver usando o Visual Studio 2010 Professional ou Standard, clique em *Debug*. Na caixa de diálogo *Visual Studio Just-In-Time Debugger*, na caixa de listagem *Possible Debuggers*, selecione *MathsOperators – Microsoft Visual Studio: Visual Studio 2010* e então clique em *Yes*:



6. Se você estiver utilizando o Visual C# 2010 Express, clique em *Close Program*. No menu *Debug*, clique em *Start Debugging*. Digite **John** na caixa de texto do operando esquerdo e clique em *Calculate*.
7. O Visual Studio 2010 exibe o seu código e destaca a instrução que causou a exceção, juntamente com uma caixa de diálogo que descreve essa exceção. Nesse caso, a informação é a seguinte: "Input string was not in a correct format." (Formato incorreto na string de entrada). Você pode ver que a exceção foi lançada pela chamada a *int.Parse* dentro do método *calculateClick*. O problema é que esse método é incapaz de processar o texto "John" em um número válido.



Nota Você só pode visualizar o código que causou uma exceção se realmente tiver o código-fonte disponível no computador.

8. Na barra de ferramentas *Debug*, clique no botão *Stop Debugging*. O programa é encerrado.
9. Exiba o código para o arquivo *MainWindow.xaml.cs* na janela *Code and Text Editor* e localize o método *calculateClick*.
10. Adicione um bloco *try* (incluindo chaves) em torno das quatro instruções dentro desse método, como mostrado no texto em negrito aqui:

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
```

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

11. Adicione um bloco *catch* logo após a chave de fechamento para esse novo bloco *try*, como a seguir:

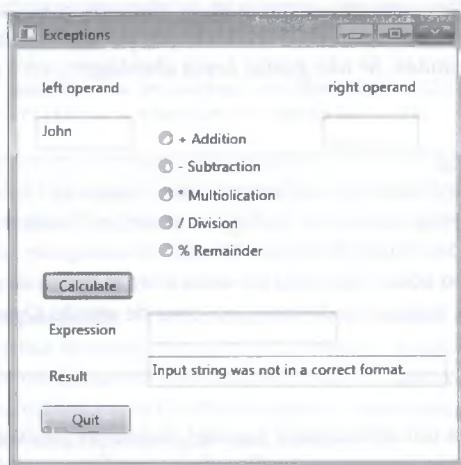
```
catch (FormatException fEx)  
{  
    result.Text = fEx.Message;  
}
```

Essa rotina de tratamento *catch* captura a *FormatException* lançada por *int.Parse* e então a exibe na caixa *result* de texto, na parte inferior do formulário, o texto na propriedade *Message* da exceção.

12. No menu *Debug*, clique em *Start Without Debugging*.

13. Digite **John** na caixa de texto do operando esquerdo e 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.



14. Substitua John pelo número **10**, digite **Sharp** na caixa de texto do operando direito e então clique em *Calculate*.

O bloco *try* cerca 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.

15. Substitua Sharp por **20** na caixa de texto do *operando da direita*, clique no botão *Addition* e depois clique em *Calculate*. O aplicativo funciona como previsto e exibe o valor **30** na caixa de texto *Result*.

16. Clique em *Quit* para retornar ao ambiente de programação do Visual Studio 2010.

Utilizando aritmética verificada e não verificada de números inteiros

No Capítulo 2, você aprendeu a utilizar os operadores aritméticos binários, como + e *, em tipos de dados primitivos, como *int* e *double*. Você também viu 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 programas e adicionar a sobrecarga de verificar o estouro (overflow) em cada expressão de números inteiros pode levar a um desempenho muito pobre. Em muitos casos, o risco é aceitável porque você sabe (ou espera!) que seus valores *int* nunca atinjam seus limites. Se não gostar dessa abordagem, você pode ativar a verificação de overflow.



Dica Você pode ativar o desativar a verificação de overflow no Visual Studio 2010 definindo as propriedades do projeto. No *Solution Explorer*, clique em *SeuProjeto* (onde *SeuProjeto* é o nome 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.

Escrevendo instruções verificadas

Uma instrução verificada é um bloco precedido por uma 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"); // Esta linha não será executada
}
```



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 aplica ao código que executa no método que é chamado.

Você também pode utilizar a palavra-chave *unchecked* para criar uma instrução de bloco *não verificada*. 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"); // Esta linha será executada  
}
```

Escrevendo 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 utilizando 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. (O .NET Framework tem uma representação para infinito.)

No próximo exercício, você verá como executar a aritmética verificada quando utilizar o Visual Studio 2010.

Utilize expressões verificadas

1. Retorne ao Visual Studio 2010.
2. No menu *Debug*, clique em *Start Without Debugging*.

Agora, você tentará multiplicar dois valores grandes.

3. Digite **9876543** na caixa de texto do operando esquerdo, digite **9876543** na caixa de texto do operando direito, clique no botão *Multiplication* e então clique em *Calculate*.

O valor **-1195595903** aparece na caixa de texto *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. Clique em *Quit* e retorne ao ambiente de programação do Visual Studio 2010.
5. Na janela *Code and Text Editor* exibindo *MainWindow.xaml.cs*, localize o método *multiplyValues*. Ele é semelhante a este:

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

A instrução *return* contém a operação de multiplicação que causa silenciosamente um overflow.

6. Edite a instrução *return* para que o valor de retorno seja verificado, desta maneira:

```
return checked(leftHandSide * rightHandSide);
```

A multiplicação agora é verificada e lançará uma *OverflowException* em vez de retornar silenciosamente a resposta errada.

7. Localize o método *calculateClick*.
8. Adicione a rotina de tratamento *catch* a seguir imediatamente após a rotina de tratamento *catch FormatException* existente no método *calculateClick*:

```
catch (OverflowException oEx)
{
    result.Text = oEx.Message;
}
```

Dica A lógica dessa rotina de tratamento *catch* é a mesma da rotina de tratamento *catch FormatException*. Mas ainda vale a pena manter essas duas rotinas de tratamentos separadas em vez de simplesmente escrever uma rotina de tratamento *catch Exception* genérica, porque você poderia decidir tratar essas exceções de maneira diferente no futuro.

9. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
10. Digite **9876543** na caixa de texto do operando esquerdo, digite **9876543** na caixa de texto do operando direito, clique no botão *Multiplication* e então clique em *Calculate*.
- A segunda rotina de tratamento *catch* captura com sucesso a *OverflowException* e exibe a mensagem “*Arithmetical operation resulted in an overflow*” na caixa de texto *Result*.
11. Clique em *Quit* para retornar ao ambiente de programação do Visual Studio 2010.

Lançando 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 facilmente criar sua própria classe de exceção, mas precisa conhecer um pouco mais da linguagem C# antes de poder fazer isso.) 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 uma 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, "Criando e gerenciando 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 uma operação a ser executada.

Lance uma exceção

1. Retorne ao Visual Studio 2010.
2. No menu *Debug*, clique em *Start Without Debugging*.
3. Digite **24** na caixa de texto do operando esquerdo, digite **36** na caixa de texto do operando direito e então clique em *Calculate*.

O valor **0** aparece na caixa de texto *Result*. O fato de você não ter selecionado uma opção de operador não é óbvio. Seria útil escrever uma mensagem de diagnóstico na caixa de texto *Result*.

4. Clique em *Quit* para retornar ao ambiente de programação do Visual Studio 2010.
5. Na janela *Code and Text Editor* exibindo *MainWindow.xaml.cs*, localize e examine o método *doCalculation*. Ele é semelhante a este:

```
private int doCalculation(int leftHandSide, int rightHandSide) {  
    int result = 0;  
  
    if (addition.IsChecked.HasValue && addition.IsChecked.Value)  
        result = addValues(leftHandSide, rightHandSide);  
    else if (subtraction.IsChecked.HasValue && subtraction.IsChecked.Value)  
        result = subtractValues(leftHandSide, rightHandSide);  
    else if (multiplication.IsChecked.HasValue && multiplication.IsChecked.Value)  
        result = multiplyValues(leftHandSide, rightHandSide);  
    else if (division.IsChecked.HasValue && division.IsChecked.Value)  
        result = divideValues(leftHandSide, rightHandSide);  
    else if (remainder.IsChecked.HasValue && remainder.IsChecked.Value)  
        result = remainderValues(leftHandSide, rightHandSide);  
  
    return result;  
}
```

Os campos *addition*, *subtraction*, *multiplication*, *division* e *remainders* são os botões de opção que aparecem no formulário. Cada botão tem uma propriedade chamada *IsChecked* que indica se o usuário a selecionou. A propriedade *IsChecked* é um exemplo de um valor *nullable*, ou seja, ela pode conter um valor específico ou estar em um estado indefinido. (Discutiremos outros detalhes sobre valores nullable no Capítulo 8, "Entendendo valores e referências".) A propriedade *.IsChecked.HasValue* indica se o botão está em um estado definido, e, se estiver, a propriedade *.IsChecked.Value* indica qual é esse estado. A propriedade *.IsChecked.Value* é uma booleana que tem o valor *true* se o botão estiver selecionado ou *false* caso contrário. 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 nenhum dos botões estiver selecionado, nenhuma das instruções *if* será verdadeira e a variável *result* permanecerá com seu valor inicial (0). Essa variável armazena o valor que é retornado pelo método.

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 essa solução não é uma boa ideia porque o verdadeiro objetivo desse método não é emitir mensagens. É melhor separar a detecção e a sinalização de um erro da captura e tratamento desse erro.

6. Adicione outra instrução *else* à lista de instruções *if-else* (imediatamente antes da instrução *return*) e lance uma *InvalidOperationException* exatamente como mostrado a seguir:

```
else  
    throw new InvalidOperationException("No operator selected");
```

7. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
8. Digite **24** na caixa de texto do operando esquerdo, digite **36** na caixa de texto do operando direito e então clique em *Calculate*.

O Windows detecta que seu aplicativo lançou uma exceção e é (finalmente) exibida uma caixa de diálogo de exceção. O aplicativo lançou uma exceção, mas seu código não a captura ainda.

9. Clique em *Close program*.

O aplicativo termina e você retorna ao Visual Studio 2010.

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* exibindo MainWindow.xaml.cs, localize o método *calculateClick*.
2. Adicione a rotina de tratamento *catch* a seguir imediatamente abaixo das duas rotinas de tratamento *catch* existentes no método *calculateClick*:

```
catch (InvalidOperationException ioEx)
{
    result.Text = ioEx.Message;
}
```

Esse código captura a *InvalidOperationException* que é lançada quando nenhum botão de operador está selecionado.

3. No menu *Debug*, clique em *Start Without Debugging*.
4. Digite **24** na caixa de texto do operando esquerdo, digite **36** na caixa de texto do operando direito e então clique em *Calculate*.

A mensagem “no operator selected” aparece na caixa de texto *Result*.

5. Clique em *Quit*.

O aplicativo agora está mais robusto do que antes. Mas ainda podem surgir várias exceções que não serão capturadas e que poderão provocar a falha do aplicativo. 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 isso é escrever um número ainda maior de rotinas de tratamento *catch* dentro do método *calculateClick*. Mas uma solução melhor é adicionar uma rotina de tratamento *catch* geral que capture uma *Exception*, no final da lista de rotinas de tratamento *catch*. Isso capturará todas as exceções não tratadas.

Dica A decisão de capturar explicitamente todas 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 trate do erro ali.

Capture exceções não tratadas

1. Na janela *Code and Text Editor* exibindo Window1.xaml.cs, localize o método *calculateClick*.
2. Adicione a rotina de tratamento *catch* a seguir ao final da lista de rotinas de tratamento *catch* existentes:

```
catch (Exception ex)
{
    result.Text = ex.Message;
}
```

Essa rotina de tratamento *catch* irá capturar todas as exceções não tratadas até aqui, qualquer que seja seu tipo específico.

3. No menu *Debug*, clique em *Start Without Debugging*.

Você agora vai tentar realizar alguns cálculos conhecidos por provocar exceções e confirmar que elas sejam tratadas corretamente.

4. Digite **24** na caixa de texto do operando esquerdo, digite **36** na caixa de texto do operando direito e então clique em *Calculate*.

Confirme que a mensagem de diagnóstico “no operator selected” ainda é exibida na caixa de texto *Result*. Essa mensagem foi gerada pela rotina de tratamento *InvalidOperationException*.

5. Digite **John** na caixa de texto do operando esquerdo e clique em *Calculate*.

Confirme que a mensagem de diagnóstico “Input string was not in a correct format” é exibida na caixa de texto *Result*. Essa mensagem foi gerada pela rotina de tratamento *FormatException*.

6. Digite **24** na caixa de texto do operando esquerdo, digite **0** na caixa de texto do operando direito, clique no botão *Division* e, em seguida, em *Calculate*.

Confirme que a mensagem de diagnóstico “Attempted to divide by zero” é exibida na caixa de texto *Result*. Essa mensagem foi gerada pela rotina de tratamento *Exception* geral.

7. Clique em *Quit*.

Utilizando um 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. Veja o exemplo a seguir. É muito fácil assumir que a chamada ao *reader.Close* sempre ocorrerá quando o loop *while* terminar. Afinal de contas, é o que está no código:

```
TextReader reader = src.OpenText();
string line;
while ((line = reader.ReadLine()) != null)
{
    source.Text += line + "\n";
}
reader.Close();
```

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: se a chamada ao `src.OpenText` for bem-sucedida, então ela adquire um recurso (um handle de arquivo) e você deve garantir a chamada de `reader.Close` para liberar o recurso. Se você não fizer, cedo ou tarde você 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 ou não lançada, 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 de chamada 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 = null;
try
{
    reader = src.OpenText();
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        source.Text += line + "\n";
    }
}
finally
{
    if (reader != null)
    {
        reader.Close();
    }
}
```

Mesmo que uma exceção seja lançada, o bloco `finally` garante que a instrução `reader.Close` sempre seja executada. Você verá outra maneira de resolver esse problema no Capítulo 14, “Utilizando a coleta de lixo e o gerenciamento de recursos”.

Neste capítulo, você aprendeu a capturar e tratar exceções por meio das construções `try` e `catch`. Você viu como é possível habilitar e desabilitar a verificação de estouro de inteiros por meio das palavras-chave `checked` e `unchecked`. Você 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 você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 7.

- Se quiser sair do Visual Studio 2010:

No menu `File`, clique em `Exit`. Se vir uma caixa de diálogo `Save`, clique em `Yes` e salve o projeto.

Referência rápida do Capítulo 6

Para	Faça isto
Capturar uma exceção específica	Escreva uma rotina e tratamento <i>catch</i> que capture a classe de exceção específica. Por exemplo: Try { *** } catch(FormatException fEx) { *** }
Garantir que a aritmética de inteiros seja sempre verificada quanto a estouros	Use a palavra-chave <i>checked</i> . Por exemplo: int number = Int32.MaxValue; checked { number++; }
Lançar uma exceção	Utilize uma instrução <i>throw</i> . Por exemplo: throw new FormatException(source);
Capturar todas as exceções em uma única rotina de tratamento <i>catch</i>	Escreva uma rotina de tratamento <i>catch</i> que captura <i>Exception</i> . Por exemplo: try { *** } catch (Exception ex) { *** }
Garantir que algum código sempre seja executado, mesmo se uma exceção for lançada	Escreva o código dentro de um bloco <i>finally</i> . Por exemplo: try { *** } finally { // sempre executa }

Parte II

Entendendo a linguagem C#

Capítulo 7: Criando e gerenciando classes e objetos	161
Capítulo 8: Entendendo valores e referências	183
Capítulo 9: Criando tipos-valor com enumerações e estruturas	205
Capítulo 10: Utilizando arrays e coleções.	223
Capítulo 11: Entendendo arrays de parâmetros	251
Capítulo 12: Trabalhando com herança	263
Capítulo 13: Criando interfaces e definindo classes abstratas.	285
Capítulo 14: Utilizando a coleta de lixo e o gerenciamento de recursos	311

Capítulo 7

Criando e gerenciando 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 invocar 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.

Na Parte I, “Apresentando o Microsoft Visual C# e o Microsoft Visual Studio 2010”, você aprendeu a declarar variáveis, utilizar operadores para criar valores, chamar métodos e escrever muitas das instruções necessárias para implementar um método. Agora, você já sabe o suficiente para prosseguir para a próxima etapa – combinar métodos e dados nas suas classes.

O Microsoft .NET Framework contém milhares de classes, e você já usou várias delas, inclusive *Console* e *Exception*. As classes fornecem 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 relacionado à determinação das entidades importantes para os processos implementados pelo sistema e à execução de uma análise para ver que informações essas entidades precisam armazenar e que operações elas devem executar. Você armazena as informações contidas por uma classe como campos e utiliza métodos para implementar as operações que uma classe pode realizar.

Os capítulos da Parte II, “Entendendo a linguagem C#”, fornecem tudo o que você precisa saber para criar suas classes.

Entendendo a 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

objetos que compartilham 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.

Definindo e utilizando 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. Veja 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 realizar cálculos matemáticos e campos contendo constantes matemáticas. O campo *Math.PI* engloba 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. Você já viu como declarar variáveis no Capítulo 2, “Trabalhando com variáveis, operadores e expressões”, e como escrever métodos no Capítulo 3, “Escrevendo métodos e aplicando o escopo”, de modo que não há quase sintaxe nova aqui.

Você pode utilizar a classe *Circle* de modo semelhante para usar os outros tipos já encontrados; você cria uma variável especificando *Circle* como seu tipo e inicializa a variável com algum dado válido. Observe 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*. Anteriormente, 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;
```

Acima de tudo, 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 é discutido em mais detalhes no Capítulo 8, “Entendendo valores e referências”. Por enquanto, basta aceitar que a palavra-chave *new* cria uma nova instância de uma classe, mais chamada de objeto.

Mas você pode atribuir diretamente uma instância de uma classe a uma 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.

Controlando a acessibilidade

Surpreendentemente, a classe *Circle* não tem, atualmente, qualquer utilidade prática. 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. Em outras palavras, embora se possa criar um objeto *Circle* em um programa, não se pode 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. 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 Os programadores C++ devem notar 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 Diferentemente 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

As recomendações a seguir relacionam-se às convenções de nomes para campos e métodos com base na acessibilidade dos membros da classe:

- 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 primeiramente 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 *camelCase*.

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, não poderão utilizar sua classe.

Trabalhando com construtores

Quando você utiliza a palavra-chave *new* para criar um objeto, o runtime tem de construir esse objeto utilizando a definição da classe. O runtime tem de se apropriar de uma parte da memória do sistema operacional, preenchê-la com os campos definidos pela classe e então invocar o construtor para executar qualquer inicialização necessária.

Um *construtor* é um método especial que se executa 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 de forma muito fácil – basta adicionar um método público, com o mesmo nome da classe,

que não retorna um valor. 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 ele é gerado pelo compilador ou se ele é 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, intitulada “Sobrecregando 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*. Mas você poderá achar que os construtores privados não são tão valiosos. Entretanto, eles realmente têm suas utilidades, mas estas estão além do escopo da discussão atual.

Você agora pode utilizar a classe *Circle* e exercitar 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();
```

Sobrecregando construtores

Você quase já terminou, só falta um detalhe. Agora você pode declarar uma variável *Circle*, apontá-la para um objeto *Circle* recém-criado e então chamar seu método *Area*. Mas ainda existe 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. Entretanto, entenda que um construtor é apenas um tipo especial de método e

que ele – como todos os métodos – pode ser sobreescrito. Assim como existem várias versões do método *Console.WriteLine*, e cada uma das quais recebe parâmetros diferentes, é possível também escrever diferentes versões de um construtor. Você pode adicionar um construtor à classe *Circle*, com o raio como seu parâmetro, como este:

```
class Circle
{
    private int radius;

    public Circle() // construtor padrão
    {
        radius = 0;
    }

    public Circle(int initialRadius) // construtor sobreescrito
    {
        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 mostrado aqui:

```
Circle c;
c = new Circle(45);
```

Quando você compila o aplicativo, o compilador deduz qual construtor ele deve chamar com base nos parâmetros que você especifica para o operador *new*. Neste exemplo, você passou um *int*, portanto, o compilador gera o código que invoca o construtor que recebe um parâmetro *int*.

Você deve estar ciente de uma peculiaridade da linguagem C#: se você escrever um construtor para uma classe, o compilador não irá gerar 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 2010 para aplicativos Windows Presentation Foundation (WPF), 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 múltiplos 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.

Nota É possível definir as interfaces parciais e as estruturas do mesmo jeito.

No próximo exercício, você irá declarar uma classe que modela um ponto no espaço bidimensional. Essa classe conterá dois campos privados que contêm 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 2010 se ele ainda não estiver em execução.
2. Abra o projeto Classes, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter7\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. Localize o método *Main* na classe *Program*.

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.

5. Exiba o arquivo Point.cs na janela *Code and Text Editor*.

Esse arquivo define uma classe chamada *Point*, que você utilizará para representar a localização de um ponto definido por um par de coordenadas *x* e *y*. No momento, a classe *Point* está vazia.

6. Retorne ao arquivo Program.cs e localize o método *DoWork* da classe *Program*. Edite o corpo do método *DoWork* e substitua o comentário *// to* do pela instrução a seguir:

```
Point origin = new Point();
```

7. 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.

8. Retorne à classe *Point* no arquivo Point.cs. Substitua o comentário *// to* do por um construtor *public* que aceita dois argumentos *int* chamados *x* e *y* e que chamam o método *Console.WriteLine* para exibir os valores desses argumentos no console, como mostrado no texto em negrito no exemplo de código seguinte. A classe *Point* deve ser semelhante a esta:

```
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.

9. 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 em `DoWork` não funciona porque não há mais um construtor padrão. Você escreveu seu próprio construtor para a classe `Point`, assim o compilador não irá mais gerar o construtor padrão. Agora você corrigirá isso escrevendo seu próprio construtor padrão.

10. Edite a classe `Point` e adicione um construtor padrão `public` que chama `Console.WriteLine` para escrever a string “*default constructor called*” no console, como mostrado no texto em negrito no exemplo de código a seguir. A classe `Point` deve ser semelhante a esta:

```
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 no texto em negrito no código a seguir. Forneça os valores 1024 e 1280, que representam as coordenadas do canto inferior direito da tela com base na resolução 1024×1280 . O método `DoWork` agora deve se parecer com o exemplo seguinte:

```
static void DoWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1024, 1280);
}
```

13. No menu *Debug*, clique em *Start Without Debugging*.

O programa compila e executa, escrevendo a seguinte mensagem no console:

```
Default constructor called
x:1024, y:1280
```

14. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2010.

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 e adicione dois campos de instância *private* chamados *x* e *y* do tipo *int*, como mostrado no texto em negrito no código a seguir. A classe *Point* deve ser semelhante a esta:

```
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);
    }
}
```

Agora, você editará o segundo construtor *Point* para inicializar os campos *x* e *y* para os valores dos parâmetros *x* e *y*. Há uma armadilha potencial quando você 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 e substitua 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* em -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*. (Há outros tipos de métodos que você encontrará mais adiante neste capítulo.) No exercício a seguir, você escreverá um método de instância para a classe *Point*, chamado *DistanceTo*, que calcula a distância entre dois pontos.

Escreva e chame métodos de instância

1. No projeto Classes no Visual Studio 2010, adicione o método de instância público a seguir chamado *DistanceTo* à classe *Point* 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 um 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 initialize-a com a diferença entre *this.x* e *other.x*, como mostrado aqui em negrito:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
}
```

3. Declare outra variável *int* local chamada *yDiff* e initialize-a com 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;
}
```

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. Adicione a instrução *return* mostrada no texto em negrito no seguinte código ao final do método *DistanceTo* para realizar o cálculo:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
    return Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
}
```

Agora você testará o método *DistanceTo*.

5. Retorne ao método *DoWork* na classe *Program*. Após as instruções que declaram e inicializam as variáveis *origin* e *bottomRight* *Point*, 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 um argumento.

O método *DoWork* deve agora ser semelhante a este:

```
static void DoWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1024, 1280);
    double distance = origin.DistanceTo(bottomRight);
}
```

 **Nota** O Microsoft IntelliSense deve exibir o método *DistanceTo* quando você digitar o caractere de ponto após *origin*.

6. 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(1024, 1280);
    double distance = origin.DistanceTo(bottomRight);
    Console.WriteLine("Distance is: {0}", distance);
}
```

7. No menu *Debug*, clique em *Start Without Debugging*.
8. Confirme se o valor 1640.60537607311 é escrito na janela de console.
9. Pressione Enter para fechar o aplicativo e retornar ao Visual Studio 2010.

Entendendo 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* poderia não participar 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. 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. Nenhuma instância é necessária. Veja como o método *Sqrt* da classe *Math* real é declarado:

```
class Math
{
    public static double Sqrt(double d)
    {
        ...
    }
    ...
}
```

Quando você define um método *static*, ele não tem acesso a qualquer campo de instância definida para a classe; ele só utiliza os campos que são marcados como *static*. Além disso, ele só pode invocar diretamente outros métodos na classe que estão marcados como *static*; métodos não estáticos (de instância) requerem primeiramente a criação de um objeto para chamá-los.

Criando um campo compartilhado

Conforme mencionado na seção anterior, você também pode utilizar a palavra-chave *static* ao definir um campo. Com esse recurso, você pode criar um único campo que é compartilhado 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() // construtor padrão
    {
        radius = 0;
        NumCircles++;
    }

    public Circle(int initialRadius) // construtor sobreescrito
    {
        radius = initialRadius;
        NumCircles++;
    }
}
```

Todos os objetos *Circle* compartilham o mesmo campo *NumCircles*; portanto, a instrução *NumCircles++*; incrementa os mesmos dados toda vez que uma nova instância é criada. Você acessa o campo *NumCircles* especificando a classe *Circle* em vez de um objeto *Circle*. Por exemplo:

```
Console.WriteLine("Number of Circle objects: {0}", Circle.NumCircles);
```



Dica Convém lembrar que os métodos *static* também são chamados de métodos de *classe*. Mas os campos *static* não são normalmente chamados de campos de *classe*; eles são chamados simplesmente de campos *static* (ou, eventualmente, de variáveis *static*).

Criando 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 a abreviação de “constante”. Um campo *const* não utiliza a palavra-chave *static* na sua declaração, mas mesmo assim é estático. Contudo, por razões que estão fora do escopo deste livro, você só pode declarar um campo como *const* quando esse campo por uma enumeração, um tipo numérico como *int* ou *double* ou uma string. (Você aprenderá enumerações no Capítulo 9, “Criando tipos-valor com enumeração e estruturas”.) Por exemplo, veja como a classe *Math* declara *PI* como um campo *const*:

```
class Math
{
    ...
    public const double PI = 3.14159265358979323846;
}
```

Classes estáticas

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 dado 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 executar 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 versão própria 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 Observe que a classe *Math* real não é definida assim, porque na verdade ela realmente tem alguns métodos de instância.

No exercício final neste 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. Utilizando o Visual Studio 2010, exiba a classe *Point* na janela *Code and Text Editor*.
2. Pressione Enter para adicionar um campo *private static* chamado *objectCount* do tipo *int* à classe *Point*, antes dos construtores. Inicialize-o como 0 ao declará-lo, da seguinte maneira:

```
class Point
{
    ...
    private static int objectCount = 0;
    ...
}
```

Nota Você pode escrever as palavras-chave *private* e *static* em qualquer ordem. 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 no texto em negrito no exemplo de código que segue.

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 classe *Point* deve ser semelhante a esta:

```
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));
    }
}
```

Observe que você não pode prefixar campos e métodos *static* com a palavra-chave *this* porque eles não pertencem à instância atual da classe. (Na verdade, eles não pertencem a instância alguma.)

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; você não teria então qualquer garantia de que seu valor estava 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 um método *public static* à classe *Point* chamada *ObjectCount* que retorna um *int* mas não recebe parâmetro algum. Nesse método, retorne o valor do campo *objectCount*, como no texto em negrito a seguir:

```
class Point
{
    ...
    public static int ObjectCount()
    {
        return objectCount;
    }
}
```

5. Exiba a classe *Program* na janela *Code and Text Editor* e localize o método *DoWork*.
6. Adicione uma instrução ao método *DoWork* para gravar o valor retornado a partir do método *ObjectCount* da classe *Point* na tela, como mostrado no texto em negrito no exemplo de código a seguir. O método *DoWork* deve ser semelhante a este:

```
static void DoWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(600, 800);
    double distance = origin.distanceTo(bottomRight);
    Console.WriteLine("Distance is: {0}", distance);
Console.WriteLine("No 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.

7. No menu *Debug*, clique em *Start Without Debugging*.

Confirme que o valor 2 foi escrito na janela do console (após a mensagem que exibe o valor da variável *distance*).

8. Pressione Enter para terminar o programa e retorne para o Visual Studio 2010.

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 utilizar expressões de consulta. (Você aprenderá expressões de consulta no Capítulo 20, “Consultando dados na memória utilizando expressões de consulta”.) Por enquanto, simplesmente aceite o fato de 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 = 44 };
```

Essa classe contém dois campos públicos chamados *Name* (inicializado para a string *"John"*) e *Age* (inicializado como o inteiro 44). O compilador infere os tipos dos campos a partir dos tipos de dados que você especifica para inicializá-los.

Ao definir uma classe anônima, o compilador gera um nome próprio para a classe, mas ele não informará qual é esse nome. Portanto, classes anônimas levantam um enigma potencialmente interessante: se você não souber 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 anteriormente, 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 = 44 };
```

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 notação familiar de ponto, assim:

```
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:

```
var anotherAnonymousObject = new { Name = "Diana", Age = 45 };
```

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. Classes anônimas só podem conter campos públicos, todos esses campos precisam ser inicializados, eles não podem ser estáticos e você não pode especificar método algum.

Neste capítulo, você viu como é possível 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 você 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 você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 8.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 7

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	Escrever um método cujo nome é 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 <code>static</code>	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 <code>static</code>	Escreva o nome da classe, seguida de um ponto, seguida do nome do método. Por exemplo:
	<pre>int pointsCreatedSoFar = Point.ObjectCount();</pre>
Declarar um campo <code>static</code>	Escreva a palavra-chave <code>static</code> antes da declaração do campo. Por exemplo:
	<pre>class Point { ... private static int objectCount; }</pre>

(continua)

Para	Faça isto
Declarar um campo <i>const</i>	Escreva a palavra-chave <i>const</i> antes da declaração do campo e omita a palavra-chave <i>static</i> . Por exemplo: <pre>class Math { /** * ... */ public const double PI = ...;</pre>
Acessar um campo <i>static</i>	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

Entendendo 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*.
- Fazer o boxing de um valor inicializando ou atribuindo uma variável do tipo *object*.
- Fazer o unboxing de um valor via casting do objeto que referencia o valor na forma boxed.

No Capítulo 7, “Criando e gerenciando classes e objetos”, você aprendeu a declarar suas classes e a criar objetos utilizando a palavra-chave *new*. Você também viu como inicializar um objeto utilizando um construtor. Neste capítulo, você aprenderá qual é a diferença entre as características dos tipos primitivos, como *int*, *double* e *char*, e as características dos tipos de classe.

Copiando variáveis de tipo-valor e classes

Tipos como *int*, *float*, *double* e *char* são coletivamente chamados *tipos-valor*. 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 especifica 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 um *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 A maioria dos tipos internos da linguagem C# são tipos-valor, exceto *string*, que é um tipo-referência. A descrição de tipos-referência, como classes, deste capítulo também se aplica ao tipo *string*. Na verdade, a palavra-chave *string* no C# é apenas um alias para a classe *System.String*.

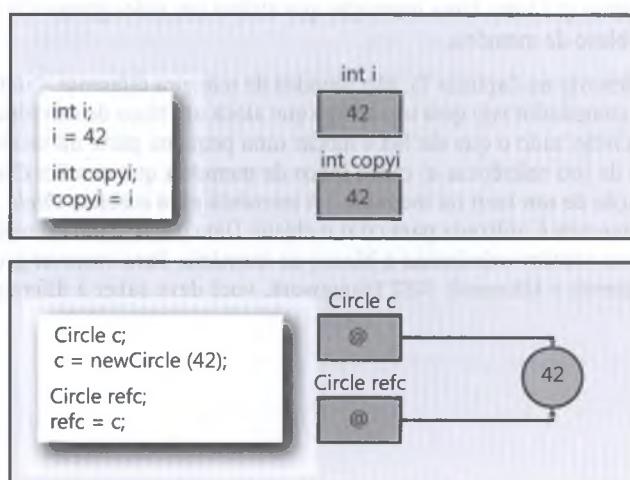
Considere a situação em que você declara uma variável chamada *i* como um *int* e atribui a ela o valor 42. Se declarar uma outra variável chamada *copyi* como um *int* e então atribuir *i* a *copyi*, *copyi* conterá o mesmo valor que *i* (42). Mas embora *copyi* e *i* contenham o mesmo valor, há dois 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 no código:

```
int i = 42;      // declara e inicializa i
int copyi = i;  // copyi contém uma cópia dos dados em i
i++;            // incrementar i não tem efeito sobre copyi
```

O efeito de declarar uma variável *c* como *Circle* (o nome de uma classe) é muito diferente. Quando você declara *c* como *Circle*, *c* pode referenciar um objeto *Circle*. Se você declarar *refc* como outro *Circle*, ele também poderá referenciar um objeto *Circle*. Se você atribuir *c* a *refc*, *refc* irá referenciar o mesmo objeto *Circle* que *c* referencia; há apenas um objeto *Circle* e tanto *refc* quanto *c* o referenciam. O que aconteceu aqui é que o compilador alocou dois blocos de memória, um a *c* e outro a *refc*, mas o endereço contido em cada bloco aponta para a mesma localização na memória que armazena o objeto *Circle* real. Vejamos isso no 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 a 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 irá explorar essa diferença no exercício a seguir.

Nota Se você realmente quiser copiar o conteúdo da variável `c` para `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 isso
```

Mas se qualquer membro da classe `Circle` for privado (como o campo `radius`), você não será capaz de copiar esses dados. Em vez disso, você deve tornar os dados nos campos privados acessíveis expondo-os como propriedades. Discutiremos como fazer isso no Capítulo 15, “Implementando propriedades para acessar campos”.

Utilize parâmetros por valor e parâmetros por referência

1. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto `Parameters` localizado na pasta `\Microsoft Press\Visual CSharp Step by Step\Chapter 8\Parameters` na sua pasta Documentos.

O projeto contém três arquivos de código C# chamados `Pass.cs`, `Program.cs` e `WrappedInt.cs`.

3. Exiba o arquivo `Pass.cs` na janela *Code and Text Editor*. Adicione um método `public static` chamado `Value` à classe `Pass`, substituindo o comentário `// to do`, como mostrado em negrito no seguinte exemplo de código. 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 `42` a `param`.

```
namespace Parameters
{
    class Pass
    {
        public static void Value(int param)
        {
            param = 42;
        }
    }
}
```

4. 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`.

5. Adicione quatro instruções ao método *DoWork* para executar as seguintes tarefas:

1. Declarar uma variável local *int* chamada *i* e inicializá-la como 0.
2. Escrever o valor de *i* no console utilizando *Console.WriteLine*.
3. Chamar *Pass.Value*, passando *i* como argumento.
4. 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* deve ficar assim:

```
static void DoWork()  
{  
    int i = 0;  
    Console.WriteLine(i);  
    Pass.Value(i);  
    Console.WriteLine(i);  
}
```

6. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o programa.
7. Confirme se o valor 0 foi escrito na janela do console duas vezes.

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.

8. Pressione a tecla Enter para fechar o aplicativo.

Você verá agora o que acontece quando você passa um parâmetro *int* que está inserido dentro de uma classe.

9. Exiba o arquivo *WrappedInt.cs* na janela *Code and Text Editor*. Adicione um campo de instância *public* chamado *Number* do tipo *int* à classe *WrappedInt*, como mostrado em negrito aqui.

```
namespace Parameters  
{  
    class WrappedInt  
    {  
        public int Number;  
    }  
}
```

10. 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* cha-

mado *param* e ter um tipo de retorno *void*. O corpo do método *Reference* deve atribuir 42 a *param.Number*, desta maneira:

```
public static void Reference(WrappedInt param)
{
    param.Number = 42;
}
```

11. Exiba o arquivo Program.cs na janela *Code and Text Editor*. Comente o código existente no método *DoWork* e adicione mais quatro instruções para executar as seguintes tarefas:

- Declarar uma variável *WrappedInt* local 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 um 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 são mostradas 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);
}
```

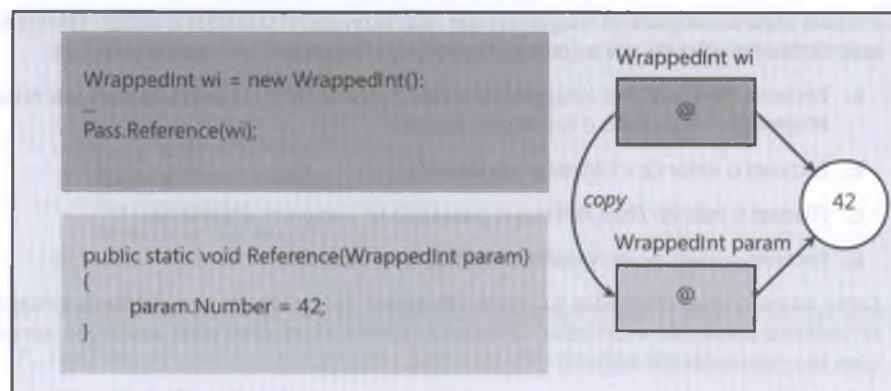
12. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.

Dessa vez, os dois valores exibidos na janela de console correspondem ao valor de *wi.Number* antes e depois de *Pass.Reference* e você deve ver a saída dos valores 0 e 42.

13. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2010.

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*:



Entendendo valores nulos e tipos nullable

Ao 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 OK, mas se você na verdade não quiser criar um novo objeto – talvez o propósito da variável seja simplesmente armazenar uma referência a um objeto existente. No exemplo de código a seguir, a variável *Circle copy* é inicializada, mas depois lhe é atribuída uma referência a uma 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, “Utilizando a coleta de lixo e o gerenciamento de recursos”. O importante a entender por enquanto é que a coleta de lixo é uma operação potencialmente lenta.

Você poderia argumentar que, se uma variável receberá uma referência a um outro objeto em algum ponto em um programa, não há sentido em inicializá-la. Mas isso é uma péssima prática de programação e pode levar a problemas no seu código. Por exemplo, você encontrará inevitavelmente a situação em que quer referenciar uma variável para um objeto somente se essa variável já não contiver uma referência, como mostra o seguinte exemplo de código:

```
Circle c = new Circle(42);
Circle copy;           // Não inicializada!!!
*** 
if (copy == // O que entra 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 maneira:

```
Circle c = new Circle(42);
Circle copy = null;      // Inicializada
*** 
if (copy == null)
    copy = c;          // copy e c referenciam o mesmo objeto
```

Utilizando tipos nullable

O valor *null* é útil para inicializar tipos-referência, mas *null* é, ele próprio, uma referência, e você não pode 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. Utilize 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 uma constante de tipo-valor para um tipo nullable
i = j;        // Copia uma variável de tipo-valor para um tipo nullable
```

Você deve observar que o contrário não é verdadeiro. Você não pode atribuir um valor 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 irá compilar:

```
int? i = 99;
Pass.Value(i); // Erro de compilador
```

Entendendo as propriedades dos tipos nullable

Tipos nullable expõem algumas propriedades que você pode utilizar e que já discutimos no Capítulo 6, "Gerenciando erros e exceções". 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)
    i = 99;
else
    Console.WriteLine(i.Value);
```

Lembre-se, a partir do que foi discutido no Capítulo 4, "Utilizando instruções de decisão", de 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*), ele atribui a essa variável o valor 99; do contrário, ele exibe o valor da variável. Neste exemplo, utilizar a propriedade *HasValue* não traz benefício algum em relação a testar quanto a um valor *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, "Criando tipo-valor com numeração 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.

Utilizando 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 que qualquer alteração no parâmetro afete o valor do argumento passado. Por exemplo, no seguinte código, a saída do valor para o console é 42 e não 43. O método *DoIncrement* incrementa uma *cópia* do argumento (*arg*) e *não* o argumento original:

```
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 principal é que, embora os dados que foram referenciados tenham mudado, o argumento passado como um 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 realmente precise modificar um argumento. O C# fornece as palavras-chave *ref* e *out* para que você possa fazer isso.

Criando parâmetros *ref*

Se você utilizar como prefixo de um parâmetro a palavra-chave *ref*, o parâmetro torna-se um alias do (ou 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 objeto. 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) // utilizando ref
{
    param++;
}
static void Main()
{
    int arg = 42;
    DoIncrement(ref arg);    // utilizando ref
    Console.WriteLine(arg); // escreve 43
}
```

Desta vez, você passa para o método *DoIncrement* uma referência ao argumento original em vez de uma cópia do argumento original, portanto, qualquer alteração que o método faz utilizando essa referência também muda o argumento original. Essa é a razão de o valor 43 ser exibido no console.

A regra de que você deve atribuir um valor a uma variável antes de poder utilizá-la ainda se aplica aos argumentos *ref*. Por exemplo, no programa a seguir, *arg* não é inicializada, portanto, esse código não será compilado. Essa falha ocorre porque *param++* dentro de *DoIncrement* é na verdade *arg++* e isso só é permitido se *arg* tiver um valor definido:

```
static void DoIncrement(ref int param)
{
    param++;
}
static void Main()
{
    int arg;           // não inicializado
    DoIncrement(ref arg);
    Console.WriteLine(arg);
}
```

Criando 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 método inicialize o parâmetro. Você pode fazer isso com a palavra-chave *out*.

A palavra-chave *out* é semelhante à palavra-chave *ref*. Você pode utilizar como prefixo do parâmetro a palavra-chave *out* 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. O exemplo a seguir não é compilado porque *DoInitialize* não atribui um valor a *param*:

```
static void DoInitialize(out int param)
{
    // Não faz nada
}
```

Entretanto, o exemplo a seguir realmente compila porque *DoInitialize* atribui um valor a *param*:

```
static void DoInitialize(out int param)
{
    param = 42;
}
```

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 inicializado
    DoInitialize(out arg);
    Console.WriteLine(arg); // escreve 42
}
```

Examinaremos parâmetros *ref* no próximo exercício.

Utilize parâmetros *ref*

1. Retorne ao projeto *Parameters* no Visual Studio 2010.
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. Descomente as quatro primeiras instruções. Edite a terceira instrução do método *DoWork* para 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 Application
{
    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 programa. Desta vez, os dois primeiros valores gravados 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 2010.



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. Se você reatribuisse o parâmetro a um objeto recém-construído, na verdade você também estaria reatribuindo o argumento ao objeto recém-construído.

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#, frequentemente dividem a memória utilizada para armazenar dados em duas partes separadas, cada uma gerenciada de uma maneira distinta. Esses dois blocos são tradicionalmente chamados pilha (*stack*) e *heap*. Pilha e heap servem para propósitos muito diferentes:

- 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 as variáveis locais é automaticamente liberada de volta para a pilha e fica disponível para ser reutilizada quando outro método for chamado.
- 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.

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:

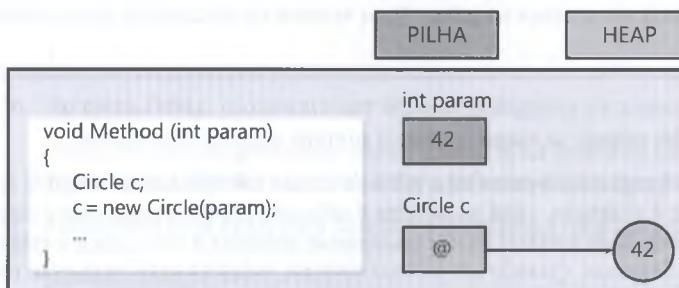
- A memória de pilha é organizada como uma pilha de caixas sobrepostas umas sobre as outras. Quando um método é chamado, cada parâmetro é colocado em uma caixa que é disposta na parte superior da pilha. Cada variável local é igualmente atribuída a uma caixa, e esta é colocada no topo da pilha de caixas. Quando um método termina, todas as suas 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 por 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.

Utilizando a pilha e o heap

Agora vamos examinar o que acontece quando o método *Method* é chamado a seguir:

```
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, um outro bloco de memória, grande o suficiente para armazenar uma referência (um endereço de memória), também é alocado da pilha, mas permanece não inicializado. (Esse bloco é 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 terminar, os parâmetros e variáveis locais sairão 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).

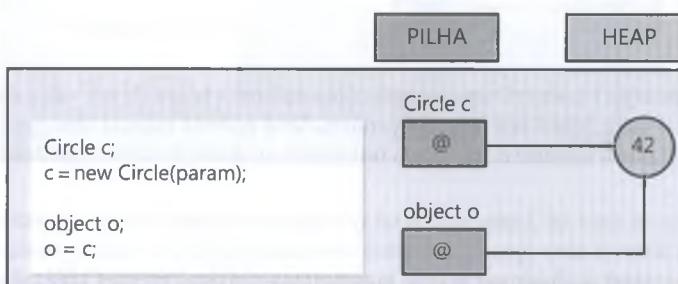
A classe *System.Object*

Um dos tipos-referência mais importante no Microsoft .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, “Trabalhando com 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 consistente com outras palavras-chave que são sinônimos para classes (como *string* para *System.String* e algumas outras que você descobrirá 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;
```

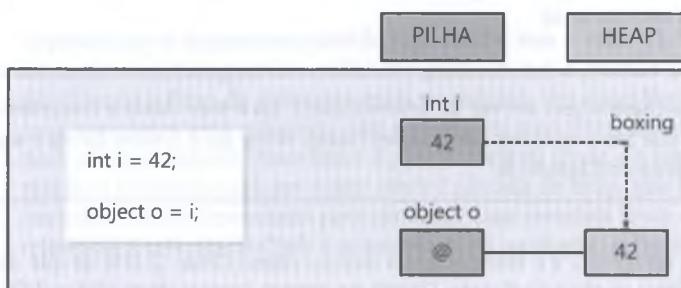


Boxing

Como você acabou de ver, as variáveis do tipo *object* podem referenciar qualquer objeto 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 requer uma pequena explicação para 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 referenciaria 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 tempo de execução 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 de uma variável, o valor no heap 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. Você poderia esperar conseguir acessar o valor *int* na forma boxed que a variável *o* referencia utilizando uma simples instrução de atribuição como esta:

```
int i = o;
```

Mas se tentar essa sintaxe, você receberá um erro de tempo de compilação. Se você pensar no assunto, é muito lógico que você não possa utilizar a sintaxe `int i = o;`. Afinal de contas, *o* pode estar referenciando qualquer coisa e não apenas um *int*. Considere o que aconteceria no código a seguir se essa instrução fosse permitida:

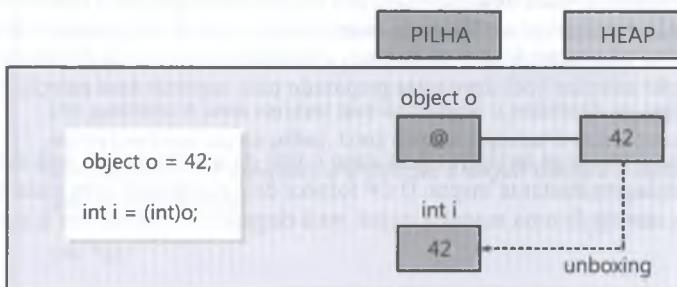
```
Circle c = new Circle();
int i = 42;
object o;

o = c; // o referencia um círculo
i = o; // o que está armazenado em i?
```

Para obter o valor da cópia boxed, você deve utilizar o que é conhecido como *casting*, uma operação que verifica se é seguro converter um tipo em outro e então faz a conversão. Você coloca o nome do tipo como prefixo da variável *object* entre parênteses, como neste exemplo:

```
int i = 42;
object o = i; // faz o boxing
i = (int)o; // compila corretamente
```

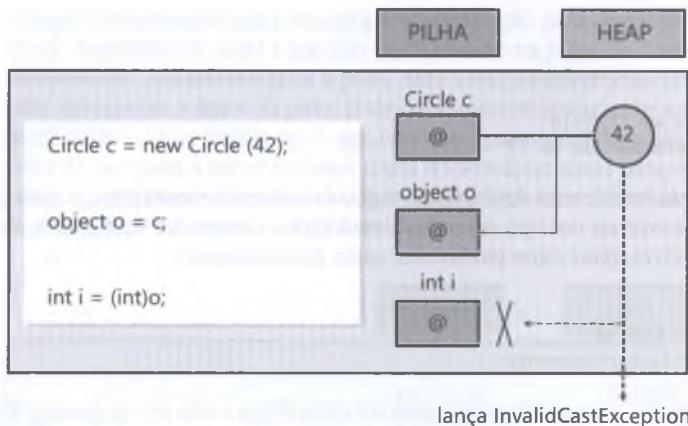
O efeito desse casting é sutil. O compilador nota que você especificou o tipo *int* no casting. Em seguida, o compilador gera um código para verificar o que *o* realmente referencia em tempo de execução. Poderia ser realmente qualquer coisa. Só porque seu casting diz que *o* referencia um *int*, isso não significa que ele realmente é um *int*. 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 boxed é então armazenado em *i*.) Isso é chamado de *unboxing*. O diagrama a seguir mostra o que está acontecendo:



Mas 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 *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 ocorre boxing porque Circle é uma variável-referência
int i = (int)o; // compila, mas lança uma exceção em tempo de execução
```

Você utilizará boxing e unboxing em exercícios posteriores. Lembre-se de que boxing e unboxing são operações caras devido à quantidade de verificação requerida 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 18, "Apresentando 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 C# confiará em você ao compilar o aplicativo, mas o runtime é mais desconfiado e realmente verificará se esse é o caso quando o aplicativo executar. 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 abordagem 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 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 iremos discuti-los novamente no Capítulo 12.

Ponteiros e código inseguro

Esta seção serve apenas para sua informação e dirige-se aos desenvolvedores que estão familiarizados com C ou C++. Se você é um 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 – os ponteiros.

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++ têm 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 que é fácil apontar para algo inválido ou simplesmente esquecer 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 a utilizar 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;
}
```

Ao compilar programas que contêm um código inseguro, você deve especificar a opção */unsafe*.

Um código inseguro também afeta a maneira como a memória é gerenciada; objetos criados em código inseguro são chamados de não gerenciados. Discutiremos esse problema com mais detalhe no Capítulo 14.

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. Você também aprendeu a utilizar as palavras-chave *ref* e *out* nos parâmetros de método para obter acesso aos argumentos. Você já 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. Você também viu 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 você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 9.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 8

Para	Faça isto
Copiar uma variável de tipo-valor	Simplesmente faça uma cópia. Como a variável é um tipo-valor, você terá duas cópias da mesma variável. Por exemplo:
	<pre>int i = 42; int copyi = i;</pre>
Copiar uma variável de tipo-referência	Simplesmente faça uma cópia. Como a variável é um tipo-referência, você terá duas referências ao mesmo objeto. Por exemplo:
	<pre>Circle c = new Circle(42); Circle refc = c;</pre>
Declarar uma variável que possa armazenar um tipo-valor ou o valor <i>null</i>	Declare a variável utilizando o modificador ? com o tipo. Por exemplo:
	<pre>int? i = null;</pre>
Passar um argumento a um parâmetro <i>ref</i>	Prefixe o argumento com a palavra-chave <i>ref</i> . 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:
	<pre>static void Main() { int arg = 42; DoWork(ref arg); Console.WriteLine(arg); }</pre>
Passar um argumento a um parâmetro <i>out</i>	Prefixe o argumento com a palavra-chave <i>out</i> . 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 o argumento real. Por exemplo:
	<pre>static void Main() { int arg = 42; DoWork(out arg); Console.WriteLine(arg); }</pre>
Fazer o boxing de um valor	Inicialize ou atribua uma variável do tipo <i>object</i> ao valor. Por exemplo:
	<pre>object o = 42;</pre>
Fazer o unboxing de 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>

(continua)

Para

Fazer o casting de um objeto de forma segura

Faça isto

Utilize o operador *is* para testar se o casting é válido.
Por exemplo:

```
WrappedInt wi = new WrappedInt();  
***  
object o = wi;  
if (o is WrappedInt)  
{  
    WrappedInt temp = (WrappedInt)o;  
    ***  
}
```

Outra alternativa é usar o operador *as* para fazer o casting e testar se o resultado é *null*. Por exemplo:

```
WrappedInt wi = new WrappedInt();  
***  
object o = wi;  
WrappedInt temp = o as WrappedInt;  
if (temp != null)  
    ***
```

Capítulo 9

Criando 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.

No Capítulo 8, “Entendendo valores e referências”, você aprendeu sobre os dois tipos fundamentais do Microsoft Visual C#: os *tipos-valor* e os *tipos-referência*. Uma variável de tipo-valor armazena seu valor diretamente na pilha, enquanto uma variável de tipo-referência armazena uma referência a um objeto no heap. No Capítulo 7, “Criando e gerenciando classes e objetos”, você aprendeu a criar seus próprios tipos-referência definindo classes. Neste capítulo, você aprenderá a criar seus próprios tipos-valor.

O C# suporta duas espécies de tipos-valor: *enumerações* e *estruturas*. Veremos uma de cada vez.

Trabalhando com 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 você atribua a ela um valor inteiro válido além de 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.

Declarando 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 }
```

Utilizando 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)
    {
        Season localVariable;
        ...
    }

    private Season currentSeason;
}
```

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. Por exemplo:

```
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:

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 permite 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. Por exemplo:

```
string name = colorful.ToString();
Console.WriteLine(name); // também escreve 'Fall'
```

Muitos dos operadores padrão que podem ser utilizados em variáveis do tipo inteiro também podem ser empregadas em variáveis de enumeração (exceto os *operadores bit a bit* e os *operadores de deslocamento*, que serão abordados no Capítulo 16, “Utilizando indexadores”). Por exemplo, você pode comparar a igualdade de duas variáveis de enumeração do mesmo tipo utilizando o operador de igualdade (`==`) e ainda executar cálculos aritméticos em uma variável de tipo enumerado (embora o resultado talvez nem sempre tenha um significado!).

Escolhendo 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. Lembre-se da discussão sobre unboxing no Capítulo 8, em 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* (na enumeração de *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 1 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 dispor de mais de um literal de enumeração ao mesmo valor subjacente. Por exemplo, no Reino Unido, o outono (*Fall*) é chamado de *Autumn*. Você pode agradar às duas culturas como mostrado a seguir:

```
enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }
```

Escolhendo o tipo subjacente de uma enumeração

Quando você declara uma enumeração, os literais de enumeração recebem valores do tipo *int*. Você também pode escolher 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*, pode fazer sentido utilizar um tipo menor de dados.

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. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto *StructsAndEnums*, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 9\StructsAndEnums na sua pasta Documentos.
3. Na janela *Code and Text Editor*, exiba o arquivo Month.cs.
O arquivo-fonte contém um namespace vazio chamado *StructsAndEnums*.
4. Adicione uma enumeração chamada *Month* para modelar os meses do ano dentro do namespace *StructsAndEnums*, como mostrado em negrito aqui. 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
    }
}
```

- 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.

- 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 primeira variável 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*.

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

O Visual Studio 2010 compila e executa o programa. Confirme que a palavra *January* está escrita no console.

- Pressione Enter para fechar o programa e retornar ao ambiente de programação do Visual Studio 2010.
- 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);
}
```

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

O Visual Studio 2010 compila e executa o programa. Confirme se as palavras *January* e *February* estão escritas no console.

Observe que realizar 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, o valor de enumeração correspondente é exibido.

11. Pressione Enter para fechar o programa e retornar ao ambiente de programação do Visual Studio 2010.
12. Modifique a primeira instrução no método *DoWork* para inicializar a primeira variável para *Month.December*, como mostrado em negrito aqui:

```
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 2010 compila e executa o programa. Desta vez, a palavra *December* é escrita no console, seguida pelo número 12. Embora você possa realizar a aritmética 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 2010.

Trabalhando com estruturas

Você viu no Capítulo 8 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 é frequentemente 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 aliases das estruturas *System.Int32*, *System.Int64* e *System.Single*, respectivamente. Essas estruturas têm campos e métodos, e você pode de fato 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. As instruções a seguir são todas elas válidas no C#:

```
int i = 99;
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. O uso dos métodos estáticos expostos por essas estruturas é muito mais comum. Por exemplo, nos capítulos anteriores você utilizou o método estático *Int32.Parse* para converter uma string no seu valor inteiro correspondente. O que você está fazendo é invocar 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 você pode armazenar 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
<i>bool</i>	<i>System.Boolean</i>	Estrutura
<i>byte</i>	<i>System.Byte</i>	Estrutura
<i>decimal</i>	<i>System.Decimal</i>	Estrutura
<i>double</i>	<i>System.Double</i>	Estrutura
<i>float</i>	<i>System.Single</i>	Estrutura
<i>int</i>	<i>System.Int32</i>	Estrutura
<i>long</i>	<i>System.Int64</i>	Estrutura
<i>object</i>	<i>System.Object</i>	Classe
<i>sbyte</i>	<i>System.SByte</i>	Estrutura
<i>short</i>	<i>System.Int16</i>	Estrutura
<i>string</i>	<i>System.String</i>	Classe
<i>uint</i>	<i>System.UInt32</i>	Estrutura
<i>ulong</i>	<i>System.UInt64</i>	Estrutura
<i>ushort</i>	<i>System.UInt16</i>	Estrutura

Declarando 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. Por exemplo, eis 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* como 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
{
    public Time(int hh, int mm, int ss)
    {
        hours = hh % 24;
        minutes = mm % 60;
        seconds = ss % 60;
    }

    public int Hours()
    {
        return hours;
    }

    ...
    private int hours, minutes, seconds;
}
```

 **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. Mas você pode declarar e implementar explicitamente operadores para seus próprios tipos de estrutura. A sintaxe para fazer isso é abordada no Capítulo 21, "Sobrecarga de operadores".

Utilize estruturas para implementar conceitos simples cujas características principais são seus valores. 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 em que 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.

Entendendo as diferenças entre estrutura e classe

Uma estrutura e uma classe são sintaticamente semelhantes, mas existem algumas diferenças importantes. Vamos examinar algumas dessas diferenças.

- 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 por que você não pode declarar seu próprio construtor padrão em uma estrutura é que o compilador *sempre* gera um. Em uma classe, o compilador só gera o construtor padrão se você não escrever seu próprio construtor. O construtor padrão gerado pelo compilador para uma estrutura sempre define os campos como *O*, *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. Se não quiser utilizar esses valores padrão, você pode inicializar os campos como valores diferentes fornecendo um construtor não padrão. Mas se você não inicializar um campo no seu construtor não padrão, o compilador não o inicializará para você. Isso significa que você deve inicializar explicitamente todos os campos em todos os construtores de estrutura ou receberá um erro de tempo de compilação. Por exemplo, embora o exemplo seguinte seja compilado e inicialize silenciosamente *seconds* como *O* 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 inicializado
}
```

- 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 <i>valores</i> e residem na pilha.	As instâncias de classe são chamadas <i>objetos</i> 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 os campos de instância em seus pontos de declaração?	Não	Sim

Existem outras diferenças entre classes e estruturas no que se refere à herança. Essas diferenças serão abordadas no Capítulo 12.

Declarando 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** Você pode criar uma versão nullable de uma variável de estrutura utilizando o modificador `?`. Você pode então atribuir o valor `null` à variável:

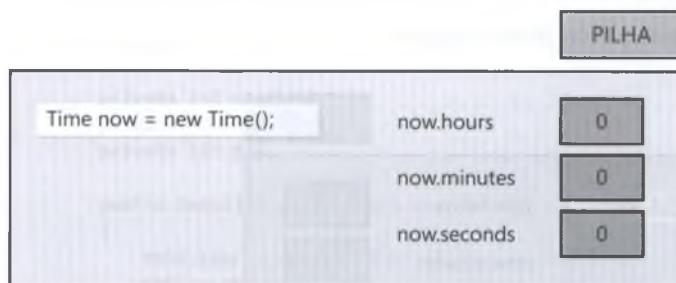
```
Time? currentTime = null;
```

Entendendo a inicialização de estruturas

Anteriormente neste capítulo, vimos como os campos em uma estrutura podem ser inicializados utilizando um construtor. Se você chamar um construtor, as várias regras descritas anteriormente garantirão que todos os campos na estrutura sejam inicializados:

```
Time now = new Time()
```

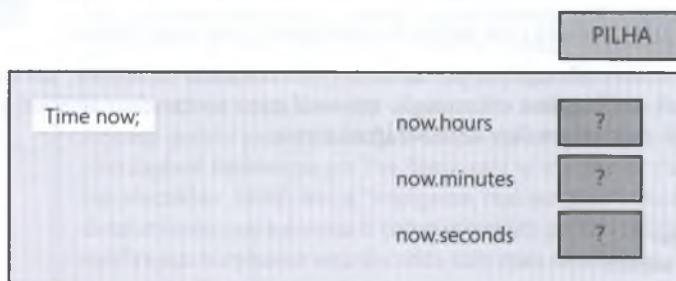
O gráfico a seguir ilustra os campos dessa estrutura:



Entretanto, como as estruturas são tipos-valor, você pode criar variáveis de estrutura sem chamar um construtor, como exemplo a seguir:

```
Time now;
```

Dessa vez, a variável é criada, mas seus campos permanecem no estado não inicializado. O gráfico 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 escreveu seu próprio construtor de estrutura, você pode também utilizá-lo para inicializar uma variável de estrutura. Conforme explicado anteriormente neste capítulo, um construtor de estrutura deve sempre inicializar explicitamente todos os seus campos. Por exemplo:

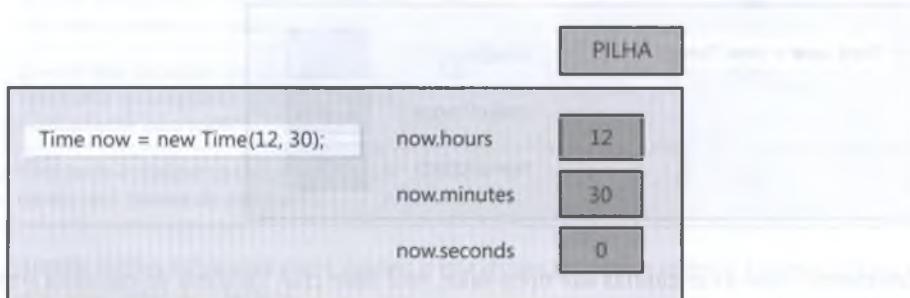
```
struct Time
{
    private int hours, minutes, seconds;
    ...
}
```

```
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 irá 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 irá 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* 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 selecionar um século diferente se preferir) e, quando o campo *day* contiver

o valor D , esse valor represente o dia $D + 1$. O construtor padrão irá então configurar os três campos como valores que representam a data de 1 de janeiro de 1900.

- 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;
    }
}
```

- 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, 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:

```
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* permite formatar dados, e opera de modo semelhante ao do 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 a string formatada como seu resultado.

5. Exiba o arquivo Program.cs na janela *Code and Text Editor*.
6. No método *DoWork*, transforme em comentários as quatro instruções existentes, adicione um código ao método *DoWork* para declarar uma variável local chamada *defaultDate* e inicialize-a como 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* agora deve ser semelhante a este:

```
static void DoWork()
{
    ***
    Date defaultDate = new Date();
    Console.WriteLine(defaultDate);
}
```

7. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o programa. Verifique se a data *January 1 1900* foi escrita no console.
8. Pressione a tecla *Enter* para retornar ao ambiente de programação do Visual Studio 2010.
9. 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 inicialize-a como July 4, 2010. (Ironicamente, eu me casei no Dia da Independência e, consequentemente, perdi a minha independência!) 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(2010, Month.July, 4);
    Console.WriteLine(weddingAnniversary);
}
```



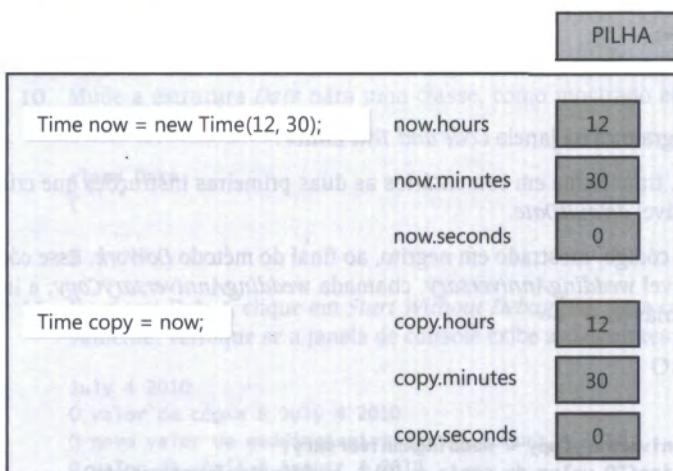
Nota Quando você digitar a palavra-chave *new*, o IntelliSense detectará automaticamente que existem dois construtores disponíveis para o tipo *Date*.

10. No menu *Debug*, clique em *Start Without Debugging*. Confirme se *July 4, 2010* está escrito no console abaixo da informação anterior.
11. Pressione Enter para fechar o programa.

Copiando 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. O gráfico mostra os resultados da atribuição.

```
Time now = new Time(12, 30);
Time copy = now;
```



A compilação do exemplo a seguir falha porque *now* não está inicializada:

```
Time now;
Time 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 Os programadores C++ devem observar que esse comportamento de cópia 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 seguinte método 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.

```
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 comentários as duas primeiras instruções 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("O valor da cópia é {0}", weddingAnniversaryCopy);
    }
```

6. Adicione as seguintes instruções ao final do método *DoWork*, que chamam o método *AdvanceMonth* da variável *weddingAnniversary*, e depois exibem o valor das variáveis *weddingAnniversary* e *weddingAnniversaryCopy*:

```
static void DoWork()
{
    /**
     * weddingAnniversaryCopy.AdvanceMonth();
     * Console.WriteLine("O novo valor de weddingAnniversary é {0}", weddingAnniversary);
     * Console.WriteLine("O valor da cópia é {0}", weddingAnniversaryCopy);
    }
```

7. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo. Verifique se a janela do console exibe as seguintes mensagens:

```
July 4 2010
O valor da cópia é July 4 2010
O novo valor de weddingAnniversary é July 4 2010
O valor da cópia é August 4 2010
```

A primeira mensagem mostra o valor inicial da variável *weddingAnniversary* (*July 4 2010*). A segunda mensagem exibe o valor da variável *weddingAnniversaryCopy*. Observe que ela contém uma cópia da data armazenada na variável *weddingAnniversary* (*July 4 2010*). A terceira mensagem exibe o valor da variável *weddingAnniversary* após mudar o mês da variável *weddingAnniversaryCopy* para *August 4 2010*. Observe que seu valor original de *July 4 2010* não mudou. A última mensagem exibe o valor da variável *weddingAnniversaryCopy*. Você pode ver que esse valor mudou para *August 4 2010*.

8. Pressione Enter e retorne ao Visual Studio 2010.
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 construir e executar o aplicativo novamente. Verifique se a janela de console exibe as seguintes mensagens:

```
July 4 2010
O valor da cópia é July 4 2010
O novo valor de weddingAnniversary é August 4 2010
O valor da cópia é August 4 2010
```

As duas primeiras mensagens e a quarta mensagem são as mesmas anteriores. Entretanto, a terceira mensagem mostra que o valor da variável *weddingAnniversary* mudou para *August 4 2010*. Lembre-se de que uma estrutura é um tipo-valor, e ao copiar uma variável tipo-valor, você cria uma cópia de todos os dados contidos na variável. Contudo, uma classe é um tipo-referência, e ao copiar uma variável tipo-referência, você copia uma referência à variável original. Se os dados contidos em uma variável de classe forem alterados, todas as referências a essa variável verão as alterações.

12. Pressione Enter para voltar ao Visual Studio 2010.

Neste capítulo, vimos como criar enumerações e estruturas. Você conheceu algumas semelhanças e diferenças entre uma estrutura e uma classe e vimos como definir construtores para inicializar os campos em uma estrutura. Você também aprendeu a representar uma estrutura como uma string, substituindo o método *ToString*.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 10.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 9

Para	Faça isto
Declarar uma enumeração	Escreva a palavra-chave <code>enum</code> , seguida pelo nome do tipo, seguido por um par de chaves contendo uma lista separada por vírgulas dos nomes literais da enumeração. Por exemplo: <code>enum Season { Spring, Summer, Fall, Winter }</code>
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 uma variável de tipo enumerado a um valor	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 <code>struct</code> , seguida pelo nome do tipo-estrutura, seguido pelo corpo da estrutura (os construtores, métodos e campos). Por exemplo: <pre>struct Time { public Time(int hh, int mm, int ss) { ... } private int hours, minutes, seconds; }</pre>
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 chamando o construtor de estrutura. Por exemplo: <code>Time lunch = new Time(12, 30, 0);</code>

Capítulo 10

Utilizando arrays e coleções

Neste capítulo, você vai aprender a:

- Declarar, inicializar e utilizar variáveis de array.
- Declarar, inicializar e utilizar variáveis de diversos tipos de coleção.

Você já viu como criar e utilizar tipos diferentes de variáveis. Mas todos os exemplos de variáveis vistos até agora têm algo em comum – eles armazenam informações sobre um único item (*int*, *float*, *Circle*, *Date* e assim por diante). O que acontece se você precisar manipular um conjunto de itens? Uma solução é criar uma variável para cada item do conjunto, mas isso levanta várias questões adicionais: de quantas variáveis você precisa? Como você deve nomeá-las? Se precisasse executar a mesma operação em cada item do conjunto (como incrementar cada uma das variáveis em um conjunto de inteiros), como evitaria 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?

Arrays e coleções fornecem mecanismos que ajudam a resolver os problemas colocados por essas questões.

O que é um array?

Um *array* é uma sequência não ordenada de elementos. Todos os elementos em um array têm o mesmo tipo (ao contrário dos campos em uma estrutura ou classe, que têm tipos diferentes). Os elementos de um array residem em um bloco contíguo da memória e são acessados por meio de um índice inteiro (ao contrário dos campos em uma estrutura ou classe, que são acessados pelo nome).

Declarando 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*, você escreveria:

```
int[] pins; // Personal Identification Numbers
```

Os programadores em Microsoft Visual Basic devem perceber o uso de colchetes no lugar de parênteses. Os programadores C e C++ devem observar que o tamanho do array não faz parte da declaração. Os programadores Java devem observar que você tem que colocar os colchetes *antes* do nome da variável.



Nota Você não está restrito aos tipos primitivos como elementos de array. Você também pode criar arrays de estruturas, enumerações e classes. Por exemplo, você pode desenvolver um array de estruturas *Time* assim:

```
Time[] times;
```



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*).

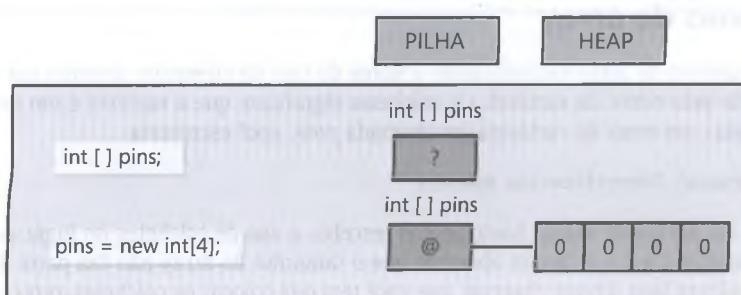
Criando uma instância de array

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, e esse bloco de memória não armazena seus elementos de array diretamente na pilha, como uma estrutura armazena. (Para recapitular valores e referências, e as diferenças entre pilha e heap, consulte o Capítulo 8, "Entendendo valores e referências".) Lembre-se de que, quando você declara uma variável de classe, a memória não é alocada para o objeto até que você crie a instância utilizando *new*. Os arrays seguem as mesmas regras – ao declarar uma variável de array, você não declara seu tamanho. Você especifica o tamanho de um array somente quando realmente cria uma instância do array.

Para criar uma instância de array, você utiliza a palavra-chave *new* seguida pelo tipo de elemento, seguido pelo tamanho do array que você está criando entre colchetes. Criar um array também inicializa seus elementos utilizando 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 anteriormente, você escreve o seguinte:

```
pins = new int[4];
```

A ilustração a seguir oferece uma representação visual dos efeitos dessa instrução:



O tamanho de uma instância de array não tem de ser uma constante; ela pode ser calculada em tempo de execução, como mostrado neste exemplo:

```
int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];
```

Você pode criar um array cujo tamanho é 0. Isso talvez pareça estranho, mas é útil em situações em que o tamanho do array é determinado dinamicamente e pode até ser 0. Um array de tamanho 0 não é um array *null*.

Inicializando variáveis de array

Quando você cria uma instância de array, todos os elementos dessa instância são inicializados com um valor padrão dependendo do seu tipo. 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 neste exemplo:

```
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 0 a *Int32.MaxValue* por padrão. O método *Next* é sobrecarregado 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 número aleatório com um valor de semente baseado na data/hora, o que reduz a possibilidade de a classe duplicar uma sequência de números aleatórios. Uma versão sobrecarregada do construtor permite fornecer seu próprio valor de semente. Assim, você 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, você pode omitir a expressão `new` e o tamanho do array. O compilador calcula o tamanho a partir do número de inicializadores e gera o código para criar o array. Por exemplo:

```
int[] pins = {9, 3, 7, 2};
```

Se criar um array de estruturas, você poderá inicializar cada estrutura do array chamando o construtor da estrutura, como mostrado neste exemplo:

```
Time[] schedule = { new Time(12,30), new Time(5,30) };
```

Criando 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 anterior, 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 no array para você, assim:

```
var names = new[]{"John", "Diana", "James", "Francesca"};
```

Neste 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 têm 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 melhor para o array implicitamente tipado):

```
var bad = new[]{"John", "Diana", 99, 100};
```

Mas, em alguns casos, o compilador converterá elementos em um tipo diferente se isso fizer sentido. No código a seguir, o array `numbers` é um array de `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 misturar tipos esperando que o compilador converta-os para você.

Arrays implicitamente tipados são mais úteis quando você está trabalhando com tipos anônimos, descritos no Capítulo 7, “Criando e gerenciando classes 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 (sim, sou mais jovem que minha esposa):

```
var names = new[] { new { Name = "John", Age = 44 },
    new { Name = "Diana", Age = 45 },
    new { Name = "James", Age = 17 },
    new { Name = "Francesca", Age = 15 } };
```

Os campos nos tipos anônimos devem ser os mesmos para cada elemento do array.

Acessando um elemento individual de um array

Para acessar um elemento individual de um array, você deve fornecer um índice indicando que elemento você quer. 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;
```

Os índices do array são baseados em zero. 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.

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 *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)
{
    ...
}
```

Iterando por um array

Todos os arrays são instâncias da classe *System.Array* no 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 por que não é necessário usar chaves para chamá-la. Você aprenderá sobre as propriedades no Capítulo 15, “Implementando 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* para que você possa 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 variável *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 no array. A instrução *foreach* é a maneira preferida de iterar por um array; ela expressa a intenção do código diretamente, 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 você 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 você 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.

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 = 44 },
                    new { Name = "Diana", Age = 45 },
                    new { Name = "James", Age = 17 },
                    new { Name = "Francesca", Age = 15 } };
foreach (var familyMember in names)
{
    Console.WriteLine("Name: {0}, Age: {1}", familyMember.Name, familyMember.Age);
}
```

Copiando 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 do array, você realmente acaba ficando com duas referências à mesma instância do array – por exemplo:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias e pins referenciam a mesma instância de array
```

Neste 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. Primeiro, você precisará criar uma nova instância do mesmo tipo e do mesmo tamanho do array que está sendo copiado, e depois copiar cada elemento de dado 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 < copy.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 – é 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 um determinado índice inicial:

```
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 alvo antes de chamar *Copy*:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

Ainda outra alternativa é 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** O método *Clone* na verdade retorna um tipo *object*, razão por que você deve fazer o casting do array para o tipo apropriado ao utilizá-lo. Além disso, todas as quatro maneiras de copiar mostradas anteriormente criam uma cópia *superficial* de um array – se os elementos no array sendo copiados contiverem referências, o loop *for* como codificado e os três métodos anteriores simplesmente copiam as referências em vez dos objetos referenciados. 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*.

Utilizando arrays multidimensionais

Os arrays apresentados até agora abrangiam apenas uma dimensão, e você pode considerá-los listas simples de valores. É possível criar arrays com mais de uma dimensão. Por exemplo, para criar um array bidimensional, especifique um array que necessite de dois índices de inteiros. O código a seguir gera um array bidimensional de 24 inteiros, chamado *items*. Se ajudar, você pode visualizar um array bidimensional como uma tabela, onde a primeira dimensão especifica o número de linhas e a segunda especifica o número de colunas.

```
int[,] items = new int[4, 6];
```

Para acessar um elemento no 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 na célula(2, 3) como 99
items[2, 4] = items[2, 3]; // copia o elemento contido na célula(2,3) para a célula(2,4)
items[2, 4]++;             // incrementa o valor inteiro na célula(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, 1] = cube[1, 2, 1] * 3;
```

Neste estágio, recomendamos cuidado ao criar arrays com mais de três dimensões. Em termos específicos, os arrays podem consumir muita memória. O array *cube* contém 125 elementos ($5 * 5 * 5$). Um array quadridimensional, em que cada dimensão tem um tamanho de 5, contém 625 elementos. Geralmente, você deve sempre estar preparado para capturar e tratar as exceções de *OutOfMemoryException*, ao utilizar arrays multidimensionais.

Utilizando arrays para jogar cartas

No exercício a seguir, você utilizará arrays para implementar um aplicativo que distribui as cartas como parte de um jogo de cartas. O aplicativo exibe um formulário WPF (Windows Presentation Foundation) 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 2010, se ele ainda não estiver em execução.
2. Abra o projeto *Cards*, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 10\Cards Using Arrays, contida em sua pasta Documentos.
3. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.

Será exibido um formulário WPF com a legenda *Card Game*, quatro caixas de texto (intituladas *North*, *South*, *East* e *West*), e um botão com a legenda *Deal* (distribuir).

4. Clique em *Deal*.
Será exibida uma mensagem como texto “DealCardFromPack – TBD”. Você ainda não implementou o código que dá as cartas.
5. Clique em *OK*, e feche a janela *Card Game* para retornar ao Visual Studio 2010.
6. Na janela *Code and Text Editor*, exiba o arquivo *Value.cs*.

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. Exiba 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* e essa classe modela uma única classe do jogo.

```

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;
    }
}

```

Esta 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 um par de métodos, chamados *CardValue* e *CardSuit*, que retornam essas informações, e ela substitui 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. Você aprenderá a fazer isso no Capítulo 15, “Implementando propriedades para acessar campos”.

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 objeto *Random*. A classe *Random* é um gerador de números aleatórios, e você utilizará *randomCardSelector* para embaralhar as cartas antes de serem distribuídas em cada mão.

```
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 *to do* (a fazer). Exclua o comentário e adicione a instrução mostrada em negrito para instanciar o array *cardPack* com o número correto de elementos:

```
public Pack()
{
    this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
}
```

11. Adicione ao construtor *Pack* o seguinte código mostrado em negrito. O loop *for* externo itera sobre a lista de valores na 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 valor especificados e o adiciona ao elemento pertinente no array *cardPack*.

```
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);
    }
}
```

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, retorná-la e depois remover a carta do baralho, para impedir que ela seja novamente selecionada.

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-a 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 no baralho do naipe selecionado. Resolva essa situação e escolha outro naipe, se necessário.

13. Localize o método *IsSuitEmpty*. O objetivo desse método é aceitar um parâmetro *Suit* e retornar um valor booleano que indica se ainda existem outras cartas desse naipe no baralho. Exclua o comentário e a instrução que lança a exceção *NotImplementedException* desse método, e adicione o seguinte código, que aparece 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 de cartas e determina se ainda existe alguma carta no array *cardPack*, com o naipe e o valor especificados, por meio do método *IsCardAlreadyDealt*, o que encerrará a etapa seguinte. 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.

14. 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 dar uma carta, o método *DealFromPack* a retira do array *cardPack* e define o elemento correspondente com *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 no array *cardPack* correspondente ao naipe e ao valor for *null*, e retorna *false* caso contrário.

15. Volte ao método *DealCardFromPack*. Adicione o seguinte loop *while* depois do código que seleciona um naipe aleatoriamente. Esse loop chama o método *IsSuitEmpty* para detectar se ainda existem cartas do naipe especificado, no baralho. Se não existirem, ele selecionará outro naipe aleatoriamente (ele pode até escolher o mesmo naipe novamente) e verificará novamente. 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);
    }
}
```

16. 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 anteriormente, não é possível assegurar que a carta com o valor escolhido ainda não tenha sido dada. Entretanto, você pode utilizar a mesma solução de antes, chamar o método *IsCardAlreadyDealt* para detectar se a carta já foi dada, e, nesse caso, escolher outra carta aleatória, e tentar novamente, repetindo o processo até que a carta seja encontrada. Para isso, adicione as seguintes instruções ao método *DealCardFromPack*:

```
public PlayingCard DealCardFromPack()
{
    ...
    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
    while (this.IsCardAlreadyDealt(suit, value))
    {
        value = (Value)randomCardSelector.Next(CardsPerSuit);
    }
}
```

17. Você acabou de selecionar uma carta aleatória que ainda não foi distribuída. Adicione o seguinte código 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;
}
```

18. A próxima etapa é adicionar a carta selecionada a uma mão. Abra o arquivo *Hand.cs* e exiba-o na janela *Code and Text Editor*. Esse arquivo contém a classe *Hand*, que implementa uma 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 tamanho de uma mão de cartas (13); o arquivo também contém um array de objetos *PlayingCard*, inicializado por meio da constante *HandSize*. O campo *playingCardCount* é 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 que aparecem em negrito:

```
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 e lança uma exceção *ArgumentException*, se ela existir. 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 Game.xaml e abra o arquivo Game.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*. O código é semelhante ao seguinte:

```
private void dealClick(object sender, RoutedEventArgs e)
{
    try
    {
        pack = new Pack();

        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);
            }
        }

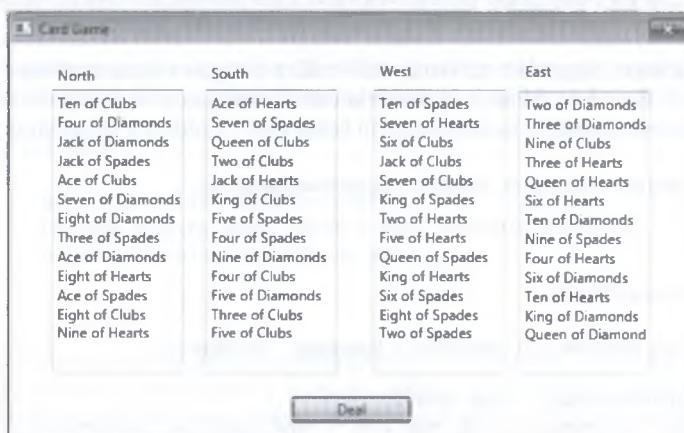
        north.Text = hands[0].ToString();
        south.Text = hands[1].ToString();
        east.Text = hands[2].ToString();
        west.Text = hands[3].ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButton.OK,
messageBoxImage. Error);
    }
}
```

A primeira instrução no bloco *try* cria um novo baralho. O loop *for* externo gera quatro mãos desse baralho e as armazena em um array chamado *hands*. 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.

Quando todas as cartas estiverem distribuídas, cada mão será exibida 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 manipulação *catch* exibirá uma caixa de mensagem com a mensagem de erro da exceção.

21. No menu *Debug*, clique em *Start Without Debugging*. Quando a janela Card Game for exibida, clique em *Deal*. A carta do baralho deve ser dada aleatoriamente a cada mão, e as cartas de cada mão devem ser exibidas no formulário, como mostra a imagem a seguir:



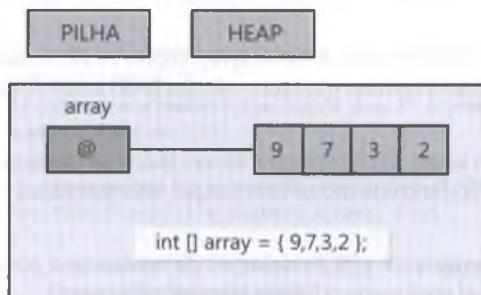
22. Clique em *Deal* novamente. Um novo conjunto de mãos é distribuído, e as cartas de cada mão mudam.
23. Feche a janela *Card Game* e retorne ao Visual Studio.

O que são classes de coleção?

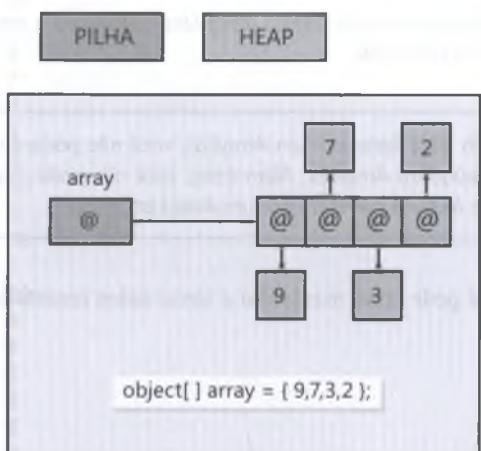
Arrays são úteis, mas têm suas limitações – uma das mais óbvias é que você precisa utilizar um índice inteiro para acessar os elementos no array. Felizmente, arrays são apenas uma das maneiras de colecionar elementos do mesmo tipo. O Microsoft .NET Framework fornece diversas classes que também colecionam elementos de outras maneiras especializadas. São as classes *Collection*, que residem no namespace *System.Collections* e seus subnamespaces.

Além da questão dos índices, existe outra diferença básica entre um array e uma coleção. Um array pode armazenar tipos-valor. As classes de coleção básicas aceitam, armazenam e retornam seus elementos como *tipos-objetos* – isto é, o tipo de elemento de uma classe de coleção é um *object*. Para entender suas implicações, é útil comparar um array de variáveis *int* (*int* é tipo-valor) com um array

de objetos (*object* é um tipo-referência). Como *int* é um tipo-valor, um array de variáveis *int* armazena seus valores *int* diretamente, como mostrado na imagem a seguir:



Agora considere o efeito quando o array é um array de objetos. Você ainda pode adicionar valores inteiros a esse array. (Na realidade, você pode adicionar valores de qualquer tipo a ele.) Quando você acrescenta um valor inteiro, ele automaticamente sofre boxing, e o elemento do array (uma referência a um objeto) referencia a cópia na forma boxed do valor inteiro. De modo semelhante, ao remover um valor de um array de objetos, você deve fazer unboxing utilizando um casting. (Para obter mais detalhes sobre como fazer um boxing, consulte o Capítulo 8.) A imagem a seguir mostra um array de *object* preenchido com valores inteiros:



As seções a seguir fornecem uma visão geral muito rápida das quatro classes de coleção mais úteis. Consulte a documentação do Microsoft .NET Framework Class Library para obter mais detalhes sobre cada classe.

Nota Existem outras classes de coleção que nem sempre usam *object* como seu tipo de elemento e que podem armazenar tipos-valor, assim como tipos-referência, mas você precisa saber um pouco mais sobre C# antes de conhecê-las. Você encontrará essas classes de coleção no Capítulo 18, "Apresentando genéricos".

A classe de coleção *ArrayList*

ArrayList é uma classe útil para reunir elementos em um array. Há certas ocasiões em que um array comum pode ser muito restritivo:

- 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 qualquer referência ao array original para que ela se refira 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 *ArrayList* oferece os seguintes recursos para ajudar a superar essas restrições:

- Você pode remover um elemento de um *ArrayList* utilizando o método *Remove*. A classe *ArrayList* reordena automaticamente seus elementos.
- Você pode adicionar um elemento ao final de um *ArrayList* utilizando o método *Add*. Você fornece o elemento a ser adicionado. O *ArrayList* se redimensiona, se necessário.
- Você pode inserir um elemento no meio de um *ArrayList* utilizando o método *Insert*. Novamente, o *ArrayList* se redimensiona, se necessário.
- Você pode referenciar um elemento existente em um objeto *ArrayList* utilizando a notação de array normal, com colchetes e o índice do elemento.



Nota Como com arrays, se utilizar *foreach* para iterar por um *ArrayList*, você não poderá utilizar a variável de iteração para modificar o conteúdo do *ArrayList*. Além disso, você não pode chamar o método *Remove*, *Add* ou *Insert* em um loop *foreach* que itera por um *ArrayList*.

Observe um exemplo que mostra como você pode criar, manipular e iterar pelos conteúdos de um *ArrayList*:

```
using System;
using System.Collections;
...
ArrayList numbers = new ArrayList();
...
// preenche o ArrayList
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 sendo inserido)
numbers.Insert(numbers.Count-1, 99);

// remove o primeiro elemento cujo valor é 7 (o elemento 4, índice 3)
numbers.Remove(7);
// remove o elemento que agora é o 7º elemento, índice 6 (10)
numbers.RemoveAt(6);

*** 
// itera pelos 10 elementos restantes utilizando uma instrução for
for (int i = 0; i < numbers.Count; i++)
{
    int number = (int)numbers[i]; // observe o casting, que faz o unboxing do valor
    Console.WriteLine(number);
}

*** 
// itera pelos 10 elementos restantes utilizando uma instrução foreach
foreach (int number in numbers) // nenhum casting é necessário
{
    Console.WriteLine(number);
}
```

A saída deste código é mostrada aqui:

```
10
9
8
7
6
5
4
3
2
99
1
10
9
8
7
6
5
4
3
2
99
1
```

Nota A maneira como você determina o número de elementos para um *ArrayList* é diferente de consultar o número de itens em um array. Ao utilizar um *ArrayList*, você examina a propriedade *Count* e, ao utilizar um array, você examina a propriedade *Length*.

A classe de coleção Queue

A classe *Queue* implementa um mecanismo FIFO (*first-in first-out*; primeiro a entrar, primeiro a sair). Um elemento é inserido no final da fila (a operação de enfileiramento) e é removido no início da fila (a operação de desenfileiramento).

Observe o exemplo de uma fila e suas operações:

```
using System;
using System.Collections;

...
Queue numbers = new Queue();

...
// preenche a fila
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Enqueue(number);
    Console.WriteLine(number + " has joined the queue"); //entrou na fila
}

...
// itera pela fila
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

...
// esvazia a fila
while (numbers.Count > 0)
{
    int number = (int)numbers.Dequeue(); // casting necessário para fazer o unboxing do valor
    Console.WriteLine(number + " has left the queue"); // saiu da fila
}
```

A saída desse código é:

```
9 has joined the queue
3 has joined the queue
7 has joined the queue
2 has joined the queue
9
3
7
2
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

A classe *Stack* 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 raramente utilizado e inevitavelmente precisará ser lavado antes de você poder colocar qualquer comida nele, uma vez que estará completamente sujo!) Observe o exemplo:

```
using System;
using System.Collections;

...
Stack numbers = new Stack();

// preenche a pilha
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Push(number);
    Console.WriteLine(number + " has been pushed on the stack"); //empilhou o valor
}

// itera pela pilha
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// esvazia a pilha
while (numbers.Count > 0)
{
    int number = (int)numbers.Pop();
    Console.WriteLine(number + " has been popped off the stack"); //desempilhou o valor
}
```

A saída do programa é:

```
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
2
7
3
9
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 *Hashtable*

Os tipos array e *ArrayList* fornecem um meio de mapear um índice inteiro para um elemento. Você fornece 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, você pode querer fornecer um mapeamento em que o tipo de origem do mapeamento não será um *int*, mas algum outro tipo qual-

quer, como *string*, *double* ou *Time*. Em outras linguagens, isso costuma ser chamado de *array associativo*. A classe *Hashtable* fornece essa funcionalidade mantendo internamente dois arrays *object*, 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 um *Hashtable*, ela determina automaticamente qual chave pertence a qual valor e permite que você recupere o valor que está associado à chave especificada rápida e facilmente. Há algumas consequências importantes no projeto da classe *Hashtable*:

- Uma *Hashtable* 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. Você pode testar se uma *Hashtable* já contém uma chave específica utilizando o método *ContainsKey*.
- Internamente, uma *Hashtable* é uma estrutura de dados esparsa que opera melhor quando tem bastante memória para funcionar. O tamanho de uma *Hashtable* na memória pode aumentar rapidamente à medida que você insere mais elementos.
- Quando utiliza uma instrução *foreach* para iterar por uma *Hashtable*, você obtém uma *DictionaryEntry*. A classe *DictionaryEntry* fornece acesso aos elementos chave e valor em ambos os arrays por meio das propriedades *Key* e *Value*.

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;
...
Hashtable ages = new Hashtable();
...
// preenche a Hashtable
ages["John"] = 44;
ages["Diana"] = 45;
ages["James"] = 17;
ages["Francesca"] = 15;
...
// itera utilizando uma instrução foreach
// o iterador gera um objeto DictionaryEntry contendo um par chave/valor
foreach (DictionaryEntry element in ages)
{
    string name = (string)element.Key;
    int age = (int)element.Value;
    Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

A saída do programa é:

```
Name: Diana, Age: 45
Name: James, Age: 17
Name: Francesca, Age: 15
Name: John, Age: 44
```

A classe de coleção *SortedList*

A classe *SortedList* é muito semelhante à classe *Hashtable* no sentido de que permite associar chaves a valores. A principal diferença é que o array de chaves está sempre ordenado. (Afinal de contas, ele se chama *SortedList*, ou seja, lista ordenada.)

Quando você insere um par chave/valor em uma *SortedList*, 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* garante automaticamente que as chaves e valores sejam alinhados, mesmo quando você adiciona e remove elementos. Isso significa que você pode inserir pares chave/valor em uma *SortedList* em qualquer sequência; eles sempre são ordenados com base no valor das chaves.

Como a classe *Hashtable*, uma classe *SortedList* não pode conter chaves duplicadas. Quando você utiliza uma instrução *foreach* para iterar por uma *SortedList*, recebe uma *DictionaryEntry*. Mas os objetos de *DictionaryEntry* 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 essa versão foi ajustada para utilizar uma classe *SortedList* em vez de uma *Hashtable*:

```
using System;
using System.Collections;
...
SortedList ages = new SortedList();
...
// preenche a SortedList
ages["John"] = 44;
ages["Diana"] = 45;
ages["James"] = 17;
ages["Francesca"] = 15;
...
// itera utilizando uma instrução foreach
// o iterador gera um objeto DictionaryEntry contendo um par chave/valor
foreach (DictionaryEntry element in ages)
{
    string name = (string)element.Key;
    int age = (int)element.Value;
    Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

A saída desse programa está ordenada alfabeticamente pelos nomes dos membros da minha família:

```
Name: Diana, Age: 45
Name: Francesca, Age: 15
Name: James, Age: 17
Name: John, Age: 44
```

Utilizando 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 um *ArrayList*, *Enqueue* para um *Queue*, *Push* para uma *Stack* e assim por diante). Você também pode inicializar *alguns* tipos de coleção ao declará-los, utilizando uma sintaxe muito semelhante àquela suportada por arrays. Por exemplo, a instrução a seguir cria e inicializa o objeto *ArrayList* chamado *numbers* mostrado anteriormente, demonstrando uma técnica alternativa de chamar repetidamente o método *Add*:

```
ArrayList numbers = new ArrayList(){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*. Consequentemente, você só pode utilizar essa sintaxe para coleções que realmente suportam o método *Add*. (As classes *Stack* e *Queue* não suportam.)

Para coleções mais complexas como *Hashtable*, que recebem pares chave/valor, você pode especificar cada par chave/valor como um tipo anônimo na lista inicializadora, como mostrado aqui:

```
Hashtable ages =
    new Hashtable(){ {"John", 44}, {"Diana", 45}, {"James", 17}, {"Francesca", 15}};
```

O primeiro item em cada par é a chave e o segundo é o valor.

Comparando arrays e coleções

Veja um resumo das principais diferenças entre arrays e coleções:

- Um array declara o tipo dos elementos que ele armazena, enquanto a coleção não faz isso devendo ao fato de as coleções armazenarem seus elementos como objetos.
- Uma instância de array tem um tamanho fixo e não pode aumentar ou 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 em uma coleção podem eles próprios ser coleções, permitindo que você simule um array multidimensional como uma coleção de coleções.

Utilizando classes de coleção para jogar cartas

No próximo exercício, você converterá o jogo de cartas desenvolvido no exercício anterior para utilizar coleções em vez de arrays.

Use coleções para implementar um jogo de cartas

1. Volte ao projeto Cards do exercício anterior.



Nota Existe uma versão completa do projeto do exercício anterior na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 10\Cards Using Arrays – Complete, na sua pasta Documentos.

- Exiba o arquivo Pack.cs, na janela *Code and Text Editor*. Observe a seguinte instrução *using*, perto do início do arquivo.

```
using System.Collections;
```

As classes de coleção estão localizadas nesse namespace.

- Na classe *Pack*, mude a definição do array bidimensional *cardPack* para um objeto *HashTable*, como mostrado em negrito a seguir:

```
class Pack
{
    ...
private Hashtable cardPack;
    ...
}
```

Lembre-se de que a classe *Hashtable* define uma coleção de *tipos-objeto*, e você não especifica o tipo *PlayingCard*. Além disso, o array original tinha duas dimensões, enquanto uma *Hashtable* só tem uma. Você emulará um array bidimensional ao utilizar os objetos da coleção *SortedList* como elementos na *Hashtable*.

- Localize o construtor de *Pack*. Modifique a primeira instrução nesse construtor para instanciar a variável *cardPack* como um novo objeto *Hashtable*, em vez de um array, como mostrado a seguir, em negrito:

```
public Pack()
{
    this.cardPack = new Hashtable();
    ...
}
```

- No loop externo *for*, declare um objeto de coleção *SortedList*, chamado *cardsInSuit*. Mude o código no loop *for* interno, de modo a adicionar o novo objeto *PlayingCard* a essa coleção, e não ao array. Após o loop *for* interno, adicione o objeto *SortedList* à *Hashtable* *cardPack*, e especifique o valor da variável *suit* como a chave para esse item. (O objeto *SortedList* contém todas as cartas no baralho do naipe especificado, e a *Hashtable* abrange uma coleção desses objetos *SortedList*.)

O código a seguir mostra o construtor finalizado, com as mudanças destacadas em negrito:

```
public Pack()
{
    this.cardPack = new Hashtable();

    for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
    {
```

```

        SortedList cardsInSuit = new SortedList();
        for (Value value = Value.Two; value <= Value.Ace; value++)
        {
            cardsInSuit.Add(value, new PlayingCard(suit, value));
        }
        this.cardPack.Add(suit, cardsInSuit);
    }
}

```

6. Encontre o método *DealCardFromPack*. Lembre-se de que 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 quaisquer alterações, mas as instruções no final do método, que recuperam a carta e a removem do array, devem ser atualizadas de modo a utilizar, em substituição, a coleção *Hashtable*.

Modifique o código depois da chave de fechamento do segundo loop *while*, como mostrado no código a seguir, em negrito:

```

public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
    while (this.IsSuitEmpty(suit))
    {
        suit = (Suit)randomCardSelector.Next(NumSuits);
    }

    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
    while (this.IsCardAlreadyDealt(suit, value))
    {
        value = (Value)randomCardSelector.Next(CardsPerSuit);
    }

    SortedList cardsInSuit = (SortedList) cardPack[suit];
    PlayingCard card = (PlayingCard)cardsInSuit[value];
    cardsInSuit.Remove(value);
    return card;
}

```

A *Hashtable* contém uma coleção de objetos *SortedList*, um para cada naipe de cartas. Esse novo código recupera a *SortedList* para a carta do naipe selecionado aleatoriamente na *Hashtable*, e depois recupera a carta com o valor selecionado nessa *SortedList*. A última instrução nova remove a carta da *SortedList*.

7. Localize o método *IsCardAlreadyDealt*. Esse método determina se a carta já foi distribuída, ao verificar 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 *SortedList* para o naipe na *cardPack Hashtable*. Atualize o método como mostrado em negrito:

```

private bool IsCardAlreadyDealt(Suit suit, Value value)
{
    SortedList cardsInSuit = (SortedList)this.cardPack[suit];
    return (!cardsInSuit.ContainsKey(value));
}

```

8. Exiba o arquivo Hand.cs na janela *Code and Text Editor*. Essa classe usa um array para armazenar as cartas da mão. Modifique a definição do array *cards*, de modo a utilizar uma coleção *ArrayList*, como mostrado em negrito:

```
class Hand
{
    public const int HandSize = 13;
    private ArrayList cards = new ArrayList();
    . . .
}
```

9. Localize o método *AddCardToHand*. Esse método verifica se a mão está cheia e, se não estiver, ele 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 *ArrayList*. Essa mudança também evita a necessidade de rastrear explicitamente a quantidade de cartas que a coleção armazena, porque você pode utilizar, em vez disso, a propriedade *Count*. Modifique a instrução *if*, que verifica se a mão está cheia, para fazer referência a essa propriedade, e exclua a variável *playingCardCount* da classe.

O método finalizado deve ficar semelhante a este:

```
public void AddCardToHand(PlayingCard cardDealt)
{
    if (this.cards.Count >= HandSize)
    {
        throw new ArgumentException("Too many cards");
    }
    this.cards.Add(cardDealt);
}
```

10. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.
11. Quando a janela Card Game for exibida, clique em *Deal*. Verifique se as cartas são distribuídas e se as mãos preenchidas aparecem como anteriormente. Clique em *Deal* novamente para gerar outro conjuntos de mãos aleatoriamente.

12. Feche o formulário e retorne ao Visual Studio 2010.

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 você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 11.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

Referência rápida do Capítulo 10

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: <code>bool[] flags;</code>
Criar uma instância de um array	Escreva a palavra-chave <code>new</code> , seguida pelo nome do tipo de elemento, seguido pelo tamanho do array entre colchetes. Por exemplo: <code>bool[] flags = new bool[10];</code>
Inicializar os elementos de um array (ou de uma coleção que suporta o método <code>Add</code>) com valores específicos	Para um array, escreva os valores específicos em uma lista separada por vírgulas incluindo-os entre chaves. Por exemplo: <code>bool[] flags = { true, false, true, false };</code> Para uma coleção, utilize o operador <code>new</code> e o tipo de coleção com os valores específicos em uma lista separada por vírgulas incluída entre chaves. Por exemplo: <code>ArrayList numbers = new ArrayList(){10, 9, 8, 7, 6, 5};</code>
Descobrir o número de elementos de um array	Utilize a propriedade <code>Length</code> . Por exemplo: <code>int [] flags = ...;</code> *** <code>int noOfElements = flags.Length;</code>
Descobrir o número de elementos em uma coleção	Utilize a propriedade <code>Count</code> . Por exemplo: <code>ArrayList flags = new ArrayList();</code> *** <code>int noOfElements = flags.Count;</code>
Acessar um elemento individual 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 array começa em 0, não em 1. Por exemplo: <code>bool initialElement = flags[0];</code>
Iterar pelos elementos de um array ou de uma coleção	Utilize uma instrução <code>for</code> ou uma instrução <code>foreach</code> . Por exemplo: <code>bool[] flags = { true, false, true, false };</code> <code>for (int i = 0; i < flags.Length; i++)</code> <code>{</code> <code>Console.WriteLine(flags[i]);</code> <code>}</code> <code>foreach (bool flag in flags)</code> <code>{</code> <code>Console.WriteLine(flag);</code> <code>}</code>

Capítulo 11

Entendendo 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.

Os arrays de parâmetros são úteis se você quer escrever métodos que podem receber qualquer número de argumentos, possivelmente de tipos diferentes, como parâmetros. Se você está familiarizado com os conceitos de orientação a objetos, talvez você esteja rangendo os dentes de frustração. Afinal de contas, o enfoque orientado a objetos para solucionar esse problema é definir métodos sobrecarregados.

Sobrecarregar é o termo técnico utilizado para declarar dois ou mais métodos com o mesmo nome no mesmo escopo. Ser capaz de sobrecarregar um método é muito útil nos casos em que você quer realizar a mesma ação em argumentos de tipos diferentes. O exemplo clássico de sobrecarga no Microsoft Visual C# é *Console.WriteLine*. O método *WriteLine* é sobrecarregado várias vezes para permitir passar qualquer argumento de tipo primitivo:

```
class Console
{
    public static void WriteLine(int parameter)
    ...
    public static void WriteLine(double parameter)
    ...
    public static void WriteLine(decimal parameter)
    ...
}
```

Embora muito útil, a sobrecarga não cobre todos os casos. Particularmente, a sobrecarga não trata facilmente uma situação em que o tipo dos parâmetros não varia, mas o número de parâmetros, sim. Contudo, e se você quisesse escrever muitos valores no console, por exemplo? Você teria de fornecer versões do *Console.WriteLine* que pudessem receber dois parâmetros, outras versões que pudessem receber três parâmetros e assim por diante? Isso rapidamente 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 de *params* resolvem esse problema, é útil compreender primeiramente as utilizações e deficiências dos arrays comuns.

Utilizando 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)
    {
        if (paramList == null || paramList.Length == 0)
        {
            throw new ArgumentException("Util.Min: not enough arguments");
        }
        int currentMin = paramList[0];
        foreach (int i in paramList)
        {
            if (i < currentMin)
            {
                currentMin = i;
            }
        }
        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 dois valores *int*, 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 valores *int*, 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: você tem de escrever um código adicional para preencher o array que você passa. Entretanto, você pode fazer o compilador escrever um pouco desse código para você utilizando a palavra-chave *params* para declarar um array *params*.

Declarando um array *params*

Você utiliza a palavra-chave *params* como um modificador dos parâmetros de array. Por exemplo, eis 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 o anterior
    }
}
```

O efeito da palavra-chave *params* no método *Min* é ela permitir que você o chame utilizando qualquer número de argumentos inteiros. Por exemplo, para descobrir o mínimo de dois valores inteiros, escreva:

```
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 ao método *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** Os programadores C e C++ poderão reconhecer *params* como um equivalente seguro das macros *varargs* do arquivo de cabeçalho *stdarg.h*.

Há vários pontos sobre os arrays *params* que vale a pena mencionar:

- Você não pode utilizar a palavra-chave *params* em arrays multidimensionais. O código no exemplo a seguir não irá compilar:

```
// erro de tempo de compilação
public static int Min(params int[,] table)
{
    ...
}
```

- Você não pode sobrepor 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:

```
// erro 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 sobreposta de um método para os casos comuns. Por exemplo:

```
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 em que 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.

- O compilador detecta e rejeita todas as sobreposições potencialmente ambíguas. Por exemplo, os dois métodos *Min* a seguir são ambíguos; não está claro qual deles deve ser chamado se você passar dois argumentos *int*:

```
// erro de tempo de compilação
public static int Min(params int[] paramList)
...
public static int Min(int, params int[] paramList)
...
```

Utilizando *params object[]*

Um array de parâmetros do tipo *int* é muito útil porque ele permite passar qualquer número de argumentos *int* para uma chamada de método. Contudo, e se não só o número de argumentos variar, 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, “Entendendo 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)
    ...
}
```

Atribuí a esse método o nome *Black.Hole* (“buraco negro”) porque nenhum argumento pode escapar dele:

- Você pode passar o método sem nenhum argumento, caso em que o compilador passará um array de objetos cujo comprimento é 0:

```
Black.Hole();
// convertido em Black.Hole(new object[0]);
```



Dica É perfeitamente seguro tentar iterar por um array de comprimento zero utilizando uma instrução *foreach*.

- 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, você pode criar manualmente o array que normalmente é 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* qualquer outro argumento de tipo diferente, 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 dessas sobrecargas 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. Eis um exemplo de uma chamada a esse método:

```
Console.WriteLine("Forename:{0}, Middle Initial:{1}, Last name: {2}, Age: {3}", fname,
    mi, lname, age);
```

O compilador resolve essa chamada para o seguinte

```
Console.WriteLine( Forename: {0}, Middle Initial: {1}, Last name: {2}, Age: {3}",
    new object[4] {fname, mi, lname, age});
```

Utilizando um array *params*

No exercício a seguir, você irá implementar e testar um método *static* chamado *Util.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 *Util.Sum* para aceitar um parâmetro *params int[]*. Você implementará duas verificações no parâmetro *params* a fim de assegurar que o método *Util.Sum* é completamente robusto. Você então chamará o método *Util.Sum* com diversos argumentos diferentes para testá-lo.

Escreva um método de array *params*

1. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto *ParamsArray*, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 11\SwitchStatement na sua pasta Documentos.
3. Exiba o arquivo *Util.cs* na janela *Code and Text Editor*.

O arquivo *Util.cs* contém uma classe vazia chamada *Util* no namespace *ParamsArray*.

4. Adicione um método estático público chamado *Sum* à classe *Util*.

O método *Sum* retorna um *int* e aceita um array *params* de valores *int*. O método *Sum* deve ser semelhante a este:

```
public static int Sum(params int[] paramList)
{
}
```

A primeira etapa 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 acionar uma *ArgumentException*. (Você poderá argumentar que a soma dos inteiros em um

array de comprimento zero é 0, mas, neste exemplo, trataremos essa situação como uma exceção.)

- Adicione código a *Sum* que lança uma *ArgumentException* se *paramList* for *null*.

O método *Sum* agora deve se parecer com isso:

```
public static int Sum(params int[] paramList)
{
    if (paramList == null)
    {
        throw new ArgumentException("Util.Sum: null parameter list");
    }
}
```

- Adicione código ao método *Sum* para lançar uma *ArgumentException* se o comprimento do *array* 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 juntos ao array.

- Você pode utilizar uma instrução *foreach* para adicionar todos os elementos juntos. Você precisará de uma variável local para armazenar o total. Declare uma variável do tipo inteiro chamada *sumTotal* e a inicialize como 0, após o código do passo anterior. Adicione uma instrução *foreach* ao método *Sum* para iterar pelo arrays *paramList*. O corpo desse loop *foreach* deve adicionar cada elemento ao array a *sumTotal*. No final do método, retorne o valor de *sumTotal* utilizando uma instrução *return*, como mostrado em negrito a seguir:

```
class Util
{
    public static int Sum(params int[] paramList)
    {
        ...
        int sumTotal = 0;
        foreach (int i in paramList)
        {
            sumTotal += i;
        }
        return sumTotal;
    }
}
```

- No menu *Build*, clique em *Build Solution*. Confirme se sua solução é compilada sem nenhum erro.

Teste o método *Util.Sum*

1. Exiba o arquivo Program.cs na janela *Code and Text Editor*.
 2. Na janela *Code and Text Editor*, localize o método *DoWork* na classe *Program*.
 3. Adicione a seguinte instrução ao método *DoWork*:
- ```
Console.WriteLine(Util.Sum(null));
```

4. 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.

5. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2010.
6. 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 nenhum argumento. O compilador converterá a lista de argumentos vazia em um array vazio.

7. 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.

8. Pressione a tecla Enter para finalizar o programa e retornar ao Visual Studio 2010.
9. 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));
```

10. No menu *Debug*, clique em *Start Without Debugging*.

Verifique se o programa compila e executa e escreve o valor 55 no console.

11. Pressione Enter para fechar o aplicativo e voltar ao Visual Studio 2010.

## Comparando arrays de parâmetros e parâmetros opcionais

No Capítulo 3, “Escrevendo métodos e aplicando escopo”, vimos 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 os argumentos fornecidos pelo chamador, nem os padrões gerados pelo compilador.
- Um método que utiliza um array de parâmetros realmente possui uma lista totalmente arbitrária de parâmetros, e nenhum deles tem valores padrão. Além disso, o método pode determinar exatamente 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.

Ainda existe uma última questão que compensa 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 de parâmetros e parâmetros opcionais

1. Retorne à solução ParamsArray no Visual Studio 2010, e exiba o arquivo Util.cs na janela *Code and Text Editor*.
2. Adicione a seguinte instrução *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, com um valor padrão 0. No corpo do método, dê saída à mensagem “Using optional parameters”, e depois calcule e retorne a soma dos quatro parâmetros. O método concluído deve ficar parecido com o seguinte:

```
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*, comente 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*.

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

- Pressione Enter e retorne ao Visual Studio.

- No método *DoWork*, mude a instrução que chama o método *Sum*, como mostrado a seguir:

```
Console.WriteLine(Util, Sum(2, 4, 6));
```

- No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo. Quando o aplicativo for executado, ele exibirá as seguintes 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.

- Pressione Enter e retorne ao Visual Studio.

- No método *DoWork*, mude a instrução que chama o método *Sum* novamente.

```
Console.WriteLine(Util, Sum(2, 4, 6, 8, 10));
```

- No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo. Quando o aplicativo for executado, ele exibirá as seguintes mensagens:

```
Using parameter list
```

```
30
```

Dessa vez, há mais parâmetros 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.

- Pressione Enter e retorne ao Visual Studio.

Neste capítulo, você aprendeu a utilizar um array *params* para definir um método que pode receber quantidades variáveis de argumentos. Você também viu como utilizar um array *params* dos tipos *object* para criar um método que aceita qualquer número de argumentos de qualquer tipo. Você também viu como o compilador resolve chamadas de método quando ele pode escolher entre chamar um método que aceita um array de parâmetros e chamar um método que aceita parâmetros opcionais.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 12.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 11

| Para                                                                         | Faça isto                                                                                                                                                                                                                                     |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Escrever um método que aceita qualquer número de argumentos de um dado tipo  | Escreva um método cujo parâmetro é 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:<br><pre>someType Method(params bool[] flags) {     ... }</pre> |
| Escrever um método que aceita qualquer número de argumentos de qualquer tipo | Escreva um método cujo parâmetro é um array <i>params</i> de elementos do tipo <i>object</i> . Por exemplo:<br><pre>someType Method(params object[] paramList) {     ... }</pre>                                                              |

# Capítulo 12

# Trabalhando com 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.

Herança é um conceito-chave no mundo da orientação a objetos. Você pode usá-la como uma ferramenta para evitar a repetição ao definir classes diferentes com várias características em comum e que estão muito claramente relacionadas entre si. Talvez sejam classes diferentes do mesmo tipo, cada uma delas com sua característica própria e distinta – por exemplo, *gerentes*, *operários* e *todos os empregados* de uma fábrica. Se você estiver escrevendo um aplicativo para simular a fábrica, como especificaria que os gerentes e operários têm várias características comuns, mas também são diferentes? Por exemplo, todos têm um número de referência de funcionário, mas os gerentes têm responsabilidades diferentes e executam tarefas diferentes dos operários.

É aí que a herança se prova útil.

## O que é herança?

Se perguntar a vários programadores experientes o que eles entendem por *herança*, em geral, você obterá respostas diferentes e contraditórias, o que deriva do fato de que a própria palavra *herança* tem vários significados sutilmente diferentes. Se alguém deixar 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 *herança* 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, você provavelmente aprendeu sobre mamíferos e percebeu 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, 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. 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 você 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 todos tipos de mamíferos, portanto, crie uma classe chamada *Mammal* que fornece 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 forneceriam automaticamente a funcionalidade da classe *Mammal* (*Breathe*, *SuckleYoung* e assim por diante), mas você também poderia adicionar a funcionalidade peculiar 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 precisar modificar a maneira como um método comum como *Breathe* funciona, você precisará alterá-lo apenas em um único lugar, a classe *Mammal*.

## Utilizando a herança

Você declara que uma classe herda de outra classe, ao utilizar a seguintes sintaxe:

```
class DerivedClass : BaseClass {
 ...
}
```

A classe derivada herda da classe base, e os métodos na classe base também 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 classes seladas no Capítulo 13, "Criando interfaces e definindo classes abstratas").

```
class DerivedSubClass : DerivedClass {
 ...
}
```



**Importante** Todas as estruturas herdam de uma classe abstrata, chamada `System.ValueType`. (Você conhecerá as classes abstratas no Capítulo 13.) Este é apenas um detalhe de implementação do modo como o .NET Framework define o comportamento comum de tipos-valor baseados em pilha. Não é possível definir uma hierarquia de herança própria com estruturas – só é possível definir uma estrutura derivada de uma classe ou de outra estrutura.

No exemplo descrito anteriormente, você poderia declarar a classe *Mammal* como mostrado aqui. 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. Por exemplo:

```
class Horse : Mammal
{

 public void Trot()
 {

 }
}

class Whale : Mammal
{

 public void Swim()
 {

 }
}
```

**Nota** Os programadores C++ devem notar que você não pode explicitar se a herança é pública, privada ou protegida. A herança no C# é sempre implicitamente pública. Os programadores Java devem notar o uso de dois-pontos e que não há qualquer palavra-chave *extends*.

Lembre-se de que a classe *System.Object* é a classe raiz de todas as classes. Todas as classes derivam implicitamente da classe *System.Object*. Consequentemente, o compilador C# reescreve discretamente a classe *Mammal* como o código a seguir (podendo ser escrita explicitamente se quiser):

```
class Mammal : System.Object
{

}
```

Qualquer método na classe *System.Object* é automaticamente rebaixado na cadeia de herança para classes que derivam de *Mammal*, como *Horse* e *Whale*. Em termos práticos, significa que todas as classes que você define automaticamente herdaram as características da classe *System.Object*. Inclui métodos, como *ToString* (já discutido no Capítulo 2, “Trabalhando com variáveis, operadores e expressões”), que é utilizado para converter um objeto em uma string, geralmente para propósitos de exibição.

## Chamando 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 normalmente é realizado em um construtor. Lembre-se de que todas as classes têm pelo menos um construtor (se você não fornecer um, o compilador vai gerar um construtor padrão). Uma boa prática é o construtor em uma classe derivada chamar o construtor para sua classe base como parte da inicialização. 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), em cujo caso esquecer de chamar o construtor correto da classe base resulta em um erro de compilação.

## Atribuindo classes

Nos exemplos anteriores, você viu como declarar uma variável utilizando um tipo classe e então como utilizar a palavra-chave *new* para criar um objeto. Viu também como as regras de verificação de tipo do C# impedem que você atribua um objeto de um tipo a uma variável declarada como 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("Neddy"); // O construtor mostrado antes espera um nome!
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 legais:

```
Horse myHorse = new Horse("Neddy");
Mammal myMammal = myHorse; // válido, Mammal é a classe base de Horse
```

Se 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 adicionar 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("Neddy");
Mammal myMammal = myHorse;
myMammal.Breathe(); // OK - Breathe é parte da classe Mammal
myMammal.Trot(); // erro - Trot não é parte da classe Mammal
```

 **Nota** Isso explica por que você pode atribuir quase tudo a uma variável *object*. Lembre-se de que *object* é um alias de *System.Object* e 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* irrestritivamente a uma variável *Horse*:

```
Mammal myMammal = new Mammal("Mammalia");
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 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 alguns outros tipos de *Mammal*, o operador *as* retornará, em vez disso, *null*.

```
Horse myHorse = new Horse("Neddy");
Mammal myMammal = myHorse; // myMammal referencia um Horse
...
Horse myHorseAgain = myMammal as Horse; // OK - myMammal era um Horse
...
Whale myWhale = new Whale("Moby Dick");
myMammal = myWhale;
...
myHorseAgain = myMammal as Horse; // retorna null - myMammal era uma Whale
```

## Declarando métodos new

Um dos problemas mais difíceis no campo da programação é a tarefa de 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, você tentará reutilizar um nome que já está em uso por uma das classes de nível mais elevado 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.

O método na classe derivada mascara (ou oculta) o método na classe base que tem a mesma assinatura. Por exemplo, se você compilar o código a seguir, o compilador irá gerar 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. Na verdade, a palavra-chave *new* diz: “Eu sei o que estou fazendo, portanto, pare de me mostrar esses avisos”.

## Declarando 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 em sua representação 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 um conceito, e todas as classes devem fornecer um método que possa ser utilizado para converter objetos em strings para propósitos de exibição ou depuração. Mas é só a implementação que é problemática. De fato, você não precisa chamar o método *ToString* definido por *System.Object* – ele é simplesmente um espaço reservado. Em vez disso, você deve 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 seu próprio método *ToString*. Dessa maneira, você pode ter certeza de que é possível chamar *ToString* em qualquer objeto e que o método retornará uma string contendo algo.

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 normalmente 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** Os desenvolvedores em Java devem notar que os métodos C# não são virtuais por padrão.

## Declarando 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. Por exemplo:

```
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 (como discutido na barra lateral, “Métodos virtuais e polimorfismo”) utilizando as palavras-chave *virtual* e *override*:

- Você não pode declarar um método privado utilizando a palavra-chave *virtual* ou *override*. Se tentar, haverá um erro de tempo de compilação. Privado realmente é privado.
- As duas assinaturas de método devem ser idênticas – isto é, elas devem ter o mesmo nome e o mesmo número e tipo de parâmetros. Além disso, os dois métodos devem retornar o mesmo tipo.
- Os dois métodos devem ter o mesmo nível de acesso. Por exemplo, se um dos dois métodos for público, o outro também deverá ser público (métodos também podem ser protegidos, como veremos na próxima seção).
- 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. Este 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 de base. 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 descrito anteriormente.
- 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 permitem 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 anteriormente:

```
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`, e, 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 não 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 ele 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 invocando um método diferente, dependendo de seu contexto, é chamado de *polimorfismo*, que literalmente significa "muitas formas".

## Entendendo o acesso *protected*

As palavras-chave de acesso *public* e *private* criam 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, as 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. Frequentemente é útil para a 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. Nessa situação, você pode utilizar a palavra-chave *protected* para marcar os membros:

- 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. Em outras palavras, 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. 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, mas esses ainda permitem que o encapsulamento seja violado pelas classes que herdam da classe.



**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 adicionará alguns métodos às duas classes derivadas que são específicas a essas classes. Você então adicionará um método virtual chamado *Drive* à classe *Vehicle* e redefinirá a implementação padrão desse método nas duas classes derivadas.

### Crie uma hierarquia de classes

1. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto *Vehicles*, localizado na pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 12\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* aparece, permitindo adicionar um novo arquivo que definirá uma classe para o projeto.

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 para os 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* aparece mais uma vez.

7. Na caixa *Name*, digite **Airplane.cs** e então 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"); // Decolando
 }

 public void Land()
 {
 Console.WriteLine("Landing"); // Aterrissando
 }
}
```

10. No menu *Project*, clique em *Add Class*.

A caixa de diálogo *Add New Item—Vehicles* aparece mais uma vez.

11. Na caixa *Name*, digite **Car.cs** e então 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"); //Acelerando
 }

 public void Brake()
 {
 Console.WriteLine("Braking"); //Acionando o freio
 }
}
```

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 mostrado 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*, crie uma instância da classe *Airplane* e teste os métodos simulando uma viagem rápida de avião, como a seguir:

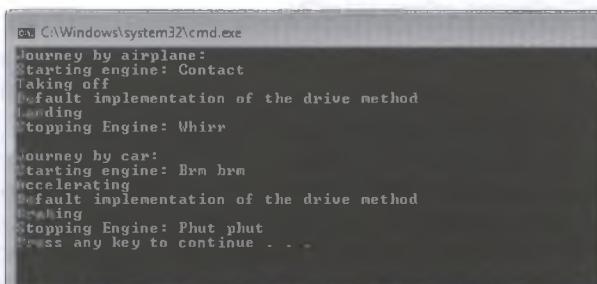
```
static void DoWork()
{
 Console.WriteLine("Journey by airplane:");
 Airplane myPlane = new Airplane();
 myPlane.StartEngine("Contact");
 myPlane.TakeOff();
 myPlane.Drive();
 myPlane.Land();
 myPlane.StopEngine("Whirr");
}
```

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 testa 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, verifique se o programa emite mensagens simulando as diferentes etapas de uma viagem de avião e de carro, como mostrado na imagem a seguir:



Observe que os dois meios de transporte invocam 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 2010.

21. Exiba a classe *Airplane* na janela *Code and Text Editor*. Redefina o método *Drive* na classe *Airplane*, desta maneira:

```
public override void Drive()
{
 Console.WriteLine("Flying");
}
```



**Nota** O IntelliSense exibe uma lista dos métodos virtuais disponíveis. Se selecionar o método *Drive* da lista IntelliSense, o Visual Studio irá inserir automaticamente no seu código uma instrução que chama o método *base.Drive*. Se isso acontecer, exclua a instrução, uma vez que ela não é necessária neste exercício.

22. Exiba a classe *Car* na janela *Code and Text Editor*. Redefina o método *Drive* na classe *Car* da seguinte maneira:

```
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*.

24. Pressione Enter para fechar o aplicativo e retornar ao Visual Studio 2010.

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* ao 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 se as mesmas mensagens aparecem como anteriormente seguidas por este texto:

```
Testing polymorphism
Motoring
Flying
```

O método *Drive* é virtual, portanto, o runtime (não o compilador) determina qual versão do método *Drive* chamar ao invocá-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 2010.

## Entendendo métodos de extensão

A herança é um recurso poderoso, pois permite que você estenda a funcionalidade de uma classe criando uma nova classe que derive dela. Mas, às vezes, o uso da herança não é o mecanismo mais apropriado para adicionar novos comportamentos, especialmente 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* – 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). Uma maneira de alcançar 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()
 {
 ...
 }
}
```

A teoria é que *NegInt32* herdará toda a funcionalidade associada ao tipo *System.Int32* além do método *Negate*. Há duas razões para você não querer seguir essa abordagem:

- Esse método só será aplicado ao tipo *NegInt32* e se você quiser utilizá-lo com as variáveis *int* existentes no seu código, você 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.

Aqui os métodos de extensão tornam-se muito úteis.

Um método de extensão permite 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 “.” para referenciar o método, desta maneira:

```
int x = 591;
Console.WriteLine("x.Negate {0}", x.Negate());
```

Observe que você não precisa 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 um dado tipo a partir de todas as classes estáticas que estão em escopo. Você também pode chamar o método *Util.Negate* passando um *int* como o parâmetro, utilizando a sintaxe comum que já vimos anteriormente, 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*. Esse método de extensão permite converter o valor que uma variável *int* contém a partir da base 10 em uma representação desse valor em uma base numérica diferente.

### Crie um método de extensão

1. No Visual Studio 2010, abra o projeto *ExtensionMethod*, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 12\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*. Essa classe está vazia, com exceção dos comentários *// to do*. Lembre-se de que você deve definir métodos de extensão dentro de uma classe estática.

3. Adicione um método estático público à classe *Util*, chamado *ConvertToBase*. O método deve receber dois parâmetros: um parâmetro *int* nomeado *i*, prefixado com a palavra-chave *this* para indicar que o método é um método de extensão para o tipo *int*, e um outro parâmetro *int* normal nomeado *baseToConvertTo*. O método converterá o valor em *i* na base indicada por *baseToConvertTo*, devendo 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)
 {
 }
}
```

4. 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 uma maneira confiável fora desse intervalo de valores. Lance uma *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());
}
```

5. Adicione as seguintes instruções mostradas em negrito ao método *ConvertToBase*, após a instrução que lança a *ArgumentException*. Esse código implementa um algoritmo bem conhecido que converte um número de base 10 em uma base numérica diferente. (Você viu uma versão desse algoritmo para converter um número decimal em octal no Capítulo 5, “Utilizando atribuição composta e instruções de iteração”.)

```
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;
}

```

6. Exiba o arquivo Program.cs na janela *Code and Text Editor*.

7. Adicione a seguinte instrução *using* após a instrução *using System*; na parte superior do arquivo:

```
using Extensions;
```

Essa instrução dá escopo ao namespace contendo a classe *Util*. O método *ConvertToBase* de extensão não será visível no arquivo Program.cs se você não realizar essa tarefa.

8. Adicione as seguintes instruções ao método *DoWork* da classe *Program*:

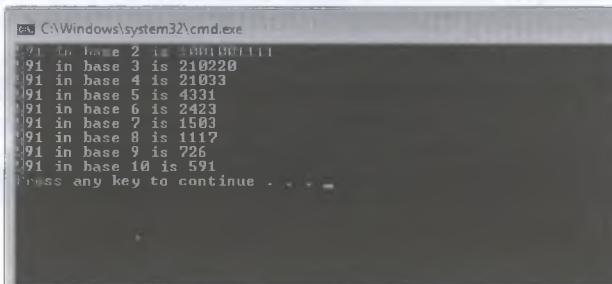
```

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 selecionar qualquer valor inteiro que quiser). 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 *x* na instrução *Console.WriteLine*.

9. No menu *Debug*, clique em *Start Without Debugging*. Confirme se o programa exibe no console as mensagens que mostram o valor 591 nas diferentes bases numéricas, assim:



10. Pressione Enter para fechar o programa.

Neste capítulo, você viu como utilizar a herança para definir uma hierarquia de classes e agora deve entender como redefinir métodos herdados e implementar métodos virtuais. Também viu como adicionar um método de extensão a um tipo existente.

■ Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 13.

■ Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save* clique em *Yes* e salve o projeto

## Referência rápida do Capítulo 12

| 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:<br><pre>class Derived : Base {     ... }</pre>                                                                                                                                   |
| Chamar um construtor de classe base como parte do construtor para uma classe que o herda | Forneça uma lista de parâmetros do construtor antes do corpo do construtor da classe derivada. Por exemplo:<br><pre>class Derived : Base {     ...     public Derived(int x) : Base(x)     {         ...     }     ... }</pre>                                                  |
| Declarar um método virtual                                                               | Use a palavra-chave <i>virtual</i> ao declarar o método. Por exemplo:<br><pre>class Mammal {     public virtual void Breathe()     {         ...     }     ... }</pre>                                                                                                          |
| Implementar um método em uma classe derivada que redefine um método virtual herdado      | Utilize a palavra-chave <i>override</i> ao declarar o método na classe derivada. Por exemplo:<br><pre>class Whale : Mammal {     public override void Breathe()     {         ...     }     ... }</pre>                                                                         |
| 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 <i>this</i> . Por exemplo:<br><pre>static class Util {     public static int Negate(this int i)     {         return -i;     } }</pre> |

## Capítulo 13

# Criando interfaces e definindo 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.
- Capturar detalhes de implementação comuns em uma classe abstrata.
- Fazer referência a uma classe por meio de uma interface.
- Implementar classes *sealed* que não podem ser utilizadas para derivar novas classes.

A herança de uma classe é um mecanismo poderoso, mas o verdadeiro poder da herança vem da herança de uma interface. Uma interface não contém qualquer código ou dado; ela simplesmente especifica os métodos e as propriedades que uma classe que herda dela deve fornecer. Utilizando uma interface você pode separar completamente os nomes e as assinaturas dos métodos de uma classe de um lado, e, de outro, a implementação do método.

Classes abstratas são bem semelhantes a interfaces, exceto por poderem conter código e dados. Mas é possível especificar que certos métodos de uma classe abstrata sejam virtuais de tal modo que uma classe que herde da classe abstrata disponha de sua própria implementação desses métodos. Você utiliza frequentemente classes abstratas com interfaces, e elas fornecem conjuntamente uma técnica fundamental que permite construir estruturas de programação extensíveis, como você descobrirá neste capítulo.

## Entendendo interfaces

Suponha que você queira definir uma nova classe de coleção que permita a um aplicativo armazenar objetos em uma sequência que depende dos tipos dos objetos que a coleção contém. Por exemplo, se a coleção armazena objetos alfanuméricos como strings, ela deve ordenar os objetos de acordo com a sequência de intercalação (collation) do computador; se a coleção armazena objetos numéricos como inteiros, ela deve ordená-los numericamente.

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. 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, “Trabalhando com 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. Normalmente não há qualquer forma de relacionamento de herança entre a classe de coleção e os objetos que armazena, portanto, um método virtual não seria muito útil. Se pensar um pouco, o problema é que a maneira como os objetos na coleção devem ser ordenados depende do tipo dos próprios objetos, não da coleção. A solução, então, é exigir que todos os objetos forneçam um método, como o método *CompareTo* aqui mostrado, que a coleção pode chamar, permitindo que a coleção compare esses objetos entre si.

```
int CompareTo(object obj)
{
 // retorna 0 se essa instância for igual a obj
 // retorna < 0 se essa instância for menor que obj
 // retorna > 0 se essa instância for maior que obj
 ...
}
```

A classe de coleção pode utilizar esse método para ordenar os objetos nela contidos.

Você pode definir uma interface para objetos colecionáveis que inclua o método *CompareTo* e especificar que a classe de coleção só pode coletar 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.

As interfaces permitem que você realmente separe o “o que” do “como”, informando apenas quais são 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 implementar, mas não como essa funcionalidade é implementada.

## Definindo uma interface

Para definir uma interface, você utiliza a palavra-chave *interface* em vez da palavra-chave *class* ou *struct*. Dentro da interface, você declara os métodos exatamente como em uma classe ou em uma estrutura, exceto pelo fato de nunca especificar um modificador de acesso (*public*, *private* ou *protected*) e substituir o corpo do método por um ponto e vírgula. Eis um exemplo:

```
interface IComparable
{
 int CompareTo(object obj);
}
```

**Dica** A documentação do Microsoft .NET Framework recomenda que você comece o nome das suas 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* como mostrado anteriormente.

## Implementando uma interface

Para implementar uma interface, você declara uma classe ou estrutura que herda da interface e implementa *todos* os métodos especificados por ela. Por exemplo, suponha que você esteja definindo a hierarquia *Mammal* descrita no Capítulo 12 e 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 pode então implementar essa interface na classe *Horse*. Você herda da interface e fornece uma implementação de cada método definido pela interface.

```
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.
- O nome do método se inicia com o nome da interface. Isso é conhecido como implementação explícita de interface e é um bom hábito a cultivar.
- Todos os métodos que implementam uma interface devem ser publicamente acessíveis. Mas se você estiver utilizando a 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.

Uma classe pode estender outra classe e implementar uma interface ao mesmo tempo. Nesse caso, o C# não indica 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 nomeada primeiramente, seguida por uma vírgula, seguida pela interface. O seguinte exemplo define *Horse* como uma classe que é um *Mammal* mas que também implementa a interface *ILandBound*:

```
interface ILandBound
{
 ...
}

class Mammal
{
 ...
}

class Horse : Mammal , ILandBound
{
 ...
}
```

## Referenciando 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, você 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, não sendo possível 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 porque permite 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)
{
 ...
}
```

Observe que, ao referenciar um objeto por meio de uma interface, você só pode invocar os métodos que são visíveis pela interface.

## Trabalhando com múltiplas interfaces

Uma classe pode ter no máximo uma classe base, mas pode implementar um número ilimitado de interfaces, devendo ainda implementar todos os métodos que herda de todas as suas interfaces.

Se uma interface, estrutura ou classe herda de mais de uma interface, você escreve as interfaces em 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 uma 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
{
 ...
}
```

## Implementando explicitamente uma interface

Os exemplos apresentados até agora mostraram classes que implementam implicitamente uma interface. Se você examinar novamente a interface *ILandBound* e a classe *Horse* (mostrada a seguir), embora a classe *Horse* implemente a interface *ILandBound*, perceberá que não há nada na implementação do método *NumberOfLegs* na classe *Horse* que indique 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 a impedir que diversas interfaces especifiquem um método com o mesmo nome, embora possa ter semânticas distintas. 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 rastrear 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 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 tange à classe *Horse*, qual método o seguinte código chamaria – 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 fazer referência ao objeto *Horse* pela interface adequada, como a seguir:

```
Horset horse = new Horse();

I Journey journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs();
ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs();
```

É recomendável que você implemente 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 construtores em uma interface. Um construtor também é considerado um detalhe de implementação de uma classe ou estrutura.
- Você não tem permissão para definir um destrutor em uma interface. Um destrutor contém as instruções utilizadas para destruir uma instância de objeto. (Os destrutores estão descritos no Capítulo 14, “Utilizando a coleta de lixo e o 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, ela estaria herdando alguma implementação.

## Definindo e utilizando interfaces

Nos exercícios a seguir, você definirá e implementará interfaces que fazem parte de um simples pacote de desenho gráfico. Você definirá duas interfaces chamadas *IDraw* e *IColor*, e também definirá classes que as implementam. Cada classe determinará uma forma que pode ser desenhada sobre um canvas, em um formulário do Windows Presentation Foundation (WPF). (Um canvas é um controle WPF que permite desenhar linhas, texto e formas.)

A interface *IDraw* define os seguintes métodos:

- ***SetLocation*** Esse método permite 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 permite 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 2010, se ele ainda não estiver em execução.
2. Abra o projeto *Drawing*, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 13\Drawing de sua pasta Documentos.

O projeto *Drawing* é um aplicativo WPF, e contém um formulário WPF chamado *DrawingPad*. Esse formulário dispõe de um controle canvas, chamado *drawingCanvas*. Você utilizará esse formulário e o canvas para testar seu código.

3. No menu *Project*, clique em *Add New Item*.

É exibida a caixa de diálogo *Add New Item – Drawing*.

4. No painel esquerdo da caixa de diálogo *Add New Item – Drawing*, clique em *Visual C#*. Se você estiver utilizando o Visual Studio 2010 Professional ou o Visual Studio 2010 Standard, clique em *Code*. (O Visual C# 2010 Express tem menos templates e não os divide em grupos, como acontece no Visual Studio.) No painel central, clique no template *Interface*. Na caixa de texto *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 é parecido com o seguinte:

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

namespace Drawing
{
 interface IDraw
 {
 }
}
```

5. No arquivo `IDraw.cs`, adicione a seguinte instrução `using` à lista localizada no início do arquivo:

```
using System.Windows.Controls;
```

Você fará uma referência à classe `Canvas` nessa interface. A classe `Canvas` está localizada no namespace `System.Windows.Controls`.

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. No painel central da caixa de diálogo *Add New Item – Drawing*, clique no template *Interface*. Na caixa de texto *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`, adicione a seguinte instrução `using` à lista localizada no início do arquivo:

```
using System.Windows.Media;
```

Você fará uma referência à classe `Color` nessa interface, localizada no namespace `System.Windows.Media`.

10. Adicione o seguinte método mostrado em negrito à definição da interface `IColor`:

```
interface IColor
{
 void SetColor(Color color);
}
```

Você acabou de definir 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*, verifique se o template *Class* está selecionado no painel central, digite **Square.cs** na caixa de texto *Name* e clique em *Add*.

O Visual Studio gera o arquivo `Square.cs` e o exibe na janela *Code and Text Editor*.

3. Adicione as seguintes instruções *using* à lista localizada no início do arquivo Square.cs:

```
using System.Windows;
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
{
}
```

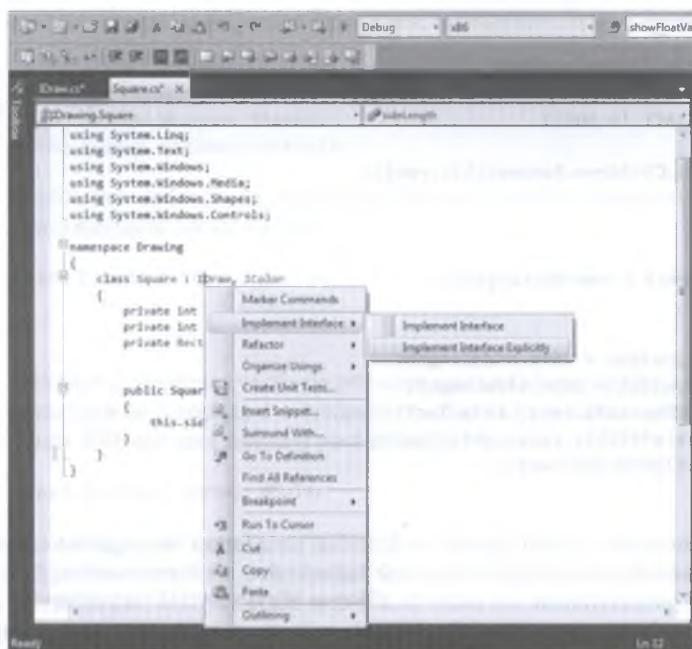
5. Adicione as seguintes variáveis privadas, apresentadas em negrito, à classe *Square*. Essas variáveis armazenarão a posição e o tamanho do objeto *Square* sobre o canvas. A classe *Rectangle* é uma classe WPF localizada no namespace *System.Windows.Shapes*. Você utilizará essa classe para desenhar o quadrado:

```
class Square : IDraw, IColor
{
 private int sideLength;
 private int locX = 0, locY = 0;
 private Rectangle rect = null;
}
```

6. Adicione o construtor, mostrado em negrito, à classe *Square*. Esse construtor inicializa o campo *sideLength* e especifica o comprimento de cada lado do quadrado.

```
class Square : IDraw, IColor
{
 ...
 public Square(int sideLength)
 {
 this.sideLength = sideLength;
 }
}
```

7. Na definição da classe *Square*, clique com o botão direito do mouse na interface *IDraw*. É exibido um menu de atalho. Nele, 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*. Espera-se que você substitua o corpo desses métodos pelo seu código.

8. No método *SetLocation*, substitua o código existente pelas instruções mostradas em negrito. Esse código armazena os valores passados pelos parâmetros nos campos *locX* e *locY*, no objeto *Square*.

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
 this.locX = xCoord;
 this.locY = yCoord;
}
```

9. Substitua o código existente no método *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(rect);
}
```

Esse método processa o objeto *Square*, ao desenhar uma forma *Rectangle* no canvas. (Um quadrado é tão somente um retângulo de quatro lados com o mesmo tamanho.) 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 (rect != null)
 {
 SolidColorBrush brush = new SolidColorBrush(color);
 rect.Fill = brush;
 }
}
```

Esse método verifica se o objeto *Square* foi realmente exibido. (O campo *rect* será null (nulo) se ele 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 objeto *SolidBrushClass* estão além do escopo desta discussão.)

11. No menu *Project*, clique em *Add Class*. Na caixa de diálogo *Add New Item – Drawing*, digite **Circle.cs** na caixa de texto *Name* e clique em *Add*.

O Visual Studio gera o arquivo Circle.cs e o exibe na janela *Code and Text Editor*.

12. Adicione as seguintes instruções *using* à lista localizada no início do arquivo Circle.cs:

```
using System.Windows;
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*. Essas variáveis armazenarão a posição e o tamanho do objeto *Circle* sobre o canvas. A classe *Ellipse* é outra classe WPF que você utilizará para desenhar o círculo.

```
class Circle : IDraw, IColor
{
 private int radius;
 private int locX = 0, locY = 0;
 private Ellipse circle = null;
}
```

15. Adicione o construtor, mostrado em negrito, à classe *Circle*. Esse construtor inicializa o campo *radius* (raio).

```
class Circle : IDraw, IColor
{
 ...
 public Circle(int radius)
 {
 this.radius = radius;
 }
}
```

16. Adicione o método *SetLocation*, mostrado a seguir, à classe *Circle*. Esse método implementa parte da interface *IDraw*, e o código é exatamente o mesmo da classe *Square*.

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
 this.locX = xCoord;
 this.locY = yCoord;
}
```

17. Adicione o método *Draw*, mostrado a seguir, à classe *Circle*. Esse método também faz parte da interface *IDraw*.

```
void IDraw.Draw(Canvas canvas)
{
 if (this.circle != null)
 {
 canvas.Children.Remove(this.circle);
 }
 else
 {
 this.circle = new Ellipse();
 }
 this.circle.Height = this.radius;
 this.circle.Width = this.radius;
 canvas.SetTop(this.circle, this.locY);
 canvas.SetLeft(this.circle, this.locX);
 canvas.Children.Add(circle);
}
```

Esse método é semelhante ao método *Draw* na classe *Square*, exceto pelo fato de que ele 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 método *SetColor* à classe *Circle*. Esse método faz parte da interface *IColor*. Como anteriormente, esse método é parecido com o da classe *Square*.

```
void IColor.SetColor(Color color)
{
 if (circle != null)
 {
 SolidColorBrush brush = new SolidColorBrush(color);
 circle.Fill = brush;
 }
}
```

Você concluiu as classes *Square* e *Circle* e já pode utilizar o formulário WPF para testá-las.

### Teste as classes *Square* e *Circle*:

1. Exiba o arquivo DrawingPad.xaml na janela *Design View*.
2. Clique na área sombreada existente no meio do formulário WPF.  
A área sombreada do formulário é o objeto *Canvas*, e esta ação define o foco neste objeto.
3. Na janela *Properties*, clique no botão *Events*. (Esse botão possui um ícone parecido com um relâmpago.)
4. Na lista de eventos, localize o evento *MouseLeftButtonDown* e clique duas vezes nesse evento.

O Visual Studio gera um método chamado *drawingCanvas\_MouseLeftButtonDown* para a classe *DrawingPadWindow*, que implementa o formulário WPF e o exibe na janela *Code and Text Editor*. Esse é um manipulador de eventos executado quando o usuário clica no botão esquerdo do mouse sobre o canvas. (Você conhecerá mais detalhes sobre os manipuladores de eventos no Capítulo 17, “Interrompendo o fluxo do programa e tratando eventos”.)

5. Adicione o código, mostrado em negrito, ao método *drawingCanvas\_MouseLeftButtonDown*:

```
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 *MouseButtonEventArgs*, *e*, para esse método fornece informações úteis sobre a posição do mouse. Mais especificamente, o método *GetPosition* retorna uma estrutura *Point* que contém as coordenadas X e Y do mouse. O código que você adicionou gera um novo objeto *Square*. Em seguida, ele verifica se esse objeto implementa a interface *IDraw* (o que é uma prática recomendada) e gera uma referência ao objeto, pela interface *IDraw*. Convém lembrar que, quando você implementa explicitamente uma interface, os métodos definidos por essa interface só estarão disponíveis ao 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 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*.

6. Adicione o seguinte código, mostrado em negrito, ao final do método *drawingCanvas\_MouseLeftButtonDown*:

```
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 enumeração *Colors* é fornecida como parte do .NET Framework.)



**Importante** Você deve chamar *Draw* antes de chamar *SetColor*. Isso deve ser feito dessa maneira porque o método *SetColor* só define 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.

7. Retorne ao arquivo DrawingPad.xaml na janela *Design View* e clique no objeto *Canvas* no meio do formulário. Na lista de eventos na janela *Properties*, clique duas vezes no evento *MouseRightButtonDown*.

O Visual Studio gera outro método, chamado *drawingCanvas\_MouseRightButtonDown*, executando-o quando o usuário clica com o botão direito do mouse sobre o canvas.

8. Adicione o código mostrado em negrito a seguir ao método *drawingCanvas\_MouseRightButtonDown*. A lógica nesse código é semelhante ao do método que manipula o botão esquerdo do mouse, exceto pelo fato de que ele exibe um objeto *Circle* em *HotPink*.

```
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);
 }
}
```

9. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.
10. Quando a janela *Drawing Pad* for exibida, clique com o botão esquerdo do mouse em qualquer lugar nessa janela. Deve ser exibido um quadrado violeta.
11. Clique com o botão direito do mouse em qualquer lugar na janela. Deve ser exibido um círculo rosa-shocking. Você pode clicar os botões direito e esquerdo do mouse quantas vezes desejar, e cada clique desenhará um quadrado ou um círculo na posição do mouse, como mostra a imagem a seguir:



12. Feche a janela e retorne ao Visual Studio.

## Classes abstratas

Você pode implementar as interfaces *ILandBound* e *IGrazable*, discutidas na seção anterior, em várias classes diferentes, dependendo de quantos tipos de mamíferos você 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.CheatGrass()
 {
 Console.WriteLine("Chewing grass");
 // código para pastar
 };
}

class Sheep : Mammal, ILandBound, IGrazable
{
 ...
 void IGrazable.CheatGrass()
 {
 Console.WriteLine("Chewing grass");
 // o mesmo código de horse para pastar
 };
}
```

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. 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. Por exemplo:

```
class GrazingMammal : Mammal, IGrazable
{
 ...
 void IGrazable.CheatGrass()
 {
 Console.WriteLine("Chewing grass");
 // código comum para pastar
 }
}

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 e 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 deve explicitar que a classe é abstrata, utilizando a palavra-chave *abstract*. Por exemplo:

```
abstract class GrazingMammal : Mammal, IGrazable
{
 ...
}
```

Se tentar instanciar um objeto *GrazingMammal*, o código não irá compilar:

```
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 (discutimos métodos virtuais no Capítulo 12), exceto por ele 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 e 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 modelando. 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 tão bem quanto uma classe base. O C# permite que você utilize a palavra-chave *sealed* para impedir que uma classe seja utilizada como uma classe base se quiser. 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.

 **Nota** Uma estrutura é implicitamente selada. Você nunca pode derivar de uma estrutura.

## 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 sobrescrever pos-

teriormente o método selado. Você pode selar apenas um método *override* e declarar o método como *sealed override*, ou seja, não pode selar um método que esteja implementando diretamente um método em uma interface. (Não é possível sobreescriver um método herdado diretamente de uma interface, só de uma classe.) 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.

## Implementando e utilizando 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* contém uma alta proporção de código duplicado. Compensa fatorar esse código em uma classe abstrata, chamada *DrawingShape*, porque esta ação facilitará 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\Drawing Using Interfaces – Complete de sua pasta Documentos.

2. No menu *Project*, clique em *Add Class*.

É exibida a caixa de diálogo *Add New Item – Drawing*.

3. Na caixa de texto *Name*, digite *DrawingShape.cs* e clique em *Add*.

O Visual Studio gera o arquivo e o exibe na janela *Code and Text Editor*.

4. No arquivo *DrawingShape.cs*, adicione as seguintes instruções *using* à lista localizada no início:

```
using System.Windows;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Controls;
```

5. O objetivo dessa classe é conter o código comum às classes *Circle* e *Square*. Um programa não poderá instanciar diretamente um objeto *DrawingShape*. Modifique a definição da classe *DrawingShape* e declare-a como *abstract*, como mostrado aqui em negrito:

```
abstract class DrawingShape
{
}
```

6. Adicione as 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 canvas, para que você possa mover esses campos para a classe abstrata. De modo semelhante, as classes *Square* e *Circle* usavam um campo para indicar o tamanho do objeto quando ele era processado; embora ele tenha outro nome em cada classe (*sideLength* e *radius*), semanticamente o campo executava 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*:

```
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. 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 (shape != null)
 {
 SolidColorBrush brush = new SolidColorBrush(color);
 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 sobrescrevam 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 ApplicationException("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(shape);
 }
}
```

Você acabou de finalizar a classe abstrata *DrawingShape*. A próxima etapa é mudar as classes *Square* e *Circle* para que elas herdem dessa classe, e remover o código duplicado das classes *Square* e *Circle*.

Modifique as classes `Square` e `Circle` para herdar da classe `DrawingShape`

- Exiba o código da classe *Square* na janela *Code and Text Editor*. 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 quaisquer interfaces.

2. Na classe *Square*, remova as definições dos campos *sideLength*, *rect*, *locX* e *locY*.
  3. Substitua o construtor existente pelo seguinte código, que chama o construtor da classe base. Perceba que o corpo desse construtor está vazio porque o construtor da classe base se encarrega de toda a inicialização necessária.

```
class Square : DrawingShape, IDraw, IColor
{
 public Square(int sideLength) : base(sideLength)
 {
 }
 ...
}
```

4. Remova os métodos *SetLocation* e *SetColor* da classe *Square*. A classe *DrawingShape* já fornece a implementação desses métodos.
5. 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)
{
 ...
}
```

6. Substitua o corpo do método *Draw* pelo código mostrado em negrito. 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* na classe *DrawingShape*.

```
public override void Draw(Canvas canvas)
{
 if (this.shape != null)
 {
 canvas.Children.Remove(this.shape);
 }
 else
 {
 this.shape = new Rectangle();
 }

 base.Draw(canvas);
}
```

7. Repita as etapas 2 a 6 para a classe *Circle*, exceto pelo fato de que o construtor deve ser chamado de *Circle* com um parâmetro chamado *radius*, 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 radius) : base(radius)
 {
 }

 public override void Draw(Canvas canvas)
 {
 if (this.shape != null)
 {
 canvas.Children.Remove(this.shape);
 }
```

```

 else
 {
 this.shape = new Ellipse();
 }

 base.Draw(canvas);
}
}

```

8. No menu *Debug*, clique em *Start Without Debugging*. Quando a janela *Drawing Pad* for exibida, verifique se os objetos *Square* aparecem quando você clica com o botão esquerdo do mouse na janela, e os objetos *Circle* são exibidos quando você clica com o botão direito do mouse na janela.
9. Feche a janela *Drawing Pad* e volte ao Visual Studio.

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 (*exigido*), ao definir métodos para interfaces e classes.

| Palavra-chave    | Interface        | Classe abstrata | Classe | Classe selada | Estrutura        |
|------------------|------------------|-----------------|--------|---------------|------------------|
| <i>abstract</i>  | não              | sim             | não    | não           | não              |
| <i>new</i>       | sim <sup>1</sup> | sim             | sim    | sim           | sim              |
| <i>override</i>  | não              | sim             | sim    | sim           | sim              |
| <i>private</i>   | não              | sim             | sim    | sim           | sim              |
| <i>protected</i> | não              | sim             | sim    | sim           | não <sup>2</sup> |
| <i>public</i>    | não              | sim             | sim    | sim           | sim              |
| <i>sealed</i>    | não              | sim             | sim    | exigido       | não              |
| <i>virtual</i>   | não              | sim             | sim    | não           | não              |

<sup>1</sup>Uma interface pode estender outra interface e introduzir um novo método com a mesma assinatura.

<sup>2</sup>Uma estrutura é implicitamente selada e não pode ser derivada.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 14.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 13

| Para                                                                                                  | Faça isto                                                                                                                                                                            |
|-------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Declarar uma interface                                                                                | Utilize a palavra-chave <i>interface</i> . Por exemplo:                                                                                                                              |
|                                                                                                       | <pre>interface IDemo {     string Name();     string Description(); }</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.Name()     {         ***     }      public string IDemo.Description()     {         ***     } }</pre>                              |
| Criar uma classe abstrata que só possa ser utilizada como uma classe base, contendo métodos abstratos | Declare a classe utilizando a palavra-chave <i>abstract</i> . Para cada método abstrato, declare o método com a palavra-chave <i>abstract</i> 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 <i>sealed</i> . Por exemplo:                                                                                                             |
|                                                                                                       | <pre>sealed class Horse {     *** }</pre>                                                                                                                                            |

## Capítulo 14

# Utilizando a coleta de lixo e o 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 um determinado momento e de modo seguro quanto a exceções escrevendo uma instrução *try/finally*.
- Liberar um recurso em um determinado momento e de modo seguro quanto a exceções escrevendo uma instrução *using*.

Você viu nos capítulos anteriores como criar variáveis e objetos, e agora já deve entender como a memória é alocada quando variáveis e objetos são criados (no caso de você ter esquecido, os tipos-valor são criados na pilha, e os tipos-referência são memória alocada a partir do heap). Os computadores não têm quantidades infinitas de memória, portanto, ela deve ser recuperada quando uma variável ou objeto não precisar 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 um objeto é destruído? Esse é o assunto deste capítulo.

## O tempo de vida de um objeto

Em primeiro lugar, vamos recapitular o que acontece quando você cria um objeto.

Você cria um objeto utilizando o operador *new*. O exemplo a seguir cria uma nova instância da classe *Square*, que você conheceu no Capítulo 13, “Criando interfaces e definido classes abstratas”.

```
Square mySquare = new Square(); // Square é um tipo-referência
```

Do seu ponto de vista, a operação *new* é atômica, mas, por baixo dos panos, 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** Os programadores C++ devem notar que no C# não é possível sobrepor o operador `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 você pode executar:

```
mySquare.Draw();
```



**Nota** Este código é baseado na versão da classe `Square` que herda da classe abstrata `DrawingShape` e que não implementa explicitamente a interface `IDraw`. Para obter mais informações, consulte o Capítulo 13.

Você pode fazer outras variáveis de referência referenciarem o mesmo objeto:

```
Square referenceToMySquare = mySquare;
```

Quantas referências a um objeto você pode criar? Quantas você quiser! Isso tem um impacto sobre o tempo de vida de um objeto. O runtime tem de manter o controle de todas essas referências. Se a variável `Square` desaparecer (saindo do escopo), outras variáveis (como `referenceToMySquare`) ainda poderão existir. O tempo de vida de um objeto não pode ser vinculado a uma determinada variável de referência. Um objeto pode ser destruído e sua memória só pode ser reivindicada quando *todas* as referências a ele desaparecerem.

Como a criação de um objeto, a destruição do objeto é um processo de duas fases, as quais espelham exatamente as duas fases de criação:

1. O runtime precisa organizar as coisas. Você pode controlar isso escrevendo um *destrutor*.
2. O runtime precisa retornar a memória que anteriormente pertencia ao objeto de volta ao heap; 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** Os programadores C++ devem notar que o C# não tem um operador `delete`. O runtime controla quando um objeto é destruído.

## Escrevendo destrutores

Você pode utilizar um destrutor para executar qualquer limpeza necessária quando um objeto vai para a coleta de lixo. Um destrutor é um método especial, parecido com um construtor, exceto pelo fato de o runtime o chamar depois de a última referência a um objeto desaparecer. A sintaxe para es-

crever um destrutor é um til (~) seguido pelo nome da classe. Como exemplo, eis uma classe simples que conta o número de instâncias existentes incrementando uma variável estática no construtor e decrementando a mesma variável estática no destrutor:

```
class Tally
{
 public Tally()
 {
 this.instanceCount++;
 }

 ~Tally()
 {
 this.instanceCount--;
 }

 public static int InstanceCount()
 {
 return this.instanceCount;
 }

 ...
}
```

Há algumas restrições importantes que dizem respeito aos destrutores:

- Os destrutores só se aplicam a tipos-referência. Você não pode declarar um destrutor em um tipo-valor, como um *struct*.

```
struct Tally
{
 ~Tally() { ... } // 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 runtime, chamada *coletor de lixo*, faz isso para você.

```
public ~Tally() { ... } // erro de tempo de compilação
```

- Um destrutor não pode aceitar quaisquer parâmetros. Novamente, isso ocorre porque você nunca chama o destrutor por conta própria.

```
~Tally(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 Tally
{
 ~Tally() { // Seu código entra aqui }
}
```

nisto:

```
class Tally
{
 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, "Gerenciando 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 perceber 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 coleto de lixo?

Você agora deve entender que nunca será possível destruir um objeto utilizando código C#. Não há uma sintaxe que faça isso. O tempo de execução se encarregará disso para você e existem boas razões para os projetistas do C# terem decidido impedi-lo de fazer isso. Se fosse *sua* responsabilidade destruir os objetos, mais cedo ou mais tarde, uma das seguintes situações poderia ocorrer:

- Você poderia esquecer 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 desalocada de volta para o heap. Você poderia esgotar a memória.
- Você poderia destruir um objeto ativo. Lembre que os objetos são acessados por referência. Se uma classe mantivesse uma referência a um objeto destruído, essa seria uma *referência oscilante*. A referência oscilante terminaria referenciando uma memória não utilizada ou talvez um objeto completamente diferente na 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.
- Você poderia tentar e 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 resistência e a segurança no alto de sua lista de objetivos de projeto. Em vez disso, o coleto de lixo é responsável por destruir os objetos para você, dando-lhe as seguintes garantias:

- Todo objeto será destruído, e seus destrutores serão executados. Quando um programa terminar, todos os objetos existentes serão destruídos.

- Cada objeto será destruído apenas uma vez.
- Cada objeto será destruído somente quando ele se tornar inacessível – isto é, quando não houver quaisquer referências 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 pode parecer uma pergunta 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 runtime só coleta o lixo quando há necessidade (quando ele verifica que a memória disponível está diminuindo) e então coleta o máximo possível. É melhor executar algumas limpezas grandes do que executar muitas pequenas!

**Nota** Você pode invocar 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 `System.GC.Collect` inicia o coletor de lixo, mas o processo executa de modo assíncrono; e o método `System.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 runtime 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 é talvez 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 você simplesmente não sabe quando. Consequentemente, 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 horas – normalmente 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 poderia precisar mover os objetos e atualizar as referências de objeto, e ele não pode fazê-lo caso o objeto esteja em uso.

**Nota** Uma thread é um caminho de execução separado em um aplicativo. O Windows utiliza threads para permitir que um aplicativo execute várias operações simultaneamente.

Os passos executados são:

1. Ele 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 as referências circulares não causam 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 requeira uma finalização é colocado em uma fila especial chamada *freachable* (pronuncia-se efe-rítchbôl).
3. Ele desaloca os objetos inacessíveis restantes (aqueles que não requerem finalização) movendo os objetos *acessíveis* para a parte inferior do heap, desfragmentando assim o heap e liberando a memória na parte superior do heap. 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 requerem finalização (agora na fila *freachable*) 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 mais lentamente. 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, exceto quando você realmente precisar deles. Por exemplo, considere a possibilidade de empregar uma instrução *using* (consulte a seção “A instrução *using*” mais adiante neste capítulo).

Você precisa ter bastante cuidado ao escrever um destrutor. Em particular, você deve estar 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 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 precisam ser liberados, e isso precisa ser feito o mais cedo possível. Nessas situações, sua única opção é você mesmo liberar o recurso, 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 sequencial de entrada. A classe *TextReader* contém um método virtual chamado *Close*, que 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*. Eis 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 coisa alguma 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 com esse 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 chamar o método de descarte 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 torna-se rapidamente difícil de manejá-lo se você remover mais de um recurso (você acaba tendo blocos *try* e *finally* aninhados).
- Em alguns casos, talvez você precise modificar o código (por exemplo, você poderia precisar reordenar a declaração da referência de recurso, lembrar-se de inicializar a referência como *null* e de 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 acidentalmente tentar utilizar o recurso após ele ter sido liberado.

A instrução *using* é projetada para solucionar todos esses problemas.

## A instrução *using*

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 em escopo. Infelizmente essa mesma palavra-chave tem dois significados diferentes.

A sintaxe para uma instrução *using* é:

```
using (tipo variável = inicialização)
{
 StatementBlock
}
```

Eis a melhor maneira de garantir que seu código sempre chamará *Close* em um *TextReader*:

```
using (TextReader reader = new StreamReader(filename))
{
 string line;
 while ((line = reader.ReadLine()) != null)
 {
 Console.WriteLine(line);
 }
}
```

A 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();
 }
 }
}
```

A variável que você declara em uma instrução *using* deve ser do tipo que implementa a interface *IDisposable*.

**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 acidentalmente tentar acessar um recurso removido.

A interface *IDisposable* reside no namespace *System* e contém apenas um método, chamado *Dispose*:

```
namespace System
{
 interface IDisposable
 {
 void Dispose();
 }
}
```

Isso acontece de tal maneira 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/finally*. Você agora tem uma solução que:

- É facilmente escalonável se precisar descartar múltiplos recursos.
- Não altera a lógica do código do programa.
- Elimina o problema e evita a repetição.
- É robusta. Você não pode utilizar a variável que foi declarada dentro da instrução *using* (nesse caso, *reader*) após essa ter terminado porque ela não está mais no escopo – você obterá um erro de tempo de compilação.

## Chamando o método *Dispose* a partir de um destrutor

Ao escrever uma classe, você deve escrever um destrutor ou implementar a interface *IDisposable*? 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, porque ela precisa que o programador se lembre de escrever uma instrução *using*. Mas é possível garantir que o método *Dispose* sempre execute chamando a partir de um destrutor. Isso funciona como um backup útil. Você poderá 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. Veja um exemplo de como fazer isso:

```
class Example : IDisposable
{
 private Resource scarce; // recurso escasso para gerenciar e descartar
 private bool disposed = false; // sinalizador para indicar se o recurso
 // foi descartado

 ...
 ~Example()
 {
 Dispose();
 }

 public virtual void Dispose()
 {
 if (!this.disposed)
 {
 try {
 // libera recursos escassos aqui
 }
 finally {
 this.disposed = true;
 GC.SuppressFinalize(this);
 }
 }
 }
}
```

```
public void SomeBehavior() // método de exemplo
{
 checkIfDisposed();
 ...
}

private void checkIfDisposed()
{
 if (this.disposed)
 {
 throw new ObjectDisposedException("Example: object has been disposed of");
 }
}
```

Observe os seguintes recursos da classe *Example*:

- A classe implementa a interface *IDisposable*.
- O destrutor chama *Dispose*.
- O método *Dispose* é público e pode ser chamado a qualquer momento.
- O método *Dispose* pode ser chamado de modo seguro muitas vezes. A variável *disposed* indica se o método já foi executado antes. O recurso escasso é liberado somente na primeira vez que o método executa.
- O método *Dispose* chama o método estático *GC.SuppressFinalize*. Esse método faz o coletor de lixo parar de chamar o destrutor no seu objeto, porque o objeto agora foi finalizado.
- Todos os métodos comuns da classe (como *SomeBehavior*) verificam se o objeto já foi descartado. Se afirmativo, eles geram uma exceção.

## Implementando descarte seguro quanto a exceções

No próximo exercício, você reescreverá uma pequena parte do código para torná-lo seguro quanto a exceções. O código abre um arquivo de texto, lê seu conteúdo linha por linha, escreve essas linhas em uma caixa de texto em um formulário na tela e então fecha o arquivo de texto. Entretanto, se surgir uma exceção enquanto o arquivo é lido ou enquanto as linhas são escritas na caixa de texto, a chamada para fechar o arquivo de texto será pulada. Você reescreverá o código para utilizar uma instrução *using*, garantindo assim que o código seja seguro quanto a exceções.

### Escreva uma instrução *using*

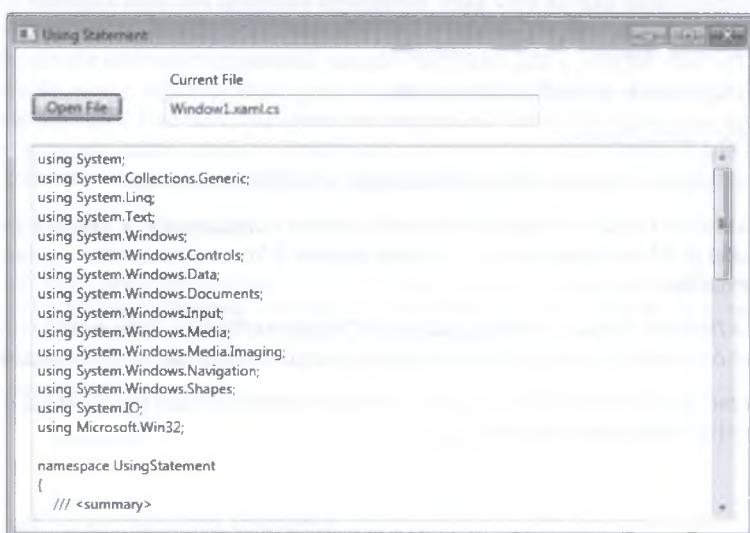
1. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto *UsingStatement*, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 14\UsingStatement na sua pasta Documentos.

3. No menu *Debug*, clique em *Start Without Debugging*. Um formulário Windows Presentation Foundation (WPF) aparece.
4. No formulário, clique em *Open File*.
5. Na caixa de diálogo *Open*, navegue para a pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 14\UsingStatement\UsingStatement na sua pasta Documentos e selecione o arquivo-fonte Window1.xaml.cs.

Esse é o arquivo-fonte do aplicativo.

6. Clique em *Open*.

O conteúdo do arquivo é exibido no formulário, como mostrado aqui:



7. Feche o formulário para retornar ao Visual Studio 2010.

8. Abra o arquivo MainWindow.xaml.cs na janela *Code and Text Editor* e então localize o método *openFileDialogFileOk*.

O método é semelhante a este:

```
private void openFileDialogFileOk(object sender,
System.ComponentModel.CancelEventArgs e)
{
 string fullPathname = openFileDialog.FileName;
 FileInfo src = new FileInfo(fullPathname);
 fileName.Text = src.Name;
 source.Clear();

 TextReader reader = new StreamReader(fullPathname);
 string line;
 while ((line = reader.ReadLine()) != null)
 {
 source.Text += line + "\n";
 }
 reader.Close();
}
```

As variáveis `fileName`, `openFileDialog` e `source` são três campos privados da classe `MainWindow`. Esse código utiliza um objeto `TextReader` chamado `reader` para abrir o arquivo especificado pelo usuário. (Os detalhes de como selecionar o arquivo estão descritos no Capítulo 23, “Obtendo a entrada do usuário”.) A instrução `while` contém a principal funcionalidade desse método; ela itera por meio do arquivo, em uma linha de cada vez, ao utilizar o método `ReadLine` do objeto `reader`, e exibe cada linha, anexando-a à propriedade `Text` do campo de texto `Source`, no formulário. Quando o método `ReadLine` retorna nulo, não existem mais dados no arquivo, o loop `while` termina e o método `Close` do objeto `reader` fecha o arquivo.

O problema com esse código é que não há garantia de que a chamada a `reader.Close` será executada. Se ocorrer uma exceção após a abertura do arquivo, o método terminará com uma exceção, mas o arquivo permanecerá aberto até o próprio aplicativo terminar.

9. Modifique o método `openFileDialogFileOk` e insira o código que processa o arquivo dentro de uma instrução `using` (incluindo as chaves de abertura e de fechamento), como mostrado em negrito. Remova a instrução que fecha o objeto `TextReader`.

```
private void openFileDialogFileOk(object sender,
System.ComponentModel.CancelEventArgs e)
{
 string fullPathname = openFileDialog.FileName;
 FileInfo src = new FileInfo(fullPathname);
 fileName.Text = src.Name;
 source.Clear();
 using (TextReader reader = new StreamReader(fullPathname))
 {
 string line;
 while ((line = reader.ReadLine()) != null)
 {
 source.Text += line + "\n";
 }
 }
}
```

Você não precisa mais chamar `reader.Close` porque ele será chamado automaticamente pelo método `Dispose` da classe `StreamReader` quando a instrução `using` for completada. Isso se aplica tanto se a instrução `using` concluir naturalmente como se concluir devido a uma exceção.

10. No menu `Debug`, clique em *Start Without Debugging*.

11. Verifique se o aplicativo ainda funciona como anteriormente e então feche o formulário.

Neste capítulo, vimos como o coletor de lixo funciona e como o .NET Framework o utiliza para desfazer 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. Você também viu como é possível utilizar a instrução `using` para implementar um descarte de recursos seguro quanto a exceções.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 15.

- Se quiser sair do Visual Studio 2010:

No menu `File`, clique em `Exit`. Se vir uma caixa de diálogo `Save`, clique em `Yes` e salve o projeto.

## Referência rápida do Capítulo 14

| 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 <i>public</i> ) 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 <i>System.GC.Collect</i> .                                                                                                                                                                                                  |
| Liberar um recurso em um determinado momento (mas com o risco de vazamentos de memória se uma exceção interromper a execução) | Escreva um método de descarte (um método que descarte um recurso) e chame-o explicitamente a partir do programa. Por exemplo:                                                                                                     |
|                                                                                                                               | <pre>class TextReader {     ***     public virtual void Close()     {         ***     } }  class Example {     void Use()     {         TextReader reader = ...;         // utilizar reader         reader.Close();     } }</pre> |

Parte III

## Criando componentes

|                                                                                          |     |
|------------------------------------------------------------------------------------------|-----|
| Capítulo 15: Implementando propriedades para acessar campos . . . . .                    | 327 |
| Capítulo 16: Utilizando indexadores . . . . .                                            | 347 |
| Capítulo 17: Interrompendo o fluxo do programa e tratando eventos . . . . .              | 361 |
| Capítulo 18: Apresentando genéricos . . . . .                                            | 385 |
| Capítulo 19: Enumerando coleções . . . . .                                               | 413 |
| Capítulo 20: Consultando dados na memória utilizando<br>expressões de consulta . . . . . | 427 |
| Capítulo 21: Sobrecarga de operadores . . . . .                                          | 451 |

## Capítulo 15

# Implementando 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.

As duas primeiras partes deste livro apresentaram a sintaxe básica da linguagem C# e mostraram como utilizar o C# para criar novos tipos empregando estruturas, enumerações e classes. Você viu também como o runtime gerencia a memória utilizada pelas variáveis e pelos objetos quando um programa é executado e agora deve entender o ciclo de vida dos objetos do C#. Os capítulos da Parte III, “Criando componentes”, são baseados nessas informações mostrando a você como utilizar o C# para criar componentes reutilizáveis – classes funcionais que você pode reutilizar em muitos aplicativos diferentes.

Este capítulo examina como definir e utilizar as propriedades para encapsular campos e dados em uma classe. Os capítulos anteriores enfatizaram que você deve tornar privados os campos em uma classe e fornecer métodos para armazenar e recuperar valores. Essa abordagem oferece acesso seguro e controlado a campos e permite encapsular a lógica e as regras adicionais em relação a valores que são permitidos. Mas a sintaxe para acessar um campo dessa maneira não é natural. Quando quer ler ou escrever uma variável, você normalmente utiliza uma instrução de atribuição; portanto, chamar um método para conseguir o mesmo efeito em um campo (que é, afinal de contas, apenas uma variável) parece algo grosseiro. As propriedades são projetadas para resolver esse problema.

## Implementando encapsulamento com métodos

Em primeiro lugar, vamos recapitular a motivação original para utilizar os métodos à 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 com essa estrutura é que ela não segue a regra principal 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 ter certeza de 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 utiliza 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; // oops
```

A maneira de resolver esse problema é criar campos privados e adicionar um método de acesso e um método modificador para ler e escrever 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 como *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, ele 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 escrever 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. As propriedades permitem 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.

A sintaxe de uma declaração de propriedade se parece com esta:

```
AccessModifier Type PropertyName
{
 get
 {
 // código de leitura de propriedade
 }

 set
 {
 // código de gravação de propriedade
 }
}
```

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 dados lidos e gravados pelos métodos de acesso *get* e *set*.

O próximo exemplo de código mostra a estrutura *ScreenPosition* reescrita utilizando propriedades. Ao ler 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 escritos, utilizando um parâmetro oculto e predefinido chamado *value*.



**Dica** Os campos e as propriedades seguem a convenção de nomes *public/private* padrão do Microsoft Visual C#. Os campos e as propriedades públicos devem iniciar com uma letra maiúscula, mas os campos e as propriedades privados devem começar com uma letra minúscula.

```
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 somente 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 desse capítulo mostrem como definir as propriedades para uma estrutura, eles são igualmente aplicáveis às classes; a sintaxe é a mesma.

## Utilizando propriedades

Ao utilizar uma propriedade em uma expressão, você pode empregá-la em um contexto de leitura (quando você estiver lendo 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* de uma 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 que você acessa as propriedades e os campos usando a mesma sintaxe. 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 traduz 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, você somente pode 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
{
 ...
 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. Por exemplo:

```
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, você somente pode utilizar a propriedade no contexto de gravação. Por exemplo, veja a propriedade *X* da estrutura *ScreenPosition* declarada como uma propriedade de gravação:

```
struct ScreenPosition
{
 ...
 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. Por exemplo:

```
Console.WriteLine(origin.X); // erro de tempo de compilação
origin.X = 200; // compila OK
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 possibilitará que você a leia. Ao tentar fazer login, o usuário pode informar a senha. Um método de login 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*) quando você a declara. Mas também é possível dentro da declaração de propriedade redefinir a acessibilidade da propriedade para os métodos de acesso *get* e *set*. Por exemplo, a versão da estrutura *ScreenPosition* mostrada aqui 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
{
 /**
 * @public int X
 */
 public int X
 {
 get { return this.x; }
 private set { this.x = rangeCheckedX(value); }
 }

 /**
 * @public int Y
 */
 public int Y
 {
 get { return this.y; }
 private set { this.y = rangeCheckedY(value); }
 }

 /**
 * @private int x, y;
 */
}
```

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 for declarada como *private*, você não poderá especificar o método de acesso de leitura como *public* (em vez disso, você tornaria a propriedade *public* e o método de acesso de gravação *private*).

## Nomes de propriedades e campos: um alerta

Embora seja uma prática comumente aceita dar às propriedades e aos campos privados o mesmo nome, diferindo apenas pela letra inicial maiúscula, você deve estar ciente de uma desvantagem. Examine o código a seguir que implementa uma classe chamada *Employee*. O campo *employeeID* é privado, mas a propriedade *EmployeeID* fornece acesso público a esse campo.

```
class Employee
{
 private int employeeID;

 public int EmployeeID;
 {
 get { return this.EmployeeID; }
 set { this.EmployeeID = value; }
 }
}
```

Esse código compilará perfeitamente bem, mas resulta em um programa que lança uma *StackOverflowException* sempre que a propriedade *EmployeeID* é 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!

## Entendendo as restrições de uma propriedade

As propriedades se parecem com e atuam como campos. Mas elas não são campos verdadeiros, e determinadas restrições se aplicam a elas:

- 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. O que isso realmente significa é que existem algumas diferenças entre campos e propriedades. 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 mais funcionar se você transformar os campos em propriedades. Retornaremos a essa questão na seção “Gerando 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 por-

que a propriedade na realidade não aponta para uma posição da memória, mas, em vez disso, para um método de acesso. Por exemplo:

```
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 são atribuídos são passados automaticamente para o método de acesso *set*, utilizando a variável *value*.
- Você não pode declarar propriedades *const*. Por exemplo:

```
const int X { get {...} set {...} } // erro de tempo de compilação
```

## Utilizando as propriedades adequadamente

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. Você poderia ser tentado a criar uma propriedade *Balance* em uma classe *BankAccount*, como esta:

```
class BankAccount
{
 /**
 * public money Balance
 {
 get { ... }
 set { ... }
 }

 private money balance;
}
```

Esse é um projeto medíocre, pois falha ao representar a funcionalidade necessária para sacar e 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:

```
class BankAccount
{
 /**
 * public money Balance { get { ... } } // Saldo
 * public void Deposit(money amount) { ... } // Depósitos
 * public bool Withdraw(money amount) { ... } // Sacar
 * private money balance;
}
```

## Declarando propriedades de interface

Já discutimos interfaces no Capítulo 13, “Criando interfaces e definindo 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. Por exemplo:

```
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*. Por exemplo:

```
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. Por exemplo:

```
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 criar uma estrutura porque estruturas são seladas implicitamente.

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). Por exemplo:

```
struct ScreenPosition : IScreenPosition
{
 ...
 int IScreenPosition.X
 {
 get { ... }
 set { ... }
 }

 int IScreenPosition.Y
 {
 get { ... }
 set { ... }
 }

 ...
 private int x, y;
}
```

## Utilizando propriedades em um aplicativo Windows

Ao configurar valores para propriedades de objetos, como controles *TextBox*, *Windows* e *Buttons*, utilizando a janela Properties no Microsoft Visual Studio 2010, você na verdade está gerando um código que configura os valores dessas propriedades em tempo de execução. Alguns componentes têm um grande número de propriedades, embora algumas sejam mais utilizadas do que outras. Você pode escrever seu próprio código para modificar boa parte dessas propriedades em tempo de execução empregando a mesma sintaxe que você viu por todo este capítulo.

No próximo exercício, você utilizará algumas propriedades predefinidas dos controles *TextBox* e da classe *Window* para criar um aplicativo simples que exibirá continuamente o tamanho da sua janela principal, mesmo quando ela for redimensionada.

### Utilize propriedades

1. Inicie o Visual Studio 2010 se ele ainda não estiver em execução.
2. Abra o projeto *WindowProperties*, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 15\WindowProperties na sua pasta Documentos.
3. No menu *Debug*, clique em *Start Without Debugging*.

O projeto compila e executa. Um formulário Windows Presentation Foundation (WPF) aparece, exibindo duas caixas de texto vazias rotuladas *Width* e *Height*.

No programa, os controles da caixa de texto são nomeados *width* e *height*. Atualmente, eles estão vazios. Você adicionará um código ao aplicativo que exibe o tamanho atual da janela, e que atualiza os valores nessas caixas de texto se a janela for redimensionada.

4. Feche o formulário e retorne ao ambiente de programação do Visual Studio 2010.
5. Exiba o arquivo MainWindow.xaml.cs na janela *Code and Text Editor* e localize o método *sizeChanged*.

Esse método é chamado pelo construtor de *MainWindow*. Você o utilizará para exibir o tamanho atual do formulário nas caixas de texto *width* e *height*. Também vai usar as propriedades *ActualWidth* e *ActualHeight* da classe *Window*, que retornam a largura e a altura atuais do formulário como valores *double*.

6. Adicione duas instruções ao método *sizeChanged* para exibir o tamanho do formulário. A primeira instrução deve ler o valor da propriedade *ActualWidth* do formulário, convertê-lo em uma string e atribuir esse valor à propriedade *Text* da caixa de texto *width*. A segunda instrução deve ler o valor da propriedade *ActualHeight* do formulário, convertê-lo em uma string e atribuir esse valor à propriedade *Text* da caixa de texto *height*.

O método *sizeChanged* deve ficar assim:

```
private void sizeChanged()
{
 width.Text = this.ActualWidth.ToString();
 height.Text = this.ActualHeight.ToString();
}
```

7. Localize o método *MainWindowSizeChanged*.

Esse método executa sempre que o tamanho da janela muda quando o aplicativo está em execução. Observe que esse método chama o método *sizeChanged* para exibir o novo tamanho da janela nas caixas de texto.

8. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o projeto.

O formulário exibe as duas caixas de texto contendo os valores *305* e *155*. Essas são as dimensões padrão do formulário, especificadas quando o formulário foi projetado.

9. Redimensione o formulário. Observe que o texto nas caixas de texto muda a fim de refletir o novo tamanho.

10. Feche o formulário e retorne ao ambiente de programação do Visual Studio 2010.

## Gerando propriedades automáticas

Este capítulo mencionou anteriormente que o principal propósito das propriedades é ocultar a implementação dos campos. Isso é bom se suas propriedades realizarem 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 abordagem. Há pelo menos duas boas razões para você definir propriedades em vez de expor dados como campos públicos:

■ **Compatibilidade com aplicativos** Os campos e as propriedades se expõem utilizando diferentes metadados nos assemblies. Se desenvolver uma classe e decidir utilizar os campos públicos, qualquer aplicativo que usar essa classe irá referenciar esses itens como campos. Embora seja possível empregar a mesma sintaxe C# para ler e gravar um campo que você utiliza ao ler e gravar uma propriedade, o código compilado é bem diferente – o compilador C# só oculta as diferenças de você. Se você mais tarde decidir que precisa transformar esses campos em propriedades (talvez os requisitos do negócio tenham mudado e você precise realizar 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. Para esse fim, 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 a 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 automaticamente gerado; se precisar incluir uma lógica adicional posteriormente, você poderá fazer isso sem quebrar aplicativos existentes. Você deve observar, porém, que é necessário especificar um método de acesso *get* e um método de acesso *set* com uma propriedade automaticamente gerada – 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*.

## Inicializando objetos com propriedades

No Capítulo 7, “Criando e gerenciando classes e objetos”, você aprendeu a definir construtores para inicializar um objeto. Um objeto pode ter múltiplos 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 você precise 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 inicializa 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 seus valores padrão e definir 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 valores para qualquer propriedade pública que tenha métodos de acesso *set*. Isso significa que 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 que o polígono contém e o comprimento desses lados.

## Defina propriedades automáticas e utilize inicializadores de objeto

1. No Visual Studio 2010, abra o projeto *AutomaticProperties*, localizado na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 15\AutomaticProperties 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.
2. No *Solution Explorer*, clique com o botão direito do mouse no projeto *AutomaticProperties*, aponte para *Add* e então clique em *Class*. Na caixa de diálogo *Add New Item – AutomaticProperties*, na caixa de texto *Name*, digite **Polygon.cs** e então 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*.

3. Adicione as propriedades automáticas *NumSides* e *SideLength*, mostradas em negrito, à classe *Polygon*:

```
class Polygon
{
 public int NumSides { get; set; }
 public double SideLength { get; set; }
}
```

4. Adicione o seguinte construtor padrão à classe *Polygon*:

```
class Polygon
{
 ...
 public Polygon()
 {
 this.NumSides = 4;
 this.SideLength = 10.0;
 }
}
```

Neste exercício, o polígono padrão é um quadrado com lados de comprimento de 10 unidades.

5. Exiba o arquivo *Program.cs* na janela *Code and Text Editor*.

6. Adicione as instruções mostradas em negrito ao método *DoWork*:

```
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 utilizando o 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 ao final do método *DoWork*:

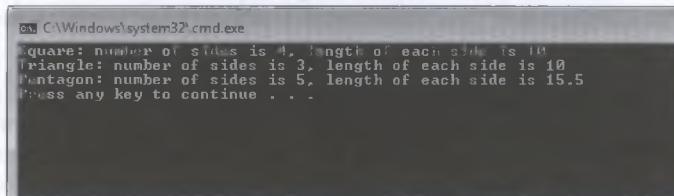
```
static void DoWork()
{
 ...
 Console.WriteLine("Square: number of sides is {0}, length of each side is {1}",
 square.NumSides, square.SideLength);
```

```
 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 se o programa constrói e executa, gravando a mensagem mostrada no console:



```
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 2010.

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 você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 16.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 15

| Para                                                                      | Faça isto                                                                                                                 |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Declarar uma propriedade de leitura/gravação para uma estrutura ou classe | Declare o tipo da propriedade, seu nome, um método de acesso <i>get</i> e um método de acesso <i>set</i> . Por exemplo:   |
|                                                                           | <pre>struct ScreenPosition {     ...     public int X     {         get { ... }         set { ... }     }     ... }</pre> |
| Declarar uma propriedade somente-leitura para uma estrutura ou classe     | Declare uma propriedade com apenas um método de acesso <i>get</i> . Por exemplo:                                          |
|                                                                           | <pre>struct ScreenPosition {     ...     public int X     {         get { ... }     }     ... }</pre>                     |
| Declarar uma propriedade somente-gravação para uma estrutura ou classe    | Declare uma propriedade com apenas um método de acesso <i>set</i> . Por exemplo:                                          |
|                                                                           | <pre>struct ScreenPosition {     ...     public int X     {         set { ... }     }     ... }</pre>                     |
| Declarar uma propriedade em uma interface                                 | Declare uma propriedade com apenas uma palavra-chave <i>get</i> ou <i>set</i> , ou ambas. Por exemplo:                    |
|                                                                           | <pre>interface IScreenPosition {     int X { get; set; } // nenhum corpo     int Y { get; set; } // nenhum corpo }</pre>  |

(continua)

| Para                                                                       | Faça isto                                                                                                                                                                                                                                                                                                                         |
|----------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Implementar uma propriedade de interface em uma estrutura ou em uma classe | <p>Na classe ou estrutura que implementa a interface, declare a propriedade e implemente os métodos de acesso. Por exemplo:</p> <pre>struct ScreenPosition : IScreenPosition {     public int X     {         get { ... }         set { ... }     }      public int Y     {         get { ... }         set { ... }     } }</pre> |
| Criar uma propriedade automática                                           | <p>Na classe ou estrutura que contém a propriedade, defina a propriedade com métodos de acesso <code>get</code> e <code>set</code> vazios. Por exemplo:</p> <pre>class Polygon {     public int NumSides { get; set; } }</pre>                                                                                                    |
| Usar propriedades para inicializar um objeto                               | <p>Ao construir o objeto, especifique as propriedades e seus valores como uma lista dentro de chaves. Por exemplo:</p> <pre>Triangle tri3 =     new Triangle { Side2Length = 12, Side3Length = 17 };</pre>                                                                                                                        |

# Capítulo 16

# Utilizando 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 os métodos de acesso `set`.
- Criar interfaces que declaram indexadores.
- Implementar indexadores em estruturas e classes que herdam de interfaces.

O capítulo anterior descreveu como implementar e usar as propriedades como um meio de fornecer acesso controlado aos campos em uma classe. As propriedades são úteis para espelhar campos que contenham um valor único. Mas os indexadores são inestimáveis se você quiser fornecer acesso aos itens que contenham 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 que você utiliza para um indexador é a mesma utilizada para um array.

A melhor maneira de entender indexadores é trabalhar com um exemplo. Primeiro, examinaremos um problema e veremos uma solução que não utiliza os indexadores. Em seguida, trabalharemos no mesmo problema e examinaremos uma solução melhor que utiliza os 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 de 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*, a saber:

- **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.
- **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 para a esquerda em duas casas, 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.
- **Operador OR (|)** É um operador binário que realiza uma operação OR 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*).
- **Operador AND (&)** Executa uma operação AND bit a bit. AND é semelhante ao operador OR bit a bit, exceto por ele retornar um valor contendo um 1 em cada posição onde os dois operandos têm um 1. Portanto, *204 & 20* é 8 (204 é *11001100*, 24 é *00011000* e 8 é *00001000*).
- **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 em 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 seguinte expressão emprega os operadores de deslocamento para a esquerda (*<<*) e o operador AND (*&*) em nível de bits para determinar se o sexto bit do *int* chamado *bits* está definido com 0 ou 1:

```
(bits & (1 << 5)) != 0
```

Vamos supor que a variável *bits* contém o valor decimal 42. Em binário, esse valor é *00101010*. O valor decimal 1 é *00000001* em binário, de modo que a expressão *1 << 5* tem o valor *00100000*. 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* tiver o valor 65, ou *01000001* em binário, o valor da expressão é *01000001 & 00100000*, que dá o resultado binário de *00000000*, ou zero.

Esse é um exemplo relativamente complexo, mas comum, quando comparado com a seguinte expressão, que utiliza o operador de atribuição composta `&=` para definir o bit na posição 4 como 0:

```
bits &= ~(1 << 4)
```

**Nota** Os operadores em nível de bits contam as posições dos bits da direita para a esquerda, de modo que o bit 0 é o bit posicionado mais à direita, e o bit na posição 6 é o bit posicionado seis casas a partir da direita.

De modo semelhante, para definir o bit na posição 4 como 1, você pode utilizar um operador OR (`|`) em nível de bits. A seguinte expressão complexa se baseia no operador de atribuição composta `|=`:

```
bits |= (1 << 4)
```

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 deixa de criar uma abstração do problema que ela soluciona.

## O mesmo exemplo utilizando indexadores

Vamos deixar de lado a solução fraca por um momento e parar 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 ele fosse um array de bits! Em outras palavras, o que queremos ser capazes de escrever para acessar o bit no índice 6 da variável `bits` é algo assim:

```
bits[6]
```

E para definir o bit no índice 4 como `true`, gostaríamos de escrever:

```
bits[4] = true
```

**Nota** Para os desenvolvedores experientes em C, o valor booleano `true` é sinônimo do valor binário 1, e o valor booleano `false` equivale ao valor binário 0. Consequentemente, a expressão `bits[4] = true` significa "definir o bit 4 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 como e seja utilizado como um array de variáveis `bool`, mas que seja implementado por meio de um `int`. Você pode alcançar essa façanha definindo um indexador. Vamos chamar esse novo tipo de `IntBits`. O `IntBits` conterá um valor `int` (inicializado 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, então faz sentido criá-lo como uma estrutura em vez de como uma classe.

```
struct IntBits
{
 public IntBits(int initialValue)
 {
 bits = initialValue;
 }

 // indexador a ser escrito aqui

 private int bits;
}
```

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 para a estrutura *IntBits* emprega um inteiro como tipo do argumento *index* e retorna um booleano, como a seguir:

```
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, desativa-o
 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 podem definir no máximo um indexador, 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 complexas expressões bit a bit 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 que elemento será acessado.

 **Nota** Você deve executar 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 1111110
IntBits bits = new IntBits(adapted);
bool peek = bits[6]; // recupera o valor bool do índice 6; deve ser true (1)
bits[0] = true; // configura o bit do índice 0 como true (1)
bits[3] = false; // configura o bit do índice 3 como false (0)
 // o valor int adaptado agora é 1110111, ou 119 em decimal
```

Essa sintaxe é certamente muito mais fácil de entender, pois captura direta e sucintamente a essência do problema.

## Entendendo 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 escrever para um indexador, o compilador traduz automaticamente seu código com lógica de array para uma chamada ao método de acesso *set* desse indexador, configurando o argumento *index* como o valor especificado entre colchetes. Por exemplo:

```
bits[4] = true;
```

Essa instrução é convertida em uma chamada ao método de acesso *set* para *bits* onde *index* é 4. Assim como com propriedades normais, os dados que você grava no indexador (nesse caso, *true*) tornam-se disponíveis dentro do método de acesso *set* utilizando 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, os métodos de acesso *get* e *set* são utilizados. Examine a próxima instrução, que utiliza o operador XOR (^) para inverter o valor do bit 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).

## Comparando indexadores e arrays

Quando você utiliza um indexador, a sintaxe é deliberadamente muito parecida com a de um array. Mas existem algumas diferenças importantes entre os indexadores e os arrays:

- Os indexadores podem utilizar subscritos não numéricos, como uma string, conforme mostrado no seguinte exemplo. Os arrays só podem utilizar subscritos inteiros:

```
public int this [string name]{...} // OK
```



**Dica** Muitas classes de coleção, como *Hashtable*, que implementam uma pesquisa associativa baseada em pares chave/valor, implementam os indexadores como uma alternativa conveniente ao uso do método *Add* para adicionar um novo valor e como uma alternativa a iterar pela propriedade *Values* a fim de localizar um valor no seu código. Por exemplo, em vez disso:

```
Hashtable ages = new Hashtable();
ages.Add("John", 42);
```

você pode utilizar isso:

```
Hashtable ages = new Hashtable();
ages["John"] = 42;
```

- 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 contêm 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* (dados):

```
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 *data* privada 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 essa não for 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 (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 abordagem pode tornar-se bem complicada e cara em termos de uso de memória. Os indexadores fornecem uma solução natural para esse problema – não exponha o array inteiro como uma propriedade; simplesmente disponibilize 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. Por exemplo:

```
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*. Por exemplo:

```
class RawInt : IRawInt
{
 ...
 public virtual bool this [int index]
 {
 get { ... }
 set { ... }
 }
 ...
}
```

Você também pode escolher implementar um indexador utilizando a sintaxe de implementação explícita abordada no Capítulo 12, “Trabalhando com herança”. Uma implementação explícita de um indexador é não pública e não virtual (e, portanto, não pode ser redefinida). Por exemplo:

```
struct RawInt : IRawInt
{
 ...
 bool IRawInt.this [int index]
 {
 get { ... }
 set { ... }
 }
 ...
}
```

## Utilizando 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. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto *Indexers*, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter16\Indexers na sua pasta Documentos.

Esse é um aplicativo Windows Presentation Foundation (WPF) que permite ao usuário procurar o número de telefone e também localizar o nome de um contato que corresponde a um dado número de telefone.

3. No menu *Debug*, clique em *Start Without Debugging*.

O projeto compila e executa. Aparece um formulário, exibindo duas caixas de texto vazias rotuladas *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. Sua tarefa é concluir o aplicativo para que esses botões funcionem.

4. Feche o formulário e retorne ao Visual Studio 2010.
5. Exiba o arquivo *Name.cs* 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 utilizando 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* no 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.

### Escreva os indexadores

1. No arquivo *PhoneBook.cs*, adicione um indexador público somente-leitura à classe *PhoneBook*, como mostrado em negrito no código a seguir. O indexador deve retornar *Name* e obter um item *PhoneNumber* como seu 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 seguinte. 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 ele o encontrar; caso contrário, *IndexOf* retorna *-1*. Se o indexador encontrar o número de telefone, ele deve retorná-lo; caso contrário, ele deve retornar um valor *Name* vazio (Observe que *Name* é uma estrutura e, como tal, o construtor padrão define seu campo *name* privado com *null*.)

```

sealed class PhoneBook
{
 ...
 public Name this [PhoneNumber number]
 {
 get
 {
 int i = Array.IndexOf(this.phoneNumbers, number);
 if (i != -1)
 {
 return this.names[i];
 }
 else
 {
 return new Name();
 }
 }
 }
 ...
}

```

3. Adicione um segundo indexador *público* somente-leitura à 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 sobre carregados podem coexistir porque eles retornam 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 todos os erros de sintaxe e então recompile, se necessário.

### Chame os indexadores

1. Exiba o arquivo *MainWindow.xaml.cs* na janela *Code and Text Editor* e então localize o método *findPhoneClick*.

Esse método é chamado quando o primeiro botão *Search by Name* é pressionado. Esse método estará vazio. Adicione o código mostrado em negrito no exemplo a seguir para realizar estas tarefas:

- 1.1. Ler o valor da propriedade *Text* a partir da caixa de texto *name* no formulário. Isso é uma string contendo o nome de contato que o usuário digitou.
- 1.2. Se a string não estiver vazia, procure o número de telefone correspondente a esse nome no *PhoneBook*, utilizando o indexador (observe que a classe *MainWindow* contém um campo *PhoneBook* privado chamado *phoneBook*). Construa um objeto *Name* a partir da string e passe-o como o parâmetro para o indexador *PhoneBook*.
- 1.3. Grave a propriedade *Text* da estrutura *PhoneNumber* retornada pelo indexador na caixa de texto *phoneNumber* no formulário.

O método *findPhoneClick* deve ser semelhante a este:

```
private void findPhoneClick(object sender, RoutedEventArgs e)
{
 string text = name.Text;
 if (!String.IsNullOrEmpty(text))
 {
 Name personsName = new Name(text);
 PhoneNumber personsPhoneNumber = this.phoneBook[personsName];
 phoneNumber.Text = personsPhoneNumber.Text;
 }
}
```

**Dica** Note o uso do método estático *IsNullOrEmpty* de *String* para determinar se uma string está vazia ou contém um valor nulo. Esse é o método preferido para testar se uma string contém um valor. Ele retorna *true* se a string tiver um valor não nulo, e *false*, caso contrário.

2. Localize o método *findNameClick* no arquivo *MainWindow.xaml.cs*. Ele está abaixo do método *findPhoneClick*.

O método *findName\_Click* é chamado quando o botão *Search by Phone* é clicado. Esse método está atualmente vazio, assim você precisa implementá-lo desta maneira (o código é mostrado em negrito no exemplo a seguir).

- 2.1. Leia o valor da propriedade *Text* a partir da caixa de texto *phoneNumber* no formulário. Isso é uma string contendo o número de telefone que o usuário digitou.
- 2.2. Se a string não estiver vazia, procure o nome correspondente a esse número de telefone no *PhoneBook*, utilizando o indexador.

- 2.3.** Grave a propriedade *Text* da estrutura *Name* retornada pelo indexador na caixa de texto *name* no formulário.

O método completo deve ser parecido com este:

```
private void findNameClick(object sender, RoutedEventArgs e)
{
 string text = phoneNumber.Text;
 if (!String.IsNullOrEmpty(text))
 {
 PhoneNumber personsPhoneNumber = new PhoneNumber(text);
 Name personsName = this.phoneBook[personsPhoneNumber];
 name.Text = personsName.Text;
 }
}
```

- 3.** No menu *Build*, clique em *Build Solution*. Corrija qualquer erro que ocorra.

### Execute o aplicativo

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

- 2.** Digite seu nome e o número de telefone nas caixas de texto e então clique em *Add*.

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 evitar confusões, certifique-se de fornecer nomes e números de telefone diferentes.

- 4.** Digite um nome que você utilizou no passo 2 na caixa de texto *Name* e então clique em *Search by Name*.

O número de telefone que você adicionou para esse contato no Passo 2 é recuperado do catálogo de telefone e exibido na caixa de texto *Phone Number*.

- 5.** Digite um número de telefone para um diferente contato na caixa de texto *Phone Number* e então clique em *Search by Phone*.

O nome do contato é recuperado do catálogo telefônico e exibido na caixa de texto *Name*.

- 6.** Digite um nome que você não inseriu no catálogo telefônico na caixa de texto *Name* e então clique em *Search by Name*.

Desta vez, a caixa de texto *Phone Number* está vazia, indicando que o nome não pôde ser encontrado no catálogo telefônico.

- 7.** Feche o formulário e retorne ao Visual Studio 2010.

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, ao utilizar uma lógica definida pelo método de acesso *get*, e aprendeu a utilizar o método de acesso *set* com um índice para preencher um valor em um indexador.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 17.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 16

| Para                                                                                                                             | Faça isto                                                                                                                                                                                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Criar um indexador para uma classe ou estrutura                                                                                  | <p>Declare o tipo do indexador, seguido pela palavra-chave <code>this</code> e então os argumentos do indexador entre colchetes. O corpo do indexador pode conter um método de acesso <code>get</code> e/ou <code>set</code>. 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 <code>get</code> e/ou <code>set</code>. 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 a 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

# Interrompendo o fluxo do programa e tratando 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 para um delegate.
- Declarar um campo de evento.
- Manipular um evento utilizando um delegate.
- Disparar um evento.

Grande parte do código escrito nos vários exercícios deste livro pressupõe que as instruções são executadas sequencialmente. Embora esse seja o cenário 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. O exemplo clássico desse estilo de programa é o formulário Windows Presentation Foundation (WPF). Um formulário WPF 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 imediatamente, mas 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 com o usuário, mas 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 aplicação, o tempo de execução tem de fornecer duas coisas: um meio de indicar que algo urgente está acontecendo e uma maneira de especificar um código que deve ser executado quando isso acontecer. Essa é a finalidade de eventos e delegates.

Começaremos examinando os delegates.

## Declarando e utilizando delegates

Um delegate é um ponteiro para um método. Você pode chamar um método por meio de um delegate ao especificar o nome desse delegate. Ao invocar um delegate, na verdade o runtime executa o método que o delegate referencia. Você pode alterar dinamicamente o método que um delegate referencia de modo que o código que chama um delegate possa realmente executar um método diferente a cada vez. A melhor maneira de entender os delegates é vê-los em ação, portanto, vamos acompanhar um exemplo.



**Nota** Se você está familiarizado com C++, um delegate é muito semelhante a um ponteiro de função. Entretanto, os delegates são fortemente tipados; você pode fazer um delegate referenciar apenas um método que corresponda à assinatura do delegate e não pode chamar um delegate que não referece um método válido.

## 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. As máquinas foram construídas e instaladas por fornecedores especialistas, e são controladas por computador e cada fornecedor disponibilizou um conjunto de APIs 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!



**Nota** O termo API significa Application Programming Interface, ou interface de programação de aplicativos. Trata-se de um método ou um conjunto de métodos expostos por um fragmento de software que pode ser usado para controlar esse software. Você pode pensar no Microsoft .NET Framework como um conjunto de APIs, porque ele fornece métodos que permitem controlar o CLR (common language runtime) do .NET e o sistema operacional Microsoft Windows.

Cada máquina tem seu processo controlado por computador (e API) para ser desligada de modo seguro. Estes estão resumidos assim:

```
StopFolding(); // Máquina de dobra e modelagem
FinishWelding(); // Máquina de solda
PaintOff(); // Máquina de pintura
```

## Implementando a fábrica sem utilizar delegates

Uma abordagem simples para implementar a funcionalidade de desligamento no programa de controle é mostrada a seguir:

```
class Controller
{
 // Campos que representam as diferentes máquinas
 private FoldingMachine folder;
 private WeldingMachine welder;
 private PaintingMachine painter;

 public void ShutDown()
 {
 folder.StopFolding();
 welder.FinishWelding();
 painter.PaintOff();
 }

}
```

Embora esse enfoque funcione, ele não é muito extensível ou flexível. Se a fábrica comprar uma nova máquina, você precisará modificar esse código; a classe *Controller* e um código para gerenciar as máquinas estão fortemente acoplados.

## Implementando a fábrica utilizando um delegate

Embora os nomes de cada método sejam diferentes, todos têm a mesma “forma”: não recebem parâmetro algum e não retornam um valor (veremos o que acontece se esse não for o caso mais adiante, portanto, tenha um pouco de paciência comigo!). O formato geral de cada método é:

```
void nomeDoMétodo();
```

Aqui um delegate é útil, pois 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 as seguintes questões:

- Utilize a palavra-chave *delegate* ao declarar um delegate.
- Um delegate define a forma dos métodos que ele pode referenciar. Você especifica o tipo de retorno (*void*, neste exemplo), um nome para o delegate (*stopMachineryDelegate*) e todos os parâmetros (não há parâmetro algum neste caso).

Após ter definido o delegate, você pode criar uma instância e fazê-la referenciar um método correspondente utilizando o operador de atribuição composta *+=*. Você pode fazer isso no construtor da classe *Controller* como a seguir:

```
class Controller
{
 delegate void stopMachineryDelegate();
 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, porém, não está realmente chamando-o. O operador *+* é sobre carregado a fim de ter esse novo significado quando utilizado com delegates (você aprenderá mais sobre sobrecarga de operadores no Capítulo 21, “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 também 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 invocando o delegate, como a seguir:

```
public void ShutDown()
{
 this.stopMachinery();
 ...
}
```

Utilize a mesma sintaxe para chamar um delegate que você utiliza para fazer uma chamada de método. Se o método que o delegate referencia receber qualquer parâmetro, você deve especificá-lo nesse momento, entre parênteses.

**Nota** Se tentar chamar um delegate que não foi inicializado e que não referencia método algum, você receberá uma *NullReferenceException*.

A principal vantagem de utilizar um delegate é que ele pode referenciar mais de um método; basta utilizar o operador `+=` para adicionar os métodos ao delegate, como mostrado a seguir:

```
public Controller()
{
 this.stopMachinery += folder.StopFolding;
 this.stopMachinery += welder.FinishWelding;
 this.stopMachinery += painter.PaintOff;
}
```

Ativar `this.stopMachinery()` no método `Shutdown` da classe `Controller` automaticamente faz 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 composto `-=`:

```
this.stopMachinery -= folder.StopFolding;
```

O esquema atual adiciona os métodos de máquina ao delegate no construtor `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. Você tem diversas opções:

- Tornar a variável do delegate, `stopMachinery`, pública:

```
public stopMachineryDelegate stopMachinery;
```

- Manter a variável do delegate `stopMachinery` privada, mas fornecer uma propriedade de leitura e gravação para proporcionar-lhe acesso:

```
public 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 o método especificado do delegate (observe que você especifica um método como um parâmetro utilizando um tipo delegate):

```

public void Add(stopMachineryDelegate stopMethod)
{
 this.stopMachinery += stopMethod;
}

public void Remove(stopMachineryDelegate stopMethod)
{
 this.stopMachinery -= stopMethod;
}

```

Se você for um purista em orientação a objetos, provavelmente optará pela abordagem *Add/Remove*. Mas as outras alternativas são mais utilizadas, razão pela qual elas são apresentadas aqui.

Qualquer que seja a técnica escolhida, você deve remover o código que adiciona os métodos de máquina ao delegate a partir do construtor *Controller*. Você pode então instanciar um *Controller* e objetos representando as outras máquinas como a seguir (este exemplo utiliza a abordagem *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();
...

```

## Utilizando delegates

No exercício a seguir, você concluirá um aplicativo que implementa um relógio internacional. O aplicativo contém um formulário WPF que exibe a hora local, assim como a hora atual em Londres, Nova York e Tóquio. Cada exibição é controlada por um objeto relógio. Cada relógio é implementado de modo diferente, para simular o cenário anterior de controle de um conjunto de máquinas independentes que funcionam em uma fábrica. Entretanto, cada relógio conta com um

par de métodos que permitem iniciar e parar o relógio. Quando você inicia um relógio, a hora em sua exibição é atualizada a cada segundo. Quando você para um relógio, a exibição não é mais atualizada. Você adicionará uma funcionalidade ao aplicativo, que inicia e para os relógios, por meio de delegates.

## Conclua o aplicativo World Clock

1. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto Clock na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 17\Clock na sua pasta Documentos.
3. No menu *Debug*, clique em *Start Without Debugging*.

O projeto é compilado e executado. Aparece um formulário que exibe a hora local, assim como a hora em Londres, Nova York e Tóquio. O relógio exibe as horas atuais como “00:00:00”.

4. Clique em *Start* para iniciar os relógios.

Nada acontece. O método *Start* ainda não foi escrito, e o botão *Stop* está desabilitado, por padrão. Sua tarefa é implementar o código associado a esses botões.

5. Feche o formulário e retorne ao ambiente do Visual Studio 2010.
6. Examine a lista de arquivos no Solution Explorer. O projeto contém alguns arquivos, como o AmericanClock.cs, EuropeanClock.cs, JapaneseClock.cs e o LocalClock.cs, com as classes que implementam os diversos relógios. Não se preocupe ainda com o funcionamento desses relógios (embora seja recomendável que você examine o código). Entretanto, o mais importante é conhecer os nomes dos métodos que iniciam e param cada tipo de relógio. A lista a seguir apresenta um resumo por tipo de relógio (todos eles são semelhantes):

- ❑ **AmericanClock.** O método que inicia é chamado de *StartAmericanClock*, e o método que interrompe é o *StopAmericanClock*. Nenhum desses métodos aceita parâmetros, e ambos têm um tipo de retorno *void*.
- ❑ **EuropeanClock.** O método que inicia é chamado de *StartEuropeanClock*, e o método que interrompe é o *StopEuropeanClock*. Nenhum desses métodos aceita parâmetros, e ambos têm um tipo de retorno *void*.
- ❑ **JapaneseClock.** O método que inicia é chamado de *StartJapaneseClock*, e o método que interrompe é o *StopJapaneseClock*. Nenhum desses métodos aceita parâmetros, e ambos têm um tipo de retorno *void*.
- ❑ **LocalClock.** O método que inicia é chamado de *StartLocalClock*, e o método que interrompe é o *StopLocalClock*. Nenhum desses métodos aceita parâmetros, e ambos têm um tipo de retorno *void*.

7. Abra o arquivo *ClockWindow.xaml.cs* na janela *Code and Text Editor*. Esse é o código do formulário WPF, parecido com o seguinte:

```
public partial class ClockWindow : Window
{
 private LocalClock localClock = null;
 private EuropeanClock londonClock = null;
 private AmericanClock newYorkClock = null;
 private JapaneseClock tokyoClock = null;

 public ClockWindow()
 {
 InitializeComponent();
 localClock = new LocalClock(localTimeDisplay);
 londonClock = new EuropeanClock(londonTimeDisplay);
 newYorkClock = new AmericanClock(newYorkTimeDisplay);
 tokyoClock = new JapaneseClock(tokyoTimeDisplay);
 }

 private void startClick(object sender, RoutedEventArgs e)
 {

 }

 private void stopClick(object sender, RoutedEventArgs e)
 {

 }
}
```

Os quatro campos privados representam os quatro objetos relógio, utilizados pelo aplicativo. O construtor inicializa cada um desses objetos relógio. Em cada caso, o parâmetro para o construtor especifica o campo de texto no formulário que o objeto relógio atualizará quando começar a funcionar. O método *startClick* é executado quando o usuário clica no botão *Start* no formulário, e seu objetivo é iniciar cada relógio. De modo semelhante, o método *stopClick* é executado quando o usuário clica no botão *Stop* e sua finalidade é parar os relógios. É possível constatar que esses dois métodos estão atualmente vazios.

Uma abordagem simplória seria tão somente chamar os métodos adequados que iniciam cada relógio no método *startClick*, e os métodos que param cada relógio, no método *stopClick*. Entretanto, como vimos anteriormente neste capítulo, essa abordagem associa o aplicativo ao modo como cada relógio é implementado, e não é muito extensível. Em vez disso, você criará um objeto *Controller* para iniciar e parar os relógios, e utilizará um par de delegates para especificar os métodos que o objeto *Controller* deve usar.

8. No menu *Project*, clique em *Add Class*. Na caixa de diálogo *Add New Item – Delegates*, na caixa de texto *Name*, digite **Controller.cs** e clique em *Add*.

O Visual Studio cria a classe *Controller* e exibe o arquivo *Controller.cs* na janela *Code and Text Editor*.

9. Na classe *Controller*, adicione os tipos delegate, *startClocksDelegate* e *stopClocksDelegate*, como mostrado em negrito a seguir. Esses tipos delegate podem fazer referência a métodos que não aceitam parâmetros e que têm um tipo de retorno *void*. Essa assinatura e o tipo de retorno correspondem a vários métodos de iniciar e parar das classes de relógio.

```
class Controller
{
 public delegate void StartClocksDelegate();
 public delegate void StopClocksDelegate();
}
```

10. Adicione dois delegates públicos, chamados *StartClocks* e *StopClocks*, à classe *Controller*, por meio desses tipos delegate, como mostrado em negrito a seguir.

```
class Controller
{
 public delegate void StartClocksDelegate();
 public delegate void StopClocksDelegate();

 public StartClocksDelegate StartClocks;
 public StopClocksDelegate StopClocks;
}
```

11. Adicione o método *StartClocksRunning* à classe *Controller*. Esse método apenas chama o delegate *StartClocks*. Todos os métodos associados a esse delegate serão executados.

```
class Controller
{
 ...
 public void StartClocksRunning()
 {
 this.StartClocks();
 }
}
```

12. Adicione o método *StopClocksRunning* à classe *Controller*. Esse método é semelhante ao *StartClocksRunning*, exceto pelo fato de que ele chama o delegate *StopClocks*.

```
class Controller
{
 ...
 public void StopClocksRunning()
 {
 this.StopClocks();
 }
}
```

13. Retorne ao arquivo *ClockWindow.xaml.cs* na janela *Code and Text Editor*. Adicione uma variável privada *Controller*, chamada *controller*, à classe *ClockWindow* e instancie essa variável, como a seguir:

```
public partial class ClockWindow : Window
{
 ...
}
```

```
private Controller controller = new Controller();
```

```
}
```

14. No construtor *ClockWindow*, adicione as instruções mostradas em negrito a seguir. Essas instruções adicionam os métodos que iniciam e param os relógios aos delegates expostos pela classe *Controller*.

```
public ClockWindow()
{
 InitializeComponent();
 localClock = new LocalClock(localTimeDisplay);
 londonClock = new EuropeanClock(londonTimeDisplay);
 newYorkClock = new AmericanClock(newYorkTimeDisplay);
 tokyoClock = new JapaneseClock(tokyoTimeDisplay);

 controller.StartClocks += localClock.StartLocalClock;
 controller.StartClocks += londonClock.StartEuropeanClock;
 controller.StartClocks += newYorkClock.StartAmericanClock;
 controller.StartClocks += tokyoClock.StartJapaneseClock;

 controller.StopClocks += localClock.StopLocalClock;
 controller.StopClocks += londonClock.StopEuropeanClock;
 controller.StopClocks += newYorkClock.StopAmericanClock;
 controller.StopClocks += tokyoClock.StopJapaneseClock;
}
```

15. No método *startClick*, chame o delegate *StartClocks* do objeto *controller*, desabilite o botão *Start* e habilite o botão *Stop*, como a seguir:

```
private void startClick(object sender, RoutedEventArgs e)
{
 controller.StartClocks();
 start.IsEnabled = false;
 stop.IsEnabled = true;
}
```

Convém lembrar que, quando você chama um delegate, todos os métodos a ele associados são executados. Nesse caso, a chamada a *StartClocks* chamará o método que inicia cada relógio.

Vários controles WPF expõem a propriedade booleana *IsEnabled*. Por padrão, os controles são habilitados quando você os adiciona ao formulário. Isso significa que você pode clicar neles e eles farão alguma coisa. Entretanto, neste aplicativo, a propriedade *IsEnabled* do botão *Stop* está definida com *false* porque não faz sentido parar os relógios antes de iniciá-los. As duas últimas instruções nesse método desabilitam o botão *Start* e habilitam o botão *Stop*.

16. No método *stopClick*, chame o delegate *StopClocks* do objeto *controller*, habilite o botão *Start* e desabilite o botão *Stop*:

```
private void stopClick(object sender, RoutedEventArgs e)
{
 controller.StopClocks();
 start.IsEnabled = true;
 stop.IsEnabled = false;
}
```

17. No menu *Debug*, clique em *Start Without Debugging*.

18. No formulário WPF, clique em *Start*.

O formulário exibe as horas corretas e atualiza a cada segundo, como na imagem a seguir. (Estou no Reino Unido, de modo que minha hora local é a mesma de Londres.)



19. Clique em *Stop*.

A exibição para de atualizar os relógios.

20. Clique em *Start* novamente.

A exibição retoma o processamento, corrige a hora e atualiza a hora a cada segundo.

21. Feche o formulário e volte ao Visual Studio 2010.

## 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 abordagem 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 pelo 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 manipular os três métodos. Então, o que é possível fazer?

## Criando 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 invocado, ele chama *FinishFolding* que, por sua vez, chama o método *StopFolding*, passando no parâmetro 0.

**Nota** O método *FinishFolding* é um exemplo clássico de um adaptador; um método que converte (ou adapta) um método 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 Gamma, Helm, Johnson e Vlissides (Addison-Wesley Professional; 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, sendo pouco provável que seja chamado em outro momento. O C# fornece expressões lambda para situações como essa.

## Utilizando uma expressão lambda como um adaptador

Uma 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# realmente retorna um valor. Se você conhece linguagens de programação funcionais como Haskell, esse conceito deve ser fácil. Para os demais, não se assustem: expressões lambda não são particularmente complicadas e, depois de se acostumar com a sintaxe, você verá que são bem úteis.

Vimos no Capítulo 3, “Escrevendo métodos e aplicando escopo”, 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 método *StopFolding* da classe *FoldingMachine*, o problema é que esse método agora recebe um parâmetro, portanto, você precisa criar um adaptador que não receba parâmetro algum e que possa ser adicionado ao delegate *stopMachinery*. Você pode utilizar a seguinte instrução para fazer isso:

```
this.stopMachinery += (o => { folder.StopFolding(0); });
```

Todo o texto à direita do operador `+ =` é uma expressão lambda, que define o método a ser adicionado ao delegate `stopMachinery`. Ele 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.
- O operador `=>`, que indica ao compilador C# que isso é uma expressão lambda.
- O corpo do método. O exemplo mostrado aqui é muito simples, ele contém uma única instrução. 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.

Estritamente falando, o corpo de uma expressão lambda pode ser um corpo de método contendo múltiplas instruções ou ele pode ter uma única expressão. Se o corpo 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), desta maneira:

```
this.stopMachinery += () => folder.StopFolding();
```

Ao invocar o delegate `stopMachinery`, ele executará o código definido pela expressão lambda.

## A forma das expressões lambda

As expressões lambda podem assumir algumas formas sutilmente diferentes. Essas expressões eram originalmente parte de uma notação matemática chamada Cálculo Lambda que fornece uma notação para descrever funções (você pode pensar em 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 da expressão lambda disponíveis no C#:

```
x => x * x // Uma expressão simples que retorna o quadrado do seu parâmetro
// O tipo do parâmetro x é inferido do contexto.

x => { return x * x; } // Semanticamente, a mesma expressão anterior
// mas utilizando um bloco de instrução C# como
// um 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 aceita parâmetro algum.
// A expressão pode ou não
// retornar um valor.
```

```
(x, y) => { x++; return x / y; } // Múltiplos parâmetros; o compilador
 // infere os tipos dos parâmetros.
 // O parâmetro x é passado por valor, assim
 // o efeito da operação ++ é
 // local à expressão.

(ref int x, int y) { x++; return x / y; } // Múltiplos parâmetros
 // com tipos explícitos
 // O parâmetro x é passado por
 // referência, portanto, o efeito da
 // operação ++ é permanente.
```

Resumindo, seguem alguns recursos das expressões lambda que você precisa conhecer:

- Se uma expressão lambda receber parâmetros, você irá especificá-los nos parênteses à esquerda do operador `=>`. Você pode omitir os tipos dos parâmetros, e o compilador C# irá inferir 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 múltiplas 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!

Você aprenderá mais sobre expressões lambda e verá outros exemplos que recebem parâmetros e retornam valores em capítulos posteriores deste livro.

## Expressões lambda e métodos anônimos

As expressões lambda são um novo recurso adicionado à linguagem C# na versão 3.0. O C# 2.0 introduziu os métodos anônimos que podem realizar uma tarefa semelhante, mas que não são tão flexíveis. Os métodos anônimos foram adicionados principalmente para que você possa 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 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`. Todos os parâmetros necessários são especificados entre chaves seguindo a palavra-chave `delegate`. Por exemplo:

```
control.Add(delegate(int param1, string param2){ /* código que usa param1 e param2 */ ...});
```

Depois de se acostumar com elas, você notará que as expressões lambda fornecem uma sintaxe mais sucinta do que aquela dos métodos anônimos e permeiam muitos dos aspectos mais avançados do C#, como veremos mais adiante neste livro. Falando em termos gerais, você deve utilizar expressões lambda em vez de métodos anônimos no seu código.

## Ativando 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` você possa invocar qualquer método indiretamente, você ainda tem de invocar o `delegate` explicitamente. Em muitos casos, seria útil que o `delegate` executasse automaticamente quando algo significativo ocorresse. Por exemplo, no cenário da fábrica automatizada, pode ser vital conseguir invocar 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 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.

## Declarando um evento

Você declara um evento em uma classe projetada para atuar como uma origem de eventos. Uma origem de eventos (*event source*) é normalmente 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. Esses métodos são às vezes chamados de *subscribers* (*assinantes*), devendo ser preparados para manipular 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:

```
evento nomeDoTipoDoDelegate nomeDoEvento
```

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 pesquisas 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 invocará 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 próxima 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 em que o evento está baseado. Examinaremos esse aspecto dos eventos a seguir.

## Fazendo 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(); });
```

```
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.

## Cancelando 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. Essa ação costuma ser referida como “cancelar a inscrição a um evento”.

## Disparando um evento

Um evento pode ser disparado, exatamente como um delegate, chamando-o como um método. Quando você dispara um evento, 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();
 }
 }
 ...
}
```

A verificação *null* é necessária porque um campo de evento é implicitamente *null* e só se torna não *null* quando um método se inscreve nele utilizando o operador `+=`. Se tentar disparar um evento *null*, você obterá uma *NullReferenceException*. Se o delegate que define o evento espera algum parâmetro, os argumentos apropriados devem ser fornecidos quando você disparar o evento. Você verá alguns exemplos mais adiante.



**Importante** Os eventos têm um recurso de segurança embutido muito útil. Um evento público (como *MachineOverheating*) só pode ser disparado por métodos da classe que o define (a classe *TemperatureMonitor*). Qualquer tentativa de disparar o método fora da classe resulta em um erro de compilação.

## Entendendo eventos de interface WPF

Como mencionado anteriormente, as classes e os controles do .NET Framework utilizados para construir interfaces gráficas do usuário (*graphical user interfaces* – GUIs) empregam eventos extensamente. Você verá e usará os eventos GUI em muitas ocasiões na segunda metade deste livro. Por exemplo, a classe WPF *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 a:

```
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 (a maneira como isso acontece está além do escopo deste livro). Essa disposição 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*:

```
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);
 ...
 }
 ...
}
```

Normalmente você não vê esse código. Quando utiliza a janela *Design View* no Visual Studio 2010 e configura a propriedade *Click* do botão *okay* como *okayClick* na descrição do formulário da Extensible Application Markup Language (XAML), o Visual Studio 2010 gera esse código para você. Tudo o que

precisa fazer é escrever a lógica do seu aplicativo no método de tratamento de evento, *okayClick*, na parte do código à qual você tem acesso, nesse 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 que os vários controles GUI geram 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 múltiplos eventos. O método 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.

Você aprenderá mais sobre como tratar eventos para controles WPF no Capítulo 22, "Apresentando o Windows Presentation Foundation".

## Utilizando eventos

No exercício anterior, você concluiu um aplicativo World Clock, que exibe a hora local, assim como a hora em Londres, Nova York e Tóquio. Você utilizou delegates para iniciar e parar os relógios. Você ainda deve se lembrar de que cada relógio tinha um construtor que esperava o nome do campo existente no formulário no qual a hora deveria ser exibida. Entretanto, um relógio deve realmente se concentrar em ser um relógio e não se preocupar, necessariamente, com o modo de exibição da hora – é melhor deixar a lógica do próprio formulário WPF se encarregar dessa funcionalidade. Neste exercício, você modificará o relógio local para acionar um evento a cada segundo. Você fará uma inscrição nesse evento, no formulário WPF, e chamará um manipulador de eventos que exibe a nova hora local.

### Modifique o aplicativo World Clock para utilizar eventos

1. Retorne à janela do Visual Studio 2010 que exibe o projeto Clock.
2. Exiba o arquivo LocalClock.cs na janela *Code and Text Editor*.

Esse arquivo contém a classe *LocalClock* que implementa o relógio local. Incluímos a seguir os principais elementos dessa classe:

- ❑ Um objeto *DispatcherTimer*, chamado *ticker*. A classe *DispatcherTimer* é fornecida como parte do .NET Framework, e seu objetivo é acionar eventos, em intervalos de tempo especificados.
- ❑ Um campo *TextBox*, chamado *display*. O construtor inicializa esse campo com o objeto *TextBox* passado como parâmetro. A classe *LocalClock* define a propriedade *Text* desse objeto para exibir a hora, no método *RefreshTime*.
- ❑ Um objeto *TimeZoneInfo*, chamado *timeZoneForThisClock*. A classe *TimeZoneInfo* também faz parte do .NET Framework. Essa classe é utilizada para obter a hora em um fuso horário específico. O construtor inicializa esse objeto com *TimeZone.Local*, que é o fuso horário local do computador que executa o aplicativo.
- ❑ O método *StartLocalClock*, que inicia o funcionamento do *DispatcherTimer*. A classe *DispatcherTimer* fornece o evento *Tick*, utilizado para especificar um método a ser executado sempre que ocorrer um evento *Tick*, e a propriedade *Interval*, usada para especificar a frequência com que ocorrem os eventos *Tick*. O código da classe *LocalClock* aciona um evento a cada segundo. Na realidade, o método *Start* da classe *DispatcherTimer* inicia o funcionamento do cronômetro. No exercício anterior, você chamou esse método por meio do delegate *StartClocksDelegate* na classe *Controller*.
- ❑ O método *StopLocalClock*, que chama o método *Stop* do objeto *DispatcherTimer*. Esse método interrompe o funcionamento do cronômetro, e ele não acionará quaisquer outros eventos antes de você chamar o método *Start* novamente. No exercício anterior, você chamou esse método por meio do delegate *StopClocksDelegate* na classe *Controller*.
- ❑ O método *OnTimedEvent*. O método *StartLocalClock* adiciona esse método ao evento *Tick* do objeto *DispatcherTimer*, e, quando ocorre um evento *Tick*, esse método é executado. Os parâmetros para esse método são exigidos pela definição do delegate utilizado pelo evento *Tick*, mas eles não são usados neste exemplo, de modo que você pode ignorá-los. Esse método recupera a data e a hora atuais, por meio da propriedade estática *Now* da classe *DateTime*. Em seguida, ele converte a hora recuperada na hora local, ao utilizar o método *TimeZoneInfo.ConvertTime*. As horas, os minutos e os segundos são extraídos da hora e passados como parâmetros para o método *RefreshTime*.

 **Nota** Na realidade, não é necessário converter do valor retornado por *DateTime.Now* para a hora local, porque o valor de *DateTime.Now* é expresso como hora local, por padrão. Entretanto, essa é uma prática recomendável, e você pode converter o valor de *DateTime.Now* em hora, em qualquer fuso horário, ao aplicar essa técnica – basta especificar o fuso horário de destino como o segundo parâmetro do método *TimeZoneInfo.ConvertTime*. É exatamente isso que as classes *AmericanClock*, *EuropeanClock* e *JapaneseClock* fazem.

- O método *RefreshTime*, que formata as horas, os minutos e segundos, passados como parâmetros em uma string, e depois exibe essa string na *TextBox* referenciada pelo campo *display*.
3. O objetivo deste exercício é retirar da classe *LocalClock* a responsabilidade pela exibição da hora, de modo que você deve transformar em comentário a definição do campo *display*:

```
class LocalClock
{

 // private TextBox display = null;

}
```

4. Remova o parâmetro do construtor *LocalClock* e comente a instrução que define o campo *display* com esse parâmetro. O construtor corrigido ficará parecido com o seguinte:

```
public LocalClock()
{
 this.timeZoneForThisClock = TimeZoneInfo.Local;
 // this.display = displayBox;
}
```

5. Adicione um delegate público, chamado *DisplayTime*, à classe *LocalClock*, antes do construtor. Esse delegate deve especificar um método que aceita um parâmetro de string e retorna um *void*. O formulário WPF fornecerá um método que combina com esse delegate. Esse método atualizará a hora exibida no formulário com a string passada como parâmetro.

```
class LocalClock
{

 public delegate void DisplayTime(string time);

}
```

6. Adicione um evento público, chamado *LocalClockTick*, à classe *LocalClock*, depois do delegate *DisplayTime*. Esse evento deve se basear no delegate *DisplayTime*.

```
class LocalClock
{

 public delegate void DisplayTime(string time);
 public event DisplayTime LocalClockTick;

}
```

7. Localize o método *RefreshTime* no final da classe *LocalClock*. Esse método define atualmente a propriedade *Text* do campo *display* com uma string formatada que contém a hora atual. Modifique esse método de modo que ele acione, em substituição, o evento *LocalClockTick*, e passe a string formatada como parâmetro para quaisquer métodos que se inscrevam nesse evento.

```
private void RefreshTime(int hh, int mm, int ss)
{
 if (this.LocalClockTick != null)
 {
 this.LocalClockTick(String.Format("{0:D2}:{1:D2}:{2:D2}", hh, mm, ss));
 }
}
```



**Nota** Neste exemplo, a string de formatação especifica que cada dígito deve ser exibido como um valor decimal de dois dígitos, com um zero à esquerda, se necessário.

- Exiba o arquivo ClockWindow.xaml.cs na janela *Code and Text Editor*. No construtor *ClockWindow*, modifique a instrução que instancia a variável *localClock* e remova o parâmetro da chamada ao construtor.

```
public ClockWindow()
{
 ...
LocalClock = new LocalClock();
 ...
}
```

- No menu *Debug*, clique em *Start Without Debugging*. Quando o formulário WPF aparecer, clique em *Start*. Você deve ver os relógios de Londres, Nova York e Tóquio funcionando como anteriormente, mas a exibição da hora local continua parada em 00:00:00. Isso ocorre porque, embora o objeto *LocalClock* acione eventos a cada segundo, você ainda não se inscreveu nesses eventos. Feche o formulário WPF e retorne ao Visual Studio.

- No arquivo *ClockWindow.xaml.cs*, adicione o seguinte método ao final da classe *ClockWindow*:

```
private void displayLocalTime(string time)
{
 localTimeDisplay.Text = time;
}
```

Esse método exibe a string passada como parâmetro no *TextBox localTimeDisplay*, no formulário.

- No método *startClick*, adicione a instrução mostrada em negrito a seguir, que faz uma inscrição do método *displayLocalTime* no evento *LocalClockTick* do objeto *localClock*:

```
private void startClick(object sender, RoutedEventArgs e)
{
 controller.StartClocks();
LocalClock.LocalClockTick += displayLocalTime;
 start.IsEnabled = false;
 stop.IsEnabled = true;
}
```

12. No método *stopClick*, cancele a inscrição do método *displayLocalTime* no evento *LocalClickTick*.

```
private void stopClick(object sender, RoutedEventArgs e)
{
 controller.StopClocks();
 localClock.LocalClockTick -= displayLocalTime;
 start.IsEnabled = true;
 stop.IsEnabled = false;
}
```

13. No menu *Debug*, clique em *Start Without Debugging*.  
14. Clique em *Start*. Dessa vez, o relógio local exibe a hora correta e é atualizado a cada segundo.  
15. Clique em *Stop* e verifique se o relógio local é interrompido. Em seguida, feche o formulário e retorne ao Visual Studio 2010.

Neste exercício, você atualizou o relógio local para sinalizar para o formulário que ele deve atualizar sua exibição, ao utilizar eventos, mas que os outros relógios devem continuar exibindo a hora. Quando tiver um tempo, modifique as demais classes de relógio da mesma maneira.

Neste capítulo, você aprendeu a utilizar delegates para fazer referência a métodos e chamar esses métodos. Você viu como definir métodos anônimos e expressões lambda que podem ser executados por meio de um delegate. Por último, você aprendeu a definir e utilizar eventos para acionar a execução de um método.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 18.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 17

| 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>                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Criar uma instância de um delegate inicializando com um único método específico | <p>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:</p> <pre>delegate void myDelegate(); private void myMethod() { ... }  myDelegate del = new myDelegate(this.myMethod);</pre>                                                                                                                                                                                         |
| Invocar um delegate                                                             | <p>Utilize a mesma sintaxe que uma chamada de método. Por exemplo:</p> <pre>myDelegate del; ***  del();</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Declarar um evento                                                              | <p>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:</p> <pre>delegate void myEvent();</pre> <pre>class MyClass {     public event myDelegate MyEvent; }</pre>                                                                                                                                                                                                                                                                                                                                                                               |
| Fazer a inscrição a um evento                                                   | <p>Crie uma instância do delegate (do mesmo tipo do evento) e vincule-a ao evento utilizando o operador <i>+=</i>. Por exemplo:</p> <pre>class MyEventHandlingClass {     private MyClass myClass = new MyClass();      public void Start()     {         myClass.MyEvent += new myDelegate             (this.eventHandlingMethod());     }      private void eventHandlingMethod()     {         ***     } }</pre> <p>Você também pode fazer o compilador gerar automaticamente o novo delegate simplesmente especificando o método inscrito:</p> <pre>public void Start() {     myClass.MyEvent += this.eventHandlingMethod(); }</pre> |

(continua)

| 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);     }     ... }</pre> <p>Ou:</p> <pre>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 <code>null</code>. Por exemplo:</p> <pre>class MyClass {     public event myDelegate MyEvent;     ...     private void RaiseEvent()     {         if (this.MyEvent != null)         {             this.MyEvent();         }     }     ... }</pre>                                   |

# Capítulo 18

# Apresentando genéricos

Neste capítulo, você vai aprender a:

- 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 independente do tipo de dados em que ele opera.

No Capítulo 8, “Entendendo valores e referências”, você aprendeu a empregar o tipo *object* para referenciar uma instância de qualquer classe. Você pode utilizar o tipo *object* para armazenar um valor de qualquer tipo e também definir parâmetros através do tipo *object*, quando precisar passar valores de qualquer tipo em um método. Um método também pode retornar valores de qualquer tipo especificando *object* como o tipo de retorno. Embora seja muito flexível, essa prática deixa ao programador a responsabilidade de lembrar que tipos de dados estão realmente sendo utilizados, e podem ocorrer erros de tempo de execução se o programador cometer um erro. Neste capítulo, você vai aprender sobre os genéricos, um recurso projetado para ajudá-lo a evitar esse tipo de erro.

## O problema com *objects*

Para entender os genéricos, vale a pena examinar detalhadamente os problemas que eles foram projetados para resolver, especificamente ao empregar o tipo *object*.

Você pode utilizar o tipo *object* para referenciar a um valor ou a uma variável de qualquer tipo. Todos os tipos-referência herdam automaticamente (direta ou indiretamente) da classe *System.Object* no Microsoft .NET Framework. Você pode utilizar essas informações para criar classes e métodos altamente generalizados. Por exemplo, muitas das classes no namespace *System.Collections* exploram esse fato, portanto, você pode criar coleções que armazenam praticamente qualquer tipo de dados. (Já vimos classes de coleção no Capítulo 10, “Utilizando arrays e coleções”.) Examinando uma classe de coleção específica, como um exemplo detalhado, você também perceberá na classe *System.Collections.Queue* que é possível criar filas contendo praticamente qualquer coisa. O exemplo de código a seguir mostra como criar e manipular uma fila de objetos *Circle*:

```
using System.Collections;

Queue myQueue = new Queue();
Circle myCircle = new Circle();
myQueue.Enqueue(myCircle);

myCircle = (Circle)myQueue.Dequeue();
```

O método *Enqueue* adiciona um *object* ao início de uma fila e o método *Dequeue* remove o *object* na outra extremidade da fila. Esses métodos são definidos assim:

```
public void Enqueue(object item);
public object Dequeue();
```

Como os métodos *Enqueue* e *Dequeue* manipulam *objects*, você pode operar em filas de *Circles*, *PhoneBooks*, *Clocks* 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 de objeto. Se não fizer o casting do valor retornado, você receberá o erro de compilador “Cannot implicitly convert type ‘object’ to ‘Circle’.” [“Não foi possível converter implicitamente o tipo ‘object’ para ‘Circle’.”]

Essa necessidade de fazer um casting explícito inibe grande parte da flexibilidade propiciada pelo tipo *object*. É muito fácil escrever um código como este:

```
Queue myQueue = new Queue();
Circle myCircle = new Circle();
myQueue.Enqueue(myCircle);

...
Clock myClock = (Clock)myQueue.Dequeue(); // erro de tempo de execução
```

Embora esse código seja compilado, ele não é válido e gera uma *System.InvalidCastException* em tempo de execução. O erro é causado por tentar armazenar uma referência a um *Circle* em uma variável *Clock* 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 abordagem *object* para criar classes e métodos generalizados é que ele pode utilizar 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 variáveis *int*:

```
Queue myQueue = new Queue();
int myInt = 99;
myQueue.Enqueue(myInt); // faz o boxing do int para um objeto

...
myInt = (int)myQueue.Dequeue(); // faz o unboxing do objeto para um int
```

O tipo de dados *Queue* espera que os itens que ele armazena sejam tipos-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 necessário e facilitar a criação de classes e métodos generalizados. Classes e métodos genéricos aceitam *parâmetros de tipo*, que especificam o tipo dos objetos em que eles operam. A biblioteca de classes do .NET Framework inclui versões genéricas de boa parte das classes de coleção e interfaces no namespace *System.Collections.Generic*. O exemplo de código a seguir mostra como utilizar a classe *Queue* genérica, encontrada nesse namespace, para criar uma fila de objetos *Circle*:

```
using System.Collections.Generic;

Queue<Circle> myQueue = new Queue<Circle>();
Circle myCircle = new Circle();
myQueue.Enqueue(myCircle);

myCircle = myQueue.Dequeue();
```

Há duas coisas novas a notar no código do exemplo anterior:

- O uso do parâmetro de tipo entre os colchetes angulares, *< Circle >*, na declaração da variável *myQueue*.
- A falta de um casting na execução do método *Dequeue*.

O parâmetro de tipo entre os colchetes angulares especifica o tipo de objetos aceitos pela fila. Todas as referências aos métodos nessa fila vão utilizar automaticamente esse tipo em vez de um *object*, tornando desnecessário o casting para o tipo *Circle*, ao invocar o método *Dequeue*. O compilador verificará se os tipos não foram misturados acidentalmente e gerará um erro de tempo de compilação, em vez de um de tempo de execução, se você tentar remover um item de *circleQueue* em um objeto *Clock*, por exemplo.

Se examinar a descrição da classe *Queue* genérica, na documentação do Microsoft Visual Studio 2010, você notará que ela está definida como:

```
public class Queue<T> : ...
```

O *T* identifica o parâmetro do tipo e 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* no exemplo anterior). Além disso, se então examinar os métodos da classe *Queue<T>*, você observará que alguns deles, como *Enqueue* e *Dequeue*, especificam *T* como um parâmetro de tipo ou um valor de retorno:

```
public void Enqueue(T item);
public T Dequeue();
```

O parâmetro de tipo, *T*, é substituído pelo tipo que você especifica ao declarar a fila. Além disso, o compilador agora tem informações suficientes para executar uma verificação de tipo estrita quando você compilar o aplicativo e pode interceptar todos os erros de não correspondência de tipo antecipadamente.

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 executa 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>();
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

Os dois primeiros exemplos criam filas de tipos-valor, enquanto o terceiro cria uma fila de filas (de *ints*). Por exemplo, para a variável *intQueue*, o compilador também gera as seguintes versões dos métodos *Enqueue* e *Dequeue*:

```
public void Enqueue(int item);
public int Dequeue();
```

Compare essas definições com aquelas da classe *Queue* não genérica mostradas 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.

Uma classe genérica também pode ter múltiplos parâmetros de tipo. Por exemplo, a classe genérica *System.Collections.Generic.Dictionary* espera dois parâmetros de tipo: um tipo para as chaves e outro para os valores. A definição a seguir mostra como especificar múltiplos parâmetros de tipo:

```
public class Dictionary<TKey, TValue>
```

Um dicionário fornece uma coleção de pares chave/valor. Você armazena os valores (tipo *TValue*) com uma chave associada (tipo *TKey*) e então os recupera especificando a chave a ser pesquisada. A classe *Dictionary* fornece um indexador que permite acessar os itens utilizando uma notação de array. Ela é definida da seguinte forma:

```
public virtual TValue this[TKey key] { get; set; }
```

Observe que o indexador acessa os valores do tipo *TValue* por meio de uma chave do tipo *TKey*. Para criar e utilizar um dicionário chamado *directory* contendo os valores *Person* identificados por chaves *string*, use o código a seguir:

```
struct Person
{
 ...
}

Dictionary<string, Person> directory = new Dictionary<string, Person>();
Person john = new Person();
directory["John"] = john;
...
Person author = directory["John"];
```

Como na classe genérica *Queue*, o compilador detecta tentativas de armazenar valores diferentes das estruturas *Person* no diretório, assim como garante que a chave seja sempre um valor *string*. Para obter mais informações sobre a classe *Dictionary*, leia a documentação do Visual Studio 2010.

 **Nota** Você também pode definir estruturas e interfaces genéricas utilizando a mesma sintaxe 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 via *casting* em tipos diferentes. Por exemplo, a classe *System.Collections.Queue* 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 você usa instâncias da mesma classe e precisa fazer um casting dos dados utilizados para e do tipo *object*.

Compare essa classe com a classe *System.Collections.Generic.Queue<T>*. Sempre que utiliza essa classe com um parâmetro de tipo (como *Queue<int>* ou *Queue<string>*), na realidade você faz o compilador gerar uma classe totalmente nova que tem funcionalidades definidas pela classe genérica. Isso significa que uma *Queue<int>* é um tipo totalmente diferente de um *Queue<string>*, 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<string>*, 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*, você pode querer garantir que todos os objetos armazenados na classe tenham um método *Print*. É possível especificar essa condição utilizando uma *restrição*.

Utilizando 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* realmente implementa a interface *IPrintable* e, caso isso não aconteça, terminará com um erro de compilação.

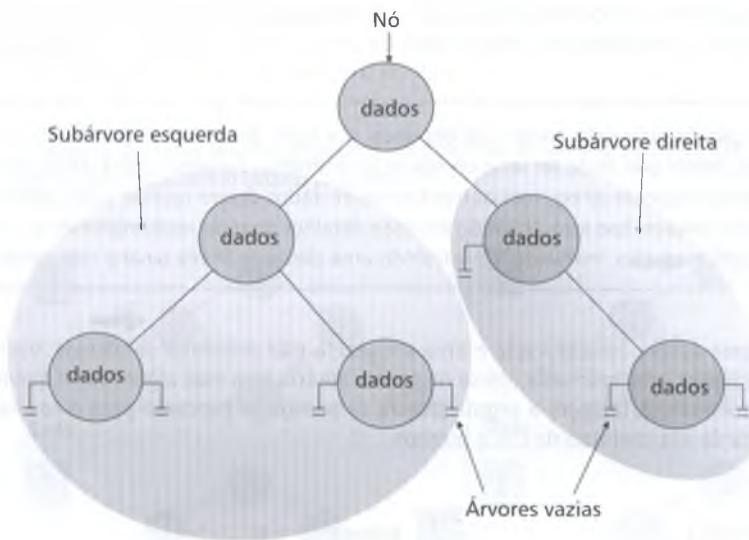
## Criando uma classe genérica

A 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 de você fazer isso, forneço 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. Esse é um exercício prático porque, casualmente, essa classe está faltando no namespace *System.Collections.Generic*. Uma árvore binária é uma estrutura de dados útil utilizada em várias operações, incluindo classificar e pesquisar dados de forma muito rápida. Há livros inteiros escritos sobre as minúcias das árvores binárias, mas abordá-las em detalhe não é a finalidade deste livro. Em vez disso, vamos examinar apenas os detalhes pertinentes. Se estiver interessado, consulte um livro como *The Art of Computer Programming, Volume 3: Sorting and Searching*, de Donald E. Knuth (AddisonWesley Professional; 2<sup>a</sup> edição, 1998).

Uma árvore binária é uma estrutura de dados recursiva (de autorreferenciação) que pode estar vazia ou conter três elementos: um dado, que é em geral 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 percorre-lá para visitar cada nó em uma sequência ordenada. O algoritmo para inserir um item em uma árvore binária ordenada é mostrado a seguir:

**Se a árvore  $T$  está vazia**

**Então**

Construa uma nova árvore  $T$ , com o novo item  $I$  como o nó, e subárvores esquerda e direita vazias

**Senão**

Examine o valor do nó atual,  $N$ , da árvore  $T$

Se o valor de  $N$  for maior do que o do novo item  $I$

**Então**

Se a subárvore esquerda de  $T$  está vazia

**Então**

Construa uma nova subárvore à esquerda de  $T$ , com o novo item  $I$  como o nó, e subárvores esquerda e direita vazias

**Senão**

Insira  $I$  na subárvore esquerda de  $T$

**Fim\_Se**

**Senão**

Se a subárvore direita de  $T$  está vazia

**Então**

Construa uma nova subárvore à direita de  $T$ , com o novo item  $I$  como o nó, e subárvores esquerda e direita vazias

**Senão**

Insira  $I$  na subárvore direita de  $T$

**Fim\_Se**

**Fim\_Se**

**Fim\_Sr**

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 outras formas de dados devem receber suas próprias maneiras de comparar valores. Isso será discutido em mais detalhes quando você implementar uma árvore binária na próxima seção, intitulada "Construindo 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.

**1** Dados: 1, 5, -2, 1, 6

Árvore: (Vazia)



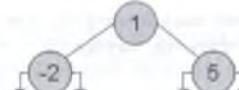
**2** Dados: 5, -2, 1, 6



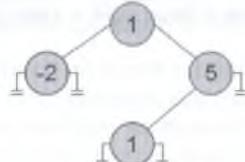
**3** Dados: -2, 1, 6



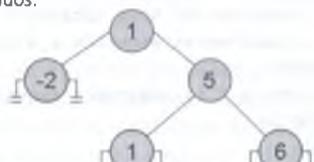
**4** Dados: 1, 6



**5** Dados: 6



**6** Dados:



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

    Exibir o conteúdo da subárvore esquerda

Fim\_Se

    Exibir o valor do nó atual

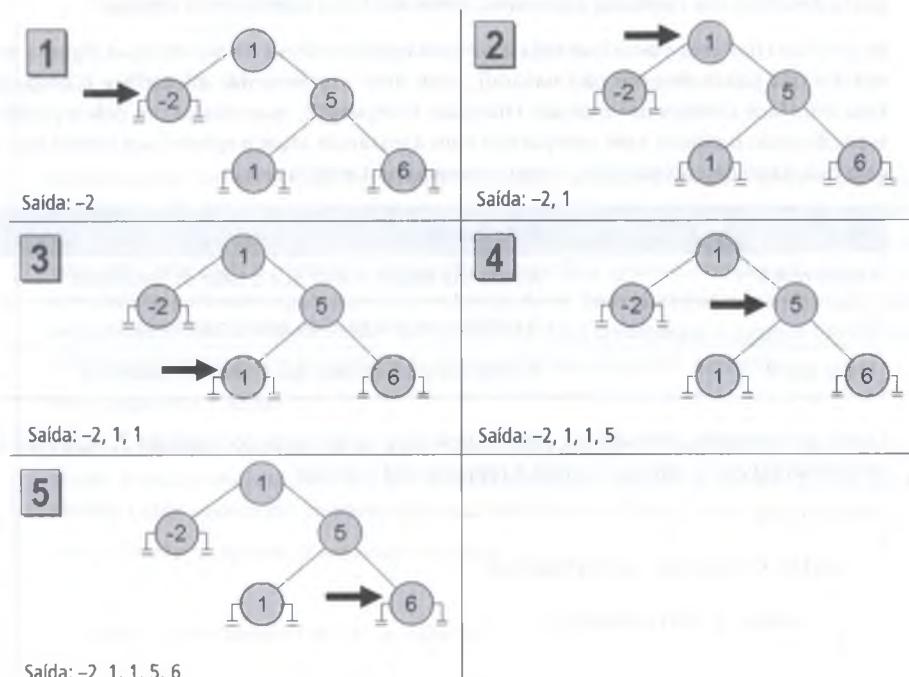
Se a árvore direita não está vazia

Então

Exibir 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.



## Construindo 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 é uma classe que pode ser útil em várias aplicações diferentes. Então você a implementará como uma biblioteca de classes, em vez de um aplicativo independente. Você pode então 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 delegados) armazenadas em um assembly. Um 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 com as 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 nos capítulos anteriores, como você vais comparar 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 um exemplo, considere a classe *Circle* que foi descrita no Capítulo 7, “Criando e gerenciando classes e objetos”, e que é reproduzida a seguir:

```
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*. Neste 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* requer o uso de um casting para conseguir 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 os seguintes métodos:

```
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* e *Equals* 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. Inicie o Visual Studio 2010 se ele ainda não estiver em execução.
2. Se você estiver utilizando o Visual Studio 2010 Standard ou o Visual Studio 2010 Professional, siga estes passos para criar um novo projeto de biblioteca de classe.
  - 2.1. No menu *File*, aponte para *New* e então clique em *Project*.
  - 2.2. Na caixa de diálogo *New Project*, no painel central, selecione o template *Class Library*.
  - 2.3. Na caixa de texto *Name*, digite **BinaryTree**.
  - 2.4. Na caixa de texto *Location*, especifique *Microsoft Press\Visual CSharp Step By Step\Chapter 18* na sua pasta Documentos.
  - 2.5. Clique em *OK*.
3. Se você estiver utilizando o Microsoft Visual C# 2010 Express, siga estes passos para criar um novo projeto de biblioteca de classe:
  - 3.1. No menu *Tools*, clique em *Options*.
  - 3.2. Na caixa de diálogo *Options*, marque a caixa de seleção *Show all settings*.
  - 3.3. Clique em *Projects and Solutions* na visualização de árvore, no painel esquerdo.
  - 3.4. No painel direito, na caixa de texto *Visual Studio projects location*, especifique a localização como a pasta *Microsoft Press\Visual CSharp Step By Step\Chapter 18* na sua pasta Documentos.
  - 3.5. Clique em *OK*.
  - 3.6. No menu *File*, clique em *New Project*.
  - 3.7. Na caixa de diálogo *New Project*, clique no ícone *Class Library*.
  - 3.8. No campo *Name*, digite **BinaryTree**.
  - 3.9. Clique em *OK*.
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 na classe *Tree<TItem>*, como mostrado a seguir em negrito, para especificar que o parâmetro de tipo *TItem* deve denotar um tipo que implementa a interface genérica *IComparable<TItem>*.

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 no texto em negrito 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 neste código. Esse método insere um valor *TItem* na árvore.

A definição do método deve ser semelhante a este:

```
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. A parte do algoritmo depois da verificação se a árvore está vazia é reproduzida 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, T

 Se o valor de N for maior do que aquele do novo item, I

 Então

 Se a subárvore esquerda de T estiver vazia

 Então

 Construir uma nova subárvore esquerda de T com o novo item I como o nó,

 e subárvore esquerda e direita vazias

 Senão

 Inserir I na subárvore esquerda de T

 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 a seguir:

```
public void Insert(TItem newItem)
{
 TItem currentNodeValue = this.NodeData;
}
```

11. Adicione a seguinte instrução *if-else*, mostrada em negrito, ao método *Insert* 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 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. Substitua o comentário `//Inserir o novo item na subárvore esquerda` pelo seguinte bloco de código:

```
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. Substitua o comentário `//Inserir o novo item na subárvore direita` pelo 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 imprimindo seu valor.

A definição do método deve ser esta:

```
public void WalkTree()
{
}
```

15. Adicione as instruções seguintes ao método `WalkTree`. Estas instruções implementam o algoritmo descrito anteriormente para imprimir o conteúdo de uma árvore binária:

```
if (this.LeftTree != null)
{
 this.LeftTree.WalkTree();
}

Console.WriteLine(this.NodeData.ToString());

if (this.RightTree != null)
{
 this.RightTree.WalkTree();
}
```

16. No menu *Build*, clique em *Build Solution*. A classe deve ser compilada inteiramente, mas corrija todos os erros que forem reportados e recompile a solução, se necessário.



**Nota** Se você estiver utilizando o Visual C# 2010 Express e o menu *Build* não estiver visível, no menu *Tools*, clique em *Settings* e, em seguida, em *Expert Settings*.

17. Se você estiver utilizando o Visual C# 2010 Express, no menu *File*, clique em *Save All*. Se a caixa de diálogo *Save Project* aparecer, clique em *Save*.

No próximo exercício, você testará a classe *Tree<TItem>* criando árvores binárias de inteiros e strings.

### Teste a classe *Tree<TItem>*

1. 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*, em vez de no projeto de *BinaryTree*.

2. Adicione um novo projeto utilizando o template *Console Application*. Nomeie o projeto como *BinaryTreeTest*. Configure *Location* como *Microsoft Press\Visual CSharp Step By Step\Chapter 18* na sua pasta Documentos e então clique em *OK*.



**Nota** Lembre-se de que uma solução do Visual Studio 2010 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>*. Essa é a maneira recomendada de testar as bibliotecas de classe.

3. Assegure-se de que o projeto *BinaryTreeTest* está selecionado no *Solution Explorer*. No menu *Project*, 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.

4. Certifique-se de que o projeto *BinaryTreeTest* ainda está selecionado no *Solution Explorer*. No menu *Project*, clique em *Add Reference*. Na caixa de diálogo *Add Reference*, clique na guia *Projects*. Clique no projeto *BinaryTree* e, então, em *OK*.

O assembly *BinaryTree* aparece na lista de referências do projeto *BinaryTreeTest* no *Solution Explorer*. 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. Pode-se fazer isso selecionando o assembly na guia *Browse* da caixa de diálogo *Add Reference*. Você utilizará essa técnica no conjunto final de exercícios deste capítulo.

5. Na janela *Code and Text Editor* que exibe a classe *Program*, adicione a seguinte diretiva *using* à lista, na parte superior da classe:

```
using BinaryTree;
```

6. Adicione as instruções em negrito no código ao método *Main* 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);
 tree1.WalkTree();
}
```

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* imprime o conteúdo da árvore, que deve aparecer em ordem crescente.



**Nota** Lembre-se de que a palavra-chave *int* no C# é, na verdade, 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>*, o que é a razão por que você pode criar objetos *Tree<int>*. Da mesma forma, a palavra-chave *string* é um alias para *System.String*, que também implementa *IComparable* and *IComparable<T>*.

7. No menu *Build*, clique em *Build Solution*. Verifique se a solução é compilada, corrigindo todos os erros, se necessário.
8. Salve o projeto e então, no menu *Debug*, clique em *Start Without Debugging*.

O programa executa e exibe os valores na seguinte sequência:

-12, -8, 0, 5, 5, 8, 8, 10, 10, 11, 14, 15

9. Pressione a tecla Enter para retornar ao Visual Studio 2010.
10. 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("!");
 tree2.WalkTree();
}
```

Essas instruções criam outra árvore binária para armazenar *strings*, preenchem-na com alguns dados de teste e, então, a imprimem. Desta vez, os dados são ordenados alfabeticamente.

11. No menu *Build*, clique em *Build Solution*. Verifique se a solução é compilada, corrigindo todos os erros, se necessário.
12. No menu *Debug*, clique em *Start Without Debugging*.

O programa executa e exibe os valores inteiros como anteriormente, seguidos pelas strings na seguinte sequência:

!, Are, Are, Feeling, Hello, Hope, How, I, Today, Well, World, You, You

13. Pressione a tecla Enter para retornar ao Visual Studio 2010.

## Criando um método genérico

Além de definir classes genéricas, você também pode utilizar o .NET Framework para criar métodos genéricos.

Com um método genérico, você pode especificar os 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 utilizados frequente-

mente junto com as classes genéricas – você precisa delas para os métodos que recebem uma classe genérica como parâmetro ou que têm um tipo de retorno que é uma classe genérica.

Você define métodos genéricos utilizando a mesma sintaxe de parâmetro de tipo utilizada para criar classes genéricas (você também pode especificar restrições). Por exemplo, você pode chamar o seguinte método *Swap*<T> genérico para trocar 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 invoca o método especificando o tipo apropriado para seu parâmetro de tipo. Os exemplos a seguir mostram como invocar 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 método genérico, portanto, mostram o mesmo comportamento, embora sobre tipos diferentes.

## Definindo um método genérico para criar uma árvore binária

O exercício anterior mostrou como criar 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*

- Utilizando o Visual Studio 2010, crie um novo projeto por meio do template Console Application. Na caixa de diálogo *New Project*, nomeie o projeto **BuildTree**. Se você estiver utilizando o Visual Studio 2010 Standard ou Visual Studio 2010 Professional, configure *Location* como *Microsoft Press\Visual CSharp Step By Step\Chapter 18* na sua pasta Documentos e selecione *Create a new Solution* na lista suspensa *Solution*. Clique em *OK*.

2. No menu *Project*, clique em *Add Reference*. Na caixa de diálogo *Add Reference* clique na guia *Browse*. Navegue para a pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 18\BinaryTree\BinaryTree\bin\Debug*, clique em *BinaryTree.dll* e então clique em *OK*.

A assembly *BinaryTree* é adicionada à lista de referências mostradas no *Solution Explorer*.

3. 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;
```

Esse namespace contém a classe *Tree<TItem>*.

4. Adicione um método chamado *InsertIntoTree* à classe *Program* após o método *Main*. Esse método deve ser um método *static* que aceita uma variável *Tree<TItem>* e um array *params* dos elementos *TItems* chamados *data*.

A definição do método deve ser esta:

```
static void InsertIntoTree<TItem>(Tree<TItem> tree, params TItem[] data)
{
}
```



**Dica** Uma maneira alternativa de implementar esse método é criar um método de extensão da classe *Tree<TItem>* prefixando o parâmetro *Tree<TItem>* com a palavra-chave *this* e definindo o método *InsertIntoTree* em uma classe estática, desta maneira:

```
public static class TreeMethods
{
 public static void InsertIntoTree<TItem>(this Tree<TItem> tree,
 params TItem[] data)
 {
 ...
 }
}
```

A principal vantagem dessa abordagem é que você pode invocar o método *InsertIntoTree* diretamente em um objeto *Tree<TItem>* em vez de passar *Tree<TItem>* como um parâmetro. Mas, neste exercício, ficaremos com o mais simples.

5. 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* apropriada, como mostrado em negrito no código a seguir:

```
static void InsertIntoTree<TItem>(Tree<TItem> tree, params TItem[] data) where TItem : IComparable<TItem>
{
}
```

6. Adicione as seguintes instruções mostradas em negrito ao método *InsertIntoTree*. Essas instruções verificam se o usuário realmente passou alguns parâmetros para o método (o array *data* poderia estar vazio) e então iteram pela lista *params*, adicionando cada item à árvore com o método *Insert*. A árvore é passada de volta como o valor de retorno:

```
static void InsertIntoTree<TItem>(Tree<TItem> tree, params TItem[] data) where TItem : IComparable<TItem>
{
 if (data.Length == 0)
 throw new ArgumentException("Must provide at least one data value");

 foreach (TItem datum in data)
 {
 tree.Insert(datum);
 }
}
```

### Teste o método *InsertIntoTree*

1. No método *Main* da classe *Program*, adicione as instruções a seguir mostradas em negrito que criam uma nova *Tree* para armazenar os dados de caracteres, preenchem-na com alguns dados de exemplo utilizando o método *BuildTree* e então a exibem utilizando o método *WalkTree* de *Tree*:

```
static void Main(string[] args)
{
 Tree<char> charTree = new Tree<char>('M');
 InsertIntoTree<char>(charTree, 'X', 'A', 'M', 'Z', 'Z', 'N');
 charTree.WalkTree();
}
```

2. No menu *Build*, clique em *Build Solution*. Verifique se a solução é compilada, corrigindo todos os erros, 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 2010.

## Variância e interfaces genéricas

No Capítulo 8, você aprendeu 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 pela classe *Wrapper*<*T*> 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 gera uma instância do tipo *Wrapper*<*string*>, que 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".

Examine agora a 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 anteriormente); 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 o seguinte:

```
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<B>*, 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 a seguir, no lugar da *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());
```

Agora, o seguinte código 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 *IWrapper*<*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 rigorosa de tipos. 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*. O código a seguir já 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 quaisquer parâmetros 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. 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*>, que você conhecerá no Capítulo 19, “Enumerando coleções”.

## Interfaces contravariantes

A contravariância é a consequência da covariância, e 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, de modo que compensa examinar um exemplo da biblioteca de classes do .NET Framework.

O namespace *System.Collections.Generic* do .NET Framework dispõe de uma interface chamada *IComparer*:

```
public interface IComparer<in T>
{
 int Compare(T x, T y);
}
```

Uma classe que implementa essa interface deve definir o método *Compare*, 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*, e 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;
```

Em princípio, essa instrução parece violar todas as regras imagináveis de segurança de tipos. Entretanto, se você considerar o que a interface *IComparer*<*T*> faz, encontrará algum sentido. O objetivo do método *Compare* é retornar um valor com base em uma comparação realizada entre os parâmetros passados. Se você puder comparar os *Objects*, certamente conseguirá comparar as *Strings*, que são apenas tipos específicos de *Objects*. Acima de tudo, uma *String* deve conseguir fazer tudo o que um *Object* pode fazer – essa é a finalidade da herança.

Contudo, tudo isso ainda parece uma certa presunção. Como o compilador C# reconhece que você 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? Ao reexaminar 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 você pode passar qualquer tipo derivado de *T*. Não é possível utilizar *T* como tipo de retorno de quaisquer métodos. Isso permite basicamente que você faça referência a 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. Basicamente, se um tipo *A* apresenta algumas operações, propriedades ou campos, então, se o tipo *B* for derivado do *A*, ele 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 alternativos no mundo dos genéricos, mas são úteis. Por exemplo, a classe de coleções genéricas *List*<*T*> 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 poder 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 especificados de tipos. Entretanto, a menos que você seja um matemático, será muito difícil lembrar o que a covariância e a contravariância realmente fazem. O que consigo me lembrar, com base nos exemplos contidos nesta seção, é o seguinte:

- **Covariância** Se os métodos de uma interface genérica puderem retornar strings, também poderão retornar objetos. (Todas as strings são objetos.)

- **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** Somente interfaces e delegates podem ser declarados covariantes ou contravariantes. Você não pode utilizar os modificadores *in* ou *out* com classes genéricas.

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, você aprendeu a definir interfaces genéricas do tipo covariantes e contravariantes que podem operar com uma hierarquia de tipos.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 19.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 18

| Para                                                                         | Faça isto                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Instanciar um objeto utilizando um tipo genérico                             | Especifique o parâmetro de tipo apropriado. Por exemplo:<br><code>Queue&lt;int&gt; myQueue = new Queue&lt;int&gt;();</code>                                                                                                                                                                                |
| Criar um novo tipo genérico                                                  | Defina a classe utilizando um parâmetro de tipo. Por exemplo:<br><code>public class Tree&lt;TItem&gt; {     ... }</code>                                                                                                                                                                                   |
| Restringir o tipo que pode ser substituído para o parâmetro de tipo genérico | Especifique uma restrição utilizando uma cláusula <code>where</code> ao definir a classe. Por exemplo:<br><code>public class Tree&lt;TItem&gt; where TItem : IComparable&lt;TItem&gt; {     ... }</code>                                                                                                   |
| Definir um método genérico                                                   | Defina o método utilizando os parâmetros de tipo. Por exemplo:<br><code>static void InsertIntoTree&lt;TItem&gt;(Tree&lt;TItem&gt; tree, params TItem[] data) {     ... }</code>                                                                                                                            |
| Invocar um método genérico                                                   | Forneça tipos para cada um dos parâmetros de tipo. Por exemplo:<br><code>InsertIntoTree&lt;char&gt;(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:<br><code>interface IRetrieveWrapper&lt;out T&gt; {     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:<br><code>public interface IComparer&lt;in T&gt; {     int Compare(T x, T y); }</code> |

## Capítulo 19

# Enumerando 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.

No Capítulo 10, “Utilizando arrays e coleções”, você aprendeu sobre arrays e classes de coleções para armazenar sequências ou conjuntos de dados. O Capítulo 10 também apresentou a instrução `foreach` que você pode empregar para percorrer passo a passo ou iterar pelos elementos de uma coleção. Naquele momento, você utilizou apenas uma instrução `foreach` como um meio rápido e conveniente de acessar o conteúdo de uma coleção, mas agora é hora de aprender um pouco mais sobre o verdadeiro funcionamento dessa instrução. Esse tópico torna-se importante quando você começa a definir suas próprias classes de coleção. Felizmente, o C# fornece iteradores para ajudá-lo a automatizar boa parte do processo.

## Enumerando os elementos em uma coleção

No Capítulo 10, você viu um exemplo do uso da instrução `foreach` para listar itens em um array simples. O código é semelhante a este:

```
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* (*enumerable collection*). Portanto, 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*, que é uma classe de coleções 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 aponta para 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. Você utiliza a propriedade *Current* para acessar o item que está sendo apontado no momento e pode usar o método *Reset* para retornar o ponteiro de volta 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<T>* e *IEnumerator<T>* devido à semelhança dos nomes. Não as misture.

Se é observador, você deve ter notado que a propriedade *Current* da interface *IEnumerator* exibe um comportamento não seguro (non-type-safe) tipado porque ele retorna um *object* em vez de um tipo específico. Mas você deve 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>*. Se estiver construindo 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 definições não genéricas.



**Nota** A interface *IEnumerator<T>* tem algumas diferenças adicionais em relação à interface *IEnumerable*; ela não contém um método *Reset*, mas estende a interface *IDisposable*.

## Implementando manualmente um enumerador

No próximo exercício, você vai definir uma classe que implementará a interface genérica *IEnumerator<T>* e criar um enumerador para a classe de árvore binária que você construiu no Capítulo 18, “Apresentando genéricos”. No Capítulo 18, você viu 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 desvio será ocultado do usuário que está fazendo a iteração pelos elementos da árvore binária!

### Crie a classe *TreeEnumerator*

1. Inicie o Microsoft Visual Studio 2010, se ele ainda não estiver executando.
2. Abra o projeto *BinaryTree*, localizado na pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 19\BinaryTree* na sua pasta Documentos. Essa solução contém uma cópia funcional do projeto *BinaryTree* que você criou no Capítulo 18.
3. Adicione uma nova classe ao projeto. No menu *Project*, clique em *Add Class*. No painel central da caixa de diálogo *Add New Item – BinaryTree*, selecione a template *Class*, digite **TreeEnumerator.cs** na caixa de texto *Name* e então clique em *Add*.
4. 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 *IEnumerator<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>*.

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> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
}
```

5. Adicione as três variáveis privadas a seguir, mostradas em negrito, à classe `TreeEnumerator<TItem>`:

```
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` é explicada na seção intitulada “Inicializando uma variável definida com um parâmetro de tipo”, mais adiante neste capítulo.

6. Acrescente um construtor `TreeEnumerator` que aceite um parâmetro `Tree<TItem>` único 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:

```
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 uma árvore binária, adicionando à fila os dados contidos na árvore. O algoritmo utilizado é semelhante àquele utilizado pelo método `WalkTree` na classe `Tree<TItem>`, que foi descrita no Capítulo 18. A principal diferença é que, em vez de o método gerar valores `NodeData` na tela, ele armazena esses valores na fila.

8. Retorne à definição da classe `TreeEnumerator<TItem>`. Clique com o botão direito do mouse em qualquer lugar da interface `IEnumerator<TItem>` na declaração da classe, aponte para *Implement Interface* e, então, clique em *Implement Interface Explicitly*.

Essa ação gera stubs (“esqueletos” de código) para os métodos da interface `IEnumerator<TItem>` e da interface `IEnumerator` e os adiciona ao final da classe, assim como gera o método `Dispose` para a interface `IDisposable`.



**Nota** A interface `IEnumerator<TItem>` herda das interfaces `IEnumerator` e `IDisposable`, que é a razão por que seus métodos também aparecem. Na verdade, o único item que pertence à interface `IEnumerator<TItem>` é a propriedade `Current` genérica. Os métodos `MoveNext` e `Reset` pertencem à interface `IEnumerator` não genérica. A interface `IDisposable` foi descrita no Capítulo 14, “Utilizando a coleta de lixo e o gerenciamento de recursos”.

9. Examine o código que foi gerado. O corpo das propriedades e dos métodos contém uma implementação padrão que simplesmente lança uma `NotImplementedException`. Você substituirá esse código por uma implementação real nos passos a seguir.
10. Substitua o corpo do método `MoveNext` pelo código mostrado em negrito:

```
bool System.Collections.IEnumerator.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 o `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 nesse exemplo. É importante lembrar que o `MoveNext` não retorna itens de dados – esta é a finalidade da proprie-

dade *Current*. Tudo o que o *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 para o seguinte:

```
TItem IEnumator<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.IEnumerator.Current*, com sua implementação padrão.

A propriedade *Current* examina a variável *enumData* para assegurar que o *MoveNext* foi chamado. (Essa variável será *null* antes da primeira chamada a *MoveNext*.) Se não for esse o caso, a propriedade irá gerar 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*. Desative a instrução `throw new NotImplementedException();` com caracteres de comentário, como mostrado a seguir em negrito. O enumerador não utiliza recurso algum que requeira disponibilidade explícita, portanto, esse método não precisa fazer coisa alguma. 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 todos os erros reportados.

## 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 nulo, 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*.)

## Implementando a interface *IEnumerable*

No próximo exercício, você modificará a classe da árvore binária para implementar a interface *IEnumerable*. 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, aponte para *Implement Interface* e, então, clique em *Implement Interface Explicitly*.

Essa ação gera implementações dos métodos *IEnumerable<TItem>.GetEnumerator* e *IEnumerable.GetEnumerable* e os adiciona à classe. O método da interface *IEnumerable* não genérico é implementado porque a interface *IEnumerable<TItem>* herda de *IEnumerable*.

4. Localize o método `IEnumerable<TItem>.GetEnumerator` genérico, próximo ao final da classe. Modifique o corpo do método `GetEnumerator()`, substituindo a instrução `throw` existente como mostrado em negrito aqui:

```
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.

O projeto deve compilar sem problemas, mas corrija qualquer erro que for reportado e recompile a solução, 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

1. No *Solution Explorer*, clique com o botão direito do mouse na solução `BinaryTree`, aponte para *Add* e clique em *New Project*. Adicione um novo projeto utilizando o template *Console Application*. Nomeie o projeto **EnumeratorTest**, configure *Location* como `\Microsoft Press\Visual CSharp Step By Step\Chapter 19` na sua pasta Documentos e clique em *OK*.
2. Clique com o botão direito do mouse no projeto `EnumeratorTest` no *Solution Explorer* e, então, clique em *Set as Startup Project*.

3. No menu *Project*, clique em *Add Reference*. Na caixa de diálogo *Add Reference*, clique na guia *Projects*. Clique no projeto `BinaryTree` e, então, em *OK*.

O assembly `BinaryTree` aparece na lista de referências, do projeto `EnumeratorTest`, no *Solution Explorer*.

4. Na janela *Code and Text Editor*, que exibe a classe `Program`, adicione a diretiva seguinte `using` à lista na parte superior do arquivo:

```
using BinaryTree;
```

5. Adicione ao método `Main` as instruções seguintes, mostradas em negrito, que 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. Compile a solução, corrigindo qualquer erro, se necessário.  
8. No menu *Debug*, clique em *Start Without Debugging*.

O programa executa e exibe os valores na sequência seguinte:

-12, -8, 0, 5, 5, 10, 10, 11, 14, 15

9. Pressione Enter para retornar ao Visual Studio 2010.

## 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# inclui iteradores que podem automatizar grande parte desse processo.

Um *iterador* é um bloco de código que produz uma sequência ordenada de valores. Além disso, um iterador não é realmente um 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 requer um pouco de reflexão para ser compreendido corretamente, então vamos considerar um exemplo básico, antes de retornar às árvores binárias e à recursão.

## 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);
 }

 IEnumerator<T> IEnumerable<T>.GetEnumerator()
 {
 foreach (var datum in data)
 yield return datum;
 }

 IEnumerator IEnumerable.GetEnumerator()
 {
 // Não implementado nesse exemplo
 }
}
```

O método *GetEnumerator* parece ser simples e direto, mas requer 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 classe *IEnumerable<T>* que contém um método *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 você precisa escrever. É possível invocar o enumerador gerado pelo iterador da maneira usual, como mostrado neste bloco de código:

```
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:

```
public IEnumerable<T> Reverse
{
 get
 {
 for (int i = data.Count - 1; i >= 0; i--)
 yield return data[i];
 }
}
```

Você pode invocar 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
```

## Definindo 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 recursivo mais natural, semelhante ao método *WalkTree* discutido no Capítulo 18.

### Adicione um enumerador à classe *Tree<TItem>*

1. Utilizando o Visual Studio 2010, abra a solução *BinaryTree* localizada na pasta *|Microsoft Press\Visual CSharp Step By Step\Chapter 19\IteratorBinaryTree* na sua pasta Documentos. Essa solução contém uma outra cópia do projeto *BinaryTree* que você criou no Capítulo 18.
2. Exiba 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 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, aponte para *Implement Interface* e, então, clique em *Implement Interface Explicitly*.

Os métodos *IEnumerable<TItem>.GetEnumerator* e *IEnumerable.GetEnumerator* são adicionados à 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 18 para imprimir 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 entregado (*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 então clique em *Existing Project*. Na caixa de diálogo *Add Existing Project*, abra a pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 19\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 anteriormente neste capítulo.

2. Clique com o botão direito do mouse no projeto EnumeratorTest no *Solution Explorer* e clique em *Set as Startup Project*.

3. Expanda o nó References do projeto EnumeratorTest no *Solution Explorer*. Clique com o botão direito do mouse no assembly BinaryTree e clique em *Remove*.

Essa ação remove a referência ao antigo assembly BinaryTree (Capítulo 18) do projeto.

4. No menu *Project*, clique em *Add Reference*. Na caixa de diálogo *Add Reference*, clique na guia *Projects*. Clique no projeto BinaryTree e, então, em *OK*.

O assembly BinaryTree aparece na lista de referências do projeto EnumeratorTest no *Solution Explorer*.

 **Nota** Esses dois passos garantem que o projeto EnumeratorTest referencia a versão do assembly BinaryTree que usa o iterador para criar seu enumerador, em vez da versão anterior.

5. Exiba o arquivo Program.cs para o projeto EnumeratorTest na janela *Code and Text Editor*. Reveja 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>*, preenche-o com alguns dados e, então, utiliza uma instrução *foreach* para exibir seu conteúdo.

6. Compile a solução, corrigindo todos os erros, 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 2010.

Neste capítulo, vimos como implementar as interfaces *IEnumerable* e *IEnumerator* com uma classe de coleção, para permitir que os aplicativos possam interagir por meio dos itens contidos na mesma coleção. Você também viu como implementar um enumerador por meio de um iterador.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 20.

- Se você quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 19

| Para                                                                             | Faça isto                                                                                                                                                                                                |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Criar uma classe enumerável, permitindo que ela suporte o comando <i>foreach</i> | Implemente a interface <i>IEnumerable</i> e forneça um método <i>GetEnumerator</i> que retorna um objeto <i>IEnumerator</i> . Por exemplo:                                                               |
| Implementar um enumerador sem usar um iterador                                   | 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: |
| 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:                                                               |

```
public class Tree<TItem> : IEnumerable<TItem>
{
 ...
}
```

```
 I IEnumerator<TItem> GetEnumerator()
 {
 ...
 }
}
```

Defina uma classe enumeradora que implemente a interface *IEnumerator* e que forneça a propriedade *Current* e o método *MoveNext* (e opcionalmente o método *Reset*). Por exemplo:

```
public class TreeEnumerator<TItem> :
I Enumerator<TItem>
{
 ...
 TItem Current
 {
 get
 {
 ...
 }
 }
 bool MoveNext()
 {
 ...
 }
}
```

Implemente o enumerador para indicar quais itens devem ser retornados (utilizando a instrução *yield*) e em qual ordem. Por exemplo:

```
I Enumerator<TItem> GetEnumerator()
{
 for (...)

 yield return ...
}
```

## Capítulo 20

# Consultando dados na memória utilizando expressões de consulta

Neste capítulo, você vai aprender a:

- Definir consultas em Language Integrated Query (LINQ) para examinar o conteúdo das coleções enumeráveis.
- Utilizar métodos de extensão e operadores de consulta LINQ.
- Explicar como a LINQ posterga a avaliação de uma consulta e como você pode forçar a execução imediata e armazenar em cache os resultados de uma consulta LINQ.

Você já viu a maioria dos recursos da linguagem C#. No entanto, ainda não discutimos um aspecto importante da linguagem que provavelmente é utilizado por muitos aplicativos – o suporte que o C# oferece a consultas de dados. Vimos que é possível definir estruturas e classes para modelar dados e que você pode utilizar coleções e arrays para armazenar dados temporariamente na memória. Mas como realizar tarefas comuns, como procurar itens em uma coleção que correspondem a um conjunto específico de critérios? Por exemplo, se você tiver uma coleção de objetos *Customer*, como encontrar todos os clientes localizados em Londres ou como pode descobrir qual cidade tem mais clientes para seus serviços? Você pode escrever seu próprio código para iterar por uma coleção e examinar os campos em cada objeto, mas esses tipos de tarefas ocorrem tão frequentemente que os projetistas do C# decidiram incluir recursos para minimizar a quantidade de código que você precisa escrever. Neste capítulo, você aprenderá a utilizar esses recursos avançados da linguagem C# para consultar e manipular 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 realizar essas operações. Mas essa estratégia pode fazer o código em um aplicativo tornar-se excessivamente amarrado à estrutura dos dados que ele processa; se as estruturas dos dados mudarem, talvez você precise realizar 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 tornar a vida de um desenvolvedor de aplicativos mais fácil, 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 projetistas 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 con-

sultar 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 invoca 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 liberada) sem que o desenvolvedor do aplicativo precise modificar as instruções SQL utilizadas pelo mesmo.

A LINQ fornece uma sintaxe e semântica muito semelhante àquelas da SQL, com vantagens parecidas. Você pode mudar a estrutura subjacente dos dados em consulta sem precisar 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.

## Utilizando a LINQ em um aplicativo C#

Talvez a maneira mais fácil de explicar como utilizar os recursos do C# que suportam a LINQ é 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          | Orlando   | Gee        | A Bike Store     |
| 2          | Keith     | Harris     | Bike World       |
| 3          | Donna     | Carreras   | A Bike Store     |
| 4          | Janet     | Gates      | Fitness Hotel    |
| 5          | Lucy      | Harrington | Grand Industries |
| 6          | David     | Liu        | Bike World       |
| 7          | Donald    | Blanton    | Grand Industries |
| 8          | Jackie    | Blackwell  | Fitness Hotel    |
| 9          | Elsa      | Leavitt    | Grand Industries |
| 10         | Eric      | Lang       | Distant Inn      |

\* N. de R.T.: Os dados do exemplo estão em inglês, assim como os códigos-fonte que acompanham o livro (no CD).

## Informações de endereços

| CompanyName      | City     | Country        |
|------------------|----------|----------------|
| A Bike Store     | New York | United States  |
| Bike World       | Chicago  | United States  |
| Fitness Hotel    | Ottawa   | Canada         |
| Grand Industries | London   | United Kingdom |
| Distant Inn      | Tetbury  | United Kingdom |

A LINQ requer que os dados sejam armazenados em uma estrutura de dados que implementa a interface *IEnumerable*, como descrito no Capítulo 19, “Enumerando coleções”. Não importa qual estrutura você utiliza (um array, uma *HashTable*, uma *Queue* ou qualquer outro tipo de coleção ou mesmo uma que você mesmo define) contanto que seja enumerável. Mas, para facilitar, os exemplos neste capítulo supõem que as informações dos clientes e dos endereços são mantidas nos arrays *customers* e *addresses* mostrados no seguinte exemplo de código.

**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. Você aprenderá mais sobre os recursos fornecidos pelo .NET Framework para recuperar informações a partir de um banco de dados na Parte V deste livro, “Gerenciando dados”.

```
var customers = new[] {
 new { CustomerID = 1, FirstName = "Orlando", LastName = "Gee",
 CompanyName = "A Bike Store" },
 new { CustomerID = 2, FirstName = "Keith", LastName = "Harris",
 CompanyName = "Bike World" },
 new { CustomerID = 3, FirstName = "Donna", LastName = "Carreras",
 CompanyName = "A Bike Store" },
 new { CustomerID = 4, FirstName = "Janet", LastName = "Gates",
 CompanyName = "Fitness Hotel" },
 new { CustomerID = 5, FirstName = "Lucy", LastName = "Harrington",
 CompanyName = "Grand Industries" },
 new { CustomerID = 6, FirstName = "David", LastName = "Liu",
 CompanyName = "Bike World" },
 new { CustomerID = 7, FirstName = "Donald", LastName = "Blanton",
 CompanyName = "Grand Industries" },
 new { CustomerID = 8, FirstName = "Jackie", LastName = "Blackwell",
 CompanyName = "Fitness Hotel" },
 new { CustomerID = 9, FirstName = "Elsa", LastName = "Leavitt",
 CompanyName = "Grand Industries" },
 new { CustomerID = 10, FirstName = "Eric", LastName = "Lang",
 CompanyName = "Distant Inn" }
};

var addresses = new[] {
 new { CompanyName = "A Bike Store", City = "New York", Country = "United States" },
 new { CompanyName = "Bike World", City = "Chicago", Country = "United States" },
 new { CompanyName = "Fitness Hotel", City = "Ottawa", Country = "Canada" },
}
```

```

new { CompanyName = "Grand Industries", City = "London",
 Country = "United Kingdom"},

new { CompanyName = "Distant Inn", City = "Tetbury", Country = "United Kingdom"}

};
```

**Nota** As seções a seguir – “Selecionando dados”, “Filtrando dados”, “Ordenando, agrupando e agregando dados” e “Junção de dados” – 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 a sintaxe funciona. Mas é útil pelo menos examinar as seções a seguir para que você possa entender como os operadores de consulta fornecidos com o C# realizam as tarefas.

## Selecionando dados

Suponha que você queira exibir uma lista que abrange o nome de cada cliente no 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 requer explicação, começando pelo uso do método *Select* do array *customers*.

O método *Select* permite recuperar dados específicos do array – nesse caso, apenas o valor no campo *FirstName* de cada item no array. Como funciona? O parâmetro para o método *Select* é na verdade um 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 pensar em *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 posteriormente. 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*. Ele é um método de extensão da classe *Enumerable*, que está localizada no namespace *System.Linq* e fornece um conjunto substancial de métodos estáticos para consultar objetos que implementam a interface *IEnumerable<T>* genérica.

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:

```
Orlando
Keith
Donna
Janet
Lucy
David
Donald
Jackie
Elsa
Eric
```

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 examinar a definição do método *Enumerable.Select* no namespace *System.Linq* na documentação fornecida com o Microsoft Visual Studio 2010, verá que ele se parece a:

```
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*. O *TSource* é o tipo da coleção que você está gerando para um conjunto enumerável de resultados (objetos *customer*, nesse caso) e o *TResult* é o tipo dos dados no conjunto enumerável de resultados (objetos *string*, nesse 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 (uma coleção genérica de objetos *customer* que implementa a interface *IEnumerable*, nesse caso). O parâmetro *selector* especifica um método genérico que identifica os campos a serem recuperados (*Func* é o nome de um tipo de delegate genérico no .NET Framework que você pode utilizar para encapsular um método genérico). O método referenciado pelo parâmetro *selector* recebe um parâmetro *TSource* (nesse caso, *customer*) e entrega (yield) uma coleção de objetos *TResult* (nesse caso, *string*). O valor retornado pelo método *Select* é uma coleção enumerável de objetos *TResult* (novamente, *string*).



**Nota** Se precisar revisar como métodos de extensão funcionam e o papel do primeiro parâmetro para um método de extensão, volte para consultar o Capítulo 12, “Trabalhando com herança”.

A questão 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 múltiplos 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> customerFullName =
 customers.Select(cust => cust.FirstName + " " + cust.LastName);
```

- Você pode definir um novo tipo que envolve os nomes e sobrenomes e utilizar o método *Select* para construir instâncias desse tipo, assim:

```
class Names
{
 public string FirstName{ get; set; }
 public string LastName{ get; set; }
}

IEnumerable<Names> customerName =
 customers.Select(cust => new Names
 {
 FirstName = cust.FirstName,
 LastName = cust.LastName
 });

```

A segunda opção é talvez a 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 customerName =
 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.

## Filtrando dados

O método *Select* permite especificar ou “projetar” os campos que você 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 de endereços localizados 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 que você especifica. Este exemplo utiliza uma outra expressão lambda. O tipo *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 linha contendo cada campo a partir 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 uma outra coleção enumerável de objetos *string* (o tipo *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:

```
A Bike Store
Bike World
```

## Ordenando, agrupando e agregando dados

Se estiver familiarizado com a linguagem SQL, você sabe que ela permite realizar 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 e também é 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 ocorre com os métodos *Select* e *Where*, esse espera um método como seu argumento e identifica as expressões que você quer utilizar para ordenar os dados. Por exemplo, você pode exibir os nomes 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:

```
A Bike Store
Bike World
Distant Inn
Fitness Hotel
Grand Industries
```

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 próximo exemplo 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);
 }
}
```

Nesse ponto, você deve reconhecer o padrão! O método *GroupBy* espera um método que especifica os campos para agrupar os dados. Há, porém, algumas diferenças sutis entre o método *GroupBy* e os outros métodos que você viu até agora.

A questão 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 na 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 para 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: United States 2 companies
 A Bike Store
 Bike World
Country: Canada 1 companies
 Fitness Hotel
Country: United Kingdom 2 companies
 Grand Industries
 Distant Inn

```

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 desse bloco de código se parece a:

```
Number of companies: 5
```

Nesse 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, estritamente 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 isso:

```

int numberOfCountries = addresses.Select(addr => addr.Country).Count();
Console.WriteLine("Number of countries:{0}", numberOfCountries);

```

A saída se parece a:

```
Number of countries: 5
```

De fato, há somente três diferentes países no array *addresses*; isso acontece porque os Estados Unidos ("United States") e o Reino Unido ("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 irá gerar a saída do resultado esperado:

```
Number of countries: 3
```

## Junção de dados

Assim como a SQL, a LINQ permite fazer junção de múltiplos 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 os nomes dos países onde estão localizados:

```
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 citiesAndCustomers)
{
 Console.WriteLine(row);
}
```

Os nomes e sobrenomes dos clientes estão disponíveis no array *customers*, mas o país para 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* a uma 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 = Orlando, LastName = Gee, Country = United States }
{ FirstName = Keith, LastName = Harris, Country = United States }
{ FirstName = Donna, LastName = Carreras, Country = United States }
{ FirstName = Janet, LastName = Gates, Country = Canada }
{ FirstName = Lucy, LastName = Harrington, Country = United Kingdom }
{ FirstName = David, LastName = Liu, Country = United States }
{ FirstName = Donald, LastName = Blanton, Country = United Kingdom }
```

```
{ FirstName = Jackie, LastName = Blackwell, Country = Canada }
{ FirstName = Elsa, LastName = Leavitt, Country = United Kingdom }
{ FirstName = Eric, LastName = Lang, 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 que 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 tivesse uma empresa correspondente e que cada empresa tivesse um endereço próprio. As coleções não impõem o mesmo nível de integridade de dados, ou seja, você poderia 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.

## Utilizando 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 para permitir 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 o nome e o sobrenome de cada cliente, você pode utilizar a seguinte instrução (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*:

```
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}\t{1} companies",
 companiesPerCountry.Key, companiesPerCountry.Count());
 foreach (var companies in companiesPerCountry)
 {
 Console.WriteLine("\t{0}", companies.CompanyName);
 }
}
```

Você pode invocar 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*, assim:

```
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 invocar o método *Count* diretamente sobre a coleção original, assim:

```
int numberOfCompanies = addresses.Count();
```

Você pode utilizar o operador *join* para combinar duas coleções sobre 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 (A LINQ atualmente suporta apenas equi-joins).

```
var companiesAndCustomers = 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 de 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 realizar operações no 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 informações adicionais, consulte a documentação fornecida com o Visual Studio 2010.

## Consultando 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 implementa 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. Você inicialmente chamará 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 2010 se ele ainda não estiver em execução.
2. Abra o projeto *QueryBinaryTree*, localizado na pasta `\Microsoft Press\Visual CSharp Step by Step\Chapter 20\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—Query BinaryTree*, digite **Employee.cs** na caixa *Name* e então clique em *Add*.

4. Adicione as propriedades automáticas mostradas em negrito à classe *Employee*:

```
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 à classe *Employee*. As classes 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* no arquivo *Employee.cs* a fim de 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 ela precisa comparar elementos ao inseri-los na árvore.

8. Substitua o corpo do método *CompareTo* pelo código mostrado em negrito. 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** Para uma descrição da interface *IComparable*, consulte o Capítulo 18, "Apresentando genéricos".

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 20\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 clique em *Add Reference*. Na caixa de diálogo *Add Reference*, clique na guia *Projects*, selecione o projeto *BinaryTree* e então clique em *OK*.
11. Exiba o arquivo Program.cs file do projeto *QueryBinaryTree* na janela *Code and Text Editor*, e verifique se a lista de instruções *using* no início do arquivo contém a seguinte linha de código:

```
using System.Linq;
```

12. Adicione a seguinte instrução *using* à lista na parte superior do arquivo Program.cs para trazer o namespace *BinaryTree* ao escopo:

```
using BinaryTree;
```

13. No método *DoWork* na classe *Program*, adicione as seguintes instruções mostradas no texto 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 = "Janet", LastName = "Gates", Department = "IT" });
 empTree.Insert(new Employee
 { Id = 2, FirstName = "Orlando", LastName = "Gee", Department = "Marketing" });
 empTree.Insert(new Employee
 { Id = 4, FirstName = "Keith", LastName = "Harris", Department = "IT" });
 empTree.Insert(new Employee
 { Id = 6, FirstName = "Lucy", LastName = "Harrington", Department = "Sales" });
}

```

```
empTree.Insert(new Employee
 { Id = 3, FirstName = "Eric", LastName = "Lang", Department = "Sales" });
empTree.Insert(new Employee
 { Id = 5, FirstName = "David", LastName = "Liu", 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* que 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 2010.

17. Modifique a instrução que cria a coleção enumerável dos departamentos como mostrado em negrito:

```
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*.

Verifique se 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 2010.
20. Adicione a seguinte instrução 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.

```
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 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, e a instrução *foreach* interna exibe os nomes dos funcionários em cada departamento.

```
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 a:

List of departments

Department: IT

Department: Marketing

Department: Sales

Employees in the IT department

Id: 1, Name: Janet Gates, Dept: IT

Id: 4, Name: Keith Harris, Dept: IT

All employees grouped by department

Department: IT

    Janet Gates

    Keith Harris

Department: Marketing

    Orlando Gee

    David Liu

```
Department: Sales
 Eric Lang
 Lucy Harrington
```

23. Pressione Enter para retornar ao Visual Studio 2010.

### Recupere os dados a partir 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 seguinte instrução mostrada em negrito, com base nos 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:

```
//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*. Verifique se a saída do aplicativo é a mesma que a anterior.

5. Pressione Enter para retornar ao Visual Studio 2010.

## 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 pela coleção. 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 modificar os dados no array *addresses* entre a definição da coleção *usCompanies* e a iteração pela coleção (por exemplo, se adicionar uma nova empresa sediada nos Estados Unidos), você verá estes novos dados. Essa estratégia é chamada avaliação *postergada*.

Você pode forçar a avaliação de uma consulta LINQ 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;
```

Dessa vez, a lista de empresas é fixada quando você define a consulta. Se adicionar mais empresas norte-americanas ao array *addresses*, você não irá vê-las ao iterar pela coleção *usCompanies*. A LINQ também fornece o método *ToArrayList*, que armazena a coleção em cache como um array.

No exercício final deste capítulo, você irá comparar os efeitos do uso da avaliação postergada de uma consulta LINQ à 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 2010, exiba o projeto *QueryBinaryTree* e edite o arquivo Program.cs.
2. Comente o conteúdo do método *DoWork* separadamente das instruções que constroem a árvore binária *empTree*, como mostrado aqui:

```
static void DoWork()
{
 Tree<Employee> empTree = new Tree<Employee>(new Employee
 { Id = 1, FirstName = "Janet", LastName = "Gates", Department = "IT" });
 empTree.Insert(new Employee
 { Id = 2, FirstName = "Orlando", LastName = "Gee", Department = "Marketing" });
 empTree.Insert(new Employee
 { Id = 4, FirstName = "Keith", LastName = "Harris", Department = "IT" });
 empTree.Insert(new Employee
 { Id = 6, FirstName = "Lucy", LastName = "Harrington", Department = "Sales" });
 empTree.Insert(new Employee
 { Id = 3, FirstName = "Eric", LastName = "Lang", Department = "Sales" });
}
```

```

 empTree.Insert(new Employee
 { Id = 5, FirstName = "David", LastName = "Liu", Department = "Marketing" });

 // transforme em comentário o resto do método

}

```



**Dica** Você pode transformar um bloco de código em comentário 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 ao método *DoWork*, após construir a árvore binária *empTree*:

```

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:

```

empTree.Insert(new Employee
{
 Id = 7,
 FirstName = "Donald",
 LastName = "Blanton",
 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 pela coleção *allEmployees* mais uma vez.

5. No menu *Debug*, clique em *Start Without Debugging*. Verifique se a saída do aplicativo se parece a:

```

All employees
Id: 1, Name: Janet Gates, Dept: IT

```

```
Id: 2, Name: Orlando Gee, Dept: Marketing
Id: 3, Name: Eric Lang, Dept: Sales
Id: 4, Name: Keith Harris, Dept: IT
Id: 5, Name: David Liu, Dept: Marketing
Id: 6, Name: Lucy Harrington, Dept: Sales
```

```
Employee added
All employees
Id: 1, Name: Janet Gates, Dept: IT
Id: 2, Name: Orlando Gee, Dept: Marketing
Id: 3, Name: Eric Lang, Dept: Sales
Id: 4, Name: Keith Harris, Dept: IT
Id: 5, Name: David Liu, Dept: Marketing
Id: 6, Name: Lucy Harrington, Dept: Sales
Id: 7, Name: Donald Blanton, Dept: IT
```

Observe que na segunda vez em que o aplicativo itera pela coleção *allEmployees* a lista exibida inclui Donald Blanton, ainda que esse funcionário só tenha sido adicionado depois que a coleção *allEmployees* foi definida.

6. Pressione Enter para retornar ao Visual Studio 2010.
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;
```

A LINQ fornece versões genéricas e não genéricas dos métodos *ToList* e *ToArray*. Se possível, utilize as versões genéricas desses métodos para assegurar que o resultado seja fortemente tipado. O dado retornado pelo operador *select* é um objeto *Employee*, e o código mostrado nesse passo gera *allEmployees* como uma coleção *List<Employee>* genérica. Se especificar o método *ToList* não genérico, a coleção *allEmployees* será uma *List* de tipos *object*.

8. No menu *Debug*, clique em *Start Without Debugging*. Verifique se a saída do aplicativo se parece a:

```
All employees
Id: 1, Name: Janet Gates, Dept: IT
Id: 2, Name: Orlando Gee, Dept: Marketing
Id: 3, Name: Eric Lang, Dept: Sales
Id: 4, Name: Keith Harris, Dept: IT
Id: 5, Name: David Liu, Dept: Marketing
Id: 6, Name: Lucy Harrington, Dept: Sales
```

```
Employee added
All employees
Id: 1, Name: Janet Gates, Dept: IT
Id: 2, Name: Orlando Gee, Dept: Marketing
Id: 3, Name: Eric Lang, Dept: Sales
Id: 4, Name: Keith Harris, Dept: IT
Id: 5, Name: David Liu, Dept: Marketing
Id: 6, Name: Lucy Harrington, Dept: Sales
```

Observe que na segunda vez em que o aplicativo itera pela coleção `allEmployees` a lista exibida não inclui Donald Blanton. Isso ocorre porque a consulta é avaliada e os resultados são armazenados em cache antes de Donald Blanton ser adicionado à árvore binária `empTree`.

**9.** Pressione Enter para retornar ao Visual Studio 2010.

Neste capítulo, você aprendeu como o LINQ usa a interface `IEnumerable<T>` e métodos de extensão para fornecer um mecanismo para consultar dados. Vimos também que esses recursos aceitam a sintaxe de expressão de consultas no C#.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 21.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 20

| Para                                                             | Faça isto                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Projetar campos especificados a partir de uma coleção enumerável | <p>Utilize o método <i>Select</i> e especifique uma expressão lambda que identifica os campos a projetar. Por exemplo:</p> <pre>var customerFirstNames = customers.Select(cust =&gt; 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 as 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 =&gt;         String.Equals(addr.Country, "United States"))     .Select(usComp =&gt; 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 para identificar o campo a utilizar para ordenar as linhas. Por exemplo:</p> <pre>var companyNames =     addresses.OrderBy(addr =&gt; addr.CompanyName)     .Select(comp =&gt; 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 para identificar o campo a utilizar para agrupar as linhas. Por exemplo:</p> <pre>var companiesGroupedByCountry =     addresses.GroupBy(addr =&gt; 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>                                                                                              |

(continua)

| 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 companiesAndCustomers =     customers         .Select(c =&gt; new { c.FirstName, c.LastName,     c.CompanyName }).         Join(addresses, custs =&gt; custs.CompanyName,             addrs =&gt; addrs.CompanyName,             (custs, addrs) =&gt; new {custs.FirstName, custs.     LastName,             addrs.Country });</pre> <p>Ou utilize o operador de consulta <i>join</i>. Por exemplo:</p> <pre>var companiesAndCustomers =     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 para 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&lt;Employee&gt;()     select e;</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

## Capítulo 21

# 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.

Você já fez uso dos símbolos de operadores padrão (como `+` e `-`) para executar operações padrão (como adição e subtração) em tipos (como `int` e `double`). Grande parte dos tipos predefinidos vem com seus comportamentos predefinidos para cada operador. Você também pode definir como os operadores das suas estruturas e classes devem se comportar, o que é o tópico deste capítulo.

## Entendendo os operadores

Compensa rever alguns aspectos básicos dos operadores antes de examinar os detalhes de seu funcionamento e como sobrecarregá-los. Em resumo:

- 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  $\alpha + b * c$  é o mesmo que  $\alpha + (b * c)$ .
- Cada operador também tem uma associatividade para definir se o operador avalia da esquerda para a direita ou da direita para a esquerda. Por exemplo, o operador `=` tem uma associatividade à direita (ele avalia da direita para a esquerda), portanto  $\alpha = b = c$  é o mesmo que  $\alpha = (b = c)$ .
- 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

Vimos ao longo do livro que o C# permite sobrestrararar métodos quando você define seus próprios tipos. O C# também permite sobrestrararar 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 precisa criar um método para 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 você pudesse, as coisas poderiam se complicar!
- Há alguns símbolos de operador que não podem ser sobrestrarregados. Por exemplo, você não pode sobrestrarar 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, você pode usar as propriedades para simular a atribuição (`=`) como um operador e pode usar delegates para simular uma chamada de função como um operador.

## Operadores sobrestrarregados

Para definir o comportamento do seu operador, você deve sobrestrararar 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 você está declarando. Por exemplo, o código a seguir mostra uma estrutura fixada 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 público. 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. (Programadores C++ devem observar que operadores nunca têm um parâmetro *this* oculto.)

**Dica** Ao 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 o 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 (caso contrário, seu código não compilará): 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*. Neste exem-

plo, os dois parâmetros são objetos *Hour*. Mas pode haver ocasiões em que você queira definir implementações adicionais do *operator+* que adiciona, 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 invocação de operador e também garante que você não possa alterar o significado dos operadores predefinidos.

## Criando 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 você pode somar uma *Hour* e um *int* – você precisa apenas utilizar primeiramente o construtor *Hour* para converter o *int* em um *Hour*. Por exemplo:

```
Hour a = ...;
int b = ...;
Hour sum = a + new Hour(b);
```

Esse é certamente um código válido, mas não é tão claro ou 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 é uma *Hour* e o segundo é um *int*. O código a seguir mostra o enfoque recomendado:

```
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 à esquerda, e um *int*, como o operando à direita. Ele não declara como somar um *int*, como o operando à esquerda, e uma *Hour*, como o operando à direita:

```
int a = ...;
Hour b = ...;
Hour sum = a + b; // erro de tempo de compilação
```

Isso é constraintuitivo. Se você pode escrever a expressão  $a + b$ , pode esperar também ser capaz de escrever  $b + a$ . Mas você deve fornecer outra sobrecarga do *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 ou 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 CLR (*common language runtime*) suportam ou entendem a sobrecarga de operadores. Se você estiver criando classes para serem utilizadas em outras linguagens, se você sobrecarregar um operador, deverá oferecer um mecanismo alternativo com suporte para a mesma funcionalidade. Por exemplo, vamos supor que você implemente o *operador+* para a estrutura *Hour*:

```
public static Hour operator +(Hour lhs, int rhs)
{

}
```

Se precisar utilizar sua classe a partir de um aplicativo Visual Basic, você também deverá fornecer um método *Add* que realiza a mesma coisa:

```
public static Hour Add(Hour lhs, int rhs)
{

}
```

## Entendendo a avaliação da atribuição composta

Um operador de atribuição composta (como *+=*) é sempre avaliado em termos do seu operador associado (como *+*). Em outras palavras, a instrução

```
a += b;
```

é automaticamente avaliada assim:

```
a = a + b;
```

Em geral, a expressão *a @= b* (em que *@* representa qualquer operador válido) é sempre avaliada como *a = a @ b*. Se você declarou o operador simples apropriado, a versão sobrecarregada é automaticamente chamada quando você utiliza seu operador de atribuição composta associado. Por exemplo:

```
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*. Observe, porém, que você não pode escrever a expressão *b += a* porque ela é igual a *b = b + a*. Embora a soma seja válida, a atribuição não é, porque não há como atribuir uma *Hour* ao tipo *int* predefinido.

## Declarando operadores de incremento e decremento

O C# permite que você declare sua própria versão de operadores de incremento (`++`) e decremento (`--`). As regras usuais se aplicam ao declarar estes operadores: eles precisam ser públicos, estáticos e unários (eles só podem aceitar um único 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 (postfix) 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, código inválido em C#
```

O resultado da expressão prefixada é o valor de retorno do operador. O compilador do C# converte o código

```
Hour now = new Hour(9);
Hour prefix = ++now;
```

em:

```
Hour now = new Hour(9);
now = Hour.operator++(now); // pseudocódigo, código inválido em 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.

## Comparando operadores em estruturas e classes

Saiba que a implementação do operador de incremento na estrutura *Hour* só funciona porque *Hour* é uma estrutura. Se alterar *Hour* para uma classe, mas deixar a implementação do seu operador de incremento inalterada, você verá que a versão sufixada não dará a resposta correta. Se lembrar que uma classe é um tipo-referência e rever as traduções do compilador explicadas anteriormente, você entenderá por que isso ocorre:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator++(now); // pseudocódigo, código inválido em C#
```

Se *Hour* for uma classe, a instrução de atribuição *postfix = now* faz 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 faz uma cópia de *now* em *postfix* e todas as alterações em *now* deixam *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, o *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 sobrecarga de coleta de lixo. Portanto, é recomendável que você limite as sobrecargas de operador ao definir os tipos. Essa recomendação se aplica a todos os operadores e não apenas ao operador de incremento.

## Definindo 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!=*, você deve definir ambos. Essa regra, “ne-

nhum 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. Eis os operadores `==` e `!=` para a estrutura `Hour`:

```
struct Hour
{
 public Hour(int initialValue)
 {
 this.value = initialValue;
 }

 /**
 * @param lhs The left-hand side of the comparison.
 * @param rhs The right-hand side of the comparison.
 */
 public static bool operator ==(Hour lhs, Hour rhs)
 {
 return lhs.value == rhs.value;
 }

 /**
 * @param lhs The left-hand side of the comparison.
 * @param rhs The right-hand side of the comparison.
 */
 public static bool operator !=(Hour lhs, Hour rhs)
 {
 return lhs.value != rhs.value;
 }

 /**
 * @return The value of this hour.
 */
 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!

 **Nota** Se definir `operator ==` e `operator !=` em uma classe, você também deve redefinir os métodos `Equals` e `GetHashCode` herdados do `System.Object` (ou `System.ValueType`, se você estiver criando uma estrutura). O método `Equals` deve exibir exatamente o mesmo comportamento que o `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 2010.) 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.)

## Implementando 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  $(x + yi)$ , em que  $x$  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 de sua aparência obscura e teórica, os números complexos têm muitas aplicações nas áreas da eletrônica, matemática aplicada, física e em diversos aspectos da engenharia.



**Nota** O .NET Framework 4.0 já dispõe de um tipo chamado *Complex* no namespace *System.Numerics*, que implementa números complexos, de modo que não há mais necessidade de definir uma implementação própria. 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 operandos 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. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver em execução.
2. Abra o projeto ComplexNumbers, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 21\ComplexNumbers de 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 menu *Project*, clique em *Add Class*. Na caixa de diálogo *Add New Item – Complex Numbers*, digite **Complex.cs** na caixa de texto *Name* 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 de inteiros, *Real* e *Imaginary*, à classe *Complex*, como mostrado em negrito a seguir. Você utilizará essas duas propriedades para armazenar os componentes real e imaginário de um número complexo.

```
class Complex
{
 public int Real { get; set; }
 public int Imaginary { get; set; }
}
```

5. Adicione o construtor mostrado em negrito a seguir à classe *Complex*. Esse construtor aceita dois parâmetros *int* e os utiliza para preencher as propriedades *Real* e *Imaginary*.

```
class Complex
{
 ...
 public Complex (int real, int imaginary)
 {
 this.Real = real;
 this.Imaginary = imaginary;
 }
}
```

6. Substitua o método *ToString*, como mostrado em negrito a seguir. Esse método retorna uma string que representa o número complexo, na forma  $(x + yi)$ .

```
class Complex
{
 ...
 public override string ToString()
 {
 return String.Format("{0} + {1}i", this.Real, this.Imaginary);
 }
}
```

7. Adicione o operador **+** sobrecarregado, mostrado em negrito a seguir, à classe *Complex*. Esse é o operador de adição binária. Ele utiliza dois objetos *Complex* e soma esses objetos, ao efetuar o cálculo apresentado na tabela apresentada no início do exercício. O operador retorna um novo objeto *Complex* que contém os resultados desses cálculos.

```
class Complex
{
 ...
 public static Complex operator +(Complex lhs, Complex rhs)
 {
 return new Complex(lhs.Real + rhs.Real, lhs.Imaginary + rhs.Imaginary);
 }
}
```

8. Adicione o operador **-** sobrecarregado à classe *Complex*. Esse operador segue a mesma forma do operador sobrecarregado **+**.

```
class Complex
{
 ...
 public static Complex operator -(Complex lhs, Complex rhs)
 {
 return new Complex(lhs.Real - rhs.Real, lhs.Imaginary - rhs.Imaginary);
 }
}
```

9. Implemente os operadores **\*** e **/**. 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.)

```

class Complex
{
 ...
 public static Complex operator *(Complex lhs, Complex rhs)
 {
 return new Complex(lhs.Real * rhs.Real + lhs.Imaginary * rhs.Real,
 lhs.Imaginary * rhs.Imaginary + 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);
 }
}

```

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*:

```

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, depois exibe os resultados em cada caso.

11. No menu *Debug*, clique em *Start Without Debugging*.

Verifique se o aplicativo exibe os resultados mostrados na imagem a seguir.

```
C:\Windows\system32\cmd.exe
first is (10 + 4i)
second is (5 + 2i)
Add result is (15 + 6i)
Subtract: result is (5 + 2i)
Multiply: result is (20 + 20i)
Divide: result is (2 + 8i)
Press any key to continue . . .
```

12. Feche o aplicativo e retorne ao ambiente de programação do Visual Studio 2010.

Você acabou de criar um tipo que modela números complexos e suporta operações aritméticas básicas. No próximo exercício, você estenderá a classe *Complex* e proverá os operadores de igualdade, `==` e `!=`. Lembre-se de que, ao implementar esses operadores, você também deve sobrescrever os métodos *Equals* e *GetHashCode* que a classe herda do tipo *Object*.

### Implemente operadores de igualdade

1. No Visual Studio 2010, exiba o arquivo *Complex.cs* na janela *Code and Text Editor*.
2. Adicione os operadores `==` e `!=` à classe *Complex*, como mostrado em negrito a seguir. Observe que esses dois operadores utilizam o método *Equal*, que compara uma instância de uma classe com outra instância especificada como argumento. Ele retornará *true* se forem iguais, e *false*, caso contrário.

```
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));
 }
}
```

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.GetHashCode()
```

```
'ComplexNumbers.Complex' defines operator == or operator != but does not override
Object.Equals(object o)
```

Se você definir os operadores `!=` e `==`, também deverá sobrescrever os métodos *Equal* e *GetHashCode*, herdados de *SystemObject*.



**Nota** Se a janela *Error List* não for exibida, no menu *View*, clique em *Error List*.

- Substitua o método *Equals* na classe *Complex*, como mostrado em negrito a seguir:

```
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* utiliza 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, e o parâmetro passado; se forem iguais, o método retornará *true*, ou retornará *false*, caso contrário. 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*, que chama o método *Equals* novamente, o que resulta um loop infinitamente 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 se a solução é compilada sem emitir quaisquer avisos.

- Exiba o arquivo Program.cs na janela *Code and Text Editor*. Adicione o seguinte código 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 uma 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.

- No menu *Debug*, clique em *Start Without Debugging*. Verifique se as duas últimas mensagens exibidas são as seguintes:

```
Comparison: temp != first
Comparison: temp == temp
```

- Feche o aplicativo e retorne ao Visual Studio 2010.

## Entendendo os 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 cujo tipo não seja *double*, mas somente se esse valor puder ser convertido em um *double*. Por exemplo, você pode fornecer um argumento *int*. Nesse caso, o compilador gera um código que converte o argumento, de um *int* para um *double*, quando o método é chamado.

## Fornecendo conversões predefinidas

Os tipos predefinidos têm algumas conversões predefinidas. Por exemplo, como mencionado anteriormente, um *int* pode ser implicitamente convertido em um *double*. Uma conversão implícita não requer 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.

Por outro lado, um *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 requer 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 *OverflowException*. O C# permite fornecer operadores de conversão para seus próprios tipos definidos pelo usuário a fim de controlar se há sentido em converter valores em outros tipos e se essas conversões são implícitas ou explícitas.

## Implementando operadores de conversão definidos pelo usuário

A sintaxe para declarar um operador de conversão definido pelo usuário é semelhante àquela para declarar um operador sobrecarregado. Um operador de conversão deve ser *public* e também deve ser *static*. 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;
}
```

O tipo de origem da conversão é declarado como o único parâmetro (nesse caso, *Hour*) e o tipo de destino da conversão é declarado como o nome do tipo, após a palavra-chave *operator* (nesse 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 operadores de conversão 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 implicitamente (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 requer um casting explícito:

```
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, ela 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 "Hello, World", em uma *Hour*?)

## Criando 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 do *operator+* (*Hour + Hour*, *Hour + int* e *int + Hour*) para a estrutura *Hour*, como mostrado anteriormente, você pode fornecer uma única versão do *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 *Hour* ao *int* (em qualquer ordem), o compilador do C# converterá automaticamente o *int* em *Hour* e, então, chamará o *operator+* com dois argumentos *Hour*:

```
void Example(Hour a, int b)
{
 Hour eg1 = a + b; // b convertido em uma Hour
 Hour eg2 = b + a; // b convertido em uma Hour
}
```

## Escrevendo operadores de conversão

No exercício a seguir, 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 deverá descartar o elemento imaginário. Por conseguinte, você implementará esse operador de conversão como explícito.

### Implemente os operadores de conversão

1. Retorne ao Visual Studio 2010 e exiba o arquivo Complex.cs na janela *Code and Text Editor*. Adicione o construtor, mostrado em negrito a seguir, à classe *Complex*. Esse construtor aceita um único parâmetro *int* que ele utiliza para inicializar a propriedade *Real*. A propriedade imaginária 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*. Esse operador converte de um *int* para um objeto *Complex*, retornando uma nova instância da classe *Complex* por meio do construtor criado na etapa anterior.

```
class Complex
{
 ...
 public static implicit operator Complex(int from)
 {
 return new Complex(from);
 }
}
```

3. Adicione o operador de conversão explícita, mostrado a seguir, à classe *Complex*. Esse operador aceita um objeto *Complex* e retorna o valor da propriedade *Real*. Essa conversão descarta o elemento imaginário do número complexo.

```
class Complex
{
 ...
 public static explicit operator int(Complex from)
 {
 return from.Real;
 }
}
```

4. Exiba o arquivo Program.cs na janela *Code and Text Editor*. Adicione o seguinte código 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* a um *int*. O compilador gera um código que converte o *int* em um objeto *Complex* primeiramente, e depois chama o operador *==* da classe *Complex*. A instrução que soma 2 à variável *temp* converte o valor 2 do *int* em um objeto *Complex* e depois utiliza o operador *+* da classe *Complex*.

5. 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*.

6. 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.

7. 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 se a solução já está compilando, e se a saída das quatro últimas instruções 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 2010.

Neste capítulo, você aprendeu a sobrepor operadores e a fornecer uma funcionalidade específica para uma classe ou estrutura. Você implementou alguns operadores aritméticos comuns, e também criou operadores que permitem comparar instâncias de uma classe. Por último, você aprendeu a criar operadores de conversão implícita e explícita.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 22.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 21

| 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> |
| 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

# Construindo Aplicativos WPF

|                                                                  |     |
|------------------------------------------------------------------|-----|
| Capítulo 22: Apresentando o Windows Presentation Foundation..... | 475 |
| Capítulo 23: Obtendo a entrada do usuário.....                   | 509 |
| Capítulo 24: Realizando validações .....                         | 541 |

## Capítulo 22

# Apresentando o Windows Presentation Foundation

Neste capítulo, você vai aprender a:

- Criar aplicativos Microsoft Windows Presentation Foundation (WPF).
- Utilizar controles WPF comuns como rótulos, caixas de texto e botões.
- Definir estilos para controles WPF.
- Alterar as propriedades dos formulários WPF e dos controles em tempo de projeto e por meio de código em tempo de execução.
- Tratar eventos expostos por formulários e controles WPF.

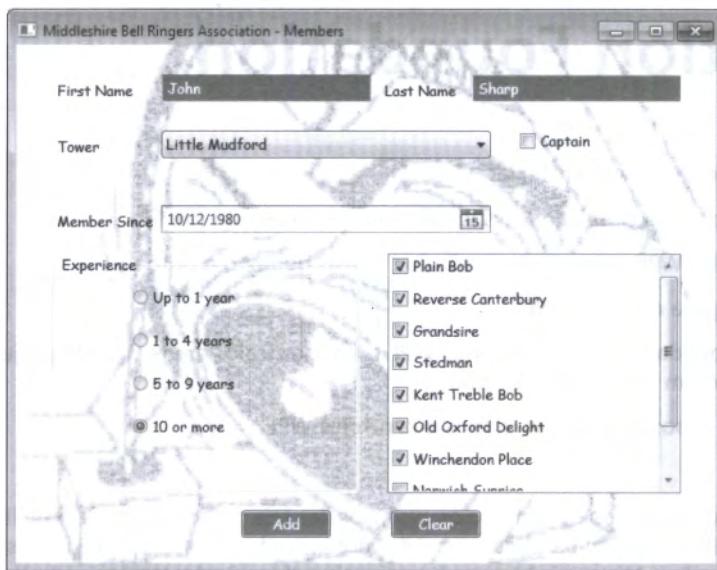
Agora que concluiu os exercícios e examinou os exemplos das três primeiras partes deste livro, você deve estar bem familiarizado com a linguagem C#. Você aprendeu a escrever programas e a criar componentes utilizando o Microsoft C# e conheceu muitos dos principais tópicos da linguagem, como métodos de extensão, expressões lambda e diferenças entre tipos-valor e tipos-referência. Agora você domina a essência da linguagem e, na Parte IV, aumentará suas habilidades usando o C# para tirar proveito das bibliotecas GUI (*graphical user interface*), fornecidas como parte do Microsoft .NET Framework. Especificamente, você verá como utilizar os objetos do namespace *System.Windows* para criar os aplicativos WPF.

Neste capítulo, você vai aprender a compilar um aplicativo WPF básico utilizando os componentes comuns, que são um recurso da maioria dos aplicativos GUI. Você verá como configurar as propriedades dos formulários e controles WPF por meio do uso das janelas *Design View* e *Properties* e também a Extensible Application Markup Language, ou XAML. Você vai saber ainda como empregar estilos WPF para construir interfaces com o usuário que podem ser facilmente adaptadas, a fim de corresponder aos padrões de apresentação da sua organização. Por fim, você vai aprender a interceptar e tratar alguns eventos que formulários e controles WPF expõem.

## Criando um aplicativo WPF

Como exemplo, você criará um aplicativo que um usuário pode usar a fim de inserir e exibir detalhes para membros da Middleshire Bell Ringers Association, um respeitado grupo dos mestres em *companologia*, a arte de tocar sinos. Inicialmente, você manterá o aplicativo muito simples, concentrando-se no layout do formulário e certificando-se de que tudo funciona. Também discutiremos alguns recursos que o WPF oferece para construir interfaces com o usuário altamente adaptáveis. Nos capítulos finais, você fornecerá menus e aprenderá a implementar validação para garantir que os dados inseridos façam sentido. A figura a seguir mostra como o aplicativo ficará

quando estiver concluído. (Você pode ver a versão concluída compilando e executando o projeto BellRingers na pasta Microsoft Press\Visual CSharp Step by Step\Chapter 22 \ BellRingers – Completed\ na sua pasta Documentos.)



## Construindo o aplicativo WPF

Neste exercício, você vai começar compilando o aplicativo da Middleshire Bell Ringers Association criando um novo projeto, fazendo o layout do formulário e adicionando os controles ao formulário. Você utilizou aplicativos WPF existentes no Microsoft Visual Studio 2010 nos capítulos anteriores, então boa parte dos dois primeiros exercícios será uma revisão.

### Crie o projeto da Middleshire Bell Ringers Association

1. Inicie o Visual Studio 2010 se ele ainda não estiver em execução.
2. Se você estiver utilizando o Visual Studio 2010 Standard ou o Visual Studio 2010 Professional, siga estes passos para criar um novo aplicativo WPF:
  - 2.1. No menu *File*, aponte para *New* e então clique em *Project*. A caixa de diálogo *New Project* se abre.
  - 2.2. No painel da esquerda, expanda *Installed Templates* (se ainda não estiver expandido), expanda *Visual C#*, e clique em *Windows*.
  - 2.3. No painel central, clique no ícone *WPF Application*.

- 2.4. No campo *Location*, digite **\Microsoft Press\Visual CSharp Step By Step\Chapter 22** na sua pasta Documentos.
  - 2.5. No campo *Name*, digite **BellRingers**.
  - 2.6. Clique em *OK*.
3. Se você estiver utilizando o Microsoft Visual C# 2010 Express, siga estes passos para criar um novo aplicativo gráfico.
- 3.1. No menu *File*, clique em *New Project*.
  - 3.2. Na caixa de diálogo *New Project*, no painel à esquerda, em *Installed Templates*, clique em *Visual C#*.
  - 3.3. *No painel central, clique em *WPF Application*; no campo *Name*, digite **BellRingers** e clique em *OK*.*
  - 3.4. Quando o Visual Studio criar o projeto, clique em *Save All* no menu *File*.
  - 3.5. No campo *Location* da caixa de diálogo *Save Project*, especifique a localização **Microsoft Press\Visual CSharp Step By Step\Chapter 22** em sua pasta Documentos, e clique em *Save*.

O novo projeto é criado e contém um formulário em branco chamado *MainWindow*.

## Examine o formulário e o layout Grid

1. Examine o formulário no painel *XAML* abaixo da janela *Design View*. Observe que a definição XAML do formulário se parece com isso:

```
<Window x:Class="BellRingers.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="MainWindow" Height="350" Width="525">
 <Grid>
 </Grid>
</Window>
```

O atributo *Class* especifica o nome completamente qualificado da classe que implementa o formulário. Nesse caso, ele se chama *MainWindow* no namespace *BellRingers*. O template WPF Application utiliza o nome do aplicativo como o namespace padrão para formulários. O atributo *xmlns* especifica os namespaces XML que definem os esquemas utilizados pelo WPF; todos os controles e outros itens que você pode incorporar a um aplicativo WPF têm definições que residem nesses namespaces. (Se você não estiver familiarizado com namespaces XML, ignore esses atributos *xmlns* por enquanto.) O atributo *Title* especifica o texto que aparece na barra de título do formulário, e os atributos *Height* e *Width* especificam a altura e largura padrão do formulário. Você pode modificar esses valores alterando-os no painel *XAML* ou utilizando a janela

*Properties*. Você também pode alterar o valor dessas e de muitas outras propriedades, dinamicamente, escrevendo código C# que executa quando o formulário é executado.

2. Clique no formulário MainWindow na janela *Design View*. Na janela *Properties*, localize e clique na propriedade *Title*, digite **Middleshire Bell Ringers Association – Members** e pressione Enter para alterar o texto da barra de título do formulário.

Observe que o valor no atributo *Title* do formulário muda no painel *XAML* e o novo título é exibido na barra de título do formulário na janela *Design View*.



**Nota** O formulário MainWindow contém um controle filho que examinaremos no próximo passo. Se a janela *Properties* exibir as propriedades para um controle *System.Windows.Controls.Grid*, clique no texto *MainWindow*, no formulário MainWindow. Essa ação seleciona o formulário, em vez da grade, e a janela *Properties* então exibe as propriedades para o controle *System.Windows.Window*.

3. No painel *XAML*, observe que o elemento *Window* contém um elemento filho chamado *Grid*.

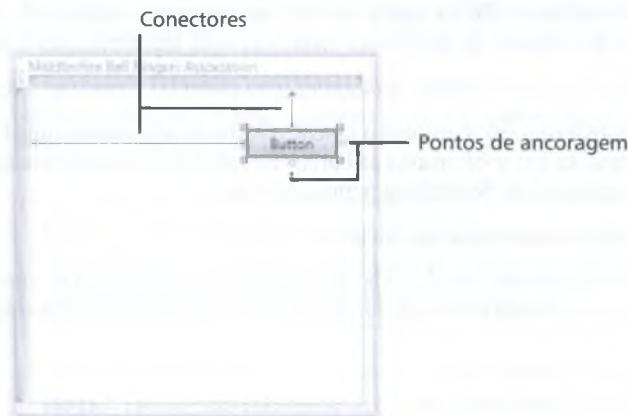
Em um aplicativo WPF, você posiciona controles, por exemplo, botões, caixas de texto e rótulos, em um painel em um formulário. O painel gerencia o layout dos controles que ele contém. O painel padrão adicionado pelo template WPF Application é um *Grid*, que pode ser utilizado para especificar precisamente a localização dos seus controles em tempo de projeto. Há outros tipos de painéis disponíveis que fornecem diferentes estilos de layout. Por exemplo, *StackPanel* posiciona automaticamente os controles em um arranjo vertical, com cada controle organizado diretamente abaixo do anterior. Outro exemplo é *WrapPanel*, que organiza os controles em uma fileira da esquerda para a direita e depois quebra o conteúdo na próxima linha, se a linha atual estiver cheia. O objetivo principal de um painel de layout é determinar como os controles serão posicionados se o usuário redimensionar a janela em tempo de execução; os controles são automaticamente redimensionados e repositionados de acordo com o tipo do painel.



**Nota** O painel *Grid* é flexível, mas complexo. Por padrão, você pode pensar no painel *Grid* como definindo uma única célula em que você pode adicionar controles e determinar sua localização. Mas é possível configurar as propriedades de um painel *Grid* para especificar múltiplas linhas e colunas (daí seu nome, "grade") e também é possível adicionar controles em cada uma das células definidas por essas linhas e colunas. Neste capítulo, para não dificultar, utilizamos somente uma célula.

4. Na janela *Design View*, clique no formulário MainWindow e então clique na guia *Toolbox*.
5. Na seção *Common WPF Controls* da caixa de ferramentas, clique em *Button* e clique na parte superior direita do formulário.

Um controle de botão que exibe dois conectores com âncoras, os quais encaixam esse controle nas bordas superior e esquerda do formulário, é adicionado ao formulário, assim:



Embora você tenha clicado no formulário, o controle *Button* é adicionado ao controle *Grid* contido no formulário. A grade ocupa todo o formulário, exceto a barra de título na parte superior. Os conectores mostram que o botão está encaixado na bordas superior e direita da grade. No tempo de execução, se você redimensionar o formulário, o botão se deslocará para manter essas ligações e a mesma distância das bordas ligadas. Para ancorar o botão em várias bordas do formulário, clique nos pontos de ancoragem do controle ou altere as propriedades *HorizontalAlignment* e *VerticalAlignment* do botão, como descrito na próxima etapa.

6. Examine o código no painel *XAML*. O elemento *Grid* e seu conteúdo agora devem parecer com isso (embora seus valores para a propriedade *Margin* talvez sejam diferentes):

```
<Grid>
 <Button Content="Button" HorizontalAlignment="Left"
 Margin="374,40,0,0" Name="button1" Width="75" Height="23"
 VerticalAlignment="Top"/>
</Grid>
```

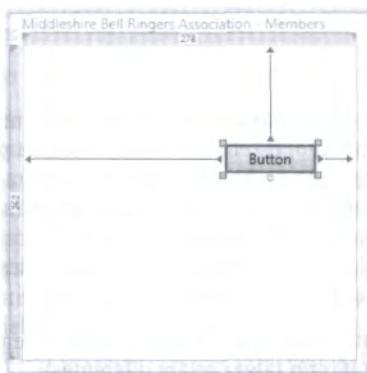
**Nota** Ao longo deste capítulo, as linhas do painel *XAML* são mostradas divididas e recuadas para que caibam na página impressa.

Ao posicionar um controle em uma grade, você pode conectar um ou todos os pontos de ancoragem à borda correspondente da grade. Se você mover o controle ao redor, ele continuará ligado às mesmas bordas, até que você altere as respectivas propriedades de alinhamento.

As propriedades *HorizontalAlignment* e *VerticalAlignment* do botão indicam as bordas às quais o botão está atualmente conectado, e a propriedade *Margin* sinaliza a distância entre essas bordas. Lembre-se do Capítulo 1, “Bem-vindo ao C#”, em que a propriedade *Margin* contém quatro valores que especificam a distância das bordas esquerda, superior, direita e inferior da grade,

respectivamente. No fragmento de XAML mostrado anteriormente, o botão tem 84 unidades a partir da borda superior da grade e 34 unidades da borda direita. (Cada unidade tem 1/96 de uma polegada.) Os valores de margem de 0 indicam que o botão não está conectado à borda correspondente. Como mencionado na etapa anterior, ao executar o aplicativo, o runtime do WPF tentará manter essas distâncias uniformes mesmo se você redimensionar o formulário.

7. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
8. Quando o formulário aparecer, redimensione a janela, clicando e arrastando uma borda de cada vez. Observe que, à medida que você arrasta as bordas do formulário, a distância do botão entre as bordas superior e esquerda do formulário permanece fixa.
9. Feche o formulário e retorne ao Visual Studio 2010.
10. Na janela *Design View*, clique no controle de botão e depois clique no ponto de ancoragem direito para anexar o controle à borda direita do formulário, como mostrado na ilustração a seguir:



No painel *XAML*, observe que a propriedade *HorizontalAlignment* não está mais especificada. O valor padrão para as propriedades *HorizontalAlignment* e *VerticalAlignment* é um valor chamado *Stretch*, de forma a indicar que o controle está ancorado nas duas bordas opostas. Também observe que a propriedade *Margin* agora aponta um valor diferente de zero para a margem esquerda.



**Nota** Você também pode clicar no ponto de ancoragem que está conectado à borda da grade para remover a conexão.

11. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo novamente.
12. Quando o formulário aparece, tente torná-lo mais estreito e mais largo. Observe que o botão não se move mais para a esquerda ou direita, porque ele está ancorado nas bordas esquerda e direita do formulário. Em vez disso, o botão fica mais largo ou mais estreito à medida que as bordas se movem.
13. Feche o formulário e retorne ao Visual Studio 2010.

14. Na janela *Design View*, adicione um segundo controle *Button* ao formulário a partir da *Toolbox* e posicione-o próximo do meio do formulário.
15. No painel *XAML*, configure os valores da propriedade *Margin* como 0,0,0,0; remova as propriedades *VerticalAlignment* e *HorizontalAlignment* e configure as propriedades *Width* e *Height*, como mostrado aqui:

```
<Button Content="Button" Margin="0,0,0,0" Name="button2"
Width="75" Height="23"/>
```



**Dica** Você também pode configurar várias das propriedades de um controle, como *Margin*, usando a janela *Properties*. Entretanto, às vezes é mais fácil digitar valores diretamente no painel *XAML*, desde que você os digite com atenção.



**Nota** Se você não configurar as propriedades *Width* e *Height* do controle de botão, o botão preenche o formulário inteiro.

16. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo mais uma vez.
17. Quando o formulário aparecer, redimensione o formulário. Observe que, enquanto o tamanho do formulário diminui ou aumenta, o novo botão tenta se reposicionar para manter sua posição relativa no formulário em relação a todos os quatro lados (isto é, ele tenta permanecer no centro do formulário). O novo controle de botão se move ao longo da parte superior do primeiro controle de botão se você diminuir a altura do formulário.
18. Feche o formulário e retorne ao Visual Studio 2010.

Contanto que sua abordagem seja consistente, utilizando painéis de layout, como o *Grid*, você pode construir formulários com uma boa aparência independentemente da resolução de tela do usuário e sem escrever código complexo para determinar quando o usuário redimensionou uma janela. Além disso, com o WPF, você pode modificar a aparência e o comportamento dos controles que um aplicativo utiliza e, mais uma vez, sem precisar escrever uma grande quantidade de código complexo. Com esses recursos, é possível construir aplicativos que podem ser facilmente personalizados para corresponder a qualquer estilo interno exigido pela sua organização. Você vai examinar alguns desses recursos nos próximos exercícios.

### Adicione uma imagem de fundo ao formulário

1. Na janela *Design View*, clique no formulário *MainWindow*.
2. Na *Toolbox*, na seção *Common WPF Controls*, clique em *Image* e então clique em qualquer lugar no formulário.

Você utilizará esse controle de imagem para exibir uma imagem contra o fundo do formulário.



**Nota** Você pode utilizar muitas outras técnicas para exibir uma imagem no fundo de um *Grid*. O método mostrado neste exercício é o mais simples, embora outras estratégias possam fornecer mais flexibilidade.

3. No painel *XAML*, configure a propriedade *Margin* do controle de imagem e remova todos os outros valores de propriedade, exceto *Name*, como mostrado aqui:

```
<Image Margin="0,0,0,0" Name="image1"/>
```

O controle de imagem se expande para ocupar a grade, embora os dois controles de botão permaneçam visíveis.

4. No *Solution Explorer*, clique com o botão direito do mouse no projeto BellRingers, aponte para *Add* e então clique em *Existing Item*. Na caixa de diálogo *Add Existing Item – BellRingers*, acesse a pasta Microsoft Press\Visual CSharp Step By Step\Chapter 22 na sua pasta Documentos. Na caixa suspensa ao lado da caixa de texto *File name*, selecione *All Files (\*.\*)*. Selecione o arquivo *bell.gif* e clique em *Add*.

Essa ação adiciona ao seu aplicativo o arquivo de imagem *Bell.gif* como um recurso. O arquivo *Bell.gif* contém um desenho de um sino tocando.

5. No painel *XAML*, modifique a definição do controle de imagem, como mostrado em negrito aqui. A propriedade *Image.Source* é um exemplo de propriedade composta que contém um ou mais elementos filhos. Observe que você precisa substituir o delimitador da tag de fechamento (*>*) do controle de imagem por uma tag comum de caractere delimitador (*>*) e adicionar uma tag de fechamento *</Image>* para encapsular a propriedade *Image.Source*:

```
<Image Margin="0,0,0,0" Name="image1" >
 <Image.Source>
 <BitmapImage UriSource="bell.gif" />
 </Image.Source>
</Image>
```

O objetivo de um controle de imagem é exibir uma imagem. Você pode especificar a fonte da imagem de diversas maneiras. O exemplo mostrado aqui carrega a imagem a partir do arquivo *Bell.gif* que você acabou de adicionar como um recurso para o projeto.

A imagem agora deve aparecer no formulário, assim:



Mas há um problema. A imagem não está no fundo e ela esconde totalmente os dois controles de botão. A questão é que, a menos que você especifique algo diferente, todos os controles posicionados em um painel de layout têm uma ordem z implícita que exibe os controles adicionados por último na descrição XAML acima dos controles adicionados primeiro.



**Nota** O termo *ordem z* refere-se às posições relativas de profundidade dos itens no eixo z de um espaço tridimensional (sendo o eixo y vertical e o eixo y horizontal). Os itens com um valor mais alto para a ordem z aparecem na frente daqueles itens com um valor mais baixo.

Há pelo menos duas maneiras de mover o controle da imagem para trás dos botões. A primeira é deslocar as definições XAML dos botões para que elas apareçam depois do controle de imagem, e a segunda é especificar explicitamente um valor para a propriedade *ZIndex* do controle. Os controles com um valor *ZIndex* mais alto aparecem na frente daqueles no mesmo painel com um *ZIndex* mais baixo. Se dois controles tiverem o mesmo valor *ZIndex*, a precedência relativa será determinada pela ordem em que eles ocorrem na descrição XAML, como anteriormente.

6. No painel *XAML*, configure as propriedades *ZIndex* do botão e dos controles de imagem como mostrado em negrito no código a seguir:

```
<Button Panel.ZIndex="1" Content="Button" Margin="379,84,49,0"
 Name="button1" Height="23" VerticalAlignment="Top" />
<Button Panel.ZIndex="1" Content="Button" Height="23" Margin="0,0,0,0"
 Name="button2" Width="75" />
<Image Panel.ZIndex="0" Margin="0,0,0,0" Name="image1" >
 <Image.Source>
 <BitmapImage UriSource="Bell.gif" />
 </Image.Source>
</Image>
```

Os dois botões agora devem reaparecer na frente da imagem.

Com o WPF, você pode criar estilos para modificar a maneira como controles, por exemplo, botões, caixas de texto e rótulos, aparecem em um formulário. Você investigará esse recurso no próximo exercício.

### Crie um estilo para gerenciar a aparência e o comportamento dos controles no formulário

1. No painel *XAML*, modifique a definição do primeiro botão no formulário, como mostrado em negrito no código a seguir. A propriedade *Button.Resources* é outro exemplo de propriedade composta, e você deve modificar a definição do elemento *Button* para encapsular essa propriedade – substituindo o delimitador da tag de fechamento (*/>*) do controle de botão por um caractere comum de delimitador de tag (*>*) e adicionando uma tag de fechamento *</Button>*. É uma prática recomendável dividir a descrição XAML de um controle que contém valores de propriedades filhas compostas, como *Button.Resources*, em várias linhas, para facilitar a leitura e manutenção do código:

```
<Button Panel.ZIndex="1" Content="Button" Margin="169,84,34,0"
Name="button1" Height="23" VerticalAlignment="Top">
 <Button.Resources>
 <Style x:Key="buttonStyle">
 <Setter Property="Button.Background" Value="Gray"/>
 <Setter Property="Button.Foreground" Value="White"/>
 <Setter Property="Button.FontFamily" Value="Comic Sans MS"/>
 </Style>
 </Button.Resources>
</Button>
```

Este exemplo especifica os valores das cores de fundo e de frente de um botão, assim como a fonte utilizada no texto do botão. Estilos são recursos, e você os adiciona a um elemento *Resources* do controle. Para atribuir a cada estilo um nome exclusivo, utilize a propriedade *Key*.



**Nota** Quando você compila uma janela WPF, o Visual Studio adiciona todos os recursos que estão na janela a uma coleção associada a essa janela. Precisamente falando, a propriedade *Key* não especifica o nome do estilo, mas, em vez disso, um identificador para o recurso nessa coleção. Você também pode especificar a propriedade *Name* se quiser manipular o recurso no seu código C#, mas os controles referenciam os recursos indicando o valor *Key* para esse recurso. Controles e outros itens que você adiciona a um formulário devem ter a propriedade *Name* configurada, pois, como ocorre com recursos, é assim que você referencia esses itens no código.

Embora você tenha definido um estilo como parte da definição do botão, a aparência do botão não mudou. Para especificar um estilo a ser aplicado a um controle, utilize a propriedade *Style*.

2. Modifique a definição do botão, de modo a fazer referência ao estilo *buttonStyle*, como mostrado em negrito a seguir:

```
<Button Style="{DynamicResource buttonStyle}" Panel.ZIndex="1"
Content="Button" Margin ="169,84,34,0" Name="button1" Height="23"
VerticalAlignment="Top">
 <Button.Resources>
 <Style x:Key="buttonStyle">
 <!--
 </Style>
 </Button.Resources>
 Button
</Button>
```

A sintaxe `{DynamicResource buttonStyle}` cria um novo objeto de estilo baseado no estilo nomeado, e a propriedade *Style* aplica esse estilo ao botão. A aparência do botão no formulário já deve estar diferente.

Os estilos têm escopo. Se você tentar fazer referência ao estilo *buttonStyle* a partir do segundo botão no formulário, ele não surtirá qualquer efeito. Uma solução é criar uma cópia desse estilo e adicioná-la ao elemento *Resources* do segundo botão, e depois fazer a referência a esse estilo, como a seguir:

```
<Grid>
 <Button Style="{DynamicResource buttonStyle}" Content="Button"
 Panel.ZIndex="1" Margin="169,84,34,0" Name="button1" Height="23"
 VerticalAlignment="Top">
 <Button.Resources>
 <Style x:Key="buttonStyle">
 <Setter Property="Button.Background" Value="Gray"/>
 <Setter Property="Button.Foreground" Value="White"/>
 <Setter Property="Button.FontFamily" Value="Comic Sans MS"/>
 </Style>
 </Button.Resources>
 </Button>
 <Button Style="{DynamicResource buttonStyle}" Content="Button"
 Panel.ZIndex="1" Height="23" Margin="0,0,0,0" Name="button2"
 Width="76">
 <Button.Resources>
 <Style x:Key="buttonStyle">
 <Setter Property="Button.Background" Value="Gray"/>
 <Setter Property="Button.Foreground" Value="White"/>
 <Setter Property="Button.FontFamily" Value="Comic Sans MS"/>
 </Style>
 </Button.Resources>
 </Button>
 . . .
</Grid>
```

Entretanto, essa abordagem pode se tornar muito repetitiva e um pesadelo de manutenção se você precisar alterar o estilo dos botões. Uma estatística muito mais eficiente é definir o estilo como um recurso da janela, e depois você pode fazer uma referência a ele em todos os controles dessa janela.

3. No painel *XAML*, adicione um elemento *<Window.Resources>* acima da grade, mova a definição do estilo *buttonStyle* para esse novo elemento, e exclua o elemento *<Button.Resources>* do primeiro botão. Adicione ou modifique a propriedade *Style* dos dois botões, de modo a fazer referência a esse estilo. Incluímos a seguir o código atualizado da descrição XAML inteira do formulário, com a definição do recurso e as referências ao recurso em negrito:

```
<Window x:Class="BellRingers.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="Middleshire Bell Ringers Association - Members"
 Height="350" Width="525">
 <Window.Resources>
 <Style x:Key="buttonStyle">
 <Setter Property="Button.Background" Value="Gray"/>
 <Setter Property="Button.Foreground" Value="White"/>
 </Style>
 </Window.Resources>
 . . .

```

```

 <Setter Property="Button.FontFamily" Value="Comic Sans MS"/>
 </Style>
</Window.Resources>
<Grid>
 <Button Style="{StaticResource buttonStyle}" Panel.ZIndex="1"
 Content="Button" Margin="169,84,34,0" Name="button1" Height="23"
 VerticalAlignment="Top">
 </Button>
 <Button Style="{StaticResource buttonStyle}" Panel.ZIndex="1"
 Content="Button" Height="23" Margin="0,0,0,0" Name="button2"
 Width="76" />
 <Image Panel.ZIndex="0" Margin="0,0,0,0" Name="image1">
 <Image.Source>
 <BitmapImage UriSource="Bell.gif" />
 </Image.Source>
 </Image>
</Grid>
</Window>

```

Os dois botões agora aparecem na janela *Design View* com o mesmo estilo.

O código que você acabou de inserir referencia o estilo de botão utilizando *StaticResource* em vez da palavra-chave *DynamicResource*. As regras de escopo dos recursos estáticos são as mesmas daquelas do C#, pois elas exigem a definição de um recurso antes de você poder referenciá-lo. No passo 1 deste exercício, você referenciou o estilo *buttonStyle* acima do código XAML que o definiu, portanto, o nome do estilo não estava realmente no escopo. Essa referência fora de escopo funciona porque o uso de *DynamicResource* é adiado até o tempo de execução, momento em que a referência ao recurso é resolvida e o recurso já deve ter sido criado. Em termos gerais, recursos estáticos são mais eficientes que os dinâmicos porque eles são resolvidos quando o aplicativo é compilado, mas os recursos dinâmicos lhe dão mais flexibilidade.

Por exemplo, se o próprio recurso mudar enquanto o aplicativo executa (você pode escrever código para mudar os estilos em tempo de execução), controles que referenciam o estilo utilizando *StaticResource* não serão atualizados, mas os que referenciam o estilo usando *DynamicResource* serão.



**Nota** Há muitas outras diferenças entre o comportamento dos recursos estáticos e dinâmicos, e restrições sobre quando você pode referenciar um recurso dinamicamente. Para informações adicionais, consulte a documentação do .NET Framework fornecida com o Visual Studio 2010.

Ainda há um pouco de repetição envolvida na definição do estilo; cada uma das propriedades (fundo, primeiro plano e família de fontes) declara explicitamente que elas são propriedades de botões. Você pode remover essa repetição especificando o atribuído *TargetType* na tag *Style*.

4. Modifique a definição do estilo para especificar o atributo *TargetType*, e remova a referência *Button* de cada uma das propriedade, desta maneira:

```
<Style x:Key="buttonStyle" TargetType="Button">
 <Setter Property="Background" Value="Gray"/>
 <Setter Property="Foreground" Value="White"/>
 <Setter Property="FontFamily" Value="Comic Sans MS"/>
</Style>
```

Você pode adicionar ao formulário o número de botões que desejar e também estilizar todos eles empregando o estilo *buttonStyle*. Mas e os outros controles, como rótulos e caixas de texto?

5. Na janela *Design View*, clique no formulário *MainWindow* e então clique na guia *Toolbox*. Na seção *Common*, clique em *TextBox* e então clique em qualquer lugar na metade inferior do formulário.

O controle *TextBox* é adicionado ao formulário.

6. No painel *XAML*, altere a definição do controle de caixa de texto e especifique o atributo *Style* mostrado em negrito no exemplo a seguir, tentando aplicar o estilo *buttonStyle*:

```
<TextBox Style="{StaticResource buttonStyle}" Height="21"
Margin="114,0,44,58" Name="textBox1" VerticalAlignment="Bottom" />
```

Não surpreende que a tentativa de definir o estilo de uma caixa de texto com um estilo destinado a um *Button* venha a falhar. O painel *XAML* exibe uma linha sublinhada azul abaixo da referência ao estilo no controle *TextBox*. Se você posicionar o mouse sobre esse elemento, será exibida uma dica de ferramenta e a mensagem “‘Button’ TargetType does not match type of element ‘TextBox’” (O TargetType ‘Button’ não corresponde ao tipo de elemento ‘TextBox’). Se você tentar compilar o aplicativo, ocorrerá uma falha com a mesma mensagem de erro.

7. Para corrigir esse erro, no painel *XAML*, mude *TargetType* para *Control* na definição do estilo, altere a propriedade *Key* para *bellRingersStyle* (um nome mais significativo) e modifique as referências aos estilo nos controles de botão e caixa de texto, como mostrado em negrito a seguir:

```
<Window x:Class="BellRingers.MainWindow"
...
<Window.Resources>
 <Style x:Key="bellRingersStyle" TargetType="Control">
 <Setter Property="Background" Value="Gray"/>
 <Setter Property="Foreground" Value="White"/>
 <Setter Property="FontFamily" Value="Comic Sans MS"/>
 </Style>
</Window.Resources>
<Grid>
 <Button Style="{StaticResource bellRingersStyle}" ...>
 </Button>
 <Button Style="{StaticResource bellRingersStyle}" ... />
 ...
 <TextBox ... Style="{StaticResource bellRingersStyle}" ... />
</Grid>
</Window>
```

Configurar o atributo *TargetType* de um estilo como *Control* especifica que o estilo pode ser aplicado a qualquer controle herdado da classe *Control*. No modelo WPF, diferentes tipos de controles, incluindo caixas de texto e botões, herdam da classe *Control*. Mas você só pode fornecer elementos *Setter* para propriedades que pertencem especificamente à classe *Control*. (Botões têm algumas propriedades adicionais que não são parte da classe *Control*; se especificar uma dessas propriedades apenas de botão, você não poderá configurar *TargetType* como *Control*.)

- No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo. Digite algum texto na caixa de texto e verifique se ele aparece na cor branca utilizando a fonte *Comic Sans MS*.

Infelizmente, a escolha das cores dificulta, um pouco, ver o cursor de texto quando você clica na caixa de texto e digita o texto. Você corrigirá isso em um passo mais adiante.

- Fechе o formulário e retorne ao Visual Studio 2010.
- No painel *XAML*, edite o estilo *bellRingersStyle* e adicione o elemento *<Style.Triggers>* mostrado em negrito no código a seguir. (Se receber uma mensagem de erro de que *TriggerCollection* está selada, simplesmente recompile a solução.)

```
<Style x:Key="bellRingersStyle" TargetType="Control">
 <Setter Property="Background" Value="Gray"/>
 <Setter Property="Foreground" Value="White"/>
 <Setter Property="FontFamily" Value="Comic Sans MS"/>
 <Style.Triggers>
 <Trigger Property="IsMouseOver" Value="True">
 <Setter Property="Background" Value="Blue" />
 </Trigger>
 </Style.Triggers>
</Style>
```

Um gatilho (*trigger*) especifica uma ação a ser realizada quando o valor de uma propriedade muda. O estilo *bellRingersStyle* detecta uma alteração na propriedade *IsMouseOver* para modificar temporariamente a cor de fundo do controle sobre o qual o mouse está posicionado.



**Nota** Não confunda gatilhos com eventos. Os gatilhos respondem a alterações transitórias nos valores das propriedades. Quando o valor na propriedade com o gatilho for revertido ao inicial, a ação desencadeada será desfeita. No exemplo mostrado anteriormente, quando a propriedade *IsMouseOver* não é mais *true* para um controle, a propriedade *Background* é reconfigurada com o valor original. Já os eventos especificam uma ação a ser realizada quando um incidente significativo (como o usuário clicando em um botão) ocorre em um aplicativo; as ações realizadas por um evento não são desfeitas quando o incidente termina.

- No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo novamente. Dessa vez, ao movimentar o mouse sobre a caixa de texto, ela ficará azul para que você possa ver o cursor de texto mais facilmente. A caixa de texto volta a ficar cinza quando você afasta o mouse. Observe que os botões não se comportam exatamente da mesma maneira. Os controles de botão já implementam essa funcionalidade e adquirem um azul mais pálido

quando você posiciona o mouse sobre eles. Esse comportamento padrão redefine o gatilho especificado no estilo.

12. Feche o formulário e retorne ao Visual Studio 2010.



**Nota** Uma abordagem alternativa que você pode utilizar para aplicar uma fonte a todos os controles em um formulário é configurar as propriedades de texto da janela que contém os controles. Essas propriedades incluem *FontFamily*, *FontSize* e *FontWeight*. Mas estilos fornecem recursos adicionais, como os gatilhos, e você não fica restrito a configurar propriedades relacionadas a fontes. Se especificar as propriedades de texto para uma janela e aplicar um estilo aos controles na janela, o estilo dos controles tem precedência sobre as propriedades de texto da janela.

## Como um aplicativo WPF executa

Um aplicativo WPF pode conter vários formulários – você pode adicionar formulários a um aplicativo utilizando o comando *Add Window* no menu *Project* no Visual Studio 2010. Como um aplicativo sabe qual formulário exibir quando um aplicativo se inicia? Se lembrar do Capítulo 1, esse é o propósito do arquivo *Application.xaml*. Se abrir o arquivo *Application.xaml* do projeto *BellRingers*, você verá que ele se parece com isso:

```
<Application x:Class="BellRingers.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>
```

Ao compilar um aplicativo WPF, o compilador converte essa definição XAML em um objeto *Application*. Esse objeto controla o tempo de vida do aplicativo e é responsável por criar o formulário inicial que o aplicativo exibe. Você pode pensar no objeto *Application* como aquele que fornece o método *Main* para o aplicativo. A propriedade-chave é *StartupUri*, que especifica o arquivo XAML para a janela que o objeto *Application* deve criar. Quando você compila o aplicativo, essa propriedade é convertida em código que cria e abre o formulário WPF especificado. Se quiser exibir um formulário diferente, você simplesmente precisa alterar o valor da propriedade *StartupUri*.

É importante entender que a propriedade *StartupUri* referencia o nome do arquivo XAML e não a classe que implementa a janela nesse arquivo. Se renomear a classe a partir do padrão (*MainWindow*), o nome do arquivo não irá mudar (ele continua *MainWindow.xaml*). Correspondentemente, se você alterar o nome do arquivo, o nome da classe de janela definido nesse arquivo não mudará. Se a classe de janela e o arquivo XAML tiverem nomes diferentes, isso pode tornar-se confuso, portanto, se quiser renomear, seja consistente e altere tanto o nome do arquivo quanto o nome da classe de janela.

## Adicionando controles ao formulário

Até agora, você criou um formulário, configurou propriedades, adicionou alguns controles e definiu um estilo. Para tornar o formulário útil, você precisa adicionar mais controles e escrever algum código por conta própria, para implementar uma funcionalidade relevante. A biblioteca WPF contém uma coleção variada de controles. As finalidades de alguns são óbvias – por exemplo, *TextBox*, *ListBox*, *CheckBox* e *ComboBox* – enquanto outros controles mais poderosos podem não ser tão familiares.

## Utilizando controles WPF

No próximo exercício, você vai adicionar controles ao formulário que podem ser utilizados por um usuário para inserir detalhes sobre membros da associação de tocadores de sino. Para tanto, empregará uma variedade de controles, cada um deles ajustado para um tipo específico de entrada de dados.

Além disso, você vai usar os controles *TextBox* para inserir o nome e o sobrenome do membro. Cada membro pertence a uma “torre” (onde está o sino). O distrito de Middleshire tem diversas torres, mas a lista é estática – novas torres não são construídas com muita frequência e, felizmente, as torres antigas não costumam cair. O controle ideal para tratar esse tipo de dados é um *ComboBox*. O formulário também registra se o membro é o “capitão” da torre (a pessoa encarregada da torre que guia os outros sineiros). Um *CheckBox* é o melhor tipo de controle para isso; ele pode estar marcado (*True*) ou desmarcado (*False*).



**Dica** Os controles *CheckBox* podem na verdade ter três estados se a propriedade *IsThreeState* for definida como *True*. Os três estados são *true*, *false* e *null*. Esses estados são úteis se você está exibindo informações recuperadas de um banco de dados relacional. Algumas colunas em uma tabela de um banco de dados permitem valores *null*, indicando que o valor armazenado não está definido ou é desconhecido.

O aplicativo também reúne informações estatísticas sobre quando os membros ingressaram na associação e quanto tempo de experiência eles têm em tocar sinos (até 1 ano, entre 1 e 4 anos, entre 5 e 9 anos e 10 anos ou mais). Você pode utilizar um grupo de botões de opção (*radio buttons*) para indicar a experiência do membro. Eles fornecem um conjunto mutuamente exclusivo de valores. O WPF fornece o controle *DateTimePicker* para selecionar e exibir datas; esse controle é ideal para indicar a data em que o membro se uniu à associação.

Por fim, o aplicativo registra os tons que o membro pode tocar – de forma muito confusa, essas tonalidades são referenciadas como “métodos” pela fraternidade de sineiros. Embora um sineiro só toque um sino de cada vez, um grupo de sineiros sob a direção do capitão da torre pode tocar seus sinos em sequências diferentes e assim tocar uma música simples. Há uma variedade de métodos para tocar o sino, cujos nomes são bem curiosos, como Plain Bob, Reverse Canterbury, Grandsire, Stedman, Kent Treble Bob e Old Oxford Delight. Novos métodos estão sendo escritos com uma regu-

laridade alarmante, portanto, a lista de métodos pode variar ao longo do tempo. Em um aplicativo do mundo real, você armazenaria essa lista em um banco de dados. Neste aplicativo, você utilizará uma pequena seleção de métodos que embutirá no formulário. (Veremos como acessar e recuperar dados a partir de um banco de dados na Parte V deste livro, “Gerenciando dados”.) Um bom controle para exibir essas informações e indicar se um membro pode tocar um método é um *ListBox* contendo uma lista de controles *CheckBox*.

Após o usuário ter inserido os detalhes do membro, o botão *Add* validará e armazenará os dados. O usuário pode, então, clicar em *Clear* para redefinir os controles no formulário e cancelar qualquer entrada de dados.

## Adicione controles ao formulário

1. Verifique se *MainWindow.xaml* é exibido na janela *Design View*. Remova os dois controles de botão e o controle de caixa de texto do formulário.

 **Dica** Para remover um controle de um formulário, clique no controle e pressione a tecla **Delete**.

2. No painel *XAML*, altere a propriedade *Height* do formulário para **470** e a propriedade *Width* para **600**, como mostrado em negrito:

```
<Window x:Class="BellRingers.MainWindow"
 ...
 Title="..." Height="470" Width="600">
 ...
</Window>
```

3. Na janela *Design View*, clique no formulário *MainWindow*. A partir da *Toolbox*, arraste um controle *Label* sobre o formulário e o posicione próximo ao canto superior esquerdo. Não se preocupe em posicionar e dimensionar o rótulo precisamente porque você fará essa tarefa para vários controles mais adiante.

4. No painel *XAML*, altere o texto do rótulo para **First Name**, como mostrado em negrito:

```
<Label Content="First Name" ... />
```

 **Dica** Você também pode mudar o texto exibido por um rótulo e muitos outros controles configurando a propriedade *Content* na janela *Properties*.

5. Na janela *Design View*, clique no formulário *MainWindow*. A partir da *Toolbox*, arraste um controle *TextBox* sobre o formulário e posicione-o à direita do rótulo.



**Dica** Você pode usar as guias exibidas pela janela *Design View* para ajudar a alinhar os controles. (As linhas de guia são exibidas depois que você solta o controle no formulário.)

6. No painel *XAML*, altere a propriedade *Name* da caixa de texto para **firstName**, como mostrado em negrito:

```
<TextBox ... Name="firstName" .../>
```

7. Adicione um segundo controle *Label* ao formulário. Posicione-o à direita do caixa de texto *firstName*. No painel *XAML*, modifique a propriedade *Content* para alterar o texto do rótulo para **Last Name**.
8. Adicione um outro controle *TextBox* ao formulário e o posicione à direita do rótulo *Last Name*. No painel *XAML*, altere a propriedade *Name* dessa caixa de texto para **lastName**.
9. Adicione um terceiro controle *Label* ao formulário e o posicione diretamente sob o rótulo *First Name*. No painel *XAML*, mude o texto do rótulo para **Tower**.
10. Adicione um controle *ComboBox* ao formulário. Coloque-o sob a caixa de texto *firstName* e à direita do rótulo *Tower*. No painel *XAML*, altere a propriedade *Name* dessa caixa de combinação para **towerNames**.
11. Adicione um controle *CheckBox* ao formulário. Coloque-o sob a caixa de texto *lastName* e à direita do caixa de combinação *towerNames*. No painel *XAML*, mude a propriedade *Name* da caixa de seleção para **isCaptain** e altere o texto exibido por essa caixa de seleção para **Captain**.
12. Adicione um quarto *Label* ao formulário e o posicione sob o rótulo *Tower*. No painel *XAML*, altere o texto desse rótulo para **Member Since**.
13. Na *Toolbox*, expanda a seção *Controls*. Adicione um controle *DatePicker* ao formulário e posicione-o sob a caixa de combinação *towerNames*. Mude a propriedade *Name* desse controle para **memberSince**.
14. Adicione o controle *GroupBox* da seção *Controls* da *Toolbox* ao formulário e posicione-o sob o rótulo *Member Since*. No painel *XAML*, altere a propriedade *Name* da caixa de grupo para **yearsExperience** e mude a propriedade *Header* para **Experience**. A propriedade *Header* modifica o rótulo exibido no formulário da caixa de grupo. Defina a propriedade *Height* com 200.
15. Adicione um controle *StackPanel* ao formulário. No painel *XAML*, defina a propriedade *Margin* desse controle com “0,0,0,0”. Remova os valores de todas as outras propriedade, exceto da propriedade *Name*. O código do controle *StackPanel* deve ficar parecido com este:

```
<StackPanel Margin="0,0,0,0" Name="stackPanel1" />
```

16. No painel *XAML*, altere a definição do controle *yearsExperience GroupBox* e exclua os elementos *<Grid></Grid>*. Mova a definição do controle *StackPanel* para o código XAML do controle *GroupBox*, como a seguir:

```
<GroupBox Header="Experience" ... Name="yearsExperience" ...>
 <StackPanel Margin="0,0,0,0" Name="stackPanel1" />
</GroupBox>
```

17. Adicione um controle *RadioButton* ao formulário e posicione-o dentro do controle *StackPanel*, na parte superior. Adicione mais três controles *RadioButton* ao controle *StackPanel*. Eles devem ser automaticamente organizados na vertical.
18. No painel *XAML*, mude a propriedade *Name* de cada botão de opção e o texto exibido na propriedade *Content*, como mostrado em negrito a seguir:

```
<GroupBox ...>
 <StackPanel ...>
 <RadioButton ... Content="Up to 1 year" ... Name="novice" ... />
 <RadioButton ... Content="1 to 4 years" ... Name="intermediate" ... />
 <RadioButton ... Content="5 to 9 years" ... Name="experienced" ... />
 <RadioButton ... Content="10 or more years" ... Name="accomplished" ... />
 </StackPanel>
</GroupBox>
```

19. Adicione um controle *ListBox* ao formulário e o posicione à direita do controle *GroupBox*. No painel *XAML*, mude a propriedade *Name* da caixa de listagem para **methods**.
20. Adicione um controle *Button* ao formulário e o posicione próximo à parte inferior ou no lado esquerdo inferior do formulário, embaixo do controle *GroupBox*. No painel *XAML*, altere a propriedade *Name* desse botão para **add** e mude o texto exibido pela propriedade *Content* para **Add**.
21. Adicione um outro controle *Button* ao formulário e o posicione próximo à parte inferior à direita do botão *Add*. No painel *XAML*, mude a propriedade *Name* desse botão para **clear** e mude o texto exibido pela propriedade *Content* para **Clear**.

Você agora adicionou todos os controles requeridos pelo formulário. O próximo passo é organizar o layout. A tabela a seguir lista as propriedades de layout e os valores que você precisa atribuir a cada um dos controles. Na janela *Design View*, clique em um controle de cada vez e faça as alterações na janela *Properties*. As margens e o alinhamento dos controles são projetados para manter os controles no lugar se o usuário redimensionar o formulário. Também observe que os valores de margem especificados para os botões de opção são relativos a cada item precedente no controle *StackPanel* que os contém; o primeiro botão de opção está a 10 unidades a partir da parte superior do controle *StackPanel*, e os botões de opção restantes têm um espaço entre eles de 20 unidades verticalmente.

Controle	Propriedade	Valor
<i>label1</i>	<i>Height</i>	28
	<i>Margin</i>	29, 25, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
<i>firstName</i>	<i>Width</i>	75
	<i>Height</i>	23
	<i>Margin</i>	121, 25, 0, 0
	<i>VerticalAlignment</i>	Top
<i>label2</i>	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	175
	<i>Height</i>	28
	<i>Margin</i>	305, 25, 0, 0
<i>lastName</i>	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	75
	<i>Height</i>	23
<i>label3</i>	<i>Margin</i>	380, 25, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	175
<i>towerNames</i>	<i>Height</i>	28
	<i>Margin</i>	29, 72, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
<i>isCaptain</i>	<i>Width</i>	75
	<i>Height</i>	23
	<i>Margin</i>	121, 72, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	275
	<i>VerticalAlignment</i>	Top
	<i>Height</i>	420, 72, 0, 0

Controle	Propriedade	Valor
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	75
<i>Label4</i>	<i>Height</i>	28
	<i>Margin</i>	29, 134, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	90
<i>memberSince</i>	<i>Height</i>	23
	<i>Margin</i>	121, 134, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	275
<i>yearsExperience</i>	<i>Height</i>	200
	<i>Margin</i>	29, 174, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	258
<i>stackPanel1</i>	<i>Margin</i>	0,0,0,0
<i>novice</i>	<i>Height</i>	16
	<i>Margin</i>	0, 10, 0, 0
	<i>Width</i>	120
<i>intermediate</i>	<i>Height</i>	16
	<i>Margin</i>	0, 20, 0, 0
	<i>Width</i>	120
<i>experienced</i>	<i>Height</i>	16
	<i>Margin</i>	0, 20, 0, 0
	<i>Width</i>	120
<i>accomplished</i>	<i>Height</i>	16
	<i>Margin</i>	0, 20, 0, 0
	<i>Width</i>	120
<i>methods</i>	<i>Height</i>	200
	<i>Margin</i>	310, 174, 0, 0

(Continua)

Controle	Propriedade	Valor
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	245
<i>add</i>	<i>Height</i>	23
	<i>Margin</i>	188, 388, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	75
<i>clear</i>	<i>Height</i>	23
	<i>Margin</i>	313, 388, 0, 0
	<i>VerticalAlignment</i>	Top
	<i>HorizontalAlignment</i>	Left
	<i>Width</i>	75

Como um toque final, em seguida você aplicará um estilo aos controles. Você pode utilizar o estilo *bellRingersStyle* para controles como os botões e as caixas de texto, mas os rótulos, a caixa de combinação (combo box), caixa de grupo (group box) e os botões de opção (radio buttons) provavelmente não serão exibidos em um fundo cinza.

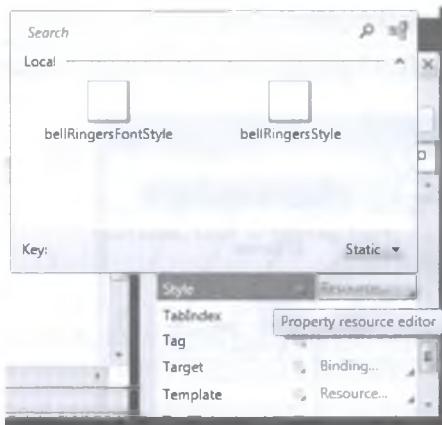
### Aplique estilos aos controles e teste o formulário

1. No painel *XAML*, adicione o estilo *bellRingersFontStyle*, mostrado em negrito no código a seguir, ao elemento *<Windows.Resources>*. Deixe o estilo *bellRingersStyle* existente como está. Observe que esse estilo só muda a fonte.

```
<Window.Resources>
 <Style x:Key="bellRingersFontStyle" TargetType="Control">
 <Setter Property="FontFamily" Value="Comic Sans MS"/>
 </Style>
 <Style x:Key="bellRingersStyle" TargetType="Control">

 </Style>
</Window.Resources>
```

2. No formulário, clique no controle *label1* *Label* que exibe o texto *First Name*. Na janela *Properties*, localize a propriedade *Style* desse controle. Clique no rótulo *Resource...* mostrado como valor dessa propriedade. É exibida uma lista dos estilos disponíveis como recursos no formulário, como mostra a imagem a seguir.



3. Verifique se a lista suspensa no canto inferior direito da caixa de listagem está definida com *Static*, e clique duas vezes em *bellRingersFontStyle*.
4. No painel *XAML*, verifique se o estilo *bellRingersFontStyle* foi aplicado ao controle *label1*, como mostrado em negrito a seguir:
 

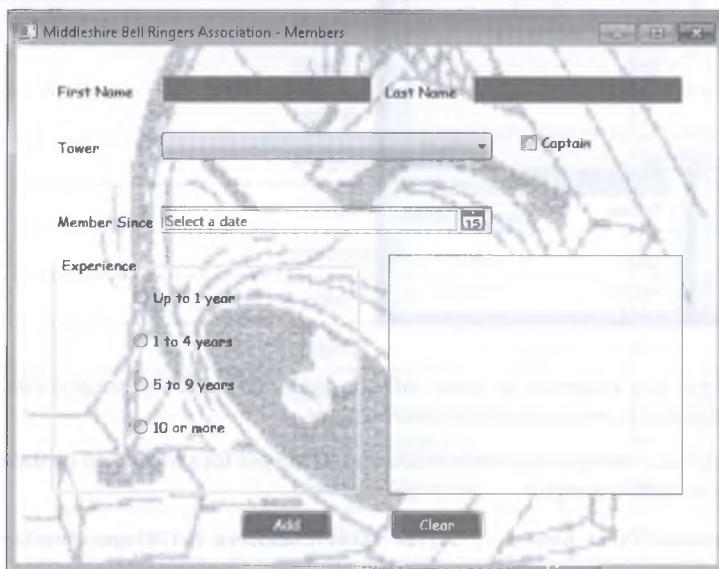
```
<Label Content="First Name" ... Style="{StaticResource bellRingersFontStyle}" />
```
5. Aplique o mesmo estilo aos seguintes controles. Você pode utilizar o editor de recursos de propriedades ou adicionar manualmente o estilo a cada controle, editando as definições XAML:
  - *label2*
  - *label3*
  - *isCaptain*
  - *towerNames*
  - *label4*
  - *yearsExperience*
  - *methods*

**Nota** Aplicar o estilo à caixa de grupo *yearsExperience* e à caixa de listagem *methods* faz o estilo ser utilizado automaticamente pelos itens exibidos nesses controles.

6. Aplique o estilo *bellRingersStyle* a estes controles:
  - *firstName*
  - *lastName*
  - *add*
  - *clear*

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

Quando o formulário for executado, ele ficará parecido com o da imagem a seguir:



Observe que a caixa de listagem *methods* está vazia. Você vai adicionar um código para preenchê-la em um exercício posterior.

- Clique na seta drop-down na caixa de combinação *Tower*. A lista das torres está vazia. Mais uma vez, você vai escrever um código para preencher essa caixa de combinação em um exercício a seguir.
- Feche o formulário e retorne ao Visual Studio 2010.

## Alterando as propriedades dinamicamente

Você já utilizou a janela *Design View*, a janela *Properties* e o painel *XAML* para configurar propriedades estaticamente. Quando o formulário for executado, será útil redefinir o valor de cada controle para um valor padrão inicial. Para fazer isso, você precisará escrever algum código (finalmente). Nos exercícios seguintes, você criará um método *private* chamado *Reset*. Posteriormente, você invocará esse método quando o formulário iniciar pela primeira vez e quando o usuário clicar no botão *Clear*.

### Crie o método *Reset*

- Na janela *Design View*, clique com o botão direito do mouse no formulário e então em *View Code*. A janela *Code and Text Editor* abre e exibe o arquivo *MainWindow.xaml.cs* para que você possa adicionar o código C# ao formulário.

2. Adicione o seguinte método *Reset*, mostrado em negrito, à classe *MainWindow*:

```
public partial class MainWindow : Window
{
 ...
 public void Reset()
 {
 firstName.Text = String.Empty;
 lastName.Text = String.Empty;
 }
}
```

Essas duas instruções garantem que as caixas de texto *firstName* e *lastName* estarão em branco atribuindo uma string vazia à sua propriedade *Text*.

Você também precisa inicializar as propriedades para os controles restantes no formulário e preencher a caixa de combinação *towerNames* e a caixa de listagem *methods*.

Se lembrar, a caixa de combinação *towerName* conterá uma lista de todas as torres de sino do distrito de Middleshire. Essa informação normalmente seria armazenada em um banco de dados e você escreveria um código para recuperar a lista de torres e preencher o *ComboBox*. Para esse exemplo, o aplicativo utilizará uma coleção codificada diretamente. Um *ComboBox* tem uma propriedade chamada *Items* que contém uma lista de dados a serem exibidos.

3. Adicione o seguinte array de strings chamado *towers*, mostrado em negrito, que contém uma lista codificada diretamente dos nomes das torres, à classe *MainWindow*:

```
public partial class MainWindow : Window
{
 private string[] towers = { "Great Shevington", "Little Mudford",
 "Upper Gumtree", "Downley Hatch" };
 ...
}
```

4. Adicione as seguintes instruções mostradas em negrito ao método *Reset*. Esse código limpa a caixa de combinação *towerName* (isso é importante porque do contrário você poderia acabar tendo muitos valores duplicados na lista) e adicione as torres localizadas no array *towers*. Uma caixa de combinação contém uma propriedade chamada *Items*, que possui uma coleção de itens a serem exibidos. A instrução depois do loop *foreach* faz a primeira torre ser exibida como o valor padrão:

```
public void Reset()
{
 ...
 towerNames.Items.Clear();
 foreach (string towerName in towers)
 {
 towerNames.Items.Add(towerName);
 }
 towerNames.Text = towerNames.Items[0] as string;
}
```



**Nota** Você também pode especificar valores codificados diretamente no tempo de projeto na descrição XAML de uma caixa de combinação, desta maneira:

```
<ComboBox Text="towerNames">
 <ComboBox.Items>
 <ComboBoxItem>
 Great Shevington
 </ComboBoxItem>
 <ComboBoxItem>
 Little Mudford
 </ComboBoxItem>
 <ComboBoxItem>
 Upper Gumtree
 </ComboBoxItem>
 <ComboBoxItem>
 Downley Hatch
 </ComboBoxItem>
 </ComboBox.Items>
</ComboBox>
```

5. Você deve preencher a caixa de listagem de métodos com uma lista de métodos para tocar sino. Como ocorre com uma caixa de combinação, uma caixa de listagem tem uma propriedade chamada *Items* que contém uma coleção dos valores a serem exibidos. Além disso, como a *ComboBox*, ela pode ser preenchida a partir de um banco de dados. Mas, como antes, para este exemplo, você apenas fornecerá alguns valores digitados diretamente no código. Adicione o seguinte array de string, mostrado em negrito, que contém a lista de métodos, à classe *MainWindow*:

```
public partial class MainWindow : Window
{
 ...
 private string[] ringingMethods = { "Plain Bob", "Reverse Canterbury",
 "Grandsire", "Stedman", "Kent Treble Bob", "Old Oxford Delight",
 "Winchendon Place", "Norwich Surprise", "Crayford Little Court" };
 ...
}
```

6. Para especificar quais métodos um membro poderá tocar, a caixa de listagem *methods* deve exibir uma lista de caixas de seleção em vez de strings de texto normais. Devido à flexibilidade do modelo WPF, você pode especificar vários tipos diferentes de conteúdo para controles como caixas de listagem e caixas de combinação. Adicione o seguinte código mostrado em negrito ao método *Reset* a fim de preencher a caixa de listagem *methods* com os métodos no array *ringingMethods*. Observe que desta vez cada item é uma caixa de seleção. Você pode especificar o texto exibido pela caixa de seleção configurando a propriedade *Content* e também definir o espaçamento entre itens na lista configurando a propriedade *Margin*; esse código insere um espaçamento de 10 unidades depois de cada item:

```
public void Reset()
{
 ...
 methods.Items.Clear();
```

```
CheckBox method = null;
foreach (string methodName in ringingMethods)
{
 method = new CheckBox();
 method.Margin = new Thickness(0, 0, 0, 10);
 method.Content = methodName;
 methods.Items.Add(method);
}
```

 **Nota** A maioria dos controles WPF tem uma propriedade *Content* que você pode utilizar para configurar e ler o valor exibido pelo controle. Na verdade, essa propriedade é um *object*, portanto, você pode configurá-la em quase qualquer tipo, contanto que faça sentido exibi-lo!

7. A caixa de seleção *isCaptain* deve assumir o padrão de falso. Para fazer isso, você precisa definir a propriedade *IsChecked*. Adicione a seguinte instrução mostrada em negrito ao método *Reset*:

```
public void Reset()
{

 isCaptain.IsChecked = false;
}
```

8. O formulário contém quatro botões de opção que indicam o número de anos de experiência como sineiro que o membro tem. Um botão de opção (*radio button*) é semelhante a uma *CheckBox* no fato de poder conter um valor *true* ou *false*. Mas o poder dos botões de opção aumenta quando você os agrupa em uma *GroupBox*. Nesse caso, os botões de opção formam uma coleção mutuamente exclusiva – no máximo, apenas um botão de opção em um grupo pode ser selecionado (definido como *true*) e todos os outros serão automaticamente desmarcados (definidos como *false*). Por padrão, nenhum dos botões estará selecionado. Você deve retificar isso definindo a propriedade *IsChecked* do botão de opção *novice*. Adicione a seguinte instrução mostrada em negrito ao método *Reset*:

```
public void Reset()
{

 novice.IsChecked = true;
}
```

9. Você deve assegurar de que o controle *member Since DatePicker* assuma o padrão da data atual. Você pode fazer isso configurando a propriedade *Text* do controle e obter a data atual a partir do método estático *Today* da classe *DateTime*.

Adicione o seguinte código mostrado em negrito ao método *Reset* para inicializar o controle *DatePicker*.

```
public void Reset()
{

 memberSince.Text = DateTime.Today.ToString();
}
```

10. Por fim, você precisa planejar o método *Reset* para ser chamado quando o formulário for exibido pela primeira vez. Um bom lugar para fazer isso é no construtor *MainWindow*. Insira uma chamada ao método *Reset* depois da instrução que chama o método *InitializeComponent*, como mostrado em negrito:

```
public MainWindow()
{
 InitializeComponent();
this.Reset();
}
```

11. No menu *Debug*, clique em *Start Without Debugging* para verificar se o projeto compila e executa.
12. Quando o formulário se abrir, clique na caixa de combinação *Tower*.

Você verá a lista de torres de sinos e poderá selecionar uma delas.

13. Clique no ícone do lado direito do selecionador de data/hora *Member Since*.

Você verá um calendário de datas. O valor padrão será a data atual. É possível clicar em uma data e utilizar as setas para selecionar um mês. Você pode também clicar no nome do mês para exibir os meses como uma lista e pode também clicar no ano para exibir uma lista de anos.

14. Clique em cada um dos botões de opção da caixa de grupo *Experience*.

Observe que você não pode selecionar mais de um botão de opção por vez.

15. Na caixa de listagem *Methods*, clique em algum método para marcar a caixa de seleção correspondente. Se clicar em um método uma segunda vez, limpará a caixa de seleção correspondente, assim como você esperaria.

16. Clique nos botões *Add* e *Clear*.

Neste momento, esses botões não fazem coisa alguma. Você adicionará essa funcionalidade no conjunto final de exercícios deste capítulo.

17. Feche o formulário e retorne ao Visual Studio 2010.

## Tratando eventos em um formulário WPF

Se está familiarizado com o Microsoft Visual Basic, Microsoft Foundation Classes (MFC) ou qualquer uma das outras ferramentas disponíveis para a criação de aplicativos GUI para o Windows, você sabe que o Windows utiliza um modelo orientado a eventos para determinar quando executar o código. No Capítulo 17, “Interrompendo o fluxo do programa e tratando eventos”, você viu como publicar seus eventos e inscrever-se neles. Os formulários e controles WPF têm seus próprios eventos predefinidos para os quais você pode se inscrever, e esses eventos devem ser suficientes para tratar os requisitos da maioria das interfaces com o usuário.

## Processando eventos no Windows Forms

A tarefa do desenvolvedor é capturar os eventos que são relevantes para o aplicativo e escrever o código que responde a esses eventos. Um exemplo familiar é o controle *Button*, que dispara um evento “Alguém clicou em mim” quando um usuário clica nele com o mouse ou pressiona Enter quando o botão tem o foco. Se você quiser que o botão faça algo, escreva o código que responde a esse evento. É isso que faremos no próximo exercício.

### Manipule os eventos *Click* para o botão *Clear*

1. Exiba o arquivo MainWindow.xaml na janela *Design View*. Dê um clique duplo no botão *Clear* no formulário.



**Nota** Após modificar o código que respalda um formulário WPF e compilar o aplicativo, na próxima vez em que você exibir o formulário na janela *Design View*, ele poderá apresentar a seguinte mensagem no início da janela: “An assembly or related document has been updated which requires the designer to be reloaded. Click here to reload” (Foi atualizado um assembly ou um documento relacionado, que exibe o recarregamento do designer. Clique aqui para recarregar). Se isso acontecer, clique na mensagem e deixe o formulário ser recarregado.

A janela *Code and Text Editor* aparece e cria um método chamado *clear\_Click*. Esse é um método de evento que será invocado quando o usuário clicar no botão *Clear*. Observe que esse método recebe dois parâmetros: o parâmetro *sender* (um *object*) e um parâmetro adicional de argumentos (um objeto *RoutedEventArgs*). O runtime do WPF preencherá esses parâmetros com informações sobre a origem do evento e quaisquer informações adicionais que possam ser úteis para tratar o evento. No entanto, você não utilizará esses parâmetros neste exercício.

Os controles WPF podem disparar uma variedade de eventos. Quando você dá um clique duplo em um controle ou em um formulário na janela *Design View*, o Visual Studio gera o stub (esqueleto de código) de um método de evento para o evento padrão do controle; para um botão, o evento padrão é o evento *Click*. (Se der um clique duplo em um controle de caixa de texto, o Visual Studio irá gerar o stub de um método de evento para tratar o evento *TextChanged*.)

2. Quando o usuário clicar no botão *Clear*, você quer que o formulário seja redefinido com os seus valores padrão. No corpo do método *clear\_Click*, chame o método *Reset*, como mostrado em negrito:

```
private void clear_Click(object sender, RoutedEventArgs e)
{
 this.Reset();
}
```

Os usuários clicarão no botão *Add* quando tiverem preenchido todos os dados de um membro e quiserem armazenar as informações. O evento *Click* do botão *Add* deve validar as informações inseridas para garantir que elas façam sentido (por exemplo, você deve permitir que um capitão de torre tenha menos de um ano de experiência?) e, se estiver tudo certo, organizar os dados

para serem enviados para um banco de dados ou outro armazenamento persistente. Você aprenderá mais sobre validação e armazenamento de dados nos capítulos posteriores. Por enquanto, o código para o evento *Click* do botão *Add* simplesmente exibirá uma caixa de mensagem simulando a entrada de dados.

3. Retorne à janela *Design View* para exibir o formulário *MainWindow.xaml*. No painel *XAML*, localize o elemento que define o botão *Add* e comece a inserir o seguinte código mostrado em negrito:

```
<Button Content="Add" ... Click= />
```

Observe que, quando você digita o caractere `=`, um menu de atalho aparece, exibindo dois itens: `<New Event Handler>` e `clear_Click`. Se dois botões realizarem uma ação comum, você poderá compartilhar o mesmo método de rotina de tratamento de evento entre eles, como `clear_Click`. Se quiser gerar um método inteiramente novo de tratamento de evento, selecione em vez disso o comando `<New Event Handler>`.

4. No menu de atalho, dê um clique duplo no comando `<New Event Handler>`.

O texto `add_Click` aparece no código XAML do botão.



**Nota** Você não está limitado ao tratamento do evento *Click* de um botão. Quando você edita o código XAML para um controle, a lista do IntelliSense exibe as propriedades e os eventos para o controle. Para tratar um evento além do evento *Click*, simplesmente digite o nome do evento e então selecione ou digite o nome do método que você quer que o trate. Para uma lista completa dos eventos suportados por cada controle, veja a documentação do Visual Studio 2010.

5. Mude para a janela *Code and Text Editor* que exibe o arquivo *MainWindow.xaml.cs*. Observe que o método `add_Click` foi adicionado à classe *MainWindow*.



**Dica** Não é necessário utilizar os nomes padrão gerados pelo Visual Studio 2010 para os métodos de rotina de tratamento de evento. Em vez de clicar no comando `<New Event Handler>` no menu de atalho, você pode simplesmente digitar o nome de um método. Mas você precisará adicionar manualmente o método à classe de janela. Esse método deve ter a assinatura correta; ele deve retornar um `void` e receber dois argumentos – um parâmetro `object` e um parâmetro `RoutedEventArgs`.



**Importante** Se a seguir decidir remover um método de evento, como `add_Click` do arquivo *MainWindow.xaml.cs*, você também deve editar a definição XAML do controle correspondente e remover a referência `Click="add_Click"` ao evento; do contrário, seu aplicativo não compilará.

6. Adicione o seguinte código mostrado em negrito ao método *add\_Click*:

```
private void add_Click(object sender, RoutedEventArgs e)
{
 string nameAndTower = String.Format(
 "Member name: {0} {1} from the tower at {2} rings the following methods:",
 firstName.Text, lastName.Text, towerNames.Text);

 StringBuilder details = new StringBuilder();
 details.AppendLine(nameAndTower);

 foreach (CheckBox cb in methods.Items)
 {
 if (cb.IsChecked.Value)
 {
 details.AppendLine(cb.Content.ToString());
 }
 }

 MessageBox.Show(details.ToString(), "Member Information");
}
```

Esse bloco de código cria uma variável *string* chamada *nameAndTower* que é preenchida com o nome do membro e a torre à qual o membro pertence.

Observe como o código acessa a propriedade *Text* dos controles da caixa de texto e da caixa de combinação para ler os valores atuais desses controles. Além disso, o código utiliza o método estático *String.Format* para formatar o resultado. Esse método opera de modo semelhante ao método *Console.WriteLine*, exceto por ele retornar a *string* formatada como resultado em vez de exibi-la na tela.

O código cria então um objeto *StringBuilder* chamado *details*. O método utiliza esse objeto para construir uma representação de *string* das informações que ele exibirá. O texto na *string* *nameAndTower* é utilizado inicialmente para preencher o objeto *details*. O código em seguida itera pela coleção *Items* na caixa de listagem *methods*. Se lembrar, essa caixa de listagem contém controles de caixa de seleção. Cada caixa de seleção é examinada uma a uma e, se o usuário selecionou uma delas, o texto na propriedade *Content* da caixa de seleção é acrescentado ao objeto *details* *StringBuilder*. Existe uma pequena reviravolta aqui. Lembre-se de que uma *CheckBox* pode ser definida com *true*, *false* ou *null*. Na realidade, a propriedade *IsChecked* retorna um valor nullable *bool?*. Você acessa o valor booleano da propriedade *IsChecked* por meio da propriedade *Value*.

Por fim, a classe *MessageBox* fornece métodos estáticos para exibir as caixas de diálogo na tela. O método *Show* utilizado aqui exibirá o conteúdo da *string* *details* no corpo da mensagem e colocará o texto “Member Information” na barra de título. Esse é um método sobrecarregado, e existem outras variantes que podem ser usadas para especificar ícones e botões para a caixa de mensagem.



**Nota** Você poderia utilizar concatenação normal de strings em vez de um objeto *StringBuilder*, mas a classe *StringBuilder* é muito mais eficiente, além de ser a abordagem recomendável para realizar o tipo de tarefa requerido nesse código. No .NET Framework e no C#, o tipo de dados *string* é imutável; quando você modifica o valor em uma *string*, o runtime na verdade cria uma nova *string* contendo o valor modificado e então descarta a *string* antiga. A modificação repetida de uma *string* pode fazer seu código se tornar ineficiente porque uma nova *string* deve ser criada na memória a cada alteração (as *strings* antigas acabarão sofrendo coleta de lixo). A classe *StringBuilder*, no namespace *System.Text*, é projetada para evitar essa ineficiência. Você pode adicionar e remover os caracteres de um objeto *StringBuilder* usando os métodos *Append*, *Insert* e *Remove* sem criar um novo objeto a cada vez.

7. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
8. Digite alguns dados de exemplo para o primeiro nome e sobrenome do membro, selecione uma torre e alguns métodos. Clique no botão *Add* e verifique se a caixa de mensagem *Member Information* aparece, exibindo os detalhes do novo membro e os métodos que ele pode tocar. Na caixa de mensagem *Member Information*, clique em *OK*.
9. Clique no botão *Clear* e verifique se os controles no formulário são reinicializados para os valores padrão corretos.
10. Feche o formulário e retorne ao Visual Studio 2010.

No exercício final deste capítulo, você adicionará uma rotina de tratamento de evento para tratar o evento *Closing* da janela para que os usuários possam confirmar se eles realmente querem fechar o aplicativo. Esse evento é disparado quando o usuário tenta fechar o formulário, mas antes de o formulário fechar realmente. Você pode utilizá-lo para solicitar que o usuário salve todos os dados não gravados ou mesmo perguntar se o usuário quer realmente fechar o formulário – se não quiser, você pode cancelar o evento na rotina de tratamento de eventos e impedir o fechamento do formulário.

### Manipule o evento *Closing* para o formulário

1. Na janela *Design View*, no painel *XAML*, comece inserindo o código mostrado em negrito para a descrição XAML da janela *MainWindow*:

```
<Window x:Class="BellRingers.MainWindow"
 ...
 Title="..." ... Closing=>
```

2. Quando o menu de atalho aparecer depois que você digitar a aspa de abertura, dê um clique duplo no comando *<New Event Handler>*.

O Visual Studio gera um método de evento chamado *Window\_Closing* e o associa ao evento *Closing* do formulário, assim:

```
<Window x:Class="BellRingers.MainWindow"
 ...
 Title="..." ... Closing="Window_Closing">
```

3. Mude para a janela *Code and Text Editor* que exibe o arquivo *MainWindow.xaml.cs*.

Um stub para o método de evento *Window\_Closing* foi adicionado à classe *MainWindow*:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{}
```

Observe que o segundo parâmetro para esse método tem o tipo *CancelEventArgs*. A classe *CancelEventArgs* tem uma propriedade booleana chamada *Cancel*. Se você definir *Cancel* como *true* na rotina de tratamento de eventos, o formulário não fechará. Se definir *Cancel* como *false* (o valor padrão), o formulário fechará quando a rotina de tratamento de eventos terminar.

4. Adicione as seguintes instruções mostradas em negrito ao método *memberFormClosing*:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
 MessageBoxResult key = MessageBox.Show(
 "Are you sure you want to quit",
 "Confirm",
 MessageBoxButton.YesNo,
 MessageBoxImage.Question,
 MessageBoxResult.No);
 e.Cancel = (key == MessageBoxResult.No);
}
```

Essas instruções exibem uma caixa de mensagem pedindo ao usuário para confirmar se quer sair do aplicativo. A caixa de mensagem conterá os botões *Yes* e *No* e um ícone de ponto de interrogação. O parâmetro final, *MessageBoxResult.No*, indica o botão padrão se o usuário simplesmente pressionar a tecla Enter – é mais seguro supor que o usuário não quer fechar o aplicativo do que correr o risco de perder accidentalmente os detalhes que o usuário acabou de digitar. Quando o usuário clicar em um dos dois botões, a caixa de mensagem fechará, e o botão clicado retornará como o valor do método (como um *MessageBoxResult* – uma enumeração identificando qual botão foi clicado). Se o usuário clicar em *No*, a segunda instrução definirá a propriedade *Cancel* do parâmetro *CancelEventArgs* (*e*) para *true*, impedindo que o formulário se feche.

5. No menu *Debug*, clique em *Start Without Debugging* para executar o aplicativo.  
6. Tente fechar o formulário. Na caixa de mensagem que aparece, clique em *No*.  
O formulário deve continuar executando.  
7. Tente fechar o formulário novamente. Desta vez, na caixa de mensagem, clique em *Yes*.  
O formulário se fecha e o aplicativo termina.

Neste capítulo, vimos como utilizar os recursos essenciais do WPF para construir uma interface funcional com o usuário. O WPF contém vários outros recursos para o pequeno espaço que temos aqui, especialmente em relação a algumas das suas interessantes capacidades para tratar animação e elementos gráficos bidimensionais e tridimensionais. Se quiser aprender mais sobre o WPF, consulte um livro como *Applications = Code+Markup: A Guide to the Microsoft Windows Presentation Foundation*, de Charles Petzold (Microsoft Press, 2006).

- Se você quiser seguir para o próximo capítulo agora:  
Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 23.
- Se você quiser sair do Visual Studio 2010:  
No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 22

Para	Faça isto
Criar um aplicativo WPF	Use o modelo WPF Application.
Adicionar controles a um formulário	Arraste o controle da <i>Toolbox</i> para o formulário.
Alterar as propriedades de um formulário ou de um controle	<p>Clique no formulário ou no controle na janela <i>Design View</i>. Realize então um dos seguintes procedimentos:</p> <ul style="list-style-type: none"> <li>■ Na janela <i>Properties</i>, selecione a propriedade que você quer alterar e insira o novo valor.</li> <li>■ No painel <i>XAML</i>, especifique a propriedade e o valor no elemento <i>&lt;Window&gt;</i> ou no elemento que define o controle.</li> </ul>
Exibir o código referente a um formulário	<p>Realize um dos seguintes procedimentos:</p> <ul style="list-style-type: none"> <li>■ No menu <i>View</i>, clique em <i>Code</i>.</li> <li>■ Clique com o botão direito do mouse na janela <i>Design View</i> e então clique em <i>View Code</i>.</li> <li>■ No <i>Solution Explorer</i>, expanda a pasta correspondente ao arquivo <i>.xaml</i> para o formulário e então dê um clique duplo no arquivo <i>.xaml.cs</i> que aparece.</li> </ul>
Definir um conjunto de botões de opção mutuamente exclusivos	<p>Adicione um controle de painel, como <i>StackPanel</i>, ao formulário. Adicione os botões de opção (radio button) ao painel. Os botões de opção que estiverem no mesmo painel são mutuamente exclusivos.</p>
Preencher uma caixa de combinação ou uma caixa de listagem utilizando código C#	<p>Use o método <i>Add</i> da propriedade <i>Items</i>. Por exemplo:</p> <pre>towerNames.Items.Add("Upper Gumtree");</pre> <p>Você talvez precise limpar primeiramente a propriedade <i>Items</i>, quer queira ou não manter o conteúdo existente da lista. Por exemplo:</p> <pre>towerNames.Items.Clear();</pre>
Inicializar um controle de caixa de seleção (check box) ou um controle de botão de opção	<p>Defina a propriedade <i>IsChecked</i> como <i>true</i> ou <i>false</i>. Por exemplo:</p> <pre>novice.IsChecked = true;</pre>
Tratar um evento de um controle ou formulário	No painel <i>XAML</i> , adicione código para especificar o evento, após selecionar um método existente que tem a assinatura apropriada ou clique no comando <i>&lt;Add New Event&gt;</i> no menu de atalho que aparece e então escreva o código que trata o evento no método de evento que é criado.

## Capítulo 23

# Obtendo a entrada do usuário

Neste capítulo, você vai aprender a:

- Criar menus para aplicativos Microsoft Windows Presentation Foundation (WPF) utilizando as classes *Menu* e *MenuItem*.
- Realizar o processamento em resposta a eventos de menu quando um usuário clica em um comando de menu.
- Criar menus pop-up sensíveis ao contexto utilizando a classe *ContextMenu*.
- Manipular menus por meio de código e criar menus dinâmicos.
- Utilizar caixas de diálogo comuns do Windows em um aplicativo para solicitar ao usuário o nome de um arquivo.
- Construir aplicativos WPF que podem se beneficiar de diversas threads para melhorar a capacidade de resposta.

No Capítulo 22, “Apresentando o Windows Presentation Foundation”, vimos como criar um aplicativo WPF simples composto de uma seleção de controles e eventos. Muitos aplicativos profissionais baseados em Microsoft Windows também fornecem menus contendo comandos e opções, dando ao usuário a capacidade de executar várias tarefas relacionadas ao aplicativo. Neste capítulo, você aprenderá a criar menus e adicioná-los aos formulários utilizando o controle *Menu*. Você verá como responder quando o usuário clica em um comando de um menu. Entenderá como criar menus pop-up cujos conteúdos variam de acordo com o contexto atual. Finalmente, compreenderá as classes de diálogo comuns fornecidas como parte da biblioteca WPF. Com essas classes de diálogo, você pode solicitar ao usuário itens frequentemente utilizados, como arquivos e impressoras, de maneira rápida, fácil e familiar.

## Diretrizes e estilos de menu

Se examinar a maioria dos aplicativos baseados em Windows, você verá que alguns itens da barra de menus tendem a aparecer sempre no mesmo lugar, e o conteúdo desses itens é previsível. Por exemplo, o menu *File* normalmente é o primeiro item na barra de menus e nesse menu você vai encontrar comandos para criar um novo documento, abrir um documento existente, salvá-lo, imprimi-lo e sair do aplicativo.

**Nota** O termo *documento* engloba os dados que o aplicativo manipula. No Microsoft Office Excel, é uma planilha; no aplicativo BellRingers que você criou no Capítulo 22, poderia ser os detalhes de um novo membro.

A ordem em que esses comandos aparecem tende a ser a mesma em todos os aplicativos; por exemplo, o comando *Exit* é invariavelmente o último comando do menu *File*. Também podem existir no menu *File* outros comandos específicos do aplicativo.

Um aplicativo geralmente tem um menu *Edit* que contém comandos, como *Cut*, *Paste*, *Clear* e *Find*. Normalmente existem menus adicionais específicos do aplicativo na barra de menus, mas, novamente, a convenção determina que o menu final seja o menu *Help*, que contém acesso ao sistema de Ajuda para seu aplicativo, assim como informações “sobre”, que contêm detalhes de licença e direitos autorais do aplicativo. Em um aplicativo bem projetado, a maioria dos menus é previsível e ajuda a garantir que o aplicativo seja fácil de aprender e utilizar.

**Dica** A Microsoft publica um conjunto completo de diretrizes para criar interfaces com o usuário, incluindo o design de menus, no site da Microsoft na Web em <http://msdn2.microsoft.com/en-us/library/Aa286531.aspx>.

## Menus e eventos de menu

O WPF fornece o controle *Menu* como um contêiner para itens de menu. Esse controle fornece um shell básico para definir um menu e, como ocorre com a maioria dos aspectos do WPF, ele é muito flexível, de forma a permitir que você defina uma estrutura de menus contando em praticamente qualquer tipo de controle WPF. Provavelmente você já conhece menus que contenham itens de texto nos quais é possível clicar para executar um comando. Os menus WPF também podem conter botões, caixas de texto, caixas de combinação, etc. Você pode definir menus utilizando o painel *XAML* na janela *Design View* bem como construir menus em tempo de execução utilizando código em Microsoft Visual C#. A disposição de um menu é apenas metade da história. Quando um usuário clica em um comando em um menu, ele espera que algo aconteça! Seu aplicativo atua sobre os comandos capturando eventos de menu e executando um código quase da mesma maneira como trata os eventos de controle.

## Criando um menu

No próximo exercício, você utilizará o painel *XAML* para criar menus para o aplicativo da Middleshire Bell Ringers Association. Veremos como manipular e criar menus por meio de código mais adiante neste capítulo.

### Crie o menu do aplicativo

1. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra a solução *BellRingers*, localizada na pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 23\BellRingers* na sua pasta Documentos. Essa é uma cópia do aplicativo que você construiu no Capítulo 22.

3. Exiba o arquivo MainWindow.xaml na janela *Design View*.
4. Na *Toolbox*, arraste um controle *DockPanel* a partir da seção *All WPF Controls* para qualquer lugar no formulário. Certifique-se de soltá-lo sobre o formulário e não sobre um dos controles existentes no formulário. Na janela *Properties*, configure a propriedade *Width* do *DockPanel* como **Auto**, configure a propriedade *HorizontalAlignment* como *Stretch*, a propriedade *VerticalAlignment* como **Top**, e a propriedade *Margin* como **0**.

 **Nota** Configurar a propriedade *Margin* como **0** é o mesmo que configurá-la como **0, 0, 0, 0**.

O controle *DockPanel* deve aparecer na parte superior do formulário, ocupando toda a largura dele. (Ele abrangerá os elementos da interface com o usuário *First Name*, *Last Name*, *Tower* e *Captains*.)

O controle *DockPanel* é um controle de painel que você pode utilizar para controlar a disposição de outros controles adicionados, como os controles *Grid* e *StackPanels* que vimos no Capítulo 22. Você pode adicionar diretamente um menu a um formulário, mas uma prática melhor é posicioná-lo em um *DockPanel* porque você então pode manipular mais facilmente o menu e o posicionamento no formulário. Por exemplo, se quiser posicionar o menu na parte inferior ou em um dos lados, você pode reposicionar o menu inteiro em outra parte no formulário simplesmente movendo o painel em tempo de projeto ou em tempo de execução por meio de código.

5. Na *Toolbox*, arraste um controle *Menu* a partir da seção *All WPF Controls* até o controle *DockPanel*. Na janela *Properties*, configure a propriedade *DockPanel.Dock* como **Top**, a propriedade *Width* como **Auto**, a propriedade *HorizontalAlignment* como **Stretch**, e a propriedade *VerticalAlignment* como **Top**.

O controle *Menu* aparece como uma barra cinza ao longo da parte superior do *DockPanel*. Se examinar o código para os controles *DockPanel* e *Menus* no painel *XAML*, eles devem se parecer com isso:

```
<DockPanel Height="100" HorizontalAlignment="Stretch" Margin="0"
Name="dockPanel1" VerticalAlignment="Top" Width="Auto">
 <Menu Height="23" Name="menu1" Width="Auto" DockPanel.Dock="Top"
 VerticalAlignment="Top"/>
</DockPanel>
```

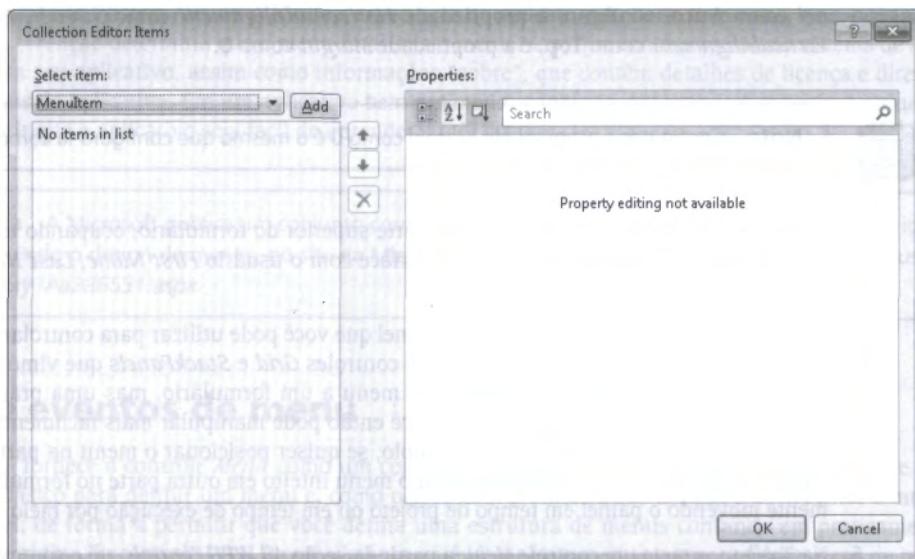
A propriedade *HorizontalAlignment* não aparece no código *XAML* porque o valor “*Stretch*” é o valor padrão para essa propriedade.

 **Nota** Em todo este capítulo, as linhas do painel *XAML* são mostradas divididas e recuadas para caber na página impressa.

6. Clique no controle *Menu* no formulário. Na janela *Properties*, localize a propriedade *Items*. O valor dessa propriedade é informado como (*Collection*). Um controle *Menu* contém uma

coleção de elementos *MenuItem*. Atualmente, o menu não tem quaisquer itens de menu, de modo que a coleção está vazia. Clique no botão de reticências (...) ao lado do valor.

A caixa de diálogo *Collection Editor: Items* é exibida, como na imagem a seguir:



7. Na caixa de diálogo *Collection Editor: Items*, clique em *Add*. Um novo elemento *MenuItem* é criado e aparece na caixa de diálogo. No painel *Properties*, defina a propriedade *Header* como \_File (incluindo o sublinhado).

O atributo *Header* do elemento *MenuItem* especifica o texto que aparece para o item de menu. O sublinhado () na frente de uma letra fornece acesso rápido a esse item de menu quando o usuário pressiona a tecla Alt e a letra depois do sublinhado (nesse caso, Alt+F, de “File”). Essa é outra convenção comum. Em tempo de execução, quando o usuário pressiona a tecla Alt, o F no início de File aparece com um sublinhado. Não use a mesma tecla de acesso mais de uma vez em nenhum menu porque você confundirá o usuário (e provavelmente o aplicativo).

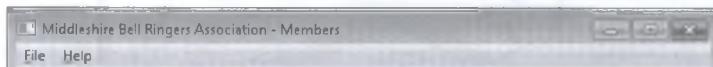
8. Clique em *Add* novamente. No painel *Properties*, defina a propriedade *Header* do segundo elemento *MenuItem* como \_Help, e clique em **OK** para fechar a caixa de diálogo.
9. No painel *XAML*, examine a definição do controle *Menu*, que deve estar parecida com esta (os novos itens aparecem em negrito):

```
<Menu Height="22" Name="menu1" Width="Auto" DockPanel.Dock="Top">
 VerticalAlignment="Top" HorizontalAlignment="Stretch" >
 <MenuItem Header="_File" />
 <MenuItem Header="_Help" />
 </Menu>
```

Observe que os elementos de *MenuItem* aparecem como itens filhos do controle *Menu*. Para criar itens de menu, digite o código diretamente no painel XAML, em vez de utilizar a caixa de diálogo *Collection Editor*, se preferir.

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

Quando o formulário aparece, você deve ver o menu na parte superior da janela embaixo da barra de título. Pressione a tecla Alt; o menu deve receber o foco e o "F" em "File" e o "H" em "Help" devem conter um sublinhado, assim:



Se clicar em um item de menu, nada acontece atualmente porque você não definiu os menus filhos que cada um desses itens conterá.

- Feche o formulário e retorne ao Visual Studio 2010.
- No painel *XAML*, modifique a definição do item do menu *\_File*, remova o caractere *" /"* do final da tag e adicione os itens do menu secundário com um elemento de fechamento *</MenuItem>* como mostrado em negrito:

```
<MenuItem Header="_File" >
 <MenuItem Header="_New Member" Name="newMember" />
 <MenuItem Header="_Save Member Details" Name="saveMember" />
 <Separator/>
 <MenuItem Header="E_xit" Name="exit" />
</MenuItem>
```

Esse código XAML adiciona *New Member*, *Save Member Details* e *Exit* como comandos ao menu *File*. O elemento *<Separator/>* aparece como uma barra quando o menu é exibido e é convenциionalmente utilizado para agrupar itens de menu relacionados. Exceto o separador, cada item de menu também recebe um nome porque você precisará fazer referência a eles, mais adiante, em seu aplicativo.

**Dica** Você também pode adicionar itens de menu filhos a um elemento *MenuItem*, na caixa de diálogo *Collection Editor: Items*. Como acontece com o controle *Menu*, cada elemento *MenuItem* tem uma propriedade chamada *Items*, que é uma coleção de elementos *MenuItem*. Você pode clicar no botão de reticências, que aparece na propriedade *Items* no painel *Properties* de um elemento *MenuItem*, para abrir outra instância da caixa de diálogo *Collection Editor: Items*. Os itens que você adicionar serão exibidos como itens filhos do elemento *MenuItem*.

- Modifique a definição do item de menu *\_Help* e adicione esse item de menu secundário como mostrado em negrito:

```
<MenuItem Header="_Help" >
 <MenuItem Header="_About Middleshire Bell Ringers" Name="about" />
</MenuItem>
```

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

Quando o formulário aparecer, clique no menu *File*. Você deve ver os itens do menu secundário, assim:



Você também pode clicar no menu *Help* para exibir o item de menu secundário *About Middleshire Bell Ringers*.

Entretanto, nenhum dos itens de menu filhos faz qualquer coisa quando você clica neles. Na próxima seção, veremos como associar itens de menu a ações.

15. Feche o formulário e retorne ao Visual Studio 2010.

Como mais um toque, você pode adicionar ícones a itens de menu. Muitos aplicativos, incluindo o Visual Studio 2010, utilizam ícones nos menus para fornecer uma legenda visual adicional.

16. No *Solution Explorer*, clique com o botão direito do mouse no projeto *BellRingers*, aponte para *Add* e então clique em *Existing Item*. Na caixa de diálogo *Add Existing Item – BellRingers*, acesse a pasta *Microsoft Press\Visual CSharp Step By Step\Chapter 23* na sua pasta Documentos. Na caixa suspensa ao lado da caixa de texto *File name*, selecione *All Files (\*.\*)*. Selecione os arquivos *Face.bmp*, *Note.bmp* e *Ring.bmp*, e clique em *Add*.

Esta ação adiciona os três arquivos de imagem como recursos a seu aplicativo.

17. No painel *XAML*, modifique as definições dos itens do menu *newMember*, *saveMember* e *about* e adicione elementos *MenuItem.Icon* filhos que referenciam cada um dos três arquivos de ícone adicionados ao projeto no passo anterior, como mostrado em negrito. Você também deve remover o caractere "/" da tag de fechamento de cada elemento *MenuItem*, e adicionar uma tag *</MenuItem>*:

```

<Menu Height="22" Name="menu1" ... >
 <MenuItem Header="_File" >
 <MenuItem Header="_New Member" Name="newMember" >
 <MenuItem.Icon>
 <Image Source="Face.bmp"/>
 </MenuItem.Icon>
 </MenuItem>
 <MenuItem Header="_Save Member Details" Name="saveMember" >
 <MenuItem.Icon>
 <Image Source="Note.bmp"/>
 </MenuItem.Icon>
 </MenuItem>
 <Separator/>
 <MenuItem Header="E_xit" Name="exit" />
 </MenuItem>

```

```

<MenuItem Header="_Help">
 <MenuItem Header="_About Middleshire Bell Ringers" Name="about" >
 <MenuItem.Icon>
 <Image Source="Ring.bmp"/>
 </MenuItem.Icon>
 </MenuItem>
</MenuItem>
</Menu>

```

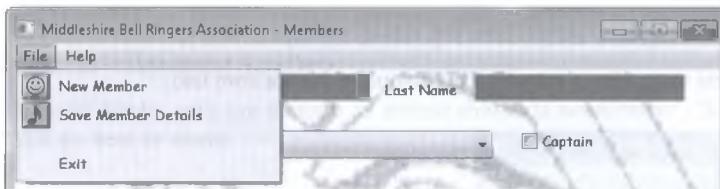
18. O ajuste final é assegurar que o texto para os itens do menu esteja estilizado de uma maneira consistente com o restante do formulário. No painel *XAML*, edite a definição do elemento *menu1* de primeiro nível e configure a propriedade *Style* como o estilo *BellRingersFontStyle*, como mostrado em negrito:

```
<Menu Style="{StaticResource bellRingersFontStyle}" ... Name="menu1" ... >
```

Observe que os itens de menu secundário herdam automaticamente o estilo do item do menu de primeiro nível que os contém.

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

Quando o formulário aparecer, clique no menu *File*. Você agora deve ver que o texto dos itens do menu é exibido na fonte correta e que os ícones aparecem junto aos itens de menu secundário, assim:



20. Feche o formulário e retorne ao Visual Studio 2010.

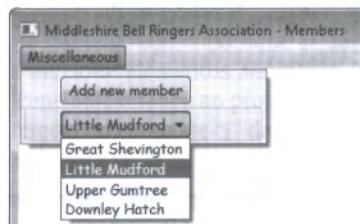
## Tipos de itens de menu

Você já utilizou o elemento *MenuItem* para adicionar itens de menu secundários a um controle *Menu*. Vimos que é possível especificar os itens no menu de primeiro nível como elementos *MenuItem*s e então adicionar elementos *MenuItem*s aninhados para definir a estrutura do seu menu. Os próprios elementos *MenuItem*s aninhados podem conter outros elementos *MenuItem*s aninhados se você quiser criar menus em cascata. Teoricamente, você pode prosseguir nesse processo até um nível muito além, mas na prática você provavelmente não deve passar de dois níveis de aninhamento.

Mas você não está limitado ao uso do elemento `MenuItem`. Você também pode adicionar caixas de combinação, caixas de texto e a maioria dos outros tipos de controles a menus WPF. Por exemplo, a seguinte estrutura de menus contém um botão e uma caixa de combinação (`ComboBox`):

```
<Menu ...>
 <MenuItem Header="Miscellaneous">
 <Button Content="Add new member" />
 <ComboBox>
 <ComboBox.Items>
 <ComboBoxItem>
 Great Shevington
 </ComboBoxItem>
 <ComboBoxItem>
 Little Mudford
 </ComboBoxItem>
 <ComboBoxItem>
 Upper Gumtree
 </ComboBoxItem>
 <ComboBoxItem>
 Downley Hatch
 </ComboBoxItem>
 </ComboBox.Items>
 </ComboBox>
 </MenuItem>
</Menu>
```

Em tempo de execução, a estrutura de menus se parece com isto:



Embora você tenha liberdade ao projetar seus menus, você deve se esforçar para manter as coisas simples, pois um menu assim não é muito intuitivo!

## Tratando eventos de menu

A aparência do menu que você construiu até agora é muito boa, mas nenhum dos itens faz algo quando clicados. Para torná-los funcionais, você precisa escrever um código que trate os vários even-

tos de menu. Diversos eventos podem ocorrer quando um usuário seleciona um item de menu, sendo alguns mais úteis que outros. O *Click* é o evento utilizado com mais frequência, que ocorre quando o usuário clica no item de menu. Normalmente, você captura esse evento para executar as tarefas associadas ao item de menu.

No próximo exercício, você aprenderá mais sobre os eventos de menu e como processá-los. Além disso, criará os eventos *Click* para os itens de menu *newMember* e *exit*.

O propósito do comando *New Member* é que o usuário possa inserir os detalhes de um novo membro. Portanto, até o usuário clicar em *New Member*, todos os campos no formulário devem estar desativados, assim como o comando *Save Member Details*. Quando o usuário clica no comando *New Member*, é recomendável ativar todos os campos, redefinir o conteúdo do formulário, para que o usuário possa começar a adicionar informações sobre um novo membro, e ativar o comando *Save Member Details*.

### Trate os eventos de item de menu *New Member* e *Exit*

1. No painel *XAML*, clique na definição da caixa de texto *firstName*. Na janela *Properties*, desmarque a propriedade *IsEnabled*. (Essa ação configura *IsEnabled* como *False* na definição XAML.)

Repita esse processo para os controles *lastName*, *towerNames*, *isCaptain*, *memberSince*, *yearsExperience*, *methods* e *clear* e para o item de menu *saveMember*.

2. Na janela *Design View*, no painel *XAML*, comece a digitar o código mostrado aqui em negrito na descrição XAML do item do menu *\_New Member*:

```
<MenuItem Header="_New Member" Name="newMember" Click=>
```

3. Quando o menu de atalho aparece depois que você digitar o caractere =, dê um clique duplo no comando *<New Event Handler>*.

O Visual Studio gera um método de evento chamado *newMember\_Click* e o associa ao evento *Click* do item de menu.

 **Dica** Sempre atribua um nome significativo a um item de menu se pretende definir métodos de evento para ele. Se não fizer isso, o Visual Studio irá gerar um método de evento chamado *MenuItem\_Click* para o evento *Click*. Se você então criar métodos de evento *Click* para outros itens de menu que também não tenham nomes, eles serão chamados *MenuItem\_Click\_1*, *MenuItem\_Click\_2*, e assim por diante. Se você tiver vários desses métodos de evento, pode ser difícil determinar qual método de evento pertence a qual item de menu.

4. Alterne para a janela *Code and Text Editor*, que exibe o arquivo *MainWindow.xaml.cs*. (No menu *View*, clique em *Code*.)

O método de evento *newMember\_Click* já constará no final da definição da classe *MainWindow*:

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

5. Adicione as seguintes instruções mostradas em negrito ao método *newMember\_Click*:

```
private void newMember_Click(object sender, RoutedEventArgs e)
{
 this.Reset();
 saveMember.IsEnabled = true;
 firstName.IsEnabled = true;
 lastName.IsEnabled = true;
 towerNames.IsEnabled = true;
 isCaptain.IsEnabled = true;
 memberSince.IsEnabled = true;
 yearsExperience.IsEnabled = true;
 methods.IsEnabled = true;
 clear.IsEnabled = true;
}
```

Esse código chama o método *Reset* e então ativa todos os controles. Vimos no Capítulo 22 que o método *Reset* redefine os controles do formulário para os seus valores padrão. (Se você não se lembrar de como o método *Reset* funciona, role a janela *Code and Text Editor* para exibir o método e refrescar sua memória.)

Em seguida, você criará um método de evento *Click* para o comando *Exit*. Esse método fará o formulário ser fechado.

6. Retorne à janela *Design View* para exibir o arquivo *MainWindow.xaml*. Utilize a técnica que você seguiu no Passo 2 para criar um método de evento *Click* para o item de menu *exit* chamado *exit\_Click*. (Esse é o nome padrão gerado, ao selecionar <New Event Handler>.)
7. Mude para a janela *Code and Text Editor*. No corpo do método *exit\_Click*, digite a instrução mostrada em negrito no seguinte código:

```
private void exit_Click(object sender, RoutedEventArgs e)
{
 this.Close();
}
```

O método *Close* de um formulário tenta fechar o formulário. Lembre-se de que, se o formulário interceptar o evento *Closing*, você poderá evitar que o formulário seja fechado. O aplicativo da Middleshire Bell Ringers Association faz exatamente isso e pergunta se o usuário deseja sair. Se ele disser que não, o formulário não será fechado e o aplicativo continuará a executar.

O próximo passo é tratar o item de menu *saveMember*. Quando o usuário clica nesse item de menu, os dados no formulário devem ser salvos em um arquivo. Por enquanto, você salvará as informações

em um arquivo texto comum chamado Members.txt na pasta atual. Mais tarde, você modificará o código para que o usuário possa selecionar outro nome e localização de arquivo.

### Trate o evento do item de menu Save Member Details

1. Retorne à janela *Design View* para exibir o arquivo MainWindow.xaml. No painel *XAML*, localize a definição do item de menu *saveMember* e utilize o comando *<New Event Handler>* para gerar um método de evento *Click* chamado *saveMember\_Click*. (Esse é o nome padrão gerado ao selecionar *<New Event Handler>*.)
2. Na janela *Code and Text Editor*, exiba o arquivo MainWindow.xaml.cs, role até a parte superior do arquivo e adicione a seguinte instrução *using* à lista:

```
using System.IO;
```

3. Localize o método de evento *saveMember\_Click* no final do arquivo. Adicione as seguintes instruções mostradas em negrito ao corpo do método:

```
private void saveMember_Click(object sender, RoutedEventArgs e)
{
 using (StreamWriter writer = new StreamWriter("Members.txt"))
 {
 writer.WriteLine("First Name: {0}", firstName.Text);
 writer.WriteLine("Last Name: {0}", lastName.Text);
 writer.WriteLine("Tower: {0}", towerNames.Text);
 writer.WriteLine("Captain: {0}", isCaptain.IsChecked.ToString());
 writer.WriteLine("Member Since: {0}", memberSince.Text);
 writer.WriteLine("Methods: ");
 foreach (CheckBox cb in methods.Items)
 {
 if (cb.IsChecked.Value)
 {
 writer.WriteLine(cb.Content.ToString());
 }
 }
 MessageBox.Show("Member details saved", "Saved");
 }
}
```

Esse bloco de código cria um objeto *StreamWriter* que o método utiliza para escrever um texto no arquivo Member.txt. Utilizar a classe *StreamWriter* é semelhante a exibir o texto em um aplicativo console empregando o objeto *Console* – você pode simplesmente usar o método *WriteLine*.

Depois que os detalhes foram gravados, uma caixa de mensagem é exibida fornecendo um feedback ao usuário (sempre uma boa ideia).

4. O botão *Add* e seu método de evento associado estão obsoletos agora, portanto, na janela *Design View*, exclua botão *Add*. Em seguida, na janela *Code and Text Editor*, transforme o método *add\_Click* em comentário.

O item de menu remanescente é o *about*, que deve exibir uma caixa de diálogo fornecendo informações sobre a versão do aplicativo, editor e quaisquer outras informações úteis. Você adicionará um método de evento para tratar esse evento no próximo exercício.

### Trate o evento do item de menu *About Middleshire Bell Ringers*

1. No menu *Project*, clique em *Add Window*.

2. Na caixa de diálogo *Add New Item – BellRingers*, no painel central, clique em *Window (WPF)*. Na caixa de texto *Name*, digite **About.xaml** e então clique em *Add*.

Depois de adicionar os controles apropriados, você exibirá essa janela quando o usuário clicar no comando *About Middleshire Bell Ringers*, no menu *Help*.

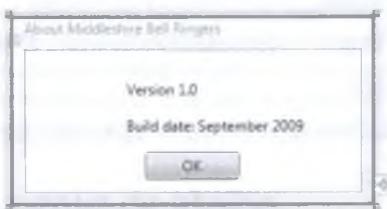


**Nota** O Visual Studio fornece o template de janela *About Box*. Porém, ele gera uma janela Windows Forms em vez de uma janela WPF.

3. Na janela *Design View*, clique no formulário *About.xaml*. Na janela *Properties*, altere a propriedade *Title* para **About Middleshire Bell Ringers**, configure a propriedade *Width* como **300** e a propriedade *Height* como **156**. Depois, configure a propriedade *ResizeMode* como **NoResize** para impedir que o usuário mude o tamanho da janela quando ela for exibida. (Esta é a convenção para esse tipo de caixa de diálogo.)
4. Na caixa *Name* na parte superior da janela *Properties*, digite **AboutBellRingers**.
5. Na *ToolBox*, adicione dois controles de rótulo e um controle de botão ao formulário. No painel *XAML*, modifique suas propriedades como mostrado aqui em negrito (ou altere o texto exibido pelo rótulo *buildDate*, se preferir):

```
<Window x:Class="BellRingers.About"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="About Middleshire Bell Ringers" Height="156" Width="300"
 Name="AboutBellRingers" ResizeMode="NoResize">
 <Grid>
 <Label Content="Version 1.0" Height="28" HorizontalAlignment="Left"
 Margin="80,20,0,0" Name="version" VerticalAlignment="Top"
 Width="75" />
 <Label Content="Build date: September 2009" Height="28"
 HorizontalAlignment="Left" Margin="80,50,0,0" Name="buildDate"
 VerticalAlignment="Top" Width="160" />
 <Button Content="OK" Height="23" HorizontalAlignment="Left"
 Margin="100,85,0,0" Name="ok" VerticalAlignment="Top"
 Width="78" />
 </Grid>
</Window>
```

O formulário completo deve ser semelhante a este:



6. Na janela *Design View*, dê um clique duplo no botão *OK*.

O Visual Studio gera um método de evento, chamado *ok\_Click*, para o evento *Click* do botão e adiciona esse método ao arquivo *About.xaml.cs*.

7. Na janela *Code and Text Editor* que exibe o arquivo *About.xaml.cs*, adicione a instrução mostrada em negrito ao método *ok\_Click*:

```
private void ok_Click(object sender, RoutedEventArgs e)
{
 this.Close();
}
```

Quando o usuário clicar no botão *OK*, a janela *About Middleshire Bell Ringers* se fechará.

8. Retorne à janela *Design View* para exibir o arquivo *MainWindow.xaml*. No painel *XAML*, localize a definição do item de menu *about* e utilize o comando *<New Event Handler>* para especificar um método de evento *Click* chamado *about\_Click*. (Esse é o nome padrão gerado.)
9. Na janela *Code and Text Editor* que exibe o arquivo *MainWindow.xaml.cs*, adicione as seguintes instruções mostradas em negrito ao método *about\_Click*:

```
private void about_Click(object sender, RoutedEventArgs e)
{
 About aboutWindow = new About();
 aboutWindow.ShowDialog();
}
```

Na verdade, os formulários WPF são apenas classes que herdam da classe *System.Windows.Window*. Você pode criar uma instância de um formulário WPF da mesma maneira como cria qualquer outra classe. Esse código gera uma nova instância da janela *About* e então chama o método *ShowDialog* para exibi-la. Esse método é herdado da classe *Windows* e exibe o formulário WPF na tela. O método *ShowDialog* não retorna até que a janela *About* se feche (quando o usuário clicar no botão *OK*).

### Teste os eventos de menu

1. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo. Observe que todos os campos do formulário estão desativados.

2. Clique no menu *File*.

O comando *Save Member Details* está desativado.

3. No menu *File*, clique em *New Member*. Os campos no formulário estão agora disponíveis.

4. Insira alguns detalhes para um novo membro.

5. Clique no menu *File* novamente. O comando *Save Member Details* agora está disponível.

6. No menu *File*, clique em *Save Member Details*.

Depois de um breve tempo, a mensagem “Member details saved” aparece. Clique em *OK* nesta caixa de mensagem.

7. Utilizando o Windows Explorer, vá para a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 23\BellRingers\BellRingers\bin\Debug na sua pasta Documentos.

Você deve ver um arquivo chamado *Members.txt* nesta pasta.

8. Dê um clique duplo em *Members.txt* para exibir seu conteúdo utilizando o Notepad (Bloco de Notas). Esse arquivo deve conter os detalhes do novo membro. O texto a seguir apresenta um exemplo:

```
First Name: John
Last Name: Sharp
Tower: Little Mudford
Captain: False
Member Since: 15/01/2000
Methods:
Plain Bob
Reverse Canterbury
Grandsire
Stedman
Kent Treble Bob
Old Oxford Delight
Winchendon Place
```

9. Feche o Bloco de Notas e retorne ao aplicativo Middleshire Bell Ringers.

10. No menu *Help*, clique em *About Middleshire Bell Ringers*.

A janela *About* aparece. Observe que você não pode redimensionar essa janela e não pode clicar em item algum no formulário *Members* enquanto a janela *About* ainda estiver visível.

11. Clique em *OK* para retornar ao formulário *Members*.

12. No menu *File*, clique em *Exit*.

O formulário tenta se fechar. É perguntado a você se tem certeza de que quer fechar o formulário. Se você clicar em *No*, o formulário permanecerá aberto; se clicar em *Yes*, o formulário fechará e o aplicativo terminará.

13. Clique em *Yes* para fechar o formulário.

## Menus de atalho

Muitos aplicativos baseados em Windows fazem uso de menus pop-up que aparecem quando você clica com o botão direito do mouse em um formulário ou controle. Esses menus são normalmente relacionados ao contexto e exibem comandos que só são aplicáveis ao controle ou formulário que tem o foco no momento. Eles são normalmente conhecidos como menus de *contexto* ou de *atalho*. Você pode adicionar facilmente menus de atalho a um aplicativo WPF utilizando a classe *ContextMenu*.

## Criando menus de atalho

Nos próximos exercícios, você criará dois menus de atalho. O primeiro menu de atalho será anexado aos controles de caixa de texto *firstName* e *lastName* e permitirá que o usuário limpe esses controles. O segundo menu de atalho será anexado ao formulário e conterá comandos para salvar as informações de membro exibidas no momento e limpar o formulário.

 **Nota** Os controles *TextBox* são associados com um menu de atalho padrão que fornece os comandos *Cut*, *Copy* e *Paste* para realizar edição de texto. O menu de atalho que você definirá no próximo exercício redefinirá esse menu padrão.

### Crie o menu de atalho para *firstName* e *lastName*

1. Na janela *Design View* que exibe *MainWindow.xaml*, adicione o seguinte elemento *ContextMenu* mostrado em negrito ao final dos recursos de janela no painel *XAML* depois das definições de estilo:

```
<Window.Resources>
 ...
 <ContextMenu x:Key="textBoxMenu" Style="{StaticResource bellRingersFontStyle}" >
 </ContextMenu>
</Window.Resources>
```

Esse menu de atalho será compartilhado pelas caixas de texto *firstName* e *lastName*. Adicionar o menu de atalho aos recursos de janela torna-o disponível a quaisquer controles na janela.

2. Adicione o seguinte elemento *MenuItem* mostrado em negrito ao menu de atalho *textBoxMenu*:

```
<Window.Resources>
 ...
 <ContextMenu x:Key="textBoxMenu" Style="{StaticResource bellRingersFontStyle}">
 <MenuItem Header="Clear Name" Name="clearName" />
 </ContextMenu>
</Window.Resources>
```

Este código adiciona um item de menu, chamado *clearName* com a legenda “Clear Name”, ao menu de atalho.

3. No painel *XAML*, modifique as definições dos controles das caixas de texto *firstName* e *lastName* e adicione a propriedade *ContextMenu*, mostrada em negrito:

```
<TextBox ... Name="firstName" ContextMenu="{StaticResource textBoxMenu}" ... />

<TextBox ... Name="lastName" ContextMenu="{StaticResource textBoxMenu}" ... />
```

A propriedade *ContextMenu* determina qual menu (se houver) será exibido quando o usuário clicar com o botão direito do mouse no controle.

4. Retorne à definição do estilo *textBoxMenu* e adicione um método de evento *Click* chamado *clearName\_Click* ao item de menu *clearName*. (Esse é o nome padrão gerado pelo comando *<New Event Handler>*.)

```
<MenuItem Header="Clear Name" Name="clearName" Click="clearName_Click" />
```

5. Na janela *Code and Text Editor* que exibe *MainWindow.xaml.cs*, adicione as seguintes instruções ao método de evento *clearName\_Click* que o comando *<New Event Handler>* gerou:

```
firstName.Clear();
lastName.Clear();
```

Esse código desmarca as duas caixas de texto quando o usuário clica no comando *Clear Name* no menu de atalho.

6. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo. Quando o formulário for exibido, clique em *File* e, então, clique em *New Member*.

7. Digite um nome nas caixas de texto *First Name* e *Last Name*. Clique com o botão direito do mouse na caixa de texto *First Name*. No menu de atalho, clique no comando *Clear Name* e verifique se ambas as caixas de texto estão desmarcadas.

8. Digite um nome nas caixas de texto *First Name* e *Last Name*. Desta vez, clique com o botão direito do mouse na caixa de texto *Last Name*. No menu de atalho, clique no comando *Clear Name* e verifique mais uma vez se as duas caixas de texto estão desmarcadas.

9. Clique com o botão direito do mouse em quaisquer controles, exceto no controle *Member Since*. Clique com o botão direito do mouse em qualquer local no formulário, fora das caixas de texto *First Name* e *Last Name*.

À exceção do controle *Member Since*, somente as caixas de texto *First Name* e *Last Name* têm menus de atalho, de modo que nenhum menu pop-up deve aparecer em qualquer outro lugar.



**Nota** O controle *Member Since* exibe um menu pop-up com os comandos *Cut*, *Copy* e *Paste*. Por padrão, essa funcionalidade está incorporada ao *DatePicker*.

10. Feche o formulário e retorne ao Visual Studio 2010.

Agora você pode adicionar o segundo menu de atalho, que contém comandos que o usuário pode utilizar para salvar informações sobre membros e desmarcar os campos no formulário. Para variar

um pouco e mostrar como é fácil criar menus de atalho dinamicamente, no próximo exercício você criará o menu de atalho utilizando código. O melhor lugar para colocar esse código é no construtor do formulário. Você então adicionará um código para ativar o menu de atalho da janela quando o usuário criar um novo membro.

### Crie o menu de atalho da janela

1. Mude para a janela *Code and Text Editor* que exibe o arquivo MainWindow.xaml.cs.
2. Adicione a seguinte variável privada mostrada em negrito à classe *MainWindow*:

```
public partial class MainWindow : Window
{
 ...
 private ContextMenu windowContextMenu = null;
 ...
}
```

3. Localize o construtor da classe *MainWindow*. Esse é, de fato, o primeiro método da classe e se chama *MainWindow*. Adicione as instruções mostradas em negrito após o código que chama o método *Reset* a fim de criar os itens de menu para salvar detalhes sobre membros:

```
public MainWindow()
{
 InitializeComponent();
 this.Reset();

 MenuItem saveMemberMenuItem = new MenuItem();
 saveMemberMenuItem.Header = "Save Member Details";
 saveMemberMenuItem.Click += new RoutedEventHandler(saveMember_Click);
}
```

Esse código configura a propriedade *Header* para o item do menu e então especifica que o evento *Click* deve invocar o método de evento *saveMember\_Click*; esse é o mesmo método que você escreveu em um exercício anterior neste capítulo. O tipo *RoutedEventHandler* é um delegate que representa os métodos para tratar os eventos disparados por muitos controles WPF. (Para mais informações sobre delegates e eventos, consulte o Capítulo 17, “Interrompendo o fluxo do programa e tratando eventos”.)

4. No construtor de *MainWindow*, adicione as seguintes instruções mostradas em negrito a fim de criar os itens de menu para desmarcar os campos do formulário e redefinir os valores padrão:

```
public MainWindow()
{
 ...
 MenuItem clearFormMenuItem = new MenuItem();
 clearFormMenuItem.Header = "Clear Form";
 clearFormMenuItem.Click += new RoutedEventHandler(clear_Click);
}
```

Esse item do menu invoca o método de evento *clear\_Click* quando clicado pelo usuário.

5. No construtor de *MainWindow*, adicione as seguintes instruções mostradas em negrito para construir o menu de atalho e preenchê-lo com os dois itens de menu que você acabou de criar:

```
public MainWindow()
{
 ...
 windowContextMenu = new ContextMenu();
 windowContextMenu.Items.Add(saveMemberMenuItem);
 windowContextMenu.Items.Add(clearFormMenuItem);
}
```

O tipo *ContextMenu* contém uma coleção chamada *Items* que armazena os itens de menu.

6. No final do método de evento *newMember\_Click*, adicione a instrução mostrada em negrito para associar o menu de contexto ao formulário:

```
private void newMember_Click(object sender, RoutedEventArgs e)
{
 ...
 this.ContextMenu = windowContextMenu;
}
```

Observe que o aplicativo só associa o menu de atalho ao formulário depois de a nova funcionalidade de membro estar disponível. Se você fosse configurar a propriedade *ContextMenu* do formulário no construtor, os itens de menu de atalho *Save Member Details* e *Clear Details* estariam disponíveis mesmo quando os controles no formulário estivessem desativados, que não é a maneira como você quer que esse aplicativo se comporte.



**Dica** Você pode desassociar um menu de atalho de um formulário configurando a propriedade *ContextMenu* do formulário como *null*.

7. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
8. Quando o formulário aparece, clique com o botão direito do mouse no formulário e verifique se o menu de atalho não aparece.
9. No menu *File*, clique em *New Member* e então insira alguns detalhes para um novo membro.
10. Clique com o botão direito do mouse no formulário. No menu de atalho, clique em *Clear Form* e verifique se os campos no formulário são reinicializados para seus valores padrão.
11. Insira mais detalhes sobre o membro. Clique com o botão direito do mouse no formulário. No menu de atalho, clique em *Save Member Details*. Verifique se a caixa de mensagem "Member details saved" aparece e então clique em *OK*.
12. Feche o formulário e retorne ao Visual Studio 2010.

## Caixas de diálogo comuns do Windows

O aplicativo BellRingers agora permite que você salve as informações sobre membros, mas ele sempre salva os dados no mesmo arquivo, sobreescrevendo tudo que já estava lá. Agora está na hora de tratar dessa questão.

Algumas tarefas cotidianas exigem que o usuário especifique as mesmas informações, independentemente da funcionalidade do aplicativo que o usuário está executando. Por exemplo, se o usuário quiser abrir ou salvar um arquivo, normalmente se solicita ao usuário qual arquivo abrir ou onde salvar. Você talvez tenha notado que a mesma caixa de diálogo é utilizada por muitos aplicativos diferentes. Isso não é resultado da falta de imaginação dos desenvolvedores de aplicativos; é exatamente porque sua funcionalidade é tão comum que a Microsoft padronizou-a e a disponibilizou como uma “caixa de diálogo comum” – um componente fornecido com o sistema operacional Microsoft Windows que você pode utilizar em seus próprios aplicativos. A biblioteca de classes do Microsoft .NET Framework fornece as classes *OpenFileDialog* e *SaveFileDialogs* que agem como empacotadores para essas caixas de diálogo comuns.

### Utilizando a classe *SaveFileDialog*

No próximo exercício, você vai utilizar a classe *SaveFileDialog*. No aplicativo da BellRingers, quando o usuário salva os detalhes em um arquivo, você vai solicitar o nome e o local do arquivo exibindo a caixa de diálogo comum Save File.

#### Utilize a classe *SaveFileDialog*

1. Na janela *Code and Text Editor* que exibe MainWindow.xaml.cs, adicione a seguinte diretiva *using* à lista na parte superior do arquivo:

```
using Microsoft.Win32;
```

A classe *SaveFileDialog* está no namespace *Microsoft.Win32* (até mesmo nas versões de 64 bits do sistema operacional Windows).

2. Localize o método *saveMember\_Click* e adicione o código mostrado em negrito ao início desse método, substituindo *YourName* pelo nome da sua conta em seu computador\*:

```
private void saveMember_Click(object sender, RoutedEventArgs e)
{
 SaveFileDialog saveDialog = new SaveFileDialog();
 saveDialog.DefaultExt = "txt";
 saveDialog.AddExtension = true;
 saveDialog.FileName = "Members";
 saveDialog.InitialDirectory = @"C:\Users\YourName\Documents\";
 saveDialog.OverwritePrompt = true;
 saveDialog.Title = "Bell Ringers";
 saveDialog.ValidateNames = true;

}
```

\* N. de R. T.: Dependendo do idioma da sua instalação do Microsoft Windows, talvez seja necessário ajustar todo o caminho na propriedade InitialDirectory.

Esse código cria uma nova instância da classe *SaveFileDialog* e configura suas propriedades. A tabela a seguir descreve o propósito dessas propriedades.

Propriedade	Descrição
<i>DefaultExt</i>	A extensão de nome de arquivo padrão a ser usada se o usuário não especificar a extensão ao fornecer o nome de arquivo.
<i>AddExtension</i>	Faz a caixa de diálogo adicionar a extensão do nome de arquivo indicada pela propriedade <i>DefaultExt</i> ao nome do arquivo especificado pelo usuário, se o usuário omitir a extensão.
<i>FileName</i>	O nome do arquivo selecionado no momento. Você pode preencher essa propriedade para especificar um nome de arquivo padrão ou limpá-la se você não quiser um nome de arquivo padrão.
<i>InitialDirectory</i>	O diretório padrão a ser usado pela caixa de diálogo.
<i>OverwritePrompt</i>	Faz a caixa de diálogo alertar o usuário quando é feita uma tentativa de sobreescriver um arquivo existente com o mesmo nome. Para funcionar, a propriedade <i>ValidateNames</i> também deve estar configurada como <i>True</i> .
<i>Title</i>	Uma string que é exibida na barra de título da caixa de diálogo.
<i>ValidateNames</i>	Indica se os nomes de arquivo são validados. Ela é usada por algumas outras propriedades, como <i>OverwritePrompt</i> . Se a propriedade <i>ValidateNames</i> estiver configurada como <i>true</i> , a caixa de diálogo também verificará se o nome de arquivo digitado pelo usuário contém apenas caracteres válidos.

3. Adicione a seguinte instrução *if* (e a chave de fechamento), mostrada em negrito, ao método *saveMember\_Click*. Essa instrução encapsula o código anterior que cria o objeto *StreamWriter* e grava os detalhes do membro em um arquivo:

```
if (saveDialog.ShowDialog().Value)
{
 using (StreamWriter writer = new StreamWriter("Members.txt"))
 {
 // código existente

 }
}
```

O método *ShowDialog* exibe a caixa de diálogo *Save File*, a qual é modal, ou seja, o usuário não pode continuar a utilizar qualquer outro formulário no aplicativo até que essa caixa de diálogo seja fechada clicando em um de seus botões. A caixa de diálogo *Save File* tem um botão *Save* e um botão *Cancel*. Se o usuário clicar em *Save*, o valor retornado pelo método *ShowDialog* é *true*; do contrário, *false*.

O método *ShowDialog* solicita ao usuário o nome de um arquivo a ser salvo, mas na verdade não salva nada – você ainda tem de providenciar esse código. Tudo o que ele faz é fornecer o nome do arquivo que o usuário selecionou na propriedade *FileName*.

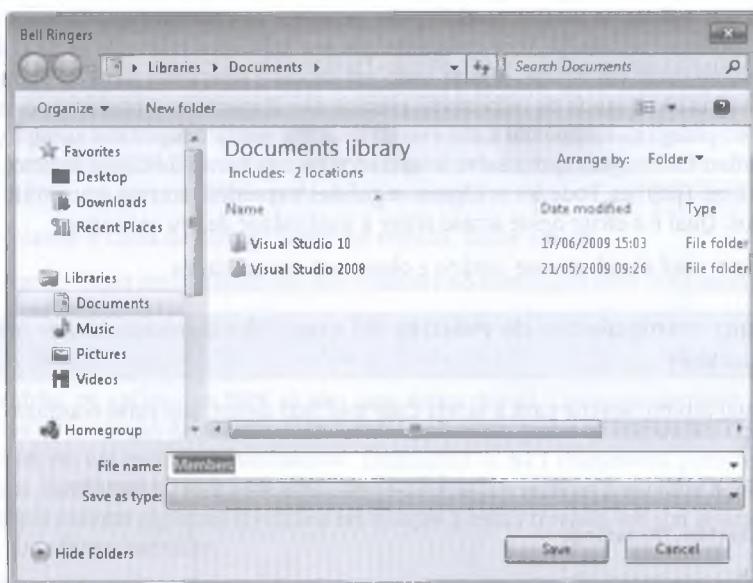
4. No método `saveMember_Click`, modifique a instrução que cria o objeto `StreamWriter` como mostrado em negrito:

```
using (StreamWriter writer = new StreamWriter(saveDialog.FileName))
{
 ...
}
```

O método `saveMember_Click` gravará agora no arquivo especificado pelo usuário em vez de em `Members.txt`.

5. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
6. No menu *File*, clique em *New Member* e então adicione alguns detalhes a um novo membro.
7. No menu *File*, clique em *Save Member Details*.

A caixa de diálogo *Save File* deve aparecer, com a legenda “Bell Ringers”. A pasta padrão deve ser sua pasta Documentos, e o nome do arquivo padrão deve ser `Members`, como mostrado na imagem a seguir:



Se você omitir a extensão do arquivo, a extensão `.txt` será automaticamente adicionada quando o arquivo for salvo. Se você selecionar um arquivo existente, a caixa de diálogo o avisa antes de fechar.

8. Mude o valor na caixa de texto *File name* para **TestMember** e então clique em *Save*.
9. No aplicativo *BellRingers*, verifique se a mensagem “Member details saved” aparece, clique em *OK* e então feche o aplicativo.

10. Utilizando o Windows Explorer, acesse a sua pasta Documentos.

Verifique se o arquivo TestMember.txt foi criado.

11. Dê um clique duplo nesse arquivo e verifique se ele contém os detalhes do membro que você adicionou. Feche o Bloco de Notas depois de terminar.

Você pode utilizar uma técnica semelhante para abrir um arquivo: crie um objeto *OpenFileDialog* e ative-o utilizando o método *ShowDialog* e recupere a propriedade *FileName* quando o método retornar, no caso de o usuário ter clicado no botão *Open*. Você poderá então abrir o arquivo, ler seu conteúdo e preencher os campos na tela. Para obter mais detalhes sobre o uso da classe *OpenFileDialog*, consulte a biblioteca MSDN do Visual Studio 2010.

## Aprimorando a capacidade de resposta de um aplicativo WPF

A finalidade das bibliotecas WPF é propiciar uma base para construir aplicativos equipados com interfaces gráficas do usuário. Consequentemente, os aplicativos WPF são inherentemente interativos. Ao executar um aplicativo WPF, o usuário pode percorrer os controles que formam a interface do usuário em qualquer sequência. Você constatou que seu aplicativo responde ao usuário que executa operações, como clicar em botões, digitar texto em caixas e selecionar itens de menu, por meio de um código executado quando os respectivos eventos são disparados. Entretanto, o que acontece se a execução do código que responde a um evento demorar muito tempo? Por exemplo, vamos supor que o aplicativo BellRingers tenha salvo seus dados em um banco de dados remoto, localizado em algum ponto na Internet. Pode levar alguns segundos para efetivamente transmitir esses dados e armazená-los. Qual é o efeito desse atraso sobre a usabilidade de seu aplicativo?

Neste exercício, você simulará esse cenário e observará os resultados.

### Simule um manipulador de eventos de execução demorada em um aplicativo WPF

1. No Visual Studio, alterne para a janela *Code and Text Editor*, que exibe o arquivo *MainWindow.xaml.cs*.
2. Adicione a seguinte instrução *using* à lista localizada no início do arquivo.

```
using System.Threading;
```

3. Adicione a seguinte instrução, mostrada em negrito, à instrução *using*, que grava os dados dos membros no arquivo especificado pelo usuário:

```
private void saveMember_Click(object sender, RoutedEventArgs e)
{
 ...
 if (saveDialog.ShowDialog().Value)
 {
 using (StreamWriter writer = new StreamWriter(saveDialog.FileName))
 {
 ...
 Thread.Sleep(10000);
 MessageBox.Show("Member details saved", "Saved");
 }
 }
}
```

O método estático *Sleep* da classe *Thread* no namespace *System.Threading* instrui a thread atual no aplicativo a parar de responder durante um período de tempo especificado. Esse intervalo de tempo é expresso em milissegundos, e esse código faz a thread parar por 10 segundos.



**Nota** Uma thread é um caminho de execução em um aplicativo. Todos os aplicativos têm pelo menos uma thread, e você pode criar aplicativos que utilizam várias threads. Se um computador tiver diversas CPUs ou um processador multicore (multinúcleo), ele poderá executar várias threads simultaneamente. Se você criar mais threads do que a quantidade disponível de CPUs ou de núcleos de processador, o sistema operacional alocará uma parcela do tempo da CPU para cada thread, para dar a impressão de uma execução simultânea. Você conhecerá mais detalhes sobre as threads e sobre como executar operações em paralelo no Capítulo 27, “Introdução à Task Parallel Library”.

4. No menu *Debug*, clique em *Start Without Debugging*.
5. Quando o formulário WPF for exibido, clique em *New Member* no menu *File* e digite alguns detalhes do novo membro.
6. No menu *File*, clique em *Save Member Details*. Na caixa de diálogo *Bell Ringers*, aceite o nome de arquivo padrão e clique em *Save* (e substitua o arquivo, se receber uma solicitação para fazê-lo).
7. Quando a caixa de diálogo *Bell Ringers* se fechar, tente clicar em qualquer um dos controles no formulário WPF. Observe que o formulário deixa de responder. (O formulário pode aparecer em branco, e a barra de título pode exibir o texto “Not Responding” [Não está respondendo].)
8. Quando a caixa de diálogo *Saved* for exibida, clique em *OK*.
9. Agora, clique em qualquer um dos controles no formulário WPF. O formulário já responde corretamente.
10. Feche o formulário e retorne ao Visual Studio 2010.

Por padrão, os aplicativos WPF só têm uma única thread. Consequentemente, um manipulador de eventos de execução demorada pode fazer o aplicativo parar de responder. Certamente, isso não é aceitável em um programa profissional. Entretanto, o .NET Framework permite que você crie várias threads. Sendo assim, você pode executar as tarefas de execução demorada nessas threads. Contudo, saiba que existem algumas restrições em relação a essas threads, em um aplicativo WPF, como você verá no próximo exercício.

### Execute uma operação de execução demorada em uma nova thread

1. No Visual Studio, exiba o arquivo *MainWindow.xaml.cs* na janela *Code and Text Editor*.
2. Adicione um novo método privado à classe *MainWindow*, chamado *saveData*. Esse método deve aceitar um parâmetro de string que especifica um nome de arquivo e não deve retornar um valor, como a seguir:

```
private void saveData(string fileName)
{
}
```

3. Localize o método *saveMember\_Click*. Copie a instrução *using* e o código de fechamento desse método para o método *saveData*. O método *saveData* deve ficar parecido com o seguinte:

```
private void saveData(string fileName)
{
 using (StreamWriter writer = new StreamWriter(saveDialog.FileName))
 {
 writer.WriteLine("First Name: {0}", firstName.Text);
 writer.WriteLine("Last Name: {0}", lastName.Text);
 writer.WriteLine("Tower: {0}", towerNames.Text);
 writer.WriteLine("Captain: {0}", isCaptain.IsChecked.ToString());
 writer.WriteLine("Member Since: {0}", memberSince.Text);
 writer.WriteLine("Methods: ");
 foreach (CheckBox cb in methods.Items)
 {
 if (cb.IsChecked.Value)
 {
 writer.WriteLine(cb.Content.ToString());
 }
 }
 }

 Thread.Sleep(10000);
 MessageBox.Show("Member details saved", "Saved");
}
}
```

4. Na instrução *using*, mude o código que chama o construtor para o objeto *StreamWriter* e substitua a referência ao *saveDialog.FileName* pelo parâmetro *fileName*, como mostrado em negrito a seguir:

```
using (StreamWriter writer = new StreamWriter(fileName))
{
 ...
}
```

5. No método *saveMember\_Click*, remova a instrução *using* e o bloco de código de fechamento, e substitua-a pelas instruções mostradas em negrito a seguir:

```
private void saveMember_Click(object sender, RoutedEventArgs e)
{
 ...
 if (saveDialog.ShowDialog().Value)
 {
 Thread workerThread = new Thread(
 () => this.saveData(saveDialog.FileName));
 workerThread.Start();
 }
}
```

Esse código cria um novo objeto *Thread*, chamado *workerThread*. O construtor para a classe *Thread* espera um delegate que faça referência a um método ser executado quando a thread for executada. Esse exemplo utiliza uma expressão lambda para criar um delegate anônimo, que chama o método *saveData*.

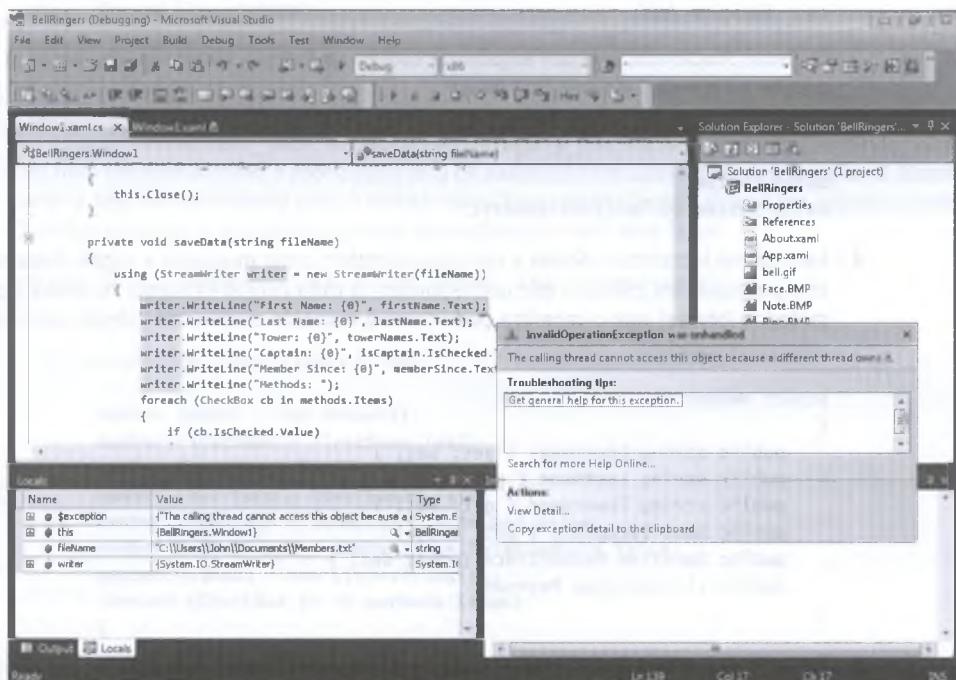
O método *Start* da classe *Thread* inicia a execução da thread. A thread executa no modo assíncrono – o método *Start* não espera o término do método executado pela thread.

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

**Importante** Não execute o aplicativo sem depurar.

- Quando o formulário WPF for exibido, clique em *New Member* no menu *File*, forneça alguns dados para o novo membro e clique em *Save Member Details*. Na caixa de diálogo *Bell Ringers*, selecione o arquivo *Members.txt* e clique em *Save* (e substitua o arquivo, se solicitado a fazê-lo).

O aplicativo para no método *saveData* e informa a exceção “*InvalidOperationException was unhandled*”, mostrada na seguinte imagem.



O texto da exceção é “The calling thread cannot access this object because a different thread owns it”, e a linha destacada é o código que lê os dados da caixa de texto `firstName`.

8. No menu *Debug*, clique em *Stop Debugging* e retorne ao Visual Studio 2010.

Você tentou utilizar uma thread para executar uma tarefa demorada em segundo plano. Essa é uma abordagem segura. O problema é que o modelo de segurança implementado pelo WPF impede que as threads diferentes daquele que criou um objeto da interface do usuário, como um controle, acessem esse objeto. Essa restrição evita que duas ou mais threads tentem assumir o controle da entrada do usuário ou modificar os dados na tela, porque isso poderia danificar seus dados.

Você pode driblar essa restrição de várias maneiras, mas a solução mais simples é reunir os dados que serão salvos em uma estrutura, no método executado pela thread da interface do usuário e depois passar essa estrutura para o método executado pela thread em segundo plano. Você fará isso no próximo exercício.

### Copie dados da thread da interface do usuário para a thread em segundo plano

1. No Visual Studio, no menu *Project*, clique em *Add New Item*.
2. Na caixa de diálogo *Add New Item – Bell Ringers*, no painel à esquerda, expanda *Visual C#* e clique em *Code*. No painel central, clique em *Code File*. Na caixa de texto *Name*, digite **Member.cs** e clique em *Add*.
3. Adicione as seguintes instruções *using* ao início do arquivo Member.cs:

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

4. No arquivo Member.cs, defina a estrutura *Member*, como mostrado a seguir. Essa estrutura contém propriedades públicas que correspondem a cada campo existente no formulário. A lista de métodos (temas) que o membro pode tocar é mantida como uma coleção *List<string>* (não uma propriedade).

```
struct Member
{
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public string TowerName { get; set; }
 public bool IsCaptain { get; set; }
 public DateTime MemberSince { get; set; }
 public List<string> Methods;
}
```

5. Retorne ao arquivo MainWindow.xaml.cs na janela *Code and Text Editor*.

6. Modifique o método *saveData* de modo a aceitar uma estrutura *Member* como o segundo parâmetro. No corpo do método *saveData*, mude o código para ler os dados do membros nessa estrutura, e não nos campos no formulário WPF. O exemplo de código a seguir apresenta o método concluído. Observe que o loop *foreach* itera pela coleção *List<string>* e sobre caixas de seleção do formulário:

```
private void saveData(string fileName, Member member)
{
 using (StreamWriter writer = new StreamWriter(fileName))
 {
 writer.WriteLine("First Name: {0}", member.FirstName);
 writer.WriteLine("Last Name: {0}", member.LastName);
 writer.WriteLine("Tower: {0}", member.TowerName);
 writer.WriteLine("Captain: {0}", member.IsCaptain.ToString());
 writer.WriteLine("Member Since: {0}", member.MemberSince.ToString());
 writer.WriteLine("Methods: ");
 foreach (string method in member.Methods)
 {
 writer.WriteLine(method);
 }

 Thread.Sleep(10000);
 MessageBox.Show("Member details saved", "Saved");
 }
}
```

7. No método *saveMember\_Click*, no bloco da instrução *if* que inicia a thread em segundo plano, crie uma variável *Member* e preencha-a com os dados do formulário. Passe essa variável *Member* como o segundo parâmetro para o método *saveData* executado pela thread em segundo plano. O código em negrito a seguir mostra as alterações que você deve fazer:

```
private void saveMember_Click(object sender, RoutedEventArgs e)
{
 ...
 if (saveDialog.ShowDialog().Value)
 {
 Member member = new Member();
 member.FirstName = firstName.Text;
 member.LastName = lastName.Text;
 member.TowerName = towerNames.Text;
 member.IsCaptain = isCaptain.IsChecked.Value;
 member.MemberSince = memberSince.SelectedDate.Value;
 member.Methods = new List<string>();
 foreach (CheckBox cb in methods.Items)
 {
 if (cb.IsChecked.Value)
 {
 member.Methods.Add(cb.Content.ToString());
 }
 }

 Thread workerThread = new Thread(
 () => this.saveData(saveDialog.FileName, member));
 workerThread.Start();
 }
}
```



**Nota** O controle *isCaptain* no formulário é um controle *CheckBox*, e a propriedade *IsChecked* pode ser nula, de modo que, para saber se o controle foi marcado, examine a propriedade *Value*. De modo semelhante, a propriedade *SelectedDate* do controle *DatePicker* também pode ser nula, e o código utiliza a propriedade *Value* para recuperar os dados relacionados à afiliação.

8. No menu *Debug*, clique em *Start Without Debugging*. No menu *File*, clique em *New Member* e adicione alguns dados. Em seguida, no menu *File*, clique em *Save MemberDetails*. Na caixa de diálogo *Bell Ringers*, especifique o nome de arquivo padrão e clique em *Save*.

Observe que o formulário WPF já está funcionando corretamente – ele responde quando você clica em qualquer um dos campos ou quando tenta inserir alguns dados.

9. Após 10 segundos, a caixa de mensagem *Saved* é exibida e indica que os dados foram salvos. Clique em *OK*, feche o formulário WPF e retorne ao Visual Studio 2010.

## Threads e a classe *BackgroundWorker*

O exemplo mostrado no exercício anterior criou um objeto *Thread* para executar um método em uma nova thread. Uma abordagem alternativa é criar um objeto *BackgroundWorker*. A classe *BackgroundWorker* reside no namespace *System.ComponentModel* e fornece um wrapper em torno das threads. O código a seguir mostra como executar o método *saveData*, por meio de um objeto *BackgroundWorker*.

```
BackgroundWorker workerThread = new BackgroundWorker();
workerThread.DoWork += (x, y) => this.saveData(saveDialog.FileName, member);
workerThread.RunWorkerAsync();
```

Para especificar o método que um objeto *BackgroundWorker* executará, faça uma inscrição no evento *DoWork*, o qual espera que você forneça um delegate *DoWorkEventHandler* (especificado como uma expressão lambda nesse exemplo). O delegate *DoWorkEventHandler* faz referência a um método que aceita dois parâmetros: um parâmetro *object* que indica o item que chamou o objeto *BackgroundWorker*, e um parâmetro *DoWorkEventArgs* que contém informações específicas, passadas para o *BackgroundWorker*, para realizar esse trabalho. O exemplo de código apresentado não utiliza esses dois parâmetros.

Para iniciar a execução de uma thread, chame o método *RunWorkerAsync* do objeto *BackgroundWorker*.

A classe *Thread* é ideal para executar operações simples em segundo plano. Entretanto, em situações mais complexas, a classe *BackgroundWorker* oferece algumas vantagens em relação à classe *Thread*. Em termos específicos, ela fornece o evento *ProgressChanged* que uma thread pode utilizar para informar o andamento de uma tarefa de execução demorada, e o evento *RunWorkerCompleted* que uma thread pode usar para indicar que ela finalizou seu trabalho.

Para obter mais informações sobre a classe *BackgroundWorker*, consulte a documentação fornecida no Visual Studio.

Você melhorou a capacidade de resposta do aplicativo ao utilizar uma thread separada para realizar uma operação demorada e ao passar os dados necessários a essa thread, como um parâmetro. Entretanto, ainda existe uma última reviravolta que você precisa solucionar. Quando o usuário salva os dados de um membro, a caixa de mensagem *Saved* é exibida e confirma que os dados foram salvos corretamente.

Inicialmente, isso funcionava bem, quando a operação de salvamento ocorria rapidamente e o usuário não podia executar quaisquer outras tarefas simultaneamente. Entretanto, atualmente, essa caixa de mensagem pode aparecer enquanto o usuário está inserindo dados de outro membro e pode virar uma perturbação porque ela captura o foco. O usuário tem que responder à caixa de mensagem para poder continuar a inserir os dados, e essa ação pode desconcentrar o usuário, gerando erros de entrada. Uma solução mais eficiente é exibir a mensagem em uma barra de status na parte inferior do formulário. Isso propicia um meio discreto de informar ao usuário que a operação de salvamento foi concluída.

Contudo, há um problema nessa abordagem: a barra de status é um controle criado e pertencente à thread da interface do usuário. Como a thread de segundo plano poderá acessar esse controle e exibir uma mensagem? A resposta está no objeto *Dispatcher* do WPF.

Você pode utilizar o objeto *Dispatcher* para solicitar que a thread da interface do usuário execute um método em nome de outra thread. O objeto *Dispatcher* enfileira essas solicitações e as executa na thread da interface do usuário, em um momento adequado – por exemplo, você pode atribuir uma prioridade a uma solicitação, ao instruir o objeto *Dispatcher* a executar a solicitação somente quando a thread da interface do usuário estiver ociosa. Você pode acessar o objeto *Dispatcher* por meio da propriedade *Dispatcher* de qualquer controle em um formulário WPF, inclusive no próprio formulário.

Para enviar uma solicitação ao objeto *Dispatcher*, chame o método *Invoke*. Esse método é sobrecarregado, mas todas as sobrecargas esperam um objeto *Delegate* que encapsula uma referência a um método que o objeto *Dispatcher* deve executar.

No último exercício deste capítulo, você corrigirá o aplicativo BellRingers para exibir o status da operação de salvamento em uma barra de status no final do formulário, utilizando um objeto *Dispatcher*.

### Use o objeto *Dispatcher* para exibir uma mensagem de status

1. No Visual Studio, exiba o arquivo MainWindow.xaml na janela *Design View*.
2. Na *Toolbox*, selecione o controle *StatusBar* na seção *Controls* e adicione-o ao final do formulário WPF.
3. Na janela *Properties*, defina as propriedades do controle *StatusBar* com os valores listados na tabela a seguir.

Propriedade	Valor
Name	status
Height	23
HorizontalAlignment	Stretch
Margin	0, 0, 0, 0
Style	{StaticResource bellRingersFontStyle}
VerticalAlignment	Bottom
Width	Auto

Essas propriedades instruem a barra de status a ocupar uma única linha no final do formulário.

4. O botão *Clear* fica parcialmente obscurecido pela barra de status, de modo que ele deve ser movido. Na janela *Properties*, mude a propriedade *Margin* do botão *Clear* para **313,378,0,0**.
5. Exiba o arquivo *MainWindow.xaml.cs* na janela *Code and Text Editor*.
6. Adicione a seguinte instrução *using* à lista localizada no início do arquivo:

```
using System.Windows.Threading;
```

7. No método *saveData*, substitua a instrução que exibe a caixa de mensagem pelo código mostrado em negrito a seguir:

```
private void saveData(string fileName, Member member)
{
 using (StreamWriter writer = new StreamWriter(fileName))
 {

 Thread.Sleep(10000);
 Action action = new Action(() => {
 status.Items.Add("Member details saved");
 });
 this.Dispatcher.Invoke(action, DispatcherPriority.ApplicationIdle);
 }
}
```

O método *Invoke* do objeto *Dispatcher* espera uma solicitação na forma de um parâmetro *Delegate* que faça referência a um método a ser executado. Entretanto, *Delegate* é uma classe abstrata. A classe *Action* do namespace *System* é uma implementação concreta da classe *Delegate*, elaborada para fazer referência a um método que não aceita parâmetros nem retorna um resultado. (Em outras palavras, o método apenas executa uma ação.)



**Nota** O tipo genérico *Func<T>* que você conheceu resumidamente no Capítulo 20, “Consultando dados na memória utilizando expressões de consulta”, é outra implementação da classe *Delegate*. Um objeto *Func<T>* faz referência a um método que retorna um objeto do tipo *T*, e é útil para invocar um método que retorna um valor por meio de um delegate.

O código mostrado aqui utiliza uma expressão lambda para definir um método anônimo que exibe a mensagem "Member details saved" (Detalhes do membro gravados) na barra de status. Um objeto *StatusBar* pode exibir vários fragmentos de informação, e para exibir um item, adicione-o à coleção *Items*.

O exemplo do método *Invoke* aqui mostrado obtém uma referência ao objeto *Dispatcher* ao utilizar a propriedade *Dispatcher* do formulário. O segundo parâmetro para o método *Invoke* especifica a prioridade que o objeto *Dispatcher* deve atribuir à solicitação. Esse é um valor da enumeração *DispatcherPriority* do namespace *System.Windows.Threading*. O valor *ApplicationIdle* instrui o objeto *Dispatcher* a executar a solicitação quando o aplicativo não estiver realizando qualquer outro trabalho.

8. No menu *Debug*, clique em *Start Without Debugging*. Quando o formulário WPF for exibido, no menu *File*, clique em *New Member* e adicione alguns detalhes do membro. Em seguida, no menu *File*, clique em *Save Member Details*. Na caixa de diálogo *Bell Ringers*, especifique o nome de arquivo padrão e clique em *Save*. Verifique se o formulário WPF continua respondendo enquanto a thread de segundo plano está em execução.
9. Após 10 segundos, verifique se a caixa de mensagem "Member details saved" aparece na barra de status, no final do formulário.
10. Feche o formulário WPF e retorne ao Visual Studio 2010.

As threads são muito importantes para preservar a capacidade de resposta em uma interface de usuário. Entretanto, ocasionalmente, o gerenciamento das threads pode se tornar realmente difícil; pode ser difícil sincronizar operações simultâneas se você precisar aguardar o término de uma ou mais threads para continuar um processo, e se você criar um número excessivo de threads, o computador pode ficar sobrecarregado e lento. O .NET Framework dispõe de uma abstração de threads chamada *Tasks*, que você pode utilizar para criar e controlar threads de modo gerenciável. Você conhecerá mais detalhes sobre a abstração *Tasks* no Capítulo 27.

Neste capítulo, vimos como gerar menus para que os usuários possam executar operações em um aplicativo, e você também criou menus de atalho, exibidos quando o usuário clica com o botão direito do mouse em um controle ou formulário. Você aprendeu a utilizar as classes de diálogo comuns para solicitar ao usuário o nome e o local de um arquivo. Por último, você conheceu o modelo de threading empregado pelos aplicativos WPF e como usar threads para desenvolver aplicativos com mais capacidade de resposta.

- Se quiser ler o próximo capítulo agora:

Mantenha o Visual Studio 2010 executando e vá para o Capítulo 24.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 23

Para	Faça isto
Criar um menu para um formulário	Adicione um controle <i>DockPanel</i> e o posicione na parte superior do formulário.
Adicione itens de menu a um menu.	Adicione um controle <i>Menu</i> ao controle <i>DockPanel</i> . Adicione elementos <i>MenuItem</i> ao controle <i>Menu</i> . Especifique o texto para um item de menu configurando a propriedade <i>Header</i> e atribua um nome a cada item de menu especificando a propriedade <i>Name</i> . Você pode especificar opcionalmente propriedades para exibir recursos como ícones e menus secundários. Você pode adicionar uma tecla de acesso a um item de menu prefixando a letra apropriada com um caractere de sublinhado.
Criar uma barra separadora em um menu	Adicione um elemento <i>Separator</i> ao menu.
Ativar ou desativar um item de menu	Configure a propriedade <i>IsEnabled</i> como <i>True</i> ou <i>False</i> na janela <i>Properties</i> em tempo de projeto ou escreva código para configurar a propriedade <i>IsEnabled</i> do item de menu como <i>true</i> ou <i>false</i> em tempo de execução.
Executar uma ação quando o usuário clicar em um item de menu	Selecione o item de menu e especifique um método de evento para o evento <i>Click</i> . Adicione seu código ao método de evento.
Criar um menu de atalho	Adicione um <i>ContextMenu</i> aos recursos de janela. Adicione itens ao menu de atalho assim como você adiciona itens a um menu comum.
Associar um menu de atalho a um formulário ou controle	Defina a propriedade <i>ContextMenu</i> do formulário ou controle para referenciar o menu de atalho.
Criar um menu de atalho dinamicamente	Crie um objeto <i>ContextMenu</i> . Preencha a coleção <i>Items</i> desse objeto com objetos <i>MenuItem</i> que definem cada um dos itens de menu. Defina a propriedade <i>ContextMenu</i> do formulário ou controle para referenciar o menu de atalho.
Solicitar o nome de um arquivo a ser salvo	Utilize a classe <i>SaveFileDialog</i> . Exiba a caixa de diálogo utilizando o método <i>ShowDialog</i> . Quando a caixa de diálogo se fecha, a propriedade <i>FileName</i> da instância de <i>SaveFileDialog</i> contém o nome do arquivo selecionado pelo usuário.
Executar uma operação em uma thread de segundo plano	Crie um objeto <i>Thread</i> que faça referência a um método a ser executado. Chame o método <i>Start</i> do objeto <i>Thread</i> para invocar o método. Por exemplo:
	<pre>Thread workerThread = new Thread(     () =&gt; doWork(...)); workerThread.Start();</pre>
Permitir que uma thread de segundo plano acesse os controles gerenciados pela thread da interface do usuário	Crie um delegate <i>Action</i> que faça referência a um método que acessa os controles. Execute o método, ao utilizar o método <i>Invoke</i> , do objeto <i>Dispatcher</i> , e, opcionalmente, especifique uma prioridade. Por exemplo:
	<pre>Action action = new Action(() =&gt; {     status.Items.Add("Member details added"); });  this.Dispatcher.Invoke(action,     DispatcherPriority.ApplicationIdle);</pre>

# Capítulo 24

# Realizando validações

Neste capítulo, você vai aprender a:

- Verificar as informações inseridas por um usuário para garantir que ele não violou as regras do negócio e do aplicativo.
- Vincular propriedades de um controle em um formulário a propriedades de outros controles.
- Utilizar regras de validação de vinculação de dados para validar informações inseridas por um usuário.
- Executar uma validação eficiente, mas discreta.

Nos dois capítulos anteriores, vimos como criar um aplicativo Microsoft Windows Presentation Foundation (WPF) que utiliza uma variedade de controles para a entrada de dados. Você também criou menus para tornar o aplicativo mais fácil de utilizar, e aprendeu a capturar eventos disparados por menus, formulários e controles para que, além de bonito, seu aplicativo seja útil. Você também utilizou threads para agilizar o aplicativo.

Embora o design cuidadoso de um formulário e o uso apropriado dos controles possa ajudar a garantir que as informações inseridas por um usuário façam sentido, com muita frequência você precisa fazer verificações adicionais. Neste capítulo, você vai aprender a validar os dados inseridos por um usuário que executa um aplicativo para garantir que eles correspondam a todas as regras de negócio especificadas pelas exigências do aplicativo.

## Validando os dados

O conceito de validação de entrada é muito simples, mas nem sempre é fácil de implementar, especialmente se a validação envolver o cruzamento de dados inseridos pelo usuário em dois ou mais controles. A regra de negócio subjacente talvez seja relativamente simples e direta, mas, muitas vezes, a validação é executada em uma hora inadequada, dificultando o uso do formulário.

## Estratégias para validar a entrada do usuário

Você pode empregar muitas estratégias para validar as informações inseridas pelos usuários dos seus aplicativos. Uma técnica comum que vários desenvolvedores de Microsoft Windows que conhecem as versões anteriores do Microsoft .NET Framework utilizam é tratar o evento *LostFocus* dos controles. Esse evento é disparado quando o usuário sai de um controle. Você pode adicionar um código a esse evento para examinar os dados do controle do qual o usuário está se afastando e assegurar que eles correspondam aos requisitos do aplicativo antes de permitir que o cursor se move. O problema dessa estratégia é que, frequentemente, é necessário confrontar os dados inseridos em um controle com os valores de outros controles, e a lógica de validação pode tornar-se bem complexa;

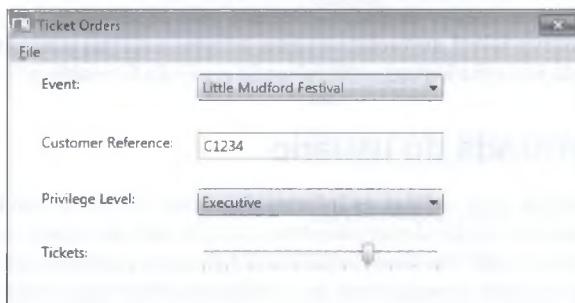
na maioria das vezes, você acaba tendo que repetir uma lógica semelhante na rotina de tratamento de evento *LostFocus* para vários controles. Além disso, você não tem poder algum sobre a sequência em que o usuário passa de um controle para outro. Os usuários podem se mover pelos controles de um formulário em uma ordem qualquer, portanto, você nem sempre pode assumir que cada controle contém um valor válido se estiver comparando um controle particular com outros do formulário.

Uma outra questão fundamental dessa estratégia é que ela pode deixar a lógica da validação dos elementos de apresentação de um aplicativo muito amarrada à lógica de negócios. Se os requisitos de negócio mudarem, talvez você precise modificar a lógica de validação, e a manutenção pode tornar-se uma tarefa complexa.

Com o WPF, você pode definir regras de validação como parte do modelo de negócios utilizado pelos seus aplicativos, podendo então referenciar essas regras a partir da descrição Extensible Application Markup Language™ (XAML) da interface com o usuário. Para fazer isso, defina as classes requeridas pelo modelo de negócios e então vincule as propriedades dos controles da interface de usuário às propriedades expostas por essas classes. (Em tempo de execução, o WPF pode criar instâncias dessas classes.) Quando você modifica os dados de um controle, eles podem ser automaticamente copiados de volta para a propriedade especificada na instância de classe apropriada do modelo de negócios e validados. Você aprenderá mais sobre a vinculação de dados na Parte V, “Gerenciando dados”. Para os propósitos deste capítulo, iremos nos concentrar nas regras de validação que você pode associar com a vinculação de dados.

## Um exemplo – order tickets (solicitar ingressos) para eventos

Considere um cenário simples. Você foi solicitado a construir um aplicativo que permita aos clientes solicitarem ingressos para eventos. Parte do aplicativo deve possibilitar que um cliente insira seus detalhes, especifique um evento e selecione o número de ingressos necessários. Um cliente tem um nível de privilégio (Standard, Premium, Executive ou Premium Executive), e quanto mais alto for esse nível, tanto maior a quantidade de ingressos que o cliente poderá requerer. (Os clientes do nível Standard, no máximo dois ingressos por evento; os clientes do nível Premium, quatro ingressos; os clientes do nível Executive, oito ingressos; e os do nível Premium Executive, dez ingressos). Você decide criar um protótipo de formulário, como o da figura a seguir.



Você precisa garantir que a entrada do usuário é válida e consistente. Mais especificamente, o cliente deve fazer o seguinte:

- Selecionar um evento. O protótipo do aplicativo utiliza um conjunto codificado de eventos. Em um aplicativo de produção, você armazena o conjunto de eventos em um banco de dados e emprega vinculação de dados para recuperar e exibir esses eventos. Você aprenderá a fazer isso no Capítulo 26, “Exibindo e editando dados com o Entity Framework e vinculação de dados”.
- Inserir um número de referência de cliente. O protótipo do aplicativo não verifica esse número de referência.
- Especificar um nível de privilégio. Mais uma vez, o protótipo do aplicativo não verifica se o cliente realmente tem o nível de privilégio informado.
- Escolher uma quantidade de ingressos maior que 0 e menor ou igual ao valor permitido pelo nível de privilégio do cliente.

Ao terminar de inserir os dados, o usuário clica no item *Purchase* no menu *File*. O aplicativo real direciona o usuário para uma tela que lhe permite informar os detalhes do pagamento. Nesse protótipo do aplicativo, o aplicativo simplesmente exibe uma caixa de mensagem que confirma a entrada do usuário.

## Realizando a validação com vinculação de dados

Nos próximos exercícios, você examinará o aplicativo Ticket Ordering e adicionará regras de validação utilizando vinculação de dados. Como precaução, um dos exercícios lhe mostrará como é fácil fazer a validação errada no momento errado e acabar tornando o aplicativo inútil!

### Examine o formulário *Ticket Orders*

1. Inicie o Microsoft Visual Studio 2010 se ele ainda não estiver executando.
2. Abra o projeto OrderTickets, localizado na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 24\OrderTickets na sua pasta Documentos.
3. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
4. Quando o formulário for exibido, não digite quaisquer dados, clique imediatamente em *Purchase* no menu *File*.

O aplicativo exibirá uma caixa de mensagem com o texto “Purchasing 0 tickets for customer: for event:”. No momento, o aplicativo permite que o usuário compre ingressos sem especificar a finalidade, o evento ou sequer o número de ingressos necessários.

5. Clique em *OK* e retorne ao formulário *Ticket Orders*.
6. Na caixa de combinação *Event*, clique em *Little Mudford Festival*.

7. Na caixa *Customer Reference*, digite **C1234**.
8. Na caixa de combinação *Privilege Level*, clique em *Premium*.
9. Utilizando o controle deslizante *Tickets*, clique nesse controle e arraste-o para o lado direito. Isso especifica 10 ingressos.
10. No menu *File*, clique em *Purchase*.

O aplicativo exibirá uma caixa de mensagem com o texto "Purchasing 10 tickets for Premium customer: C1234 for event: Little Mudford Festival". Observe que o aplicativo não verifica se o número de ingressos ultrapassa o número permitido pelo nível de privilégio do cliente.

11. Clique em *OK*, feche o formulário e retorne ao Visual Studio 2010.

Atualmente, esse formulário não é muito útil, ele não valida os dados inseridos pelo usuário e o controle deslizante dificulta a determinação exata da quantidade de ingressos que o usuário selecionou, até que ele clique no botão *Purchase*. (Você precisa contar os ingressos abaixo do controle.) Esse segundo problema é o mais fácil de corrigir, então comece por ele. No próximo exercício, você adicionará um controle *TextBox* ao formulário e usará vinculação de dados para exibir o valor atual da barra deslizante nesse controle.

### Use vinculação de dados para exibir o número de ingressos solicitados

1. No *Solution Explorer*, clique duas vezes no arquivo *TicketForm.xaml*.
2. Na *Toolbox*, adicione um controle *TextBox* ao formulário e posicione-o à direita do controle deslizante. Na janela *Properties*, defina as propriedades desse controle com os valores indicados na tabela a seguir.

Propriedade	Valor
Name	tickets
Height	23
Width	25
Margin	380, 170, 0, 0
IsReadOnly	True (selecionada)
.TextAlignment	Right
HorizontalAlignment	Left
VerticalAlignment	Top

3. No painel XAML, edite a definição do controle *tickets TextBox*. Adicione o elemento filho *TextBox.Text*, como mostrado em negrito a seguir, certificando-se de substituir a tag delimitadora de fechamento (*/>*) do controle *TextBox* por um delimitador comum (*>*), e de incluir uma tag de fechamento *</TextBox>*:

```
<TextBox Height="23" HorizontalAlignment="Left" Margin="380,170,0,0"
 Name="tickets" VerticalAlignment="Top" Width="25" TextAlignment="Right"
 IsReadOnly="True">
 <TextBox.Text>
 </TextBox.Text>
</TextBox>
```

4. No elemento filho *TextBox.Text*, adicione o elemento *Binding* mostrado em negrito a seguir:

```
<TextBox ...>
 <TextBox.Text>
 <Binding ElementName="numberOfTickets" Path="Value" />
 </TextBox.Text>
</TextBox>
```

Esse elemento *Binding* associa a propriedade *Text* do controle *TextBox* à propriedade *Value* do controle *Slider*. (O controle *Slider* recebe o nome *numberOfTickets*.) Quando o usuário alterar esse valor, o *TextBox* será automaticamente atualizado. Observe que o controle *TextBox* exibe o valor 0 na janela *Design View* – esse é o valor padrão do controle *Slider*.

5. No menu *Debug*, clique em *Start Without Debugging*.
6. Quando o formulário for exibido, arraste a barra no controle *Slider* e verifique se número de ingressos aparece no controle *TextBox*, à direita.
7. Feche o formulário e retorne ao Visual Studio.

Agora, você já pode se concentrar em validar os dados inseridos pelo usuário. Você pode adotar algumas abordagens, mas, nesse caso, a proposta recomendada abrange a criação de uma classe que possa modelar a entidade cujos dados você está inserindo. Você pode adicionar a lógica de validação a essa classe, e depois associar as propriedades existentes na classe aos diversos campos contidos no formulário. Se você digitar dados inválidos no formulário, as regras de validação na classe poderão lançar uma exceção que você poderá capturar e exibir no formulário.

Comece criando uma classe para modelar um pedido do cliente, e depois aprenda a utilizar essa classe para garantir que o usuário sempre especifique um evento e digite um número de referência do cliente.

**Crie a classe *TicketOrder* com a lógica de validação para especificar um evento e para cobrar uma entrada de um número de referência do cliente**

1. No *Solution Explorer*, clique com o botão direito do mouse no projeto *OrderTickets*, aponte para *Add* e clique em *Class*.
2. Na caixa de diálogo *Add New Item – OrderTickets*, na caixa de texto *Name*, digite **TicketOrder.cs** e clique em *Add*.
3. Na janela *Code and Text Editor* que exibe o arquivo *TickerOrder.cs*, adicione os campos privados *eventName* e *customerReference* mostrados em negrito a seguir à classe *TicketOrder*:

```
class TicketOrder
{
 private string eventName;
 private string customerReference;
}
```

4. Adicione a seguinte propriedade pública *EventName* à classe *TicketOrder*, como mostrado em negrito, com base no campo *eventName* adicionado na etapa anterior:

```
class TicketOrder
{
 ...
 public string EventName
 {
 get { return this.eventName; }
 set
 {
 if (String.IsNullOrEmpty(value))
 {
 throw new ApplicationException
 ("Specify an event");
 }
 else
 {
 this.eventName = value;
 }
 }
 }
}
```

O bloco *set* da propriedade examina o valor informado para o nome do evento, e, se estiver vazio, ele lançará uma exceção com a mensagem adequada.

5. Adicione a seguinte propriedade pública *CustomerReference* à classe *TicketOrder*, como mostrado em negrito:

```
class TicketOrder
{
 ...
 public string CustomerReference
 {
```

```

 get { return this.customerReference; }
 set
 {
 if (String.IsNullOrEmpty(value))
 {
 throw new ApplicationException
 ("Specify the customer reference number");
 }
 else
 {
 this.customerReference = value;
 }
 }
 }
}

```

Essa propriedade é semelhante à *EventName*. O bloco *set* da propriedade examina o valor informado para o número de referência do cliente, e, se estiver vazio, ele lançará uma exceção.

Agora que você já criou a classe *TicketOrder*, a próxima etapa é vincular a caixa de texto *customerReference* do formulário à propriedade *CustomerReference* da classe.

### Vincule o controle caixa de texto do formulário à propriedade da classe *TicketOrder*

1. No *Solution Explorer*, clique duas vezes no arquivo *TicketForm.xaml* para exibir o formulário na janela *Design View*.
2. No painel *XAML*, adicione a declaração de namespace XML, mostrada em negrito a seguir, à definição *Window*:

```

<Window x:Class="OrderTickets.TicketForm"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:ticketOrder="clr-namespace:OrderTickets"
 Title="Ticket Orders" Height="250" Width="480" ResizeMode="NoResize">

```

Essa declaração é semelhante a uma instrução *using* em código Microsoft C#, e permite que você faça referência aos tipos do namespace *OrderTickets*, no código XAML da janela.

3. Adicione o seguinte elemento *Window.Resources*, mostrado em negrito, à janela:

```

<Window x:Class="OrderTickets.TicketForm"

 ... ResizeMode="NoResize">
 <Window.Resources>
 <ticketOrder:TicketOrder x:Key="orderData" />
 </Window.Resources>
 <Grid>

```

Esse recurso cria uma nova instância da classe *TicketOrder*. Para fazer referência a essa instância, utilize o valor-chave *orderData* em algum local na definição XAML da janela.

4. Localize a definição da caixa de texto *customerReference* no painel XAML e modifique-a, como mostrado em negrito a seguir, certificando-se de substituir a tag delimitadora de fechamento (*>*) do controle *TextBox* por um delimitador comum (*>*), e de adicionar uma tag de fechamento *</TextBox>*:

```
<TextBox Height="23" HorizontalAlignment="Left" Margin="156,78,0,0"
 Name="customerReference" VerticalAlignment="Top" Width="205">
 <TextBox.Text>
 <Binding Source="{StaticResource orderData}"
 Path="CustomerReference" />
 </TextBox.Text>
</TextBox>
```

Esse código vincula os dados exibidos na propriedade *Text* dessa caixa de texto ao valor da propriedade *CustomerReference* do objeto *orderData*. Se o usuário atualizar o valor da caixa de texto *customerReference* no formulário, os novos dados serão automaticamente copiados no objeto *orderData*. Lembre-se de que a propriedade *CustomerReference* da classe *TicketOrder* verifica se o usuário realmente especificou um valor.

5. Modifique a definição da vinculação adicionada na etapa anterior, e adicione um elemento filho, *Binding.ValidationRules*, como mostrado em negrito a seguir:

```
<TextBox Height="23" HorizontalAlignment="Left" Margin="156,78,0,0"
 Name="customerReference" VerticalAlignment="Top" Width="205">
 <TextBox.Text>
 <Binding Source="{StaticResource orderData}"
 Path="CustomerReference">
 <Binding.ValidationRules>
 <ExceptionValidationRule/>
 </Binding.ValidationRules>
 </Binding>
 </TextBox.Text>
</TextBox>
```

O elemento *ValidationRules* de uma vinculação (binding) permite especificar a validação que o aplicativo deve executar quando o usuário inserir dados nesse controle. O elemento *ExceptionValidationRule* é uma regra interna, que procura as exceções lançadas pelo aplicativo quando os dados nesse controle forem alterados. Ao detectar alguma exceção, ele destacará o controle para que o usuário perceba a ocorrência de um problema na entrada.

6. Adicione a vinculação equivalente e a regra de vinculação à propriedade *Text* da caixa de combinação *eventList*, associando-a à propriedade *EventName* do objeto *orderData*, como mostrado em negrito a seguir:

```
<ComboBox Height="23" HorizontalAlignment="Left" Margin="156,29,0,0"
 Name="eventList" VerticalAlignment="Top" Width="205" >
 <ComboBox.Text>
 <Binding Source="{StaticResource orderData}" Path="EventName" >
 <Binding.ValidationRules>
 <ExceptionValidationRule/>
 </Binding.ValidationRules>
 </Binding>
 </ComboBox.Text>
 <ComboBox.Items>

 </ComboBox.Items>
</ComboBox >
```

7. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
8. Quando o formulário for exibido, clique em *Purchase* no menu *File*, sem inserir quaisquer dados. É exibida a mensagem "Purchasing 0 tickets for customer: for event:".
9. Clique em *OK*.
10. No formulário *Ticket Orders*, selecione *Little Mudford Festival* na caixa de combinação *Event*, digite **C1234** na caixa de texto *customerReference* e selecione *Premium* na caixa de combinação *Privilege Level*. Nada excepcional deve acontecer.
11. Clique na caixa de texto *customerReference*, exclua o número de referência inserido e clique novamente na caixa de combinação *Privilege Level*.

Dessa vez, a caixa de texto *customerReference* é destacada com uma borda vermelha. Quando a vinculação tentou copiar o valor que o usuário inseriu para o objeto *TickerOrder*, esse valor era uma string vazia, de modo que a propriedade *CustomerReference* lançou uma exceção *ApplicationException*. Nessas circunstâncias, um controle que utiliza vinculação indica que ocorreu uma exceção ao exibir uma borda vermelha.

12. Digite **C1234** na caixa de texto *customerReference* novamente e clique na caixa de combinação *Privilege Level*.

O quadro vermelho ao redor da caixa de texto *customerReference* desaparece.

13. Exclua novamente o valor contido na caixa de texto *customerReference*. No menu *File*, clique em *Purchase*. Mas que surpresa! Nenhuma borda vermelha aparece ao redor da caixa de texto *customerReference*.

14. Na caixa de mensagem, clique em *OK* e clique na caixa de combinação *Privilege Level*.

A borda vermelha volta a aparecer ao redor da caixa de texto *customerReference*.

15. Feche o formulário e retorne ao Visual Studio 2010.

Nesse ponto, há pelo menos duas perguntas que você deve fazer:

- Por que o formulário nem sempre detecta quando o usuário se esqueceu de inserir um valor em uma caixa de texto? A resposta é que a validação ocorre somente quando a caixa de texto perde o foco. Isso por sua vez só acontece quando o usuário move o foco para outro controle do formulário. Na verdade, menus não são tratados como se fossem partes dos formulários. (Eles são tratados de uma maneira diferente.) Ao selecionar um item de menu, você não se move para um outro controle no formulário e, consequentemente, a caixa de texto ainda não perdeu o foco. Somente quando você clica na caixa de combinação *Privilege Level* (ou em algum outro controle) é que o foco é movido e a validação ocorre. Além disso, a caixa de texto *customerReference* e a caixa de combinação *Event* inicialmente estão vazias. Se você sair de um desses controles sem selecionar ou digitar nada, a validação não será realizada. Só depois de digitar ou selecionar algo e então excluir isso é que a validação ocorre. Resolveremos esses problemas mais adiante neste capítulo.
- Como é possível fazer o formulário exibir uma mensagem de erro significativa em vez de apenas destacar que há um problema com a entrada em um controle? Você pode capturar a mensagem gerada por uma exceção e exibi-la em outra parte do formulário.

Veremos como fazer isso no seguinte exercício, que responderá a essas perguntas.

### Adicione um estilo para exibir as mensagens de exceção

1. Na janela *Design View* que exibe o arquivo *TicketForm.xaml*, no painel *XAML*, adicione o seguinte estilo mostrado em negrito ao elemento *Window.Resources*:

```
<Window.Resources>
 <ticketOrder:TicketOrder x:Key="orderData" />
 <Style x:Key="errorStyle" TargetType="Control">
 <Style.Triggers>
 <Trigger Property="Validation.HasError" Value="True">
 <Setter Property="ToolTip"
 Value="{Binding RelativeSource={x:Static RelativeSource.Self},
 Path=(Validation.Errors)[0].ErrorContent}" />
 </Trigger>
 </Style.Triggers>
 </Style>
</Window.Resources>
```

Esse estilo contém um gatilho que detecta quando a propriedade *Validation.HasError* do controle é configurada como *true*. Isso ocorre se uma regra de validação de vinculação para o controle gerar uma exceção. O gatilho configura a propriedade *ToolTip* do controle atual para exibir o texto da exceção. Uma explicação detalhada sobre a sintaxe de vinculação aqui apresentada está fora do escopo deste livro, mas a origem da vinculação *{Binding RelativeSource={x:Static RelativeSource.Self}}* é uma referência ao controle atual, e o caminho de vinculação *(Validation.Errors)[0].ErrorContent* associa a primeira mensagem de exceção localizada nessa origem de vinculação com a propriedade

*ToolTip.* (Uma exceção poderia lançar mais exceções, todas geram mensagens próprias. Mas a primeira mensagem é geralmente a mais significativa.)

- Aplique o estilo *errorStyle* aos controles *eventList* e *customerReference*, como mostrado em negrito:

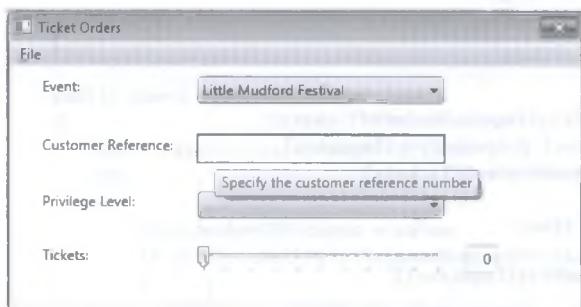
```
<ComboBox Style="{StaticResource errorStyle}" ... Name="eventList" ... >
 ...
</ComboBox>
<TextBox Style="{StaticResource errorStyle}" ... Name="customerReference" ... >
 ...
</TextBox>
```

- No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
- Quando o formulário for exibido, na caixa de combinação *Event*, selecione *Little Mudford Festival*, digite **C1234** na caixa de texto *customerReference* e clique na caixa de combinação *Privilege Level*.
- Clique na caixa de texto *customerReference*, exclua o número de referência inserido e clique novamente na caixa de combinação *Privilege Level*. A caixa de texto *customerReference* será destacada com uma borda vermelha.



**Nota** Certifique-se de excluir o conteúdo da caixa de texto *foreName* em vez de apenas redefinir o texto com espaços.

- Posicione o ponteiro do mouse sobre a caixa de texto *customerReference*. Deve aparecer uma Dica de tela (ScreenTip), com a mensagem “Specify the customer reference number,” como esta:



Essa é a mensagem da exceção *ApplicationException* lançada pela propriedade *CustomerReference* na classe *TicketOrder*.

- Feche o formulário e retorne ao Visual Studio 2010.

Ainda restam alguns problemas a serem corrigidos, mas você os solucionará depois que aprender a validar o nível de privilégio e o número de ingressos, e assegurar a respectiva consistência.

### Adicione propriedades para validar o nível de privilégio e o número de ingressos

1. Alterne para a janela *Code and Text Editor* que exibe o arquivo *TicketOrder.cs*.
2. Adicione ao arquivo a enumeração *PrivilegeLevel* mostrada em negrito, acima da classe *TicketOrder*:

```
enum PrivilegeLevel { Standard, Premium, Executive, PremiumExecutive }

class TicketOrder
{
 ...
}
```

Você utilizará essa enumeração para especificar o tipo da propriedade *PrivilegeLevel* na classe *TicketOrder*.

3. Adicione os campos privados *privilegeLevel* e *numberOfTickets* à classe *TicketOrder*, como mostrado em negrito a seguir:

```
class TicketOrder
{
 private string eventName;
 private string customerReference;
private PrivilegeLevel privilegeLevel;
private short numberOfTickets;

 ...
}
```

4. Adicione o método booleano privado *checkPrivilegeAndNumberOfTickets* à classe *TicketOrder*, como mostrado em negrito a seguir:

```
class TicketOrder
{
 ...

 private bool checkPrivilegeAndNumberOfTickets(
 PrivilegeLevel proposedPrivilegeLevel,
 short proposedNumberOfTickets)
 {
 bool retVal = false;

 switch (proposedPrivilegeLevel)
 {
 case PrivilegeLevel.Standard:
 retVal = (proposedNumberOfTickets <= 2);
 break;

 case PrivilegeLevel.Premium:
 retVal = (proposedNumberOfTickets <= 4);
 break;

 case PrivilegeLevel.Executive:
 retVal = (proposedNumberOfTickets <= 8);
 break;
 }
 }
}
```

```

 case PrivilegeLevel.PremiumExecutive:
 retVal = (proposedNumberOfTickets <= 10);
 break;
 }

 return retVal;
}
}

```

Esse método examina os valores nos parâmetros *proposedPrivilegeLevel* e *proposedNumberOfTickets* e testa a sua consistência de acordo com as regras de negócio descritas anteriormente neste capítulo. Se os valores forem consistentes, esse método retornará *true*; caso contrário, retornará *false*.

5. Adicione as propriedades públicas *PrivilegeLevel* e *NumberOfTickets*, mostradas em negrito a seguir, à classe *TicketOrder*. Observe que o tipo da propriedade *PrivilegeLevel* é a enumeração *PrivilegeLevel*:

```

class TicketOrder
{
 ...
 public PrivilegeLevel PrivilegeLevel
 {
 get { return this.privilegeLevel; }
 set
 {
 this.privilegeLevel = value;
 if (!this.checkPrivilegeAndNumberOfTickets(value, this.numberOfTickets))
 {
 throw new ApplicationException(
 "Privilege level too low for this number of tickets");
 }
 }
 }

 public short NumberOfTickets
 {
 get { return this.numberOfTickets; }
 set
 {
 this.numberOfTickets = value;
 if (!this.checkPrivilegeAndNumberOfTickets(this.privilegeLevel, value))
 {
 throw new ApplicationException(
 "Too many tickets for this privilege level");
 }

 if (this.numberOfTickets <=0)
 {
 throw new ApplicationException(
 "You must buy at least one ticket");
 }
 }
 }
}

```

Os blocos *set* dessas propriedades chamam o método *CheckPrivilegeAndNumberOfTickets* para verificar se os campos *privilegeLevel* e *numberOfTickets* são correspondentes, e acionam uma exceção se esses campos não coincidirem.

Além disso, o bloco *set* da propriedade *NumberOfTickets* verifica se o usuário especificou pelo menos um ingresso. Não é necessário verificar se o usuário informou um valor para a propriedade *PrivilegeLevel*, porque ele definirá como padrão o nível *Standard* (o primeiro item na enumeração *PrivilegeLevel*).

- Adicione o método *ToString*, mostrado em negrito a seguir, à classe *TicketOrder*:

```
class TicketOrder
{
 ...
 public override string ToString()
 {
 string formattedString = String.Format("Event: {0}\tCustomer: {1}\tPrivilege:
{2}\tTickets: {3}",
 this.eventName, this.customerReference,
 this.privilegeLevel.ToString(), this.numberOfTickets.ToString());
 return formattedString;
 }
}
```

Você usará esse método para exibir os detalhes dos pedidos de ingressos para verificar se os dados estão corretos.

A próxima etapa é vincular a caixa de combinação *privilegeLevel* e o controle deslizante *numberOfTickets* do formulário a essas novas propriedades. Entretanto, se você parar e pensar alguns instantes, perceberá que existe um pequeno problema na propriedade *PrivilegeLevel*. Você precisa vincular a propriedade *Text* da caixa de combinação *privilegeLevel* à propriedade *PrivilegeLevel* do objeto *TicketOrder* criado pelo formulário. O tipo da propriedade *Text* é *string*. O tipo da propriedade *PrivilegeLevel* é *PrivilegeLevel* (uma enumeração). Para que a vinculação funcione, você deve converter entre valores de *string* e *PrivilegeLevel*. Felizmente, com o mecanismo de vinculação implementado pelo WPF, é possível especificar uma classe conversora para executar ações como essa.

 **Nota** Uma vinculação WPF pode converter automaticamente entre uma enumeração e uma string se os valores da string forem idênticos aos nomes de cada elemento na enumeração. No aplicativo *Ticket Order*, os três primeiros itens na caixa de combinação *privilegeLevel* (*Standard*, *Premium* e *Executive*) correspondem diretamente aos elementos de mesmos nomes, existentes na enumeração *PrivilegeLevel*. Entretanto, o último item na caixa de combinação é *Premium Executive* (com um espaço), mas o elemento correspondente na enumeração é chamado *PremiumExecutive* (sem o espaço entre as duas palavras). A vinculação WPF não pode converter entre esses dois valores, dali a necessidade de uma classe conversora.

Os métodos conversores residem em classes próprias que devem implementar a interface *IValueConverter*. Essa interface define dois métodos: *Convert*, que converte entre o tipo utilizado pela propriedade na classe que está fornecendo os dados para a vinculação e o tipo exibido no formu-

lário, e *ConvertBack*, que converte os dados do tipo exibido no formulário para o tipo requerido pela classe.

## Crie a classe e os métodos conversores

1. No arquivo *TicketOrder.cs*, adicione a seguinte instrução *using* à lista na parte superior do arquivo.

```
using System.Windows.Data;
```

A interface *IValueConverter* é definida nesse namespace.

2. Adicione a classe *PrivilegeLevelConverter* mostrada a seguir ao final do arquivo, depois da classe *TicketOrder*.

```
[ValueConversion(typeof(string), typeof(PrivilegeLevel))]
public class PrivilegeLevelConverter : IValueConverter
{
}
```

O texto em colchetes imediatamente acima da classe é um exemplo de um atributo. Um atributo fornece metadados descritivos para uma classe. O atributo *ValueConversion* é utilizado pelas ferramentas como o designer de WPF na janela *Design View* para verificar se você está aplicando a classe corretamente quando você a referencia. Os parâmetros para o atributo *ValueConversion* especificam o tipo do valor exibido pelo formulário (*string*) e o tipo do valor da propriedade correspondente na classe (*PrivilegeLevel*). Veremos mais exemplos de atributos nos capítulos posteriores neste livro.

3. Na classe *PrivilegeConverter*, adicione o método *Convert* mostrado em negrito:

```
[ValueConversion(typeof(string), typeof(PrivilegeLevel))]
public class PrivilegeLevelConverter: IValueConverter
{
 public object Convert(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
 {
 PrivilegeLevel privilegeLevel = (PrivilegeLevel)value;
 string convertedPrivilegeLevel = String.Empty;

 switch (privilegeLevel)
 {
 case PrivilegeLevel.Standard:
 convertedPrivilegeLevel = "Standard";
 break;

 case PrivilegeLevel.Premium:
 convertedPrivilegeLevel = "Premium";
 break;

 case PrivilegeLevel.Executive:
 convertedPrivilegeLevel = "Executive";
 break;
 }
 }

 public object ConvertBack(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
 {
 string convertedPrivilegeLevel = value.ToString();
 PrivilegeLevel privilegeLevel = null;

 if (convertedPrivilegeLevel == "Standard")
 privilegeLevel = PrivilegeLevel.Standard;
 else if (convertedPrivilegeLevel == "Premium")
 privilegeLevel = PrivilegeLevel.Premium;
 else if (convertedPrivilegeLevel == "Executive")
 privilegeLevel = PrivilegeLevel.Executive;

 return privilegeLevel;
 }
}
```

```

 case PrivilegeLevel.PremiumExecutive:
 convertedPrivilegeLevel = "Premium Executive";
 break;
 }

 return convertedPrivilegeLevel;
}
}

```

A assinatura do método *Convert* é definida pela interface *IValueConverter*. O parâmetro *value* é o valor na classe a partir do qual você está convertendo. (Por enquanto, ignore os outros parâmetros.) O valor de retorno desse método são os dados vinculados à propriedade no formulário. Nesse caso, o método *Convert* converte um valor *PrivilegeLevel* em uma *string*. Observe que o parâmetro *value* é passado como um *object*, de modo que você deve fazer o respectivo casting para o tipo adequado antes de utilizá-lo.

4. Adicione o método *ConvertBack*, mostrado em negrito a seguir, à classe *PrivilegeLevelConverter*:

```

[ValueConversion(typeof(string), typeof(PrivilegeLevel))]
public class PrivilegeLevelConverter: IValueConverter
{
 ...
 public object ConvertBack(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
 {
 PrivilegeLevel privilegeLevel = PrivilegeLevel.Standard;

 switch ((string)value)
 {
 case "Standard":
 privilegeLevel = PrivilegeLevel.Standard;
 break;

 case "Premium":
 privilegeLevel = PrivilegeLevel.Premium;
 break;

 case "Executive":
 privilegeLevel = PrivilegeLevel.Executive;
 break;

 case "Premium Executive":
 privilegeLevel = PrivilegeLevel.PremiumExecutive;
 break;
 }

 return privilegeLevel;
 }
}

```

O método *ConvertBack* também faz parte da interface *IValueConverter*. No método *ConvertBack*, o parâmetro *value* é agora o valor no formulário que você está novamente convertendo em

um valor do tipo adequado à classe. Nesse caso, o método *ConvertBack* converte os dados de uma *string* (exibidos na propriedade *Title* na caixa de combinação) para o valor *Title* correspondente.

- No menu *Build*, clique em *Build Solution*. Verifique se a solução é compilada corretamente, corrija os erros e recompile a solução, se necessário.

### Vincule a caixa de combinação e os controles deslizantes no formulário às propriedades da classe *TicketOrder*

- Retorne à janela *Design View* que exibe o arquivo *TicketForm.xaml*.
- No painel *XAML*, adicione um objeto *PrivilegeLevelConverter* como um recurso à janela, e especifique um valor-chave de *privilegeLevelConverter*, como mostrado em negrito a seguir:

```
<Window.Resources>
 <ticketOrder:TicketOrder x:Key="orderData" />
 <ticketOrder:PrivilegeLevelConverter x:Key="privilegeLevelConverter" />
 ...
</Window.Resources>
```

- Localize a definição do controle caixa de combinação *privilegeLevel*, e atribua um estilo ao controle, por meio do estilo *errorStyle*. Após a lista dos itens da caixa de combinação, adicione o código XAML, código mostrado em negrito a seguir, para vincular a propriedade *Title* da caixa de combinação à propriedade *Title* no objeto *orderData*, especificando o recurso *titleConverter* como um objeto que fornece os métodos conversores:

```
<ComboBox Style="{StaticResource errorStyle}" ... Name="privilegeLevel" ...>
 <ComboBox.Text>
 <Binding Source="{StaticResource orderData}" Path="PrivilegeLevel"
 Converter="{StaticResource privilegeLevelConverter}" >
 <Binding.ValidationRules>
 <ExceptionValidationRule />
 </Binding.ValidationRules>
 </Binding>
 </ComboBox.Text>
 <ComboBox.Items>
 ...
 </ComboBox.Items>
</ComboBox>
```

- Modifique a definição do controle deslizante *numberOfTickets*. Aplique o estilo *errorStyle* e vincule a propriedade *Value* à propriedade *NumberOfTickets* do objeto *orderData*, como mostrado em negrito a seguir:

```
<Slider Style="{StaticResource errorStyle}" Height="22"
 HorizontalAlignment="Left" Margin="156,171,0,0", Name="numberOfTickets"
 VerticalAlignment="Top" Width="205" SmallChange="1"
 TickPlacement="BottomRight" Maximum="10" IsSnapToTickEnabled="True" >
 <Slider.Value>
 <Binding Source="{StaticResource orderData}" Path="NumberOfTickets">
 <Binding.ValidationRules>
```

```
 <ExceptionValidationRule />
 </Binding.ValidationRules>
</Binding>
</Slider.Value>
</Slider>
```

5. No menu *View*, clique em *Code* para alternar para a janela *Code and Text Editor* que exibe o arquivo *TicketForm.xaml.cs*.
6. Altere o código no método *purchaseTickets\_Click*, como mostrado em negrito a seguir:

```
private void purchaseTickets_Click(object sender, RoutedEventArgs e)
{
 Binding ticketOrderBinding =
 BindingOperations.GetBinding(privilegeLevel, ComboBox.TextProperty);
 TicketOrder ticketOrder = ticketOrderBinding.Source as TicketOrder;
 MessageBox.Show(ticketOrder.ToString(), "Purchased");
}
```

Esse código exibe os detalhes do pedido na caixa de mensagem. (Na verdade, ele ainda não salva o pedido de ingressos em lugar algum.) O método estático *GetBinding* da classe *BindingOperations* retorna uma referência ao objeto ao qual a propriedade especificada está vinculada. Nesse caso, o método *GetBinding* recupera o objeto vinculado à propriedade *Text* da caixa de combinação *title*. Esse deve ser o mesmo objeto referenciado pelo recurso *orderData*. Na realidade, o código poderia ter consultado qualquer uma das propriedades dos controles *eventList*, *customerReference*, *privilegeLevel* ou *numberOfTickets*, para recuperar a mesma referência. A referência é retornada como um objeto *Binding*. Em seguida, o código faz um casting desse objeto *Binding* em um objeto *TicketOrder* antes de exibir seus detalhes.

Você já pode executar o aplicativo novamente e acompanhar o desempenho da validação.

### Execute o aplicativo e teste a validação

1. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
2. Na caixa de combinação *Privilege Level*, clique em *Premium*.
3. Defina o controle deslizante *Tickets* como 5.

O método *CheckPrivilegeAndNumberOfTickets* na classe *TicketOrder* gera uma exceção porque o nível de privilégio e o número de ingressos não combinam. O controle deslizante *Tickets* é destacado com uma borda vermelha. Posicione o ponteiro do mouse sobre o controle deslizante *Tickets* e verifique se o texto da dica de tela (ScreenTip) "Too many tickets for this privilege level" é exibido.

4. Na caixa de combinação *Privilege Level*, clique em *Executive*.
- Embora o nível de privilégio já seja suficiente para que o cliente solicite 5 ingressos, o controle deslizante continua destacado.
5. Defina o controle deslizante *Tickets* como 6.

Observe que o destaque vermelho desaparece. A validação ocorre somente quando você muda o valor em um controle, não quando você altera o valor em um controle diferente.

6. Na caixa de combinação *Privilege Level*, clique em *Standard*.

A caixa de combinação é destacada. Se você posicionar o cursor do mouse sobre a caixa de combinação, ela deverá exibir a mensagem "Privilege level too low for this number of tickets".

7. Defina o controle deslizante *Tickets* como 5. Agora, o controle deslizante também está destacado.

8. No menu *File*, clique em *Purchase*.

Uma caixa de mensagem exibe o nível de privilégio (*Standard*) e o número de ingressos (5) do pedido. Além disso, as referências ao evento e ao cliente estão em branco. Embora o formulário contenha dados incorretos e ausentes, você ainda pode fazer a compra!

9. Clique em *OK* e digite **C1234** na caixa texto *customerReference*, mas não clique fora dessa caixa de texto.

10. No menu *File*, clique em *Purchase* novamente.

A caixa de mensagem não contém a referência ao cliente. Isso acontece porque a caixa de texto *customerReference* do formulário não perdeu o foco. Como mencionado anteriormente, a validação da vinculação de dados para uma caixa de texto só ocorre quando o usuário clica em outro controle no formulário. Por padrão, o mesmo se aplica aos próprios dados, pois são copiados para o objeto *orderDetails* somente quando a caixa de texto perde o foco. Na verdade, é exatamente a ação de copiar os dados do formulário para o objeto *orderDetails* que aciona a validação.

11. Clique em *OK*, clique na caixa de combinação *Event*, e selecione *Little Mudford Festival*.

12. No menu *File*, clique em *Purchase*.

Dessa vez, a caixa de mensagem exibe todos os detalhes do formulário.

13. Clique em *OK*, feche o aplicativo e retorne ao Visual Studio 2010.

Neste exercício, vimos que, embora a validação tenha obtido êxito na verificação cruzada dos controles *Privilege Level* e *Tickets*, ainda há mais coisas a serem feitas para que o aplicativo se torne útil.

## Alterando o ponto em que a validação ocorre

Os problemas no aplicativo ocorrem porque a validação é executada no momento errado, é aplicada de modo inconsistente, e sequer impede que o usuário forneça dados inconsistentes. Você precisa de uma abordagem alternativa para tratar da validação. A solução é verificar a entrada do usuário somente quando o usuário tentar fazer a compra. Assim, você terá certeza de que o usuário terminou de inserir todos os dados e que esses dados são consistentes. Se ocorrerem problemas, você poderá exibir uma mensagem de erro e impedir a utilização desses dados antes da correção dos problemas. No exercício a seguir, você modificará o aplicativo de modo a adiar a validação até o momento em que o usuário tentar comprar os ingressos.

## Valide os dados explicitamente

1. Retorne à janela *Design View* que exibe o arquivo *TicketForm.xaml*. No painel *XAML*, modifique a vinculação da caixa de combinação *privilegeLevel* e defina a propriedade *UpdateSourceTrigger* como “*Explicit*”, como mostrado em negrito a seguir:

```
<ComboBox ... Name="privilegeLevel" ...>
 ...
 <ComboBox.Text>
 <Binding Source="{StaticResource orderData}" Path="PrivilegeLevel"
 Converter="{StaticResource privilegeLevelConverter}"
 UpdateSourceTrigger="Explicit" >
 ...
 </Binding>
 </ComboBox.Text>
 </ComboBox>
```

A propriedade *UpdateSourceTrigger* controla quando as informações inseridas pelo usuário retornam para o objeto *TicketOrder* subjacente e são validadas. Definir essa propriedade como “*Explicit*” adia essa sincronização até que seu aplicativo a execute explicitamente, por meio do código.

2. Modifique as vinculações dos controles *eventList*, *customerReference* e *numberOfTickets* para definir a propriedade *UpdateSourceTrigger* como “*Explicit*”:

```
<ComboBox ... Name="eventList" ... >
 <ComboBox.Text>
 <Binding Source="{StaticResource orderData}" Path="EventName"
 UpdateSourceTrigger="Explicit" >
 ...
 </Binding>
 </ComboBox.Text>
 </ComboBox>
 ...
<TextBox ... Name="customerReference" ... >
 <TextBox.Text>
 <Binding Source="{StaticResource orderData}" Path="CustomerReference"
 UpdateSourceTrigger="Explicit" >
 ...
 </Binding>
 </TextBox.Text>
 </TextBox>
 ...
<Slider ... Name="numberOfTickets" ...>
 <Slider.Value>
 <Binding Source="{StaticResource orderData}" Path="NumberOfTickets"
 UpdateSourceTrigger="Explicit" >
```

```
 ...
 </Binding>
</Slider.Value>
</Slider>
```

3. Exiba o arquivo TicketForm.xaml.cs na janela *Code and Text Editor*. No método *purchaseTickets\_Click*, adicione as instruções mostradas em negrito a seguir ao início do método:

```
private void purchaseTickets_Click(object sender, RoutedEventArgs e)
{
 BindingExpression eventBe =
 eventList.GetBindingExpression(ComboBox.TextProperty);
 BindingExpression customerReferenceBe =
 customerReference.GetBindingExpression(TextBox.TextProperty);
 BindingExpression privilegeLevelBe =
 privilegeLevel.GetBindingExpression(ComboBox.TextProperty);
 BindingExpression numberOfTicketsBe =
 numberOfTickets.GetBindingExpression(Slider.ValueProperty);

 ...
}
```

Essas instruções criam o objeto *BindingExpression* para cada um dos quatro controles com regras de validação de vinculação. Você utilizará esses objetos na próxima etapa para propagar os valores do formulário para o objeto *TicketOrder* e acionar as regras de validação.

4. Adicione as instruções mostradas em negrito a seguir ao método *purchaseTickets\_Click* após o código adicionado na etapa anterior:

```
private void purchaseTickets_Click(object sender, RoutedEventArgs e)
{
 ...
 eventBe.UpdateSource();
 customerReferenceBe.UpdateSource();
 privilegeLevelBe.UpdateSource();
 numberOfTicketsBe.UpdateSource();

 ...
}
```

O método *UpdateSource* da classe *BindingExpression* sincroniza os dados em um objeto com os controles que fazem referência ao objeto por meio das vinculações. Ele retorna os valores das propriedades vinculadas dos controles do formulário para o objeto *TicketOrder*. Quando isso ocorre, os dados também são validados.

As instruções adicionadas na etapa atualizam as propriedades no objeto *TicketOrder* com os valores inseridos pelo usuário no formulário, e validam os dados simultaneamente. A classe *BindingExpression* fornece uma propriedade chamada *HasError*, que indica se o método *UpdateSource* obteve êxito ou se ocasionou uma exceção.

5. Adicione o código mostrado em negrito a seguir ao método *purchaseTickets\_Click* para testar a propriedade *HasError* de cada objeto *BindingExpression* e exibir uma mensagem se a validação falhar. Mova o código original, que exibe os detalhes do cliente, para a parte *else* da instrução *if*.

```
private void purchaseTickets_Click(object sender, RoutedEventArgs e)
{
 ...
 if (eventBe.HasError || customerReferenceBe.HasError ||
 privilegeLevelBe.HasError || numberOfTicketsBe.HasError)
 {
 MessageBox.Show("Please correct errors", "Purchase aborted");
 }
 else
 {
 Binding ticketOrderBinding =
 BindingOperations.GetBinding(privilegeLevel, ComboBox.TextProperty);
 TicketOrder ticketOrder = ticketOrderBinding.Source as TicketOrder;
 MessageBox.Show(ticketOrder.ToString(), "Purchased");
 }
}
```

### Teste o aplicativo novamente

1. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.
2. Quando o formulário *Ticket Orders* aparecer, clique em *Purchase* no menu *File*.

Verifique se a caixa de mensagem *Purchase aborted* é exibida com o texto “Please correct errors”, e se os controles *Event*, *Customer Reference* e *Tickets* estão destacados.



**Nota** O *Privilege Level* não é destacado porque está predefinido como *Standard*, como descrito anteriormente neste capítulo.

3. Clique em *OK* e retorne ao formulário *Ticket Orders*. Posicione o cursor do mouse sobre um controle destacado de cada vez, e verifique se as mensagens lançadas pela exceção *ApplicationException* para cada propriedade do objeto *TicketOrder* adjacente aparecem como Dicas de ferramenta (ToolTips).
4. Na caixa de combinação *Event*, selecione *Little Mudford Festival*. Na caixa de texto *Customer Reference*, digite **C1234**. Na caixa de combinação *Privilege Level*, selecione *Premium*. Defina o controle deslizante *Tickets* com 8, e clique em *Purchase* no menu *File*.

Verifique se a caixa de mensagem *Purchase aborted* aparece novamente, mas dessa vez, apenas o controle deslizante *Tickets* está destacado.

5. Clique em *OK* e posicione o cursor do mouse sobre o controle *Tickets*.

Verifique se a Dica de ferramenta (ToolTip) exibe a mensagem “Too many tickets for this privilege level”.

6. Na caixa de combinação *Privilege Level*, selecione *Premium Executive* e clique em *Purchase* no menu *File*.

Verifique se a caixa de mensagem *Purchased* aparece e exibe o texto “Event: Little Mudford Festival Customer: C1234 Privilege: PremiumExecutive Tickets: 8”, e se nenhum dos controles do formulário encontra-se destacado. Os dados já estão completos e consistentes.

7. Experimente outras combinações de valores, e verifique se a validação funciona como previsto. Quando terminar, feche o formulário e retorne ao Visual Studio.

Neste capítulo, vimos como é possível fazer uma validação básica, por meio do processamento de regras de validação de exceções padrão, propiciado pela vinculação de dados. Você aprendeu a definir regras de validação personalizadas para executar verificações mais complexas.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 25.

- Se você quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 24

Para	Faça isto
Utilizar vinculação de dados para vincular uma propriedade de um controle de um formulário a uma propriedade de outro controle no mesmo formulário	No código XAML da propriedade do controle, crie uma vinculação. Faça uma referência ao controle que contém a propriedade à qual será vinculado, por meio da tag <i>ElementName</i> , e à propriedade à qual se vincular, por meio da tag <i>Path</i> . Por exemplo:
	<pre>&lt;TextBox ...&gt;   &lt;TextBox.Text&gt;     &lt;Binding ElementName="numberOfTickets"       Path="Value" /&gt;   &lt;/TextBox.Text&gt; &lt;/TextBox&gt;</pre>
Usar a vinculação de dados para vincular uma propriedade de um controle em um formulário à propriedade de um objeto	No código XAML para a propriedade do controle, especifique uma origem de vinculação que identifique o objeto e o nome da propriedade do objeto à qual vincular. Por exemplo:
	<pre>&lt;TextBox ...&gt;   &lt;TextBox.Text&gt;     &lt;Binding Source="{StaticResource orderData}"       Path="ForeName" /&gt;   &lt;/TextBox.Text&gt; &lt;/TextBox&gt;</pre>
Habilitar vinculação de dados para validar os dados inseridos pelo usuário	Especifique o elemento <i>Binding.ValidationRules</i> como parte da vinculação. Por exemplo:
	<pre>&lt;Binding Source="{StaticResource orderData}"   Path="ForeName" /&gt;   &lt;Binding.ValidationRules&gt;     &lt;ExceptionValidationRule/&gt;   &lt;/Binding.ValidationRules&gt; &lt;/Binding&gt;</pre>
Exibir as informações de erro de uma maneira não intrusiva	Defina um estilo que detecte uma alteração na propriedade <i>Validation.HasError</i> do controle e então configure a propriedade <i>ToolTip</i> do controle como a mensagem retornada pela exceção. Aplique esse estilo a todos os controles que requererem validação. Por exemplo:
	<pre>&lt;Style x:Key="errorStyle" TargetType="Control"&gt;   &lt;Style.Triggers&gt;     &lt;Trigger Property="Validation.HasError"       Value="True"&gt;       &lt;Setter Property="ToolTip"         Value="{Binding RelativeSource=           {x:Static RelativeSource.Self},         Path=(Validation.Errors)[0].ErrorContent}" /&gt;     &lt;/Trigger&gt;   &lt;/Style.Triggers&gt; &lt;/Style&gt;</pre>
Validar todos os controles de um formulário via programação em vez de quando o usuário passa de um controle para outro	No código XAML para a vinculação, configure a propriedade <i>UpdateSourceTrigger</i> da vinculação como “Explicit” para adiar a validação até o aplicativo solicita-la. Para validar os dados para todos os controles, crie um objeto <i>BindingExpression</i> para cada propriedade vinculada de cada controle e chame o método <i>UpdateSource</i> . Examine a propriedade <i>HasError</i> de cada objeto <i>BindingExpression</i> . Se essa propriedade for true, a validação falhou.

# Consultando informações em um banco de dados

## Parte V

# Gerenciando dados

Capítulo 25: Consultando informações em um banco de dados . . . . .	567
Capítulo 26: Exibindo e editando dados com o Entity Framework e vinculação de dados. . . . .	597

Consultando um banco de dados por meio da ADO.NET

Este capítulo ensina como consultar e manipular dados armazenados em um banco de dados relacional. Ele mostra como usar o Entity Framework para gerenciar os dados e como usar o ADO.NET para executar consultas SQL diretas contra o banco de dados.

## Capítulo 25

# Consultando informações em um banco de dados

Neste capítulo, você vai aprender a:

- Buscar e exibir dados a partir de um banco de dados Microsoft SQL Server utilizando o Microsoft ADO.NET.
- Definir classes de entidade para armazenar os dados recuperados a partir de um banco de dados.
- Utilizar a LINQ para consultar um banco de dados e preencher as instâncias das classes de entidade.
- Criar uma classe *DataContext* personalizada para acessar um banco de dados de uma maneira fortemente tipada.

Na Parte IV, "Construindo aplicativos WPF", você aprendeu a utilizar o Microsoft Visual C# para criar interfaces de usuário e apresentar e validar informações. Na Parte V, você vai aprender a gerenciar dados utilizando a funcionalidade de acesso a dados disponível no Microsoft Visual Studio 2010 e no Microsoft .NET Framework. Os capítulos desta parte descrevem o ADO.NET, uma biblioteca de objetos projetada especificamente para facilitar o desenvolvimento de aplicativos que utilizam bancos de dados. Neste capítulo, você também vai aprender a consultar dados através da LINQ – extensões da LINQ baseadas no ADO.NET projetadas para recuperar dados de um banco de dados. No Capítulo 26, "Exibindo e editando dados com o Entity Framework e vinculação de dados", você vai aprender mais sobre o uso do ADO.NET e da LINQ para atualizar os dados.



**Importante** Para fazer os exercícios neste capítulo, o Microsoft SQL Server 2008 Express precisa estar instalado. Esse software está disponível no DVD distribuído com o Microsoft Visual Studio 2010 e com o Visual C# 2010 Express e é instalado por padrão.



**Importante** É recomendável utilizar uma conta com privilégios de administrador para fazer os exercícios deste capítulo e do restante deste livro.

## Consultando um banco de dados por meio do ADO.NET

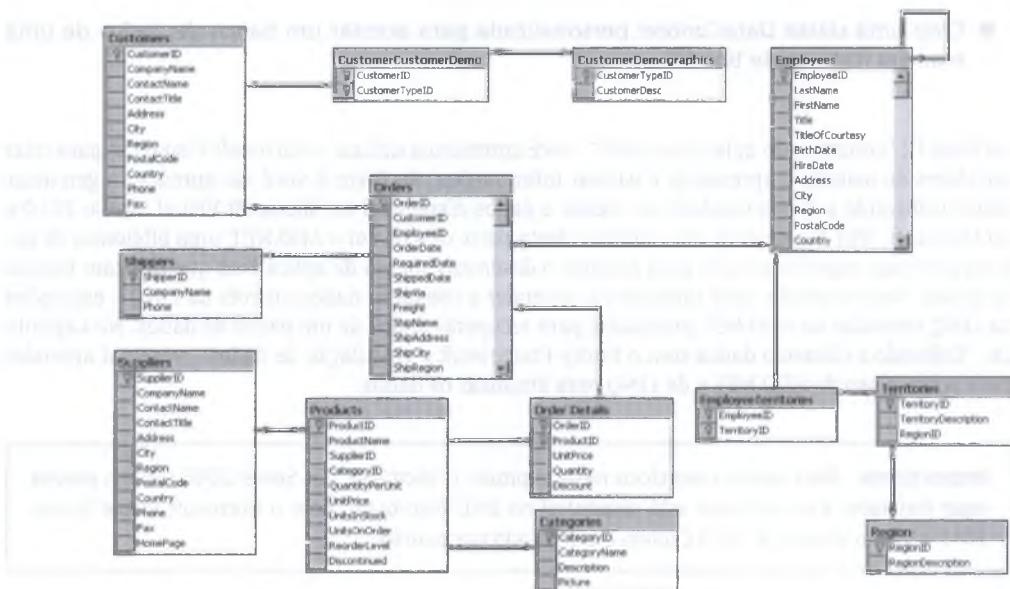
A biblioteca de classes ADO.NET contém uma estrutura abrangente para construir aplicativos que precisam recuperar e atualizar dados armazenados em um banco de dados relacional. O modelo definido pelo ADO.NET baseia-se na noção de provedores de dados. Cada sistema de gerenciamento de bancos de dados (como SQL Server, Oracle, IBM DB2 e outros) tem um provedor de dados próprio que

implementa uma abstração dos mecanismos para conectar a um banco de dados, disparar consultas e atualizar dados. Utilizando essas abstrações, você pode escrever código portátil que é independente do sistema de gerenciamento de bancos de dados subjacente.

Neste capítulo, você se conectará a um banco de dados gerenciado pelo SQL Server 2008 Express, mas as técnicas que você aprenderá são igualmente aplicáveis ao utilizar um sistema de gerenciamento de bancos de dados diferente.

## O banco de dados Northwind

A Northwind Traders é uma empresa fictícia que vende produtos alimentícios com nomes exóticos. O banco de dados Northwind contém várias tabelas com informações sobre os produtos que a Northwind Traders vende, os clientes, os pedidos feitos pelos clientes, os fornecedores dos quais a empresa obtém os produtos a serem revendidos, os distribuidores que ela usa para enviar os produtos para os clientes, e os funcionários que trabalham para a empresa. A imagem a seguir mostra todas as tabelas do banco de dados da Northwind e como elas estão relacionadas entre si. As tabelas que você utilizará neste capítulo são *Orders* e *Products*.



## Criando o banco de dados

Antes de prosseguir, você precisa criar o banco de dados da Northwind.

### Crie o banco de dados da Northwind

1. No menu *Start* do Windows, clique em *All Programs*, depois em *Accessories*, clique com o botão direito do mouse em *Command Prompt*, e depois clique em *Run as administrator*.

Se você estiver conectado utilizando uma conta com direitos de administrador, na caixa de diálogo *User Account Control*, clique em *Yes*.

Se estiver conectado utilizando uma conta sem privilégios de administrador, na caixa de diálogo *User Account Control*, digite a senha do administrador e clique em *Yes*. É exibida a janela de prompt de comando, operando como Administrador.

- Na janela de prompt de comando, digite o seguinte comando:

```
sqlcmd -S.\SQLExpress -E
```

Esse comando inicializa o utilitário sqlcmd para estabelecer conexão com sua instância local do SQL Server 2008 Express. Deve aparecer um prompt “1>”.

 **Dica** Verifique se o SQL Server 2008 Express está em execução antes de tentar executar o utilitário sqlcmd. (Por padrão, ele está definido para inicialização automática. Você receberá uma mensagem de erro se ele não estiver inicializado quando você executar o comando *sqlcmd*.) Para verificar o status do SQL Server 2008 Express e iniciar a sua execução se necessário, utilize a ferramenta SQL Configuration Manager, disponível na pasta Configuration Tools do grupo de programas Microsoft SQL Server 2008.

- Diante do prompt 1>, digite o seguinte comando, inclusive os colchetes, e pressione Enter. Substitua *computer* pelo nome de seu computador, e substitua *login* pelo nome da conta que você usou fazer login no Windows.

```
CREATE LOGIN [computer\login] FROM WINDOWS
```

Deve aparecer um prompt “2>”.

- Diante do prompt 2>, digite **GO** e pressione Enter.

O SQL Server tenta criar um login para a sua conta de usuário, para permitir que você crie o banco de dados Northwind. Se o comando obtiver êxito, o prompt “1>” deve ser reexibido. Se o comando exibir a mensagem “The server principal ‘computer\login’ already exists.”, você já tem um login do SQL Server e pode ignorar a mensagem. Se o comando exibir qualquer outra mensagem, verifique se você especificou os valores corretos para *computer* e *login*, e repita as etapas 3 e 4.

- Diante do prompt 1>, digite o seguinte comando e pressione Enter (como anteriormente, substitua *computer* pelo nome de seu computador, e *login* pelo nome da conta utilizada para o login no Windows):

```
GRANT CREATE DATABASE TO [computer\login]
```

- No prompt 2>, digite **GO** e pressione Enter.

- No prompt 1>, digite **EXIT** e pressione Enter.

Esse comando encerra o utilitário sqlcmd e você retorna ao prompt de comando do Windows.

- Feche a janela do prompt de comando.

9. No menu Windows *Start*, clique em *All Programs*, depois *Accessories*, e, por último, em *Command Prompt*.

Esta ação abre uma janela de prompt de comando que utiliza as suas credenciais e não as do administrador.

10. Na janela do prompt de comando, digite o seguinte comando para acessar a pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 25, em sua pasta Documentos. Substitua *Name* por seu nome de usuário.

```
cd "\Users\Name\Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 25"
```

11. Na janela do prompt de comando, digite o seguinte comando:

```
sqlcmd -S.\SQLEXPRESS -E -iinstnwnd.sql
```

Esse comando usa o utilitário sqlcmd para executar o script instnwnd.sql. Esse script contém os comandos SQL que criam o banco de dados Northwind Traders e as tabelas no banco de dados, e as preenche com alguns exemplos de dados.

12. Quando acabar a execução do script, feche a janela de prompt de comando.



**Nota** Você pode rodar esse comando que você executou no Passo 11 toda vez que for necessário redefinir o banco de dados Northwind Traders. O script instnwnd.sql descarta automaticamente o banco de dados, se ele existir, e então o reconstrói. Para informações adicionais, consulte o Capítulo 26.

## Utilizando ADO.NET para consultar informações de pedidos

Nos próximos exercícios, você vai escrever código para acessar o banco de dados Northwind e exibir informações em um aplicativo de console simples. O objetivo do exercício é ajudá-lo a aprender mais sobre o ADO.NET e entender o modelo de objetos que ele implementa. Nos exercícios posteriores, você vai utilizar a LINQ para consultar o banco de dados. No Capítulo 26, veremos como utilizar os assistentes incluídos no Visual Studio 2010 para gerar código que pode recuperar e atualizar dados e exibir os dados graficamente em um aplicativo Windows Presentation Foundation (WPF).

O primeiro aplicativo que você vai criar produzirá um relatório simples exibindo informações sobre os pedidos dos clientes. O programa solicitará ao usuário um ID de cliente e então exibirá os pedidos desse cliente.

### Conecte ao banco de dados

1. Inicie o Visual Studio 2010 se ele ainda não estiver em execução.
2. Crie um novo projeto chamado **ReportOrders** utilizando o modelo Console Application. Salve-o na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 25 na sua pasta Documentos.



**Nota** Lembre-se, se estiver utilizando o Visual C# 2010 Express, você poderá especificar a localização para salvar seu projeto ao salvá-lo pelo comando *Save ReportOrders* no menu *File*.

3. No *Solution Explorer*, clique com o botão direito do mouse no arquivo Program.cs para renomeá-lo para **Report.cs**. Na mensagem *Microsoft Visual Studio*, clique em *Yes* para alterar todas as referências da classe *Program* para *Report*.
4. Na janela *Code and Text Editor*, adicione as seguintes instruções *using* à lista, na parte superior do arquivo Report.cs:

```
using System.Data;
using System.Data.SqlClient;
```

O namespace *System.Data* contém vários tipos utilizados pelo ADO.NET. O namespace *System.Data.SqlClient* contém as classes do provedor de dados SQL Server para o ADO.NET. Essas classes são versões especializadas das classes ADO.NET otimizadas para funcionarem com o SQL Server.

5. No método *Main* da classe *Report*, adicione a seguinte instrução, mostrada em negrito, que cria um objeto *SqlConnection*:

```
static void Main(string[] args)
{
 SqlConnection dataConnection = new SqlConnection();
}
```

*SqlConnection* é uma subclasse de uma classe ADO.NET chamada *Connection*. Ela é projetada apenas para tratar conexões com bancos de dados SQL Server.

6. Depois da declaração da variável, adicione um bloco *try/catch* ao método *Main*, mostrado em negrito a seguir. Todo o código que você escreverá para ganhar acesso ao banco de dados é armazenado na parte *try* desse bloco. No bloco *catch*, adicione uma rotina de tratamento simples que captura as exceções *SqlException*.

```
static void Main(string[] args)
{
 ...
 try
 {
 // Você adicionará seu código aqui, mais adiante
 }
 catch (SqlException e)
 {
 Console.WriteLine("Error accessing the database: {0}", e.Message);
 }
}
```

Uma *SqlException* é lançada se um erro ocorrer ao acessar um banco de dados SQL Server.

7. Substitua o comentário no bloco *try* pelo código mostrado a seguir em negrito:

```
try
{
 SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
 builder.DataSource = ".\\SQLEXpress";
 builder.InitialCatalog = "Northwind";
 builder.IntegratedSecurity = true;
 dataConnection.ConnectionString = builder.ConnectionString;
}
```

Para conectar-se com um banco de dados SQL Server, construa uma string de conexão que especifique o banco de dados ao qual se conectar, a instância do SQL Server que armazena esse banco de dados, e como o aplicativo se identificará como um usuário válido do banco de dados perante o SQL Server. A maneira mais simples de fazer isso é usar um objeto *SqlConnectionStringBuilder*. A classe *SqlConnectionStringBuilder* apresenta propriedades para cada um dos elementos de uma string de conexão. Você pode ler uma string de conexão completa, que combina todos esses elementos no formato correto da propriedade *ConnectionString*.

Este código utiliza um objeto *SqlConnectionStringBuilder* para construir uma string de conexão para acessar o banco de dados Northwind em execução na instância do SQL Server Express em seu computador. O código especifica que a conexão usará a Autenticação do Windows (Windows Authentication) para estabelecer conexão com o banco de dados. Esse é o método de acesso preferido porque você não precisa solicitar ao usuário qualquer nome de usuário ou senha, e você não precisa codificar nomes de usuários e senhas em seu aplicativo.

A string de conexão é armazenada na propriedade *ConnectionString* do objeto *SqlConnection*, que você utilizará na próxima etapa.

Você também pode codificar vários outros elementos na string de conexão, ao utilizar a classe *SqlConnectionStringBuilder*— as propriedades mostradas neste exemplo representam um conjunto mínimo, mas eficiente. Para obter mais detalhes, consulte a documentação fornecida no Visual Studio 2010.

8. Adicione a seguinte instrução, mostrada em negrito a seguir, ao código no bloco *try*:

```
try
{
 /**
 *
 *
 dataConnection.Open();
}
```

Essa instrução utiliza a string de conexão especificada pela propriedade *ConnectionString* do objeto *dataConnection* para abrir uma conexão com o banco de dados. Se a conexão obtiver êxito, você poderá utilizar o objeto *dataConnection* para executar comandos e consultas ao banco de dados. Caso contrário, a instrução lançará uma exceção *SqlException*.

## Usando o SQL Server Authentication

O Windows Authentication é útil para autenticar os usuários que são membros de um domínio Windows. Contudo, talvez haja ocasiões em que o usuário que acessa o banco de dados não tenha uma conta Windows; por exemplo, se você estiver compilando um aplicativo projetado para ser acessado por usuários remotos na Internet. Nesses casos, você pode utilizar os parâmetros *User ID* e *Password*, como mostrado a seguir:

```
string userName = ...;
string password = ...;
// Solicite ao usuário seu nome e senha e preencha essas variáveis
string connString = String.Format(
 "User ID={0};Password={1};Initial Catalog=Northwind;" +
 "Data Source=SeuComputador\\SQLEXpress", username, password);
myConnection.ConnectionString = connString;
```

Nesse ponto, devo fazer uma advertência: nunca codifique nomes de usuário e senhas diretamente nos seus aplicativos. Qualquer pessoa que obtiver uma cópia do código-fonte (ou aqueles que praticam engenharia reversa do código compilado) pode ver essas informações, tornando inútil toda a segurança.

A próxima etapa será pedir ao cliente um ID de cliente e, então, consultar o banco de dados para encontrar todos os pedidos desse cliente.

### Consulte a tabela Orders

1. Adicione as instruções mostradas a seguir, em negrito, ao bloco *try*, depois da instrução *data-Connection.Open()*.

```
try
{
 ...
 Console.WriteLine("Please enter a customer ID (5 characters): ");
 string customerId = Console.ReadLine();
}
```

Essas instruções pedem um ID de cliente ao usuário, leem a resposta do usuário e armazenam na variável string *customerId*.

2. Digite as seguintes instruções, mostradas em negrito, depois do código que você acabou de inserir:

```
try
{
 ...
 SqlCommand dataCommand = new SqlCommand();
 dataCommand.Connection = dataConnection;
 dataCommand.CommandType = CommandType.Text;
 dataCommand.CommandText =
 "SELECT OrderID, OrderDate, ShippedDate, ShipName, ShipAddress, " +
 "ShipCity, ShipCountry " +
 "FROM Orders WHERE CustomerID = @CustomerIdParam";
}
```

A primeira instrução cria um objeto *SqlCommand*. Como ocorre com *SqlConnection*, essa é uma versão especializada de uma classe ADO.NET *Command*, projetada para realizar consultas em um banco de dados SQL Server. Um objeto ADO.NET *Command* é utilizado para executar um comando em uma fonte de dados. No caso de um banco de dados relacional, o texto do comando é uma instrução SQL.

A segunda linha do código define a propriedade *Connection* do objeto *SqlCommand* para a conexão do banco de dados que você abriu no exercício anterior. As duas instruções a seguir especificam que o objeto *SqlCommand* contém o texto de uma instrução SQL (você também pode especificar o nome de um procedimento armazenado (stored procedure) ou o nome de uma tabela individual no banco de dados) e preencher a propriedade *CommandText* com uma instrução SQL SELECT que recupera informações na tabela *Orders* de todos os pedidos que possuem um *CustomerID* especificado. O texto *@CustomerIdParam* é um marcador para um parâmetro SQL. (O símbolo *@* indica para o provedor de dados que esse é um parâmetro e não o nome de uma coluna no banco de dados.) O valor do *CustomerID* será passado como um objeto *SqlParameter* na próxima etapa.

3. Adicione as seguintes instruções, mostradas em negrito, ao bloco *try*, após o código inserido na etapa anterior:

```
try
{
 ...
 SqlParameter param = new SqlParameter("@CustomerIdParam", SqlDbType.Char, 5);
 param.Value = customerId;
 dataCommand.Parameters.Add(param);
}
```

Essas instruções criam um objeto *SqlParameter* que pode substituir o *@CustomerIdParam* quando o objeto *SqlCommand* for executado. O parâmetro é marcado como um tipo *Char* de banco de dados (o equivalente a uma string de comprimento fixo, no SQL Server), e o comprimento dessa string é especificado como 5 caracteres. O *SqlParameter* é preenchido com a string inserida pelo usuário na variável *customerId* e depois adicionada à coleção *Parameters* do *SqlCommand*. Ao executar esse comando, o SQL Server procurará na coleção *Parameters* do comando um parâmetro chamado *@CustomerIdParam* e substituirá o valor desse parâmetro pelo texto da instrução SQL.

**Importante** Se você é principiante na construção de aplicativos de bancos de dados, deve estar se perguntando por que o código cria um objeto *SqlParameter* em vez de construir uma simples instrução SQL que incorpore o valor da variável *customerId*, como a seguir:

```
dataCommand.CommandText =
 "SELECT OrderID, OrderDate, ShippedDate, ShipName, ShipAddress, " +
 "ShipCity, ShipCountry " +
 "FROM Orders WHERE CustomerID = '" + customerId + "'";
```

Essa abordagem é uma prática extremamente inadequada, porque resulta um aplicativo vulnerável a ataques de injeção de SQL. Não escreva um código como esse em seus aplicativos de produção. Para obter uma descrição do que significa um ataque de injeção de SQL e do perigo que ele representa, consulte o tópico sobre injeção de SQL no SQL Server Books Online, disponível em <http://msdn2.microsoft.com/en-us/library/ms161953.aspx>.

4. Adicione as seguintes instruções, mostradas em negrito, após o código que você acabou de inserir:

```
try
{
 ...
 Console.WriteLine("About to find orders for customer {0}\n\n", customerId);
 SqlDataReader dataReader = dataCommand.ExecuteReader();
}
```

O método *ExecuteReader* de um objeto *SqlCommand* constrói um objeto *SqlDataReader* que você pode utilizar para buscar as linhas identificadas pela instrução SQL. A classe *SqlDataReader* fornece o mecanismo mais rápido disponível (tão rápido quanto sua rede permitir) para recuperar dados de um SQL Server.

A próxima tarefa será iterar por todos os pedidos (se houver algum) e exibi-los.

### Busque e exiba os pedidos

1. No arquivo Report.cs, adicione o loop *while* mostrado aqui em negrito após a instrução que cria o objeto *SqlDataReader*:

```
try
{
 ...
 while (dataReader.Read())
 {
 // Código para exibir a linha atual
 }
}
```

O método *Read* da classe *SqlDataReader* recupera a próxima linha no banco de dados. Ele retornará *true* se outra linha tiver sido recuperada com sucesso; caso contrário, retornará *false*, normalmente porque não há mais linhas. O loop *while* que você acabou de digitar continuará lendo as linhas da variável *dataReader* e terminará quando não houver mais linhas disponíveis.

2. Adicione as instruções mostradas aqui em negrito ao corpo do loop *while* criado no passo anterior:

```
while (dataReader.Read())
{
 int orderId = dataReader.GetInt32(0);
 DateTime orderDate = dataReader.GetDateTime(1);
 DateTime shipDate = dataReader.GetDateTime(2);
 string shipName = dataReader.GetString(3);
 string shipAddress = dataReader.GetString(4);
 string shipCity = dataReader.GetString(5);
 string shipCountry = dataReader.GetString(6);
 Console.WriteLine(
 "Order: {0}\nPlaced: {1}\nShipped: {2}\n" +
 "To Address: {3}\n{4}\n{5}\n{6}\n", orderId, orderDate,
 shipDate, shipName, shipAddress, shipCity, shipCountry);
}
```

Esse bloco de código mostra como ler os dados a partir do banco de dados utilizando um objeto *SqlDataReader*. Um objeto *SqlDataReader* contém a linha mais recente recuperada do banco de dados. Você pode utilizar os métodos *GetXXX* para extrair as informações de cada coluna da linha – há um método *GetXXX* para cada tipo de dado comum. Por exemplo, para ler um valor *int*, você utiliza o método *GetInt32*; para ler uma string, você utiliza o método *GetString*; e provavelmente você pode adivinhar como ler um valor *DateTime*. Os métodos *GetXXX* recebem um parâmetro indicando a coluna a ser lida: 0 é a primeira coluna, 1 é a segunda coluna e assim por diante. O código anterior lê as várias colunas da linha *Orders* atual, armazena os valores em um conjunto de variáveis e, então, imprime os valores dessas variáveis.

## Firehose cursors

Uma das principais desvantagens de um aplicativo de banco de dados multiusuário é a existência de dados bloqueados. Infelizmente, é comum ver aplicativos que recuperam linhas de um banco de dados e mantêm essas linhas bloqueadas para impedir que outro usuário altere os dados enquanto o aplicativo está utilizando-os. Em circunstâncias extremas, um aplicativo pode ainda impedir que outros usuários leiam os dados que estão bloqueados. Se o aplicativo recuperar um grande número de linhas, ele bloqueará uma grande parte da tabela. Se houver muitos usuários executando o mesmo aplicativo ao mesmo tempo, talvez um precise esperar o outro para liberar esses bloqueios, o que acarretará uma execução lenta e frustrante.

A classe *SqlDataReader* foi projetada para remover esse empecilho. Ela recupera as linhas uma de cada vez e não mantém linha alguma bloqueada depois que ela é recuperada. Isso é ótimo para melhorar a cooperação nos seus aplicativos. A classe *SqlDataReader* é, às vezes, chamada de “firehose cursor”, porque ela emite dados em profusão, o mais rapidamente possível. (O termo *cursor* é um acrônimo de “current set of rows”, que significa conjunto atual de linhas. *Firehose* significa, literalmente, “mangueira de bombeiro”).

Quando você terminar de utilizar um banco de dados, é recomendável fechar sua conexão e liberar todos os recursos que foram utilizados.

### Desconecte-se do banco de dados e teste o aplicativo

1. Adicione a instrução mostrada aqui em negrito depois do loop *while* no bloco *try*:

```
try
{

 while(dataReader.Read())
 {

 }
 dataReader.Close();
}
```

Essa instrução fecha o objeto *SqlDataReader*. Você deve sempre fechar um *SqlDataReader* quando tiver terminado de trabalhar com ele porque você não será capaz de utilizar o objeto *SqlConnection* atual até que o tenha fechado. Isso também pode ser considerado como uma boa prática se tudo o que você fizer em seguida for fechar a *SqlConnection*.

 **Nota** Se ativar múltiplos conjuntos de resultados ativos (*multiple active result sets – MARS*) com o SQL Server 2008, você poderá abrir mais de um objeto *SqlDataReader* contra o mesmo objeto *SqlConnection* e processar múltiplos conjuntos de dados. O MARS está desativado por padrão. Para aprender mais sobre MARS e como você pode ativá-lo e utilizá-lo, consulte o Books Online do SQL Server 2008.

2. Depois do bloco *catch*, adicione o bloco *finally* a seguir:

```
catch(SqlException e)
{

}
finally
{
 dataConnection.Close();
}
```

As conexões de banco de dados são recursos escassos. Você precisa garantir que eles estejam fechados quando tiver terminado de utilizá-los. Inserir essa instrução em um bloco *finally* garante que a *SqlConnection* será fechada, mesmo que ocorra uma exceção; lembre-se de que o código no bloco *finally* será executado depois que a rotina de tratamento *catch* tiver terminado.



**Dica** Uma abordagem alternativa ao uso de um bloco *finally* é empacotar o código que cria o objeto *SqlDataConnection* em uma instrução *using*, como mostrado no código a seguir. No final do bloco definido pela instrução *using*, o objeto *SqlConnection* é fechado automaticamente, mesmo se ocorrer uma exceção:

```
using (SqlConnection dataConnection = new SqlConnection())
{
 try
 {
 SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
 /**
 */
 }
 catch (SqlException e)
 {
 Console.WriteLine("Error accessing the database: {0}", e.Message);
 }
}
```

3. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
4. No prompt de ID de cliente, digite o ID de cliente **VINET** e pressione Enter.

A instrução SQL SELECT aparece, seguida pelos pedidos desse cliente, como mostrado na imagem a seguir:

The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The user has typed the following command:

```
Please enter a customer ID (5 characters): VINET
about to find orders for customer VINET
```

Three order details are displayed:

- Order: 10248  
Placed: 04/07/1996 00:00:00  
Shipped: 16/07/1996 00:00:00  
To Address: Vins et alcools Chevalier  
59 rue de l'Abbaye  
Reims  
France
- Order: 10274  
Placed: 06/08/1996 00:00:00  
Shipped: 16/08/1996 00:00:00  
To Address: Vins et alcools Chevalier  
59 rue de l'Abbaye  
Reims  
France
- Order: 10295  
Placed: 02/09/1996 00:00:00  
Shipped: 10/09/1996 00:00:00

Você pode rolar de volta pela janela de console para ver todos os dados. Pressione a tecla Enter para fechá-la quando tiver terminado.

5. Execute o aplicativo sem depurar e digite **BONAP** quando um ID de cliente for pedido.

Aparecem algumas linhas, mas, em seguida, ocorre um erro e é exibida uma caixa de mensagem com a mensagem “ReportOrders has stopped working”. Se a mensagem “Do you want to send more information about the problem?” for exibida, clique em *Cancel*.

Uma mensagem de erro com o texto “Unhandled Exception: System.Data.SqlTypes. SqlNullValueException: Data is Null. This method or property cannot be called on Null values” aparecerá na janela de console.

O problema é que os bancos de dados relacionais permitem que algumas colunas contenham valores nulos. Um valor nulo é parecido com uma variável nula no C#: ele não tem um valor, mas se você tentar utilizá-lo, obterá um erro. Na tabela *Orders*, a coluna *ShippedDate* pode conter um valor nulo se o pedido ainda não tiver sido remetido. Você também deve observar que essa é uma *SqlNullValueException* e, consequentemente, não é capturada pela rotina de tratamento *SQLException*.

6. Pressione Enter para fechar a janela de console e retornar ao Visual Studio 2010.

## Fechando conexões

Em muitos aplicativos antigos, há uma tendência de o aplicativo abrir uma conexão quando ele se inicia e não fechá-la até que o aplicativo termine. A razão lógica por trás dessa estratégia era que a abertura e o fechamento de conexões de bancos de dados eram uma operação cara e demorada. Essa estratégia tinha um impacto na escalabilidade dos aplicativos porque cada usuário que executava o aplicativo tinha uma conexão aberta com o banco de dados enquanto o aplicativo estava executando, mesmo que ele saísse para almoçar por algumas horas. A maioria dos bancos de dados limita o número de conexões concorrentes que eles permitem (às vezes, isso ocorre por causa da licença, mas normalmente acontece porque cada conexão consome recursos no servidor de banco de dados que não são infinitos). Por fim, o banco de dados poderia atingir um limite no número de usuários que podem operar simultaneamente.

A maioria dos provedores de dados do .NET Framework (incluindo o provedor do SQL Server) implementa um *pool de conexões*. As conexões de banco de dados são criadas e mantidas em um pool. Quando um aplicativo requer uma conexão, o provedor de acesso aos dados extrai a próxima conexão disponível do pool. Quando o aplicativo fecha a conexão, ela é retornada ao pool e disponibilizada para o próximo aplicativo que quer uma conexão. Isso significa que abrir e fechar conexões de banco de dados não são mais operações caras. O fechamento de uma conexão não desconecta do banco de dados; ela apenas retorna a conexão para o pool. Abrir uma conexão é simplesmente uma questão de obter do pool uma conexão já aberta. Portanto, você só deve manter as conexões abertas pelo tempo necessário – abra uma conexão quando você precisar dela e feche-a assim que terminar.

Você deve observar que o método *ExecuteReader* da classe *SqlCommand*, que cria um *SqlDataReader*, está sobrecarregado. Você pode especificar um parâmetro *System.Data.CommandBehavior* que fecha automaticamente a conexão utilizada pelo *SqlDataReader* quando o *SqlDataReader* é fechado, assim:

```
SqlDataReader dataReader =
 dataCommand.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
```

Ao ler os dados do objeto *SqlDataReader*, você deve verificar que os dados que está lendo não são nulos. Veremos como fazer isso a seguir.

### Trate valores nulos de banco de dados

1. No método *Main* da classe *Report*, altere o código no corpo do loop *while* para que ele contenha um bloco *if...else*, como mostrado a seguir em negrito:

```
while (dataReader.Read())
{
 int orderId = dataReader.GetInt32(0);
 if (dataReader.IsDBNull(2))
 {
 Console.WriteLine("Order {0} not yet shipped\n\n", orderId);
 }
 else
 {
 DateTime orderDate = dataReader.GetDateTime(1);
 DateTime shipDate = dataReader.GetDateTime(2);
 string shipName = dataReader.GetString(3);
 string shipAddress = dataReader.GetString(4);
 string shipCity = dataReader.GetString(5);
 string shipCountry = dataReader.GetString(6);
 Console.WriteLine(
 "Order {0}\nPlaced {1}\nShipped{2}\n" +
 "To Address {3}\n{4}\n{5}\n{6}\n\n", orderId, orderDate,
 shipDate, shipName, shipAddress, shipCity, shipCountry);
 }
}
```

A instrução *if* utiliza o método *IsDBNull* para determinar se a coluna *ShippedDate* (coluna 2 na tabela) é nula. Se for nula, nenhuma tentativa de recuperá-la é feita (ou qualquer uma das outras colunas, que também devem ser nulas, se não houver valor *ShippedDate* algum); caso contrário, as colunas serão lidas e impressas como antes.

2. Compile e execute o aplicativo novamente.

3. Digite BONAP para ID do cliente quando solicitado.

Desta vez, você não obterá erro algum, mas receberá uma lista que contém pedidos que ainda não foram remetidos.

4. Quando o aplicativo terminar, pressione Enter e retorne ao Visual Studio 2010.

## Consultando um banco de dados usando LINQ to SQL

No Capítulo 20, “Consultando dados na memória utilizando expressões de consulta”, vimos como usar a LINQ para examinar o conteúdo de coleções enumeráveis armazenadas na memória. A LINQ fornece expressões de consulta que empregam uma sintaxe como de SQL para realizar consultas e gerar um conjunto de resultados que você pode então investigar. Não deve ser surpreendente que você pode utilizar uma forma LINQ estendida, chamada LINQ to SQL, para consultar e manipular o conteúdo de um banco de dados. A LINQ to SQL é construída sobre o ADO.NET e fornece um nível alto de abstração, eliminando a necessidade de se preocupar com os detalhes da construção de um objeto ADO.NET *Command*, iterar por um conjunto de resultados retornados por um objeto *DataReader* ou buscar cada coluna de dados utilizando os vários métodos *GetXXX*.

### Definindo uma classe de entidade

Vimos no Capítulo 20 que o uso da LINQ exige que os objetos que você está consultando sejam enumeráveis; eles devem ser coleções que implementam a interface *IEnumerable*. A LINQ to SQL pode criar coleções enumeráveis próprias dos objetos com base nas classes definidas por você e que são mapeadas diretamente para tabelas em um banco de dados. Essas classes são chamadas de classes de entidade. Ao conectar-se a um banco de dados e realizar uma consulta, a LINQ to SQL pode recuperar os dados identificados e criar uma instância de uma classe de entidade para cada linha buscada.

A melhor maneira de explicar a LINQ to SQL é ver um exemplo. A tabela *Products* no banco de dados Northwind consiste em colunas com informações sobre os diferentes aspectos dos vários produtos que a Northwind Traders vende. O script *instnwnd.sql*, que você executou no primeiro exercício neste capítulo, contém uma instrução *CREATE TABLE* que é semelhante a esta (algumas colunas, restrições e outros detalhes foram omitidos):

```
CREATE TABLE "Products" (
 "ProductID" "int" NOT NULL ,
 "ProductName" nvarchar (40) NOT NULL ,
 "SupplierID" "int" NULL ,
 "UnitPrice" "money" NULL ,
 CONSTRAINT "PK_Products" PRIMARY KEY CLUSTERED ("ProductID"),
 CONSTRAINT "FK_Products_Suppliers" FOREIGN KEY ("SupplierID")
 REFERENCES "dbo"."Suppliers" ("SupplierID")
)
```

Você pode definir uma classe de entidade que corresponde à tabela *Products* como mostrado a seguir:

```
[Table(Name = "Products")]
public class Product
{
 [Column(IsPrimaryKey = true, CanBeNull = false)]
 public int ProductID { get; set; }

 [Column(CanBeNull = false)]
 public string ProductName { get; set; }
```

```
[Column]
public int? SupplierID { get; set; }

[Column(DbType = "money")]
public decimal? UnitPrice { get; set; }
}
```

A classe *Product* contém uma propriedade para cada uma das colunas em que você está interessado na tabela *Products*. Você não precisa especificar cada coluna da tabela subjacente, mas qualquer coluna que você omitir não será recuperada quando você executar uma consulta com base nessa classe de entidade. Os pontos importantes a observar são os atributos *Table* e *Columns*.

O atributo *Table* identifica essa classe como uma classe de entidade. O parâmetro *Name* especifica o nome da tabela correspondente no banco de dados. Se omitir o parâmetro *Name*, a LINQ to SQL supõe que o nome da classe de entidade é o mesmo que o nome da tabela correspondente no banco de dados.

O atributo *Column* descreve como uma coluna na tabela *Products* é mapeada para uma propriedade na classe *Product*. O atributo *Column* pode receber alguns parâmetros. Aqueles mostrados neste exemplo e descritos na lista a seguir são os mais comuns:

- O parâmetro *IsPrimaryKey* especifica que a propriedade compõe parte da chave primária. (Se a tabela tiver uma chave primária composta que abrange múltiplas colunas, você deverá especificar o parâmetro *IsPrimaryKey* para cada propriedade correspondente na classe de entidade.)
- O parâmetro *DbType* especifica o tipo da coluna subjacente no banco de dados. Em muitos casos, a LINQ to SQL pode detectar e converter dados em uma coluna no banco de dados no tipo da propriedade correspondente na classe de entidade, mas em algumas situações você mesmo precisa especificar o tipo de mapeamento de dados. Por exemplo, a coluna *UnitPrice* da tabela *Products* utiliza o tipo *money* do SQL Server. A classe de entidade especifica a propriedade correspondente como um valor *decimal*.



**Nota** O padrão de mapeamento dos dados *money* no SQL Server é para o tipo *decimal* em uma classe de entidade, portanto, o parâmetro *DbType* mostrado aqui na verdade é redundante. Entretanto, eu quis mostrar a sintaxe.

- O parâmetro *CanBeNull* indica se a coluna do banco de dados pode conter um valor nulo. O valor padrão para o parâmetro *CanBeNull* é *true*. Observe que as duas propriedades da classe *Product* que correspondem a colunas que permitem valores nulos no banco de dados (*SupplierID* e *UnitPrice*) são definidas como tipos nullable na classe de entidade.



**Nota** Você também pode utilizar a LINQ to SQL para criar novos bancos de dados e tabelas com base nas definições das suas classes de entidade utilizando o método *CreateDatabase* do objeto *DataContext*. A LINQ to SQL utiliza a definição do parâmetro *DbType* para especificar se uma coluna deve permitir valores nulos. Se estiver utilizando a LINQ to SQL para criar um novo banco de dados, você deve especificar a nulidade de cada coluna em cada tabela no parâmetro *DbType* desta maneira:

```
[Column(DbType = "NVarChar(40) NOT NULL", CanBeNull = false)]
public string ProductName { get; set; }

...
[Column(DbType = "Int NULL", CanBeNull = true)]
public int? SupplierID { get; set; }
```

Como ocorre com o atributo *Table*, o atributo *Column* fornece um parâmetro *Name* que você pode utilizar para especificar o nome da coluna subjacente no banco de dados. Se omitir esse parâmetro, a LINQ to SQL supõe que o nome da coluna é o mesmo que o nome da propriedade na classe de entidade.

## Criando e executando uma consulta LINQ to SQL

Após definir uma classe de entidade, você pode utilizá-la para buscar e exibir dados da tabela *Products*. O código a seguir apresenta as etapas básicas para executar essa tarefa:

```
SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
builder.DataSource = ".\\SQLEXPRESS";
builder.InitialCatalog = "Northwind";
builder.IntegratedSecurity = true;

DataContext db = new DataContext(builder.ConnectionString);

Table<Product> products = db.GetTable<Product>();
var productsQuery = from p in products
 select p;

foreach (var product in productsQuery)
{
 Console.WriteLine("ID: {0}, Name: {1}, Supplier: {2}, Price: {3:C}",
 product.ProductID, product.ProductName,
 product.SupplierID, product.UnitPrice);
}
```

**Nota** Lembre-se de que as palavras-chave *from*, *in* e *select* nesse contexto são identificadores C#, e não elementos da sintaxe SQL. Você deve digitá-las em letras minúsculas.

A classe *DataContext* é responsável pelo gerenciamento do relacionamento entre suas classes de entidade e as tabelas no banco de dados. Você a utiliza para estabelecer uma conexão ao banco de dados

e criar coleções das classes de entidade. O construtor *DataContext* espera uma string de conexão como um parâmetro, especificando o banco de dados que você quer utilizar. Essa string de conexão é exatamente a mesma string de conexão que você utilizaria ao conectar-se por meio de um objeto ADO.NET *Connection* (a classe *DataContext* cria uma conexão ADO.NET nos bastidores).

O método *GetTable< TEntity >* genérico da classe *DataContext* espera uma classe de entidade como o tipo parâmetro *TEntity*. Esse método constrói uma coleção enumerável com base nesse tipo e retorna a coleção como um tipo *Table< TEntity >*. Você pode realizar consultas LINQ to SQL sobre essa coleção. A consulta mostrada nesse exemplo simplesmente recupera cada objeto da tabela *Products*.

**Nota** Se precisar recapitular as expressões de consulta LINQ, volte ao Capítulo 20.

A instrução *foreach* itera pelos resultados dessa consulta e exibe os detalhes de cada produto. A imagem a seguir mostra os resultados da execução desse código. (Os preços mostrados são por caixa individual, não por item individual.)

```

C:\Windows\system32\cmd.exe
1> 1. Name: Chai, Supplier: 1, Price: £10.00
1> 2. Name: Chang, Supplier: 1, Price: £19.00
1> 3. Name: Beef Jerky, Supplier: 1, Price: £10.00
1> 4. Name: Chef Anton's Cajun Seasoning, Supplier: 2, Price: £22.00
1> 5. Name: Chef Anton's Gumbo Mix, Supplier: 2, Price: £21.35
1> 6. Name: Grandma's Boysenberry Spread, Supplier: 3, Price: £25.00
1> 7. Name: Uncle Bob's Organic Dried Pears, Supplier: 3, Price: £39.00
1> 8. Name: Northwoods Cranberry Sauce, Supplier: 3, Price: £40.00
1> 9. Name: Mighty Toes Milk, Supplier: 4, Price: £7.00
1> 10. Name: Ikura, Supplier: 4, Price: £31.00
1> 11. Name: Quince Paste, Supplier: 5, Price: £21.00
1> 12. Name: Quese Mandacio La Pastora, Supplier: 5, Price: £30.00
1> 13. Name: Kombo, Supplier: 6, Price: £10.00
1> 14. Name: Tofu, Supplier: 6, Price: £23.25
1> 15. Name: Genes Shrimp, Supplier: 6, Price: £15.50
1> 16. Name: Paulsau, Supplier: 7, Price: £17.45
1> 17. Name: Alice Mutton, Supplier: 7, Price: £19.00
1> 18. Name: Carnarvon Tigers, Supplier: 7, Price: £62.50
1> 19. Name: Teatime Chocolate Biscuits, Supplier: 8, Price: £9.20
1> 20. Name: Sir Rodney's Marmalade, Supplier: 8, Price: £31.00
1> 21. Name: Sir Badney's Scones, Supplier: 9, Price: £10.00
1> 22. Name: Gustaf's Koschebeerd, Supplier: 9, Price: £21.00
1> 23. Name: Juranbirr, Supplier: 9, Price: £7.00
1> 24. Name: Guaraná Fantástica, Supplier: 10, Price: £4.50
1> 25. Name: Ho-Ho-Ca Neg-Magog-Creme, Supplier: 11, Price: £14.00

```

O objeto *DataContext* controla a conexão de banco de dados automaticamente; ele abre a conexão um pouco antes de buscar a primeira linha de dados na instrução *foreach* e então fecha a conexão depois que a última linha foi recuperada.

A consulta LINQ to SQL, mostrada no exemplo anterior, recupera cada coluna para cada linha na tabela *Products*. Nesse caso, na verdade, você pode até iterar pela coleção *products* diretamente, como a seguir:

```

Table<Product> products = db.GetTable<Product>();

foreach (Product product in products)
{
 ...
}

```

Quando a instrução `foreach` executa, o objeto `DataContext` constrói uma instrução SQL SELECT que recupera todos os dados da tabela `Products`. Se quiser recuperar uma única linha da tabela `Products`, você pode chamar o método `Single` da classe de entidade `Products`. `Single` é um método de extensão que aceita um método que identifica a linha que você quer localizar e retorna essa linha como uma instância da classe de entidade (em oposição a um conjunto de linhas em uma coleção `Table`). Você pode especificar o parâmetro do método como uma expressão lambda. Se a expressão lambda não identificar exatamente uma linha, o método `Single` retornará uma `InvalidOperationException`. O exemplo de código a seguir consulta no banco de dados Northwind o produto com o valor `ProductID` de 27. O valor retornado é uma instância da classe `Product` e a instrução `Console.WriteLine` imprime o nome do produto. Como anteriormente, a conexão com o banco de dados é aberta e fechada automaticamente pelo objeto `DataContext`.

```
Product singleProduct = products.Single(p => p.ProductID == 27);
Console.WriteLine("Name: {0} ", singleProduct.ProductName);
```

## Busca adiada e busca imediata

Um ponto importante a enfatizar é que, por padrão, a LINQ to SQL só recupera os dados do banco de dados quando você os solicita e não ao definir uma consulta LINQ to SQL ou criar uma coleção `Table`. Isso é conhecido como busca adiada (*deferred fetching*). No exemplo mostrado anteriormente, que exibe todos os produtos da tabela `Products`, a coleção `productsQuery` só é preenchida quando o loop `foreach` é executado. Esse modo de operação corresponde àquele da LINQ ao consultar objetos na memória; você sempre verá a versão mais atualizada dos dados, mesmo se os dados mudarem depois de executar a instrução que cria a coleção enumerável `productsQuery`.

Quando o loop `foreach` se inicia, a LINQ to SQL cria e executa uma instrução SQL SELECT derivada da consulta LINQ to SQL para criar um objeto ADO.NET `DataReader`. Cada iteração do loop `foreach` executa os métodos `GetXXX` necessários para buscar os dados dessa linha. Depois de a linha final ser buscada e processada pelo loop `foreach`, a LINQ to SQL fecha a conexão de banco de dados.

A busca adiada assegura que somente os dados que um aplicativo de fato utiliza sejam recuperados do banco de dados. Entretanto, se você estiver acessando um banco de dados em execução em uma instância SQL Server remota, buscar os dados linha por linha não será a utilização mais adequada da largura de banda da rede. Nesse cenário, você pode buscar e armazenar em cache todos os dados em uma única solicitação de rede forçando uma avaliação imediata da consulta LINQ to SQL. Você pode fazer isso chamando os métodos de extensão `ToList` ou `ToArray` que buscam os dados em uma lista ou em um array quando você define a consulta LINQ to SQL, como a seguir:

```
var productsQuery = from p in products.ToList()
 select p;
```

Nesse exemplo de código, *productsQuery* agora é uma lista enumerável, preenchida com as informações provenientes da tabela *Products*. Ao iterar pelos dados, a LINQ to SQL recupera-os dessa lista, em vez de enviar solicitações de busca ao banco de dados.

## Fazendo junção de tabelas e criando relações

A LINQ to SQL suporta o operador de consulta *join* para combinar e recuperar dados relacionados armazenados em múltiplas tabelas. Por exemplo, a tabela *Products* no banco de dados Northwind contém o ID do fornecedor para cada produto. Se quiser conhecer o nome de cada fornecedor, você terá de consultar a tabela *Suppliers*. A tabela *Suppliers* contém a coluna *CompanyName* que especifica o nome da empresa do fornecedor e a coluna *ContactName* contém o nome da pessoa na empresa do fornecedor que trata os pedidos da Northwind Traders. Você pode definir uma classe de entidade contendo informações relevantes sobre o fornecedor da seguinte maneira (a coluna *SupplierID* no banco de dados é obrigatória, mas *ContactName* permite valores nulos):

```
[Table(Name = "Suppliers")]
public class Supplier
{
 [Column(IsPrimaryKey = true, CanBeNull = false)]
 public int SupplierID { get; set; }

 [Column(CanBeNull = false)]
 public string CompanyName { get; set; }

 [Column]
 public string ContactName { get; set; }
}
```

Você pode então instanciar as coleções *Table<Product>* e *Table<Supplier>* e definir uma consulta LINQ to SQL para agrupar essas tabelas, como a seguir:

```
DataContext db = new DataContext(...);
Table<Product> products = db.GetTable<Product>();
Table<Supplier> suppliers = db.GetTable<Supplier>();
var productsAndSuppliers = from p in products
 join s in suppliers
 on p.SupplierID equals s.SupplierID
 select new { p.ProductName, s.CompanyName, s.ContactName };
```

Ao iterar pela coleção *productsAndSuppliers*, a LINQ to SQL executará uma instrução SQL SELECT que combina as tabelas *Products* e *Suppliers* no banco de dados com base na coluna *SupplierID*, nas duas tabelas, e recupera os dados.

Mas com a LINQ to SQL você pode especificar as relações entre as tabelas como parte da definição das classes de entidade. A LINQ to SQL pode então buscar automaticamente as informações sobre o fornecedor para cada produto sem exigir que você construa uma instrução *join* potencialmente complexa e propensa a erro. Voltando ao exemplo das tabelas de produtos e fornecedores, elas têm uma relação de “muitos para um” no banco de dados Northwind; cada produto é oferecido por

um único fornecedor, mas um único fornecedor pode oferecer vários produtos. Expressando essa relação de uma maneira um pouco diferente, uma linha na tabela *Products* pode referenciar uma única linha na tabela *Suppliers* por meio das colunas *SupplierID* nas duas tabelas, mas uma linha na tabela *Suppliers* pode referenciar um conjunto inteiro de linhas na tabela *Products*. A LINQ to SQL fornece os tipos genéricos *EntityRef< TEntity >* e *EntitySet< TEntity >* para modelar esse tipo de relação. Selecionando a classe entidade *Product* primeiro, você pode definir o lado da relação com a classe de entidade *Supplier* utilizando o tipo *EntityRef< Supplier >*, como mostrado aqui em negrito:

```
[Table(Name = "Products")]
public class Product
{
 [Column(IsPrimaryKey = true, CanBeNull = false)]
 public int ProductID { get; set; }

 [Column]
 public int? SupplierID { get; set; }

 private EntityRef<Supplier> supplier;
 [Association(Storage = "supplier", ThisKey = "SupplierID", OtherKey = "SupplierID")]
 public Supplier Supplier
 {
 get { return this.supplier.Entity; }
 set { this.supplier.Entity = value; }
 }
}
```

O campo privado *supplier* é uma referência a uma instância da classe de entidade *Supplier*. A propriedade pública *Supplier* fornece acesso a essa referência. O atributo *Association* especifica como a LINQ to SQL localiza e preenche os dados para essa propriedade. O parâmetro *Storage* identifica o campo *private* utilizado para armazenar a referência ao objeto *Supplier*. O parâmetro *ThisKey* indica qual propriedade da classe de entidade *Product* a LINQ to SQL deve utilizar para localizar a referência na tabela *Supplier* a esse produto, e o parâmetro *OtherKey* especifica qual propriedade na tabela *Supplier* a LINQ to SQL deve corresponder ao valor do parâmetro *ThisKey*. Neste exemplo, as tabelas *Product* e *Suppliers* são combinadas com base na propriedade *SupplierID* nas duas entidades.

 **Nota** O parâmetro *Storage* é opcional. Se você especificá-lo, a LINQ to SQL acessa diretamente o membro de dados correspondente ao preenchê-lo em vez de passar pelo método de acesso *set*. O método de acesso *set* é requerido para aplicativos que preenchem ou alteram manualmente o objeto entidade referenciado pela propriedade *EntityRef< TEntity >*. Embora o parâmetro *Storage* seja redundante nesse exemplo, é uma prática recomendável incluí-lo.

O método de acesso *get* na propriedade *Supplier* retorna uma referência à entidade *Supplier* utilizando a propriedade *Entity* do tipo *EntityRef< Supplier >*. O método de acesso *set* preenche essa propriedade com uma referência a uma entidade *Supplier*.

Você pode definir o lado “muitos” da relação na classe *Supplier* com o tipo *EntitySet<Product>*, como a seguir:

```
[Table(Name = "Suppliers")]
public class Supplier
{
 [Column(IsPrimaryKey = true, CanBeNull = false)]
 public int SupplierID { get; set; }

 private EntitySet<Product> products = null;
 [Association(Storage = "products", OtherKey = "SupplierID", ThisKey = "SupplierID")]
 public EntitySet<Product> Products
 {
 get { return this.products; }
 set { this.products.Assign(value); }
 }
}
```

**Dica** É uma convenção utilizar um substantivo no singular para o nome de uma classe de entidade e suas propriedades. A exceção a essa regra é que propriedades *EntitySet< TEntity >* em geral recebem a forma plural porque elas representam uma coleção em vez de uma única entidade.

Desta vez, o parâmetro *Storage* do atributo *Association* especifica o campo *EntitySet<Product>* privado. Um objeto *EntitySet< TEntity >* armazena uma coleção de referências a entidades. O método de acesso *get* da propriedade pública *Products* retorna essa coleção. O método de acesso *set* utiliza o método *Assign* da classe *EntitySet<Product>* para preencher essa coleção.

Portanto, utilizando os tipos *EntityRef< TEntity >* e *EntitySet< TEntity >* você pode definir propriedades que podem modelar uma relação de “um para muitos”, mas como você realmente preenche essas propriedades com dados? A resposta é que a LINQ to SQL as preenche para você ao buscar os dados. O código a seguir cria uma instância da classe *Table<Product>* e emite uma consulta LINQ to SQL para buscar os detalhes de todos os produtos. Esse código é semelhante ao primeiro exemplo de LINQ to SQL que vimos anteriormente. A diferença está no loop *foreach* que exibe os dados.

```
DataContext db = new DataContext(...);
Table<Product> products = db.GetTable<Product>();

var productsAndSuppliers = from p in products
 select p;

foreach (var product in productsAndSuppliers)
{
 Console.WriteLine("Product {0} supplied by {1}",
 product.ProductName, product.Supplier.CompanyName);
}
```

A instrução `Console.WriteLine` lê o valor da propriedade `ProductName` da entidade `produto` da mesma forma que antes, mas também acessa a entidade `Supplier` e exibe a propriedade `CompanyName` dessa entidade. Se executar esse código, a saída será parecida com esta:

```
C:\Windows\system32>cmd.exe
Product Perth Pasties supplied by G'day, Mate
Product Tourtiere supplied by La Maison
Product Pâté chinois supplied by La Maison
Product Gnocchi di nonna Alice supplied by Pasta Buttini s.r.l.
Product Ravioli Angelo supplied by Pasta Buttini s.r.l.
Product Escargots de Bourgogne supplied by Escargots Nouveaux
Product Raclette Courdavault supplied by Gai pâturage
Product Camembert Pierrot supplied by Gai pâturage
Product Sirup d'érable supplied by Forêt d'érables
Product Tarte au sucre supplied by Forêt d'érables
Product Vegie-spread supplied by Paulova, Ltd.
Product Wiener's gute Semmelknödel supplied by Plätzter Lebensmittelgroßmärkte AG
Product Louisiana Fiery Hot Pepper Sauce supplied by New Orleans Cajun Delights
Product Louisiana Hot Spiced Okra supplied by New Orleans Cajun Delights
Product Laughing Lumberjack Lager supplied by Bigfoot Breweries
Product Scottish Longbread supplied by Specialty Biscuits, Ltd.
Product Gudbrandsdalost supplied by Norske Meierier
Product Outback Lager supplied by Paulova, Ltd.
Product Flatenjordt supplied by Norske Meierier
Product Mozzarellla di Giovanni supplied by Formaggi Fortini s.r.l.
Product Röd Kaviar supplied by Sverrik SJÖFÉDA AB
Product LongLife Tofu supplied by Tokyo Traders
Product Rhönherz Klosterbier supplied by Plätzter Lebensmittelgroßmärkte AG
Product Lakkaribbotti supplied by Maraki Oy
Product Original Frankfurter urinie Soße supplied by Plätzter Lebensmittelgroßmärkte AG
```

À medida que o código busca cada entidade `Product`, a LINQ to SQL executa uma segunda consulta, adiada, para recuperar os detalhes do fornecedor desse produto a fim de poder preencher a propriedade `Supplier`, com base na relação especificada pelo atributo `Association` dessa propriedade na classe de entidade `Product`.

Depois de definir as entidades `Product` e `Supplier` como tendo uma relação de “um para muitos”, lógica semelhante será aplicada se você executar uma consulta LINQ to SQL contra a coleção `Table<Supplier>`, como a seguir:

```
DataContext db = new DataContext(...);
Table<Supplier> suppliers = db.GetTable<Supplier>();
var suppliersAndProducts = from s in suppliers
 select s;

foreach (var supplier in suppliersAndProducts)
{
 Console.WriteLine("Supplier name: {0}", supplier.CompanyName);
 Console.WriteLine("Products supplied");
 foreach (var product in supplier.Products)
 {
 Console.WriteLine("\t{0}", product.ProductName);
 }
 Console.WriteLine();
}
```

Nesse caso, ao buscar um fornecedor, o loop `foreach` executa uma segunda consulta (de novo adiada) para recuperar todos os produtos desse fornecedor e preencher a propriedade `Products`. Desta vez,

porém, a propriedade é uma coleção (uma `EntitySet<Product>`), portanto, é possível codificar uma instrução `foreach` aninhada para iterar pelo conjunto, exibindo o nome de cada produto. A saída desse código se assemelha a isso:

```
C:\Windows\system32\cmd.exe
Supplier name: Exotic Liquids
Products supplied
 Chai
 Chang
 #Aniseed Syrup

Supplier name: New Orleans Cajun Delights
Products supplied
 Chef Anton's Cajun Seasoning
 Chef Anton's Gumbo Mix
 Louisiana Flery Hot Pepper Sauce
 Louisiana Hot Spiced Okra

Supplier name: Grandma Kelly's Homestead
Products supplied
 Grandma's Boysenberry Spread
 Uncle Bob's Organic Dried Pears
 Northwoods Cranberry Sauce

Supplier name: Tokyo Traders
Products supplied
 Mishи Kebab Niku
 Iwana
 Longlife Tofu
```

## Busca adiada e imediata, uma retomada do assunto

Anteriormente neste capítulo, mencionamos que a LINQ to SQL adia a busca dos dados até eles serem solicitados, mas que você poderia aplicar o método de extensão `ToList` ou `ToArrayList` para recuperá-los imediatamente. Essa técnica não se aplica a dados referenciados como propriedades `EntitySet< TEntity >` ou `EntityRef< TEntity >`; mesmo se você utilizar `ToList` ou `ToArrayList`, eles ainda serão buscados apenas quando acessados. Se quiser forçar a LINQ to SQL a consultar e buscar dados referenciados imediatamente, configure a propriedade `LoadOptions` do objeto `DataContext`, como a seguir:

```
DataContext db = new DataContext(...);
Table<Supplier> suppliers = db.GetTable<Supplier>();
DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.LoadWith<Supplier>(s => s.Products);
db.LoadOptions = loadOptions;
var suppliersAndProducts = from s in suppliers
 select s;
```

A classe `DataLoadOptions` fornece o método genérico `LoadWith`. Utilizando esse método, você pode especificar se uma propriedade `EntitySet< TEntity >` de uma instância deve ser carregada quando a instância é preenchida. O parâmetro para o método `LoadWith` é uma expressão lambda que identifica os dados relacionados a serem recuperados, quando os dados de uma tabela forem pesquisados. O exemplo mostrado aqui faz a propriedade `Products` de cada entidade `Supplier` ser preenchida assim que os dados para cada entidade `Supplier` são buscados em vez de serem adiados. Se você especificar a propriedade `LoadOptions` do objeto `DataContext` junto com o método de extensão `ToList` ou `ToArrayList` de uma coleção `Table`, a LINQ to SQL carregará a coleção inteira e também os dados para as propriedades referenciadas das entidades dessa coleção na memória, assim que a consulta LINQ to SQL for avaliada.



**Dica** Se houver várias propriedades `EntitySet< TEntity >`, você poderá chamar o método `LoadWith` do mesmo objeto `LoadOptions` várias vezes, especificando em cada vez a `EntitySet< TEntity >` a carregar.

## Definindo uma classe *DataContext* personalizada

A classe *DataContext* fornece funcionalidades para gerenciar bancos de dados e conexões de banco de dados, criar classes de entidade e executar comandos para recuperar e atualizar os dados em um banco de dados. Embora você possa utilizar a classe *DataContext* bruta fornecida com o .NET Framework, uma prática melhor é usar herança e definir sua própria versão especializada que declara as várias coleções *Table< TEntity >* como membros públicos. Por exemplo, eis uma classe *DataContext* especializada que exibe as coleções *Table Products* e *Suppliers* como membros públicos:

```
public class Northwind : DataContext
{
 public Table<Product> Products;
 public Table<Supplier> Suppliers;

 public Northwind(string connectionInfo) : base(connectionInfo)
 {
 }
}
```

Observe que a classe *Northwind* também fornece um construtor que recebe uma string de conexão como um parâmetro. Você pode criar uma nova instância da classe *Northwind* e então definir e executar consultas LINQ to SQL em relação a classes de coleção *Tables* que ela expõe, como mostrado a seguir:

```
Northwind nwindDB = new Northwind(...);

var suppliersQuery = from s in nwindDB.Suppliers
 select s;

foreach (var supplier in suppliersQuery)
{

}
```

Essa prática torna seu código mais fácil de manter. Utilizando um objeto *DataContext* comum, você pode instanciar qualquer classe de entidade utilizando o método *GetTable*, independentemente do banco de dados ao qual o objeto *DataContext* se conecta. Você descobre que utilizou o objeto *DataContext* errado e se conectou ao banco de dados errado apenas em tempo de execução, ao tentar recuperar os dados. Com uma classe *DataContext* personalizada, você referencia as coleções *Table* por meio do objeto *DataContext*. (O construtor *base DataContext* utiliza um mecanismo chamado *reflexão* para examinar seus membros e instanciar automaticamente quaisquer membros que sejam coleções *Table* – os detalhes sobre como a reflexão funciona estão além do escopo deste livro.) É óbvio com qual banco de dados você precisa se conectar para recuperar dados de uma tabela específica; se o IntelliSense não exibir sua tabela no momento em que você definir a consulta LINQ to SQL, você selecionou a classe *DataContext* errada e seu código não irá compilar.

## Utilizando a LINQ to SQL para consultar informações de pedidos

No próximo exercício, você vai escrever uma versão do aplicativo de console desenvolvido no exercício anterior, que solicita ao usuário um ID de cliente e exibe os detalhes de qualquer pedido feito por esse cliente. Utilizaremos a LINQ to SQL para recuperar os dados. Você então será capaz de comparar a LINQ to SQL com o código equivalente escrito com o ADO.NET.

## Defina a classe de entidade Order

1. Utilizando o Visual Studio 2010, crie um novo projeto chamado **LINQOrders** utilizando o template Console Application. Salve-o na pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 25` na sua pasta Documentos.
2. No *Solution Explorer*, mude o nome do arquivo `Program.cs` para **LINQReport.cs**. Na caixa de mensagem *Microsoft Visual Studio*, clique em *Yes* para alterar todas as referências da classe `Program` para `LINQReport`.
3. No menu *Project*, clique em *Add Reference*. Na caixa de diálogo *Add Reference*, clique na guia `.NET`, selecione o assembly `System.Data.Linq` e então clique em *OK*.

Esse assembly contém os tipos e atributos de LINQ to SQL.

4. Na janela *Code and Text Editor*, adicione as seguintes diretivas `using` à lista, na parte superior do arquivo:

```
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Data.SqlClient;
```

5. Adicione a classe de entidade `Order` ao arquivo `LINQReport.cs`, depois da classe `LINQReport`. Marque a classe `Order` com um atributo `Table`, como a seguir:

```
[Table(Name = "Orders")]
public class Order
{
}
```

A tabela é chamada `Orders` no banco de dados Northwind. Lembre-se de que uma prática comum é utilizar o substantivo no singular para o nome de uma classe de entidade porque um objeto entidade representa uma linha do banco de dados.

6. Adicione a propriedade mostrada aqui, em negrito, à classe `Order`:

```
[Table(Name = "Orders")]
public class Order
{
 [Column(IsPrimaryKey = true, CanBeNull = false)]
 public int OrderID { get; set; }
}
```

A coluna `OrderID` é a chave primária para essa tabela no banco de dados Northwind.

7. Adicione as seguintes propriedades mostradas em negrito à classe `Order`:

```
[Table(Name = "Orders")]
public class Order
{
 ...
 [Column]
 public string CustomerID { get; set; }

 [Column]
 public DateTime? OrderDate { get; set; }

 [Column]
 public DateTime? ShippedDate { get; set; }
```

```
[Column]
public string ShipName { get; set; }

[Column]
public string ShipAddress { get; set; }

[Column]
public string ShipCity { get; set; }

[Column]
public string ShipCountry { get; set; }

}
```

Essas propriedades contêm o ID de cliente, a data do pedido e as informações de despacho para um pedido. No banco de dados, todas essas colunas permitem valores nulos, portanto, é importante utilizar a versão nullable do tipo *DateTime* para as propriedades *OrderDate* e *ShippedDate*. (Observe que *string* é um tipo-referência que permite automaticamente valores nulos.) Perceba que a LINQ to SQL mapeia automaticamente o tipo *NVarChar* do SQL Server para o tipo *string* do .NET Framework e o tipo *DateTime* do SQL Server para o tipo *DateTime* do .NET Framework.

8. Adicione a seguinte classe *Northwind* ao arquivo de *LINQReport.cs* após a classe de entidade *Order*:

```
public class Northwind : DataContext
{
 public Table<Order> Orders;

 public Northwind(string connectionInfo)
 : base (connectionInfo)
 {
 }
}
```

A classe *Northwind* é uma classe *DataContext* que expõe uma propriedade *Table* com base na classe de entidade *Order*. No próximo exercício, você utilizará essa versão especializada da classe *DataContext* para acessar a tabela *Orders* no banco de dados.

## Recupere informações sobre pedidos utilizando uma consulta LINQ to SQL

1. No método *Main* da classe *LINQReport*, adicione o código mostrado aqui, em negrito, que cria um objeto *Northwind*:

```
static void Main(string[] args)
{
 SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
 builder.DataSource = ".\\SQLExpress";
 builder.InitialCatalog = "Northwind";
 builder.IntegratedSecurity = true;

 Northwind northwindDB = new Northwind(builder.ConnectionString);
}
```

A string de conexão construída por meio do objeto *SqlConnectionStringBuilder* é exatamente a mesma do exercício anterior. O objeto *northwindDB* utiliza essa string para se conectar ao banco de dados Northwind.

2. Depois do código adicionado na etapa anterior, adicione um bloco *try/catch* ao método *Main*.

```
static void Main(string[] args)
{
 ...
 try
 {
 // Você adicionará seu código aqui mais adiante
 }
 catch (SqlException e)
 {
 Console.WriteLine("Error accessing the database: {0}", e.Message);
 }
}
```

Da mesma forma que ao utilizar código ADO.NET normal, a LINQ to SQL lança uma *SqlException* se um erro ocorrer ao acessar um banco de dados SQL Server.

3. Substitua o comentário no bloco *try* pelo seguinte código mostrado em negrito:

```
try
{
 Console.Write("Please enter a customer ID (5 characters): ");
 string customerId = Console.ReadLine();
}
```

Essas instruções pedem um ID de cliente ao usuário e salvam a resposta do usuário na variável *customerId*.

4. Digite a instrução mostrada em negrito depois do código que você acabou de inserir:

```
try
{
 ...
 var ordersQuery = from o in northwindDB.Orders
 where String.Equals(o.CustomerID, customerId)
 select o;
}
```

Essa instrução define a consulta LINQ to SQL que recuperará os pedidos para o cliente especificado.

5. Adicione a instrução *foreach* e o bloco *if...else* mostrado em negrito, após o código que você adicionou no passo anterior:

```
try
{
 ...
 foreach (var order in ordersQuery)
 {
 if (order.ShippedDate == null)
 {
 Console.WriteLine("Order {0} not yet shipped\n\n", order.OrderID);
 }
 else
 {
 // Exibe os detalhes do pedido
 }
 }
}
```

A instrução *foreach* itera pelos pedidos do cliente. Se o valor na coluna *ShippedDate* no banco de dados for *null*, a propriedade correspondente no objeto entidade *Order* também será *null* e então a instrução *if* enviará uma mensagem adequada para a saída.

6. Substitua o comentário na parte *else* da instrução *if* que você adicionou no passo anterior pelo código mostrado aqui em negrito:

```
if (order.ShippedDate == null)
{
 ...
}
else
{
 Console.WriteLine("Order: {0}\nPlaced: {1}\nShipped: {2}\n" +
 "To Address: {3}\n{4}\n{5}\n{6}\n\n", order.OrderID,
 order.OrderDate, order.ShippedDate, order.ShipName,
 order.ShipAddress, order.ShipCity,
 order.ShipCountry);
}
```

7. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo.
8. No console da janela que exibe a mensagem “Please enter a customer ID (5 characters)”, digite **VINET**.

O aplicativo deve exibir uma lista dos pedidos para esse cliente. Depois que o aplicativo terminar, pressione Enter para retornar ao Visual Studio 2010.

9. Execute o formulário novamente. Dessa vez digite **BONAP** quando o ID de um cliente for solicitado.

O pedido final para esse cliente ainda não foi despachado e contém um valor nulo para a coluna *ShippedDate*. Verifique se o aplicativo detecta e trata esse valor nulo. Quando o aplicativo terminar, pressione Enter para retornar ao Visual Studio 2010.

Neste capítulo, você viu os elementos básicos que a LINQ to SQL fornece para consultar informações, a partir de um banco de dados. A LINQ to SQL tem vários outros recursos que você pode empregar nos seus aplicativos, incluindo a capacidade de modificar os dados e atualizar um banco de dados. Examinaremos rapidamente algum desses aspectos da LINQ to SQL no próximo capítulo.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 26.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 25

Para	Faça isto
Conectar-se a um banco de dados SQL Server utilizando o ADO.NET	Crie um objeto <i>SqlConnection</i> , defina sua propriedade <i>ConnectionString</i> com detalhes que especifiquem o banco de dados a ser utilizado e chame o método <i>Open</i> .
Criar e executar uma consulta de banco de dados utilizando o ADO.NET	Crie um objeto <i>SqlCommand</i> . Defina sua propriedade <i>Connection</i> como um objeto <i>SqlConnection</i> válido. Defina sua propriedade <i>CommandText</i> como uma instrução SQL <i>SELECT</i> válida. Chame o método <i>ExecuteReader</i> para executar a consulta e crie um objeto <i>SqlDataReader</i> .
Buscar dados utilizando um objeto ADO.NET <i>SqlDataReader</i>	Certifique-se de que os dados não são nulos utilizando o método <i>IsDBNull</i> . Se os dados não forem nulos, use o método <i>GetXXX</i> apropriado (como <i>GetString</i> ou <i>GetInt32</i> ) para recuperar os dados.
Definir uma classe de entidade	Defina uma classe com propriedades públicas para cada coluna. Prefixe a definição da classe com o atributo <i>Table</i> , especificando o nome da tabela no banco de dados subjacente. Prefixe cada coluna com o atributo <i>Column</i> , e especifique parâmetros que indicam o nome, o tipo e a nulabilidade da coluna correspondente no banco de dados.
Criar e executar uma consulta usando LINQ to SQL	Crie um objeto <i>DataContext</i> e especifique uma string de conexão ao banco de dados. Crie uma coleção <i>Table</i> baseada na classe de entidade que corresponde à tabela que você deseja consultar. Defina uma consulta LINQ to SQL que identifique os dados a serem obtidos do banco de dados e retorne uma coleção enumerável de entidades. Itere pela coleção enumerável para obter os dados de cada linha e processe os resultados.

## Capítulo 26

# Exibindo e editando dados com o Entity Framework e vinculação de dados

Neste capítulo, você vai aprender a:

- Utilizar o ADO.NET Entity Framework para gerar classes de entidade.
- Utilizar vinculação de dados em um aplicativo Windows Presentation Foundation (WPF) para exibir e manter os dados recuperados de um banco de dados.
- Atualizar um banco de dados utilizando o Entity Framework.
- Detectar e resolver atualizações conflitantes feitas por múltiplos usuários.

No Capítulo 25, “Consultando informações em um banco de dados”, você aprendeu os conceitos básicos da utilização do Microsoft ADO.NET e a LINQ to SQL para fazer consultas em um banco de dados. O principal objetivo da LINQ to SQL é fornecer uma interface LINQ para o Microsoft SQL Server. Entretanto, o modelo subjacente utilizado pela LINQ to SQL é extensível, e alguns fornecedores de produtos de terceiros construíram provedores de dados que podem acessar outros sistemas de gerenciamento de bancos de dados.

O Visual Studio 2010 também dispõe de uma tecnologia, chamada Entity Framework, que você pode utilizar para consultar e manipular bancos de dados. Contudo, onde a LINQ to SQL gera código muito parecido com a estrutura do banco de dados, você pode usar o Entity Framework para gerar um modelo lógico de um banco de dados, chamado de modelo de dados de entidade, e pode escrever seu código sobre esse modelo lógico. Ao utilizar a Entity Framework, você pode construir classes que mapeiam os itens no modelo lógico (*ou entidades*) para tabelas físicas no banco de dados. Essa camada de mapeamento pode ajudar a proteger seus aplicativos contra quaisquer mudanças ocorridas na estrutura do banco de dados, posteriormente, e também pode ser aplicada para propiciar um grau de independência em relação à tecnologia usada para implementar o banco de dados. Por exemplo, você pode construir um aplicativo que utiliza o Entity Framework para acessar dados em um banco de dados Oracle e, posteriormente, migrar esse banco de dados para o SQL Server. Não será necessário mudar a lógica em seu aplicativo; basta atualizar o modo de implementação das entidades lógicas na camada de mapeamento.

O Entity Framework pode operar com uma variante da LINQ, chamada *LINQ to Entities*. Ao utilizar a LINQ to Entities, você poderá consultar e manipular os dados por meio de um modelo de objeto de entidade, através da sintaxe da LINQ.

Neste capítulo, você aprenderá a usar a Entity Framework para gerar um modelo de dados lógicos, e depois escreverá os aplicativos que utilizam vinculação de dados para exibir e modificar dados com esse modelo.



**Nota** Este capítulo apresenta apenas uma introdução resumida do Entity Framework e da LINQ to Entities. Para obter mais informações, consulte a documentação fornecida com o Microsoft Visual Studio 2010, ou acesse a página do ADO.NET Entity Framework na site da Microsoft, em [http://msdn.microsoft.com/en-us/library/bb399572\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb399572(VS.100).aspx).

## Utilizando vinculação de dados com Entity Framework

No Capítulo 24, “Realizando validações”, vimos a vinculação de dados em um aplicativo WPF quando você utilizou essa técnica para associar as propriedades dos controles em um formulário WPF com as propriedades de uma instância de uma classe. Você pode adotar uma estratégia semelhante e vincular as propriedades dos controles a objetos de entidade de modo que possa exibir e manter os dados seguros em um banco de dados utilizando uma interface gráfica com o usuário. Entretanto, você precisa primeiramente definir as classes de entidades que serão mapeadas para as tabelas no banco de dados.

No Capítulo 25, você utilizou a LINQ to SQL para construir uma série de classes de entidade e uma classe de contexto. O Entity Framework funciona de modo semelhante, mas um pouco mais expansivo, e vários contextos que você conheceu no Capítulo 25 ainda se aplicam. O Entity Framework oferece o template e os assistentes do Modelo de Dados de Entidade ADO.NET, que podem gerar classes de entidade do banco de dados. (Você também pode definir manualmente um modelo de entidade e usá-lo para criar um banco de dados.) O Entity Framework também gera uma classe de contexto personalizada, que você pode usar para acessar as entidades e para estabelecer conexão com o banco de dados.

No primeiro exercício, você utilizará o template Modelo de Dados de Entidade ADO.NET a fim de gerar um modelo de dados para gerenciar produtos e fornecedores no banco de dados Northwind.



**Importante** Os exercícios deste capítulo pressupõem que você tenha criado e preenchido o banco de dados Northwind. Para obter mais informações, consulte o exercício na seção “Criando um banco de dados”, no Capítulo 25.

## Concedendo acesso a um arquivo de banco de dados SQL Server 2008 – Visual C# 2010 Express

Se você estiver utilizando o Microsoft Visual C# 2010 Express Edition, ao definir uma conexão de banco de dados do Microsoft SQL Server para o assistente de entidade, você se conecta diretamente ao arquivo de banco de dados do SQL Server. O Visual C# 2010 Express Edition inicia uma *instância* própria do SQL Server Express, chamada instância de usuário, para acessar o banco de dados. A instância de usuário executa com as credenciais do usuário que está executando o aplicativo. Se você estiver utilizando o Visual C# 2010 Express Edition, você deverá desanexar a instância padrão do banco de dados SQL Server Express porque ele não permitirá que uma instância de usuário conecte-se a um banco de dados que ele está utilizando atualmente. Os seguintes procedimentos descrevem como realizar essas tarefas.

### Desanexe o banco de dados Northwind

1. No menu *Iniciar* do Windows, clique em *Todos os Programas*, clique em *Acessórios* e então clique em *Prompt de Comando* para abrir a janela de prompt de comando.
2. Na janela de prompt de comando, digite o seguinte comando para acessar a pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 26` na sua pasta Documentos. Substitua *Nome* pelo seu nome de usuário.  
`cd "\Users\Nome\Documentos\Microsoft Press\Visual CSharp Step by Step\Chapter 26"`
3. Na janela de prompt de comando, digite o comando a seguir:

```
sqlcmd -S.\SQLExpress -E -idetach.sql
```

**Nota** O script `detach.sql` contém o seguinte comando SQL Server, que desanexa o banco de dados `Northwind` da instância do SQL Server:

```
sp_detach_db 'Northwind'
```

4. Quando o script terminar a execução, feche a janela de prompt de comando.

**Nota** Se precisar reconstruir o banco de dados `Northwind`, execute o script `instnwnd.sql`, como descrito no Capítulo 25. Mas se desanexou o banco de dados `Northwind`, você primeiro deverá excluir os arquivos `Northwind.mdf` e `Northwind_log.ldf` na pasta `C:\Arquivos de Programas\ Microsoft SQL Server\MSSQL.10.SQLEXPRESS\MSSQL\Data`; caso contrário, o script falhará.

Após desanexar o banco de dados do SQL Server, você deverá conceder à sua conta de login acesso à pasta que contém o banco de dados e conceder Controle Total (Full Control) sobre os próprios arquivos do banco de dados. O procedimento a seguir mostra como fazer isso.

### Conceda acesso ao arquivo de banco de dados Northwind

1. Efetue o logon no seu computador utilizando uma conta com acesso de administrador.
2. Utilizando o Windows Explorer, acesse a pasta C:\Arquivos de Programas\Microsoft SQL Server\MSSQL.10.SQLEXPRESS\MSSQL.

**Nota** Se você estiver utilizando uma versão de 64 bits do Windows Vista ou Windows 7, substitua todas as referências às pasta C:\Program Files nessas instruções por C:\Program Files (x86).

3. Se aparecer uma caixa de mensagem com a mensagem "You don't currently have permission to access this folder", clique em *Continue*. Na mensagem *User Account Control* exibida a seguir, clique em *Continue* novamente.
  4. Mude para a pasta DATA, clique com o botão direito do mouse no arquivo Northwind e clique em *Properties*.
  5. Na caixa de diálogo *Northwind Properties*, clique na guia *Security*.
  6. Se a página *Security* contiver a mensagem "Do you want to continue?", clique em *Continue*. Na caixa de mensagem *User Account Control*, clique em *Continue*.
- Se a página *Security* contiver a mensagem "To change permissions, click *Edit*", clique em *Edit*. Se aparecer a caixa de mensagem *User Account Control*, clique em *Continue*.
7. Se sua conta de usuário não constar na caixa de listagem *Group or user names*, na caixa de diálogo *Permissions for Northwind*, clique em *Add*. Na caixa de diálogo *Select Users or Groups*, digite o nome de sua conta de usuário e clique em *OK*.
  8. Na caixa de diálogo *Permissions for Northwind*, na caixa de listagem *Group or user names*, clique em sua conta de usuário.
  9. Na caixa de listagem *Permissions for Account* (onde *Account* for o nome de sua conta de usuário), marque a caixa de seleção *Allow* da entrada *Full Control* e clique em *OK*.
  10. Na caixa de diálogo *Northwind Properties*, clique em *OK*.
  11. Repita as etapas 4 a 10 para o arquivo *Northwind\_log* na pasta DATA.

### Gere o Modelo de Dados de Entidade para as tabelas *Suppliers* e *Products*

1. Inicie o Visual Studio 2010, se ele ainda não estiver em execução.
2. Adicione um novo projeto usando o template WPF Application. Nomeie o projeto como **Suppliers e Products** e salve-o na pasta \Microsoft Press\Visual CSharp Step by Step\Chapter 26 na sua pasta Documentos.



**Nota** Se estiver usando o Visual C# 2010 Express Edition, você poderá especificar o local para salvar seu projeto, ao clicar em *Save Suppliers* no menu *File*.

3. No *Solution Explorer*, clique com o botão direito do mouse no projeto *Suppliers*, aponte para *Add* e clique em *New Item*.
4. Na caixa de diálogo *Add New Item – Suppliers*, no painel da esquerda, expanda *Visual C#* se ainda não estiver expandido. No painel central, faça uma rolagem para baixo e clique no template *ADO.NET Entity Data Model*, digite **Northwind.edmx** na caixa *Name* e clique em *Add*.

É exibida a caixa de diálogo *Entity Data Model Wizard*. Use essa janela para especificar as tabelas no banco de dados Northwind para as quais você quer criar classes de entidade, selecione as colunas a serem incluídas, e defina as relações entre elas.

5. Na caixa de diálogo *Entity Data Model Wizard*, selecione *Generate from database* e clique em *Next*.

O *Entity Data Model Wizard* exige a configuração de uma conexão com um banco de dados, e a página *Choose Your Data Connection* é exibida.

6. Se você estiver usando o Visual Studio 2010 Standard Edition ou Visual Studio 2010 Professional Edition, execute as seguintes tarefas:

#### 6.1. Clique em *New Connection*.

Se a caixa de diálogo *Choose Data Source* for exibida, na caixa de listagem *Data source*, clique em *Microsoft SQL Server*. Na caixa de listagem suspensa *Data provider*, selecione *.NET Framework Data Provider for SQL Server*, se ainda não estiver selecionado, e clique em *Continue*.



**Nota** Se você já tiver criado conexões com o banco de dados previamente, essa caixa de diálogo pode não aparecer e será exibida a caixa de diálogo *Connection Properties*. Nesse caso, clique no botão *Change* ao lado da caixa de texto *Data source*. É exibida a caixa de diálogo *Change Data Source*, que é a mesma caixa de diálogo *Choose Data Source*, exceto pelo fato de que o botão *Continue* tem a legenda *OK*.

- 6.2. Na caixa de diálogo *Connection Properties*, na caixa de combinação *Server name*, digite **\SQLEXPRESS**. Na seção *Log on to the server* da caixa de diálogo, selecione o botão de opção *Use Windows Authentication*. Na seção *Connect to a database* da caixa de diálogo, na caixa de combinação *Select or enter a database name*, digite **Northwind** e clique em *OK*.

7. Se estiver utilizando o Visual C# 2010 Express Edition, execute as seguintes tarefas:

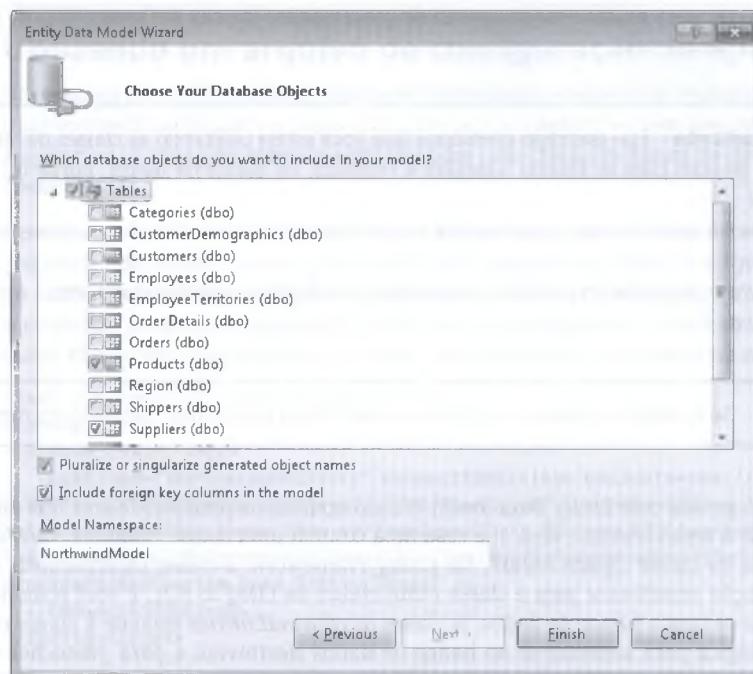
- 7.1. Clique em *New Connection*.

Se a caixa de diálogo *Choose Data Source* for exibida, na caixa de listagem *Data source*, clique em *Microsoft SQL Server Database File*. Na caixa de listagem suspensa *Data provider*, selecione *.NET Framework Data Provider for SQL Server*, se ainda não estiver selecionado, e clique em *Continue*.



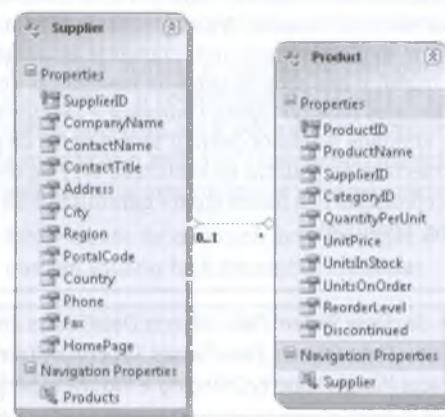
**Nota** Se você já tiver criado conexões com o banco de dados previamente, essa caixa de diálogo pode não aparecer e será exibida a caixa de diálogo *Connection Properties*. Nesse caso, clique no botão *Change* ao lado da caixa de texto *Data source*. É exibida a caixa de diálogo *Change Data Source*, que é a mesma caixa de diálogo *Choose Data Source*, exceto pelo fato de que o botão *Continue* tem a legenda *OK*.

- 7.2. Na caixa de diálogo *Connection Properties*, no texto *Database file name*, clique em **Browse**.
  - 7.3. Na caixa de diálogo *Select SQL Server Database File*, mude para a pasta C:\Program Files\Microsoft SQL Server\MSSQL10.SQLEXPRESS\MSSQL\DATA, clique no arquivo de banco de dados Northwind, e clique em *Open*.
  - 7.4. Na seção *Log on to the server* da caixa de diálogo, selecione o botão de opção *Use Windows Authentication* e clique em *OK*.
8. Na página *Choose Your Data Connection* do *Entity Data Model Wizard*, marque a caixa de seleção *Save entity connection settings in App.Config as*, digite **NorthwindEntities** (esse é o nome padrão), e clique em *Next*.
9. Na página *Choose Your Database Objects*, verifique se as caixas de seleção *Pluralize or singularize generated object names* e *Include foreign key columns in the model* estão marcadas. Na caixa de listagem *Which database objects do you want to include in your model?*, expanda *Tables* e clique nas tabelas *Products (dbo)* e *Suppliers (dbo)*. Na caixa de texto *Model Namespace*, digite **NorthwindModel** (esse é o namespace padrão). A imagem a seguir mostra a página concluída.



10. Clique em *Finish*.

O Entity Data Model Wizard gera as classes de entidade, chamadas *Supplier* e *Product*, baseadas nas tabelas *Suppliers* e *Products*, com campos de propriedade para cada coluna nas tabelas, como na imagem a seguir. O modelo de dados também define propriedades de navegação que ligam as duas entidades e mantêm a relação entre elas. Nesse caso, uma única entidade *Supplier* pode estar relacionada a muitas entidades *Product*.



Para modificar as propriedades de uma classe de entidade, selecione a classe e altere os valores das propriedades na janela *Properties*. Use também o painel *Mapping Details*, exibido na parte inferior da janela, para selecionar e editar os campos que aparecem em uma classe de entidade.

É assim que você muda o mapeamento de propriedades lógicas em uma entidade para colunas físicas de uma tabela.



**Importante** Este exercício pressupõe que você esteja utilizando as classes de entidade padrão geradas para as tabelas Suppliers e Products, no banco de dados; portanto, não altere nada!

11. No *Solution Explorer*, expanda a pasta *Northwind.edmx* e clique duas vezes em *Northwind.designer.cs*.



**Dica** Se o *Solution Explorer* não estiver visível, clique em *Solution Explorer*, no menu *View*.

O código gerado pelo Entity Data Model Wizard aparece na janela *Code and Text Editor*. Se você expandir a área *Contexts*, verá que essa área contém uma classe chamada *NorthwindEntities*, derivada da classe *ObjectContext*. Na Entity Framework, a classe *ObjectContext* desempenha uma função semelhante para a classe *DataContext* da LINQ to SQL, e você pode utilizá-la para conectar-se com o banco de dados. A classe *NorthwindEntities* estende a classe *ObjectContext* com a lógica para conectar-se ao banco de dados Northwind e para preencher as entidades *Supplier* e *Product* (exatamente como uma classe *DataContext* personalizada na LINQ to SQL).

A informação relacionada à conexão, que você especificou antes de criar as duas classes de entidade, é gravada em um arquivo de configuração de aplicativo. O armazenamento da string de conexão em um arquivo de configuração permite que você modifique essa string sem recompilar o aplicativo; basta editar o arquivo de configuração. Esse recurso será útil se você precisar mover ou renomear o banco de dados, ou passar de um banco de dados de desenvolvimento local para um banco de dados de produção que tem o mesmo conjunto de tabelas.

O código das duas classes de entidades está localizado na região *Entities* do arquivo. Essas classes de entidade são um pouco mais complicadas do que aquelas que você criou manualmente no Capítulo 25, mas os princípios gerais são semelhantes. A complexidade adicional é a consequência de as classes de entidade estarem implementando indiretamente as interfaces *INotifyPropertyChanging* e *INotifyPropertyChanged*, e das propriedades de navegação utilizadas para ligar as entidades relacionadas. As interfaces *INotifyPropertyChanging* e *INotifyPropertyChanged* definem eventos que as classes de entidade acionam quando seus valores de propriedades são alterados. Os vários controles da interface do usuário na biblioteca WPF se inscrevem nesses eventos para detectar as alterações efetuadas nos dados e para garantir que as informações exibidas em um formulário WPF estejam atualizadas.



**Nota** As classes de entidade herdam da classe *System.Data.Objects.DataClasses.EntityObject*, que, por sua vez, herda da classe *System.Data.Objects.DataClasses.StructuralObject*. A classe *StructuralObject* implementa as interfaces *INotifyPropertyChanging* e *INotifyPropertyChanged*.

## Utilizando um arquivo de configuração de aplicativo

Um arquivo de configuração de aplicativo fornece um mecanismo muito útil que permite ao usuário modificar alguns recursos utilizados por um aplicativo sem precisar recompilar todo o aplicativo. A string de conexão utilizada para conectar a um banco de dados é um exemplo desse recurso.

Quando você utiliza o Entity Data Model Wizard para gerar classes de entidade, um novo arquivo é adicionado ao seu projeto chamado app.config. Essa é a origem para o arquivo de configuração de aplicativo e ela aparece na janela *Solution Explorer*. Você pode examinar o conteúdo do arquivo app.config dando um clique duplo nele. Você verá que ele é um arquivo XML, como mostrado aqui (o texto foi reformatado para caber na página impressa):

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <connectionStrings>
 <add name="NorthwindEntities" connectionString="metadata=res://*/Northwind.
 csdl|res://*/Northwind.ssdl|res://*/Northwind.msl;provider=System.Data.
 SqlClient;provider connection string="Data Source=.\SQLEXPRESS;Initial
 Catalog=Northwind;Integrated Security=True;MultipleActiveResultSets=True";
 providerName="System.Data.EntityClient" />
 </connectionStrings>
</configuration>
```

A string de conexão é mantida no elemento `<connectionStrings>` do arquivo, e contém um conjunto de elementos na forma `propriedade=valor`. Os elementos são separados por um caractere de ponto e vírgula. As principais propriedades são os elementos *Data Source*, *Initial Catalog* e *Integrated Security*, que você deve reconhecer dos exercícios anteriores.

Quando você compila o aplicativo, o compilador C# copiará o arquivo app.config para a pasta que armazena o código compilado, e o renomeará para *application.exe.config*, em que *application* é o nome de seu aplicativo. Quando seu aplicativo conecta-se ao banco de dados, ele deve ler o valor da string de conexão a partir do arquivo de configuração em vez de utilizar os valores codificado em seu código C#. Veremos como fazer isso ao utilizar as classes de entidade geradas posteriormente neste capítulo.

Você deve instalar o arquivo de configuração do seu aplicativo (o arquivo *aplicativo.exe.config*) com o código executável para o aplicativo. Se precisar se conectar a um banco de dados diferente, você pode editar o arquivo de configuração utilizando um editor de texto e modificar o atributo `<connectionString>` do elemento `<connectionStrings>`. Quando o aplicativo for executado, ele utilizará automaticamente o novo valor.

Esteja ciente de que você deve proteger o arquivo de configuração de aplicativo e impedir que um usuário faça alterações indevidas.

Após criar o modelo de entidade para o aplicativo, você pode construir a interface do usuário para exibir as informações recuperadas, por meio da vinculação de dados.

## Crie a interface com o usuário para o aplicativo Suppliers

1. No *Solution Explorer*, clique com o botão direito do mouse no arquivo *MainWindow.xaml*, clique em *Rename* e atribua um novo nome ao arquivo *SupplierInfo.xaml*.
2. Dê um clique duplo no arquivo *Application.xaml* para exibi-lo na janela *Design View*. No painel *XAML*, altere o elemento *StartupUri* para “*SupplierInfo.xaml*”, como mostrado aqui em negrito:

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

3. No *Solution Explorer*, dê um clique duplo no arquivo *SupplierInfo.xaml* para exibi-lo na janela *Design View*. No painel *XAML*, como mostrado em negrito no seguinte trecho de código, altere o valor do elemento *x:Class* para “*Suppliers.SupplierInfo*”, configure *Title* como “*Supplier Information*”, configure *Height* como “*362*” e *Width* como “*614*”:

```
<Window x:Class="Suppliers.SupplierInfo"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="Supplier Information" Height="362" Width="614">
 ...
</Window>
```

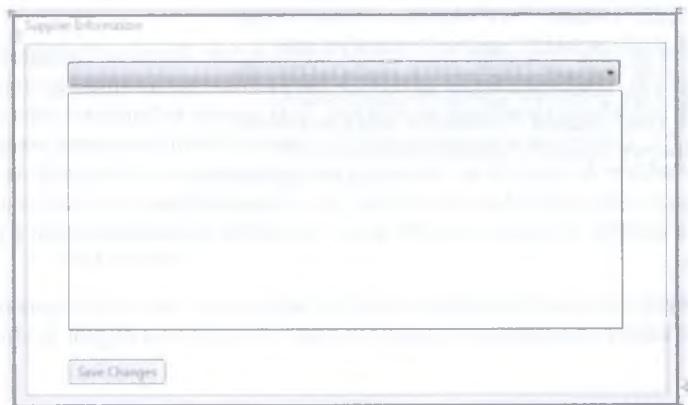
4. Exiba o arquivo *SupplierInfo.xaml.cs* na janela *Code and Text Editor*. Altere o nome da classe *MainWindow* para *SupplierInfo* e mude o nome do construtor, como mostrado a seguir em negrito:

```
public partial class SupplierInfo : Window
{
 public SupplierInfo()
 {
 InitializeComponent();
 }
}
```

5. No *Solution Explorer*, clique duas vezes no arquivo *SupplierInfo.xaml* para exibi-lo na janela *Design View*. Na seção *Common WPF Controls* da *Toolbox*, adicione um controle *ComboBox* e um controle *Button* ao formulário. (Posicione-os em qualquer lugar no formulário.) Na seção *All WPF Controls* da *Toolbox*, adicione um controle *ListView* ao formulário.
6. Utilizando a janela *Properties*, configure as propriedades desses controles com os valores especificados na tabela a seguir.

Controle	Propriedade	Valor
comboBox1	Name	suppliersList
	Height	23
	Width	Auto
	Margin	40,16,42,0
	VerticalAlignment	Top
	HorizontalAlignment	Stretch
listView1	Name	productsList
	Height	Auto
	Width	Auto
	Margin	40,44,40,60
	VerticalAlignment	Stretch
	HorizontalAlignment	Stretch
button1	Name	saveChanges
	Content	Save Changes
	IsEnabled	False (desmarque a caixa de seleção)
	Height	23
	Width	90
	Margin	40,0,0,10
	VerticalAlignment	Bottom
	HorizontalAlignment	Left

O formulário Supplier Information deve se parecer com isso, na janela *Design View*:



7. No painel *XAML*, adicione o elemento *Window.Resources*, mostrado em negrito a seguir, ao elemento *Window*, acima do elemento *Grid*:

```
<Window x:Class="Suppliers.SupplierInfo"
...
<Window.Resources>
 <DataTemplate x:Key="SuppliersTemplate">
 <StackPanel Orientation="Horizontal">
 <TextBlock Text="{Binding Path=SupplierID}" />
 <TextBlock Text=" : " />
 <TextBlock Text="{Binding Path=CompanyName}" />
 <TextBlock Text=" : " />
 <TextBlock Text="{Binding Path=ContactName}" />
 </StackPanel>
 </DataTemplate>
</Window.Resources>
<Grid>
...
</Grid>
</Window>
```

Você pode utilizar um *DataTemplate* para especificar como exibir os dados em um controle. APLICAREMOS esse template à caixa de combinação *suppliersList* no próximo passo. Esse template contém cinco controles *TextBlocks* organizados horizontalmente utilizando um *StackPanel*. O primeiro, o terceiro e o quinto controles *TextBlock* exibirão os dados das propriedades *SupplierID*, *CompanyName* e *ContactName*, respectivamente, do objeto de entidade *Supplier*, com o qual você criará um vínculo posteriormente. Os outros controles *TextBlock* exibem apenas um separador “:”.

8. No painel *XAML*, modifique a definição da caixa de combinação *suppliersList* e especifique as propriedades *IsSynchronizedWithCurrentItem*, *ItemsSource* e *ItemTemplate*, como mostrado em negrito a seguir:

```
<ComboBox ... Name="suppliersList" IsSynchronizedWithCurrentItem="True"
 ItemsSource="{Binding}" ItemTemplate="{StaticResource SuppliersTemplate}" />
```



**Dica** Se preferir, você também pode definir essas propriedades na janela Properties da caixa de combinação *suppliersList*.

Você exibirá os dados de cada fornecedor no controle *suppliersList*. Vimos no Capítulo 25 que a LINQ to SQL utilizava as classes de coleção *Table*<*T*> para armazenar as linhas de uma tabela. O Entity Framework segue uma abordagem semelhante, mas guarda as linhas em uma classe de coleção *ObjectSet*<*T*>. Configurar a propriedade *IsSynchronizedWithCurrentItem* assegura que a propriedade *SelectedItem* do controle se mantenha sincronizada com o item atual da coleção. Se você não configurar essa propriedade como *True*, quando o aplicativo se iniciar e estabelecer a vinculação com a coleção, a caixa de combinação não exibirá automaticamente o primeiro item dessa coleção.

*ItemsSource* atualmente tem uma vinculação vazia. No Capítulo 24, você definiu uma instância de uma classe como um recurso estático e especificou esse recurso como a origem de vinculação.

Se você não especificar uma origem de vinculação, o WPF vinculará a um objeto especificado na propriedade *DataContext* do controle. (Não confunda a propriedade *DataContext* de um controle com um objeto *DataContext* utilizado pela LINQ to SQL para se comunicar com um banco de dados; infelizmente eles têm o mesmo nome.) Você irá configurar a propriedade *DataContext* de um controle como um objeto da coleção *ObjectSet<Supplier>* no código.

A propriedade *ItemTemplate* especifica o template a utilizar para exibir os dados recuperados da origem da vinculação. Nesse caso, o controle *suppliersList* exibirá os campos *SupplierID*, *CompanyName* e *ContactNames*, a partir da origem de vinculação.

9. Modifique a definição da ListView *productsList* e especifique as propriedades *IsSynchronizedWithCurrentItem* e *ItemsSource*:

```
<ListView ... Name="productsList" IsSynchronizedWithCurrentItem="True"
 ItemsSource="{Binding}" />
```

A classe de entidade *Supplier* contém uma propriedade *EntityCollection<Product>* que referencia os produtos que o fornecedor pode oferecer. (A classe *EntityCollection<T>* é muito parecida com a classe *EntitySet<T>* da LINQ to SQL.) Você irá configurar a propriedade *DataContext* do controle *productsList* como a propriedade *Products* do objeto *Supplier* atualmente selecionado no código. Em um próximo exercício, você também fornecerá funcionalidade para permitir que o usuário adicione e remova produtos. Esse código modificará a lista de produtos que atuam como a origem de vinculação. Configurar a propriedade *IsSynchronizedWithCurrentItem* como *True* assegura que o produto recém-criado seja selecionado na lista quando o usuário adicionar um produto novo ou quando um item existente for selecionado se o usuário excluir um (se configurar essa propriedade como *False*, quando você excluir um produto, nenhum item da lista será selecionado subsequentemente, o que pode causar problemas no seu aplicativo se o código tentar acessar o item que estiver atualmente selecionado).

10. Adicione o elemento filho *ListView.View*, mostrado em negrito a seguir, que contém um *GridView* e definições de coluna ao controle *productsList*. Certifique-se de substituir o delimitador de fechamento (*/>*) do elemento *ListView* por um delimitador comum (*>*) e adicionar um elemento *</ListView>* de terminação.

```
<ListView ... Name="productsList" ...>
 <ListView.View>
 <GridView>
 <GridView.Columns>
 <GridViewColumn Width="75" Header="Product ID"
 DisplayMemberBinding="{Binding Path=ProductID}" />
 <GridViewColumn Width="225" Header="Name"
 DisplayMemberBinding="{Binding Path=ProductName}" />
 <GridViewColumn Width="135" Header="Quantity Per Unit"
 DisplayMemberBinding="{Binding Path=QuantityPerUnit}" />
 <GridViewColumn Width="75" Header="Unit Price"
 DisplayMemberBinding="{Binding Path=UnitPrice}" />
 </GridView.Columns>
 </GridView>
 </ListView.View>
</ListView>
```

Você pode fazer um controle *ListView* exibir dados em vários formatos, configurando a propriedade *View*. Esse código Extensible Application Markup Language (XAML) utiliza um componente *GridView*, que exibe dados em um formato tabular. Assim, cada linha na tabela tem um conjunto fixo de colunas definidas pelas propriedades *GridViewColumn* e cada coluna tem seu próprio cabeçalho que exibe o nome da coluna. A propriedade *DisplayMemberBinding* de cada coluna especifica os dados que a coluna deve exibir a partir da origem da vinculação.

Os dados para a coluna *UnitPrice* são uma propriedade *decimal* na classe de entidade *Product*. O WPF converterá essas informações em uma string e aplicará um formato numérico padrão. Idealmente, os dados dessa coluna deveriam ser exibidos como um valor monetário. Todavia, você pode reformatar os dados em uma coluna *GridView* criando uma classe *conversora* a qual já discutimos no Capítulo 24 ao converter uma enumeração em uma string. Dessa vez, a classe conversora converterá um valor *decimal?* em uma *string* contendo uma representação de um valor de moeda.

- Mude para a janela *Code and Text Editor* que exibe o arquivo *SupplierInfo.xaml.cs*. Adicione a classe *PriceConverter*, mostrada a seguir, a esse arquivo, após a classe *SupplierInfo*:

```
[ValueConversion(typeof(string), typeof(Decimal))]
class PriceConverter : IValueConverter
{
 public object Convert(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
 {
 if (value != null)
 return String.Format("{0:C}", value);
 else
 return "";
 }

 public object ConvertBack(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
 {
 throw new NotImplementedException();
 }
}
```

O método *Convert* chama o método *String.Format* para criar uma string que utiliza o formato local de moeda do seu computador. Na verdade, o usuário não modificará o preço unitário na visualização de lista, portanto, não há necessidade de implementar o método *ConvertBack* para converter uma *string* em um valor *decimal*.

- Retorne à janela *Design View* que exibe o formulário *SupplierInfo.xaml*. Adicione a seguinte declaração de namespace XML ao elemento *Window* e defina uma instância da classe *PriceConverter* como um recurso de janela, como mostrado em negrito:

```
<Window x:Class="Suppliers.SupplierInfo"
 xmlns:app="clr-namespace:Suppliers"
 ...>
 <Window.Resources>
 <app:PriceConverter x:Key="priceConverter" />
```

```

 </Window.Resources>

 </Window>
```



**Nota** A janela Design View armazena em cache as definições de controles e outros itens da interface, e nem sempre reconhece imediatamente os novos namespaces adicionados a um formulário. Se essa instrução ocasionar um erro na janela Design View, clique em *Build Solution* no menu *Build*. Esta ação atualiza o cache do WPF e deve eliminar os erros.

13. Modifique a definição de Unit Price *GridViewColumn* e aplique a classe conversora à vinculação, como mostrado em negrito a seguir:

```
<GridViewColumn ... Header ="Unit Price" DisplayMemberBinding=
 "{Binding Path=UnitPrice, Converter={StaticResource priceConverter}}" />
```

Você acabou de criar o layout do formulário. Agora, precisa escrever algum código para recuperar os dados exibidos pelo formulário e configurar as propriedades *DataContext* dos controles *suppliersList* e *productsLists* para que as vinculações funcionem corretamente.

### Escreva um código para recuperar as informações sobre o fornecedor e estabeleça as vinculações de dados

1. No arquivo *SupplierInfo.xaml*, altere a definição do elemento *Window* e adicione um método de evento *Loaded* chamado *Window\_Loaded*. (Esse é o nome padrão desse método, gerado quando você clica em *<New Event Handler>*.) O código XAML para o elemento de janela deve se parecer com isso:

```
<Window x:Class="Suppliers.SupplierInfo"

 Title="Supplier Information" ... Loaded="Window_Loaded">

 </Window>
```

2. Na janela *Code and Text Editor* que exibe a classe *SupplierInfo.xaml.cs*, adicione a seguinte diretiva *using* à lista na parte superior da classe:

```
using System.ComponentModel;
using System.Collections;
```

3. Adicione os três campos privados mostrados em negrito à classe *SupplierInfo*.

```
public partial class SupplierInfo : Window
{
 private NorthwindEntities northwindContext = null;
 private Supplier supplier = null;
 private IList productsInfo = null;

}
```

Você utilizará a variável *northwindContext* para conectar-se ao banco de dados Northwind e recuperar os dados da tabela *Suppliers*. A variável *supplier* contém os dados para o fornecedor

atual exibidos no controle *suppliersList*. A variável *productsInfo* contém os produtos oferecidos pelo fornecedor atualmente exibido. Ela estará vinculada ao controle *productsList*.

Talvez você esteja se perguntando sobre essa definição da variável *productsInfo*; afinal de contas, você aprendeu no exercício anterior que a classe *Supplier* tem uma propriedade *EntityCollection<Product>* que referencia os produtos oferecidos por um fornecedor. Na verdade, você poderia vincular essa propriedade *EntityCollection<Product>* ao controle *productsList*, mas há um problema sério com essa abordagem. Mencionei anteriormente que as classes de entidade *Supplier* e *Product* implementam indiretamente as interfaces *INotifyPropertyChanging* e *INotifyPropertyChanged*, por meio das classes *EntityObject* e *StructuralObject*. Desta forma, quando você vincula um controle WPF a uma fonte de dados, o controle inscreve-se automaticamente nos eventos expostos por essas interfaces a fim de atualizar a exibição quando os dados mudam. Mas a classe *EntityCollection<Product>* não implementa essas interfaces, portanto, o controle *ListView* não será atualizado se quaisquer produtos forem adicionados ou removidos do fornecedor. (Mas será atualizado se um produto existente mudar, porque cada item em *EntityCollection<Product>* é um objeto *Product* que envia as notificações apropriadas aos controles WPF aos quais está vinculado.)

**4.** Adicione o código mostrado em negrito a seguir ao método *Window\_Loaded*:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
 this.northwindContext = new NorthwindEntities();
 suppliersList.DataContext = this.northwindContext.Suppliers;
}
```

Quando o aplicativo se inicia carregando a janela, esse código cria uma variável *NorthwindEntities* que se conecta ao banco de dados Northwind. Lembre-se de que o Entity Data Model Wizard já criou essa classe, cujo construtor padrão lê a string de conexão de banco de dados a partir do arquivo de configuração de aplicativo. O método então configura a propriedade *DataContext* da caixa de combinação *suppliersList* como a propriedade da coleção *ObjectSet* da variável *northwindContext*. Essa ação resolve a vinculação para a caixa de combinação, cujo template de dados utilizado exibe os valores de *SupplierID*, *CompanyName* e *ContactName* para cada objeto *Supplier* na coleção.



**Nota** Se um controle for filho de outro controle, por exemplo, um *GridViewColumn* em um *ListView*, você apenas precisará configurar a propriedade *DataContext* do controle pai. Se essa propriedade de controle filho não estiver configurada, o runtime do WPF usa, em vez disso, o controle pai *DataContext*. Essa técnica torna possível compartilhar um contexto de dados entre vários controles filhos e um controle pai.

Se o controle pai imediato não tiver um contexto de dados, o runtime do WPF examina o controle avô e assim sucessivamente até o controle *Window* que define o formulário. Se nenhum contexto de dados estiver disponível, qualquer vinculação de dados para um controle será ignorada.

5. Retorne à janela *Design View*. Dê um clique duplo na caixa de combinação *suppliersList*. Esta ação criará o método de evento *suppliersList\_SelectionChanged*, que é executado sempre que o usuário seleciona um item diferente na caixa de combinação.
6. Na janela *Code and Text Editor*, adicione as instruções mostradas em negrito a seguir ao método *suppliersList\_SelectionChanged*:

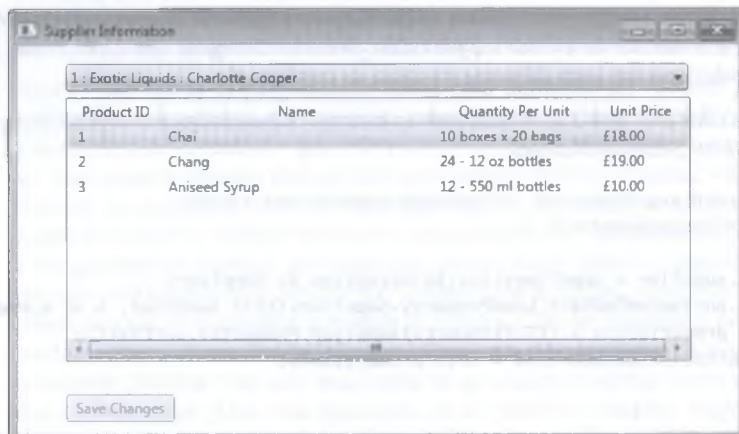
```
private void suppliersList_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
 this.supplier = suppliersList.SelectedItem as Supplier;
 this.northwindContext.LoadProperty<Supplier>(this.supplier, s => s.Products);
 this.productsInfo = ((IListSource)supplier.Products).GetList();
 productsList.DataContext = this.productsInfo;
}
```

Esse método obtém o fornecedor selecionado no momento na caixa de combinação e copia dados de sua propriedade *EntityCollection<Product>* para a variável *productsInfo*. A classe *EntityCollection<Product>* implementa a interface *IListSource*, que fornece o método *GetList* para copiar os dados do conjunto de entidades para um objeto *IList*. Como acontece na LINQ to SQL, os dados relacionados a uma entidade não são automaticamente recuperados quando uma entidade é instanciada. Nesse caso, isso significa que, sempre que um aplicativo buscar os dados de uma entidade *Supplier* no banco de dados, ele não recupera automaticamente os dados dos produtos relacionados a esse fornecedor. No Capítulo 25, vimos que a LINQ to SQL fornece o método *LoadWith* da classe *DataLoadOptions* para especificar os dados relacionados que devem ser recuperados quando uma linha for lida no banco de dados. O Entity Framework fornece o método genérico *LoadProperty<T>* da classe *ObjectContext*, que executa praticamente a mesma tarefa. A segunda instrução no código anterior instrui o Entity Framework a recuperar os produtos associados a um fornecedor sempre que ocorrer uma busca por um fornecedor.

Por último, o código define a propriedade *DataContext* do controle *productsList* com essa lista de produtos. Essa instrução permite que o controle *productsList* exiba os itens na lista de produtos.

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

Quando o formulário executa, ele deve exibir os produtos para o primeiro fornecedor – Exotic Liquids. O formulário deve se parecer com a imagem a seguir.



- Selecione um fornecedor diferente na caixa de combinação e verifique se a visualização de lista exibe os produtos desse fornecedor. Quando você tiver terminado de visualizar os dados, feche o formulário e retorne ao Visual Studio 2010.

## Utilizando a LINQ to Entities para consultar dados

O exercício anterior explorou a vinculação de dados para pesquisar e exibir informações por meio do Entity Framework, em vez de criar e fazer explicitamente consultas LINQ. Entretanto, como mencionado anteriormente, você também pode recuperar informações de um modelo de dados construído sobre o Entity Framework ao utilizar a LINQ to Entities. A sintaxe é semelhante à da LINQ to SQL, mas a principal diferença é que você baseia uma consulta LINQ to Entities em um objeto *ObjectQuery<T>*, em que *T* é um tipo *EntityObject*.

Por exemplo, você pode recuperar uma lista de nomes de produtos do *ObjectSet Products* no *ObjectContext NorthwindEntities*, mostrada nos exemplos anteriores, e exibi-los como a seguir:

```
NorthwindEntities northwindContext = new NorthwindEntities();
ObjectQuery<Product> products = northwindContext.Products;

var productNames = from p in products
 select p.ProductName;

foreach (var name in productNames)
{
 Console.WriteLine("Product name: {0}", name);
}
```

**Nota** Em termos específicos, a propriedade *Products* da classe *NorthwindEntities* tem o tipo *ObjectSet<Products>*. Mas o tipo *ObjectSet<T>* herda de *ObjectQuery<T>*, de modo que é possível atribuir a propriedade *Products* a uma variável *ObjectQuery<Products>*.

Na LINQ to Entities, há suporte para a maioria dos operadores padrão de consulta LINQ, a despeito de algumas exceções descritas na documentação fornecida com o Visual Studio 2010. É possível unir dados de várias tabelas, ordenar os resultados e efetuar operações, como agrupar dados e calcular valores agregados. Para obter mais informações, consulte o Capítulo 20, "Consultando dados na memória utilizando expressões de consulta".

A próxima etapa no aplicativo Suppliers é fornecer a funcionalidade que permite ao usuário modificar os detalhes de produtos, remover produtos e criar novos produtos. Para fazer isso, você precisa aprender a utilizar o Entity Framework para atualizar dados.

## Utilizando vinculação de dados para modificar dados

O Entity Framework fornece um canal de comunicação de duas vias com um banco de dados. Já vimos como empregar vinculação de dados com o Entity Framework para buscar dados, mas você também pode modificar as informações que recuperou e enviar essas alterações de volta ao banco de dados.

### Atualizando dados existentes

Quando você recupera dados por meio de um objeto *ObjectContext*, os objetos criados a partir desses dados são mantidos em um cache de memória dentro do aplicativo. Você pode alterar os valores dos objetos armazenados nesse cache exatamente da mesma maneira como você modifica os valores de um objeto normal qualquer – configurando suas propriedades. Mas atualizar um objeto na memória não atualiza o banco de dados. Assim, para manter as alterações no banco de dados, você precisa gerar os comandos SQL UPDATE apropriados e fazê-los serem executados pelo servidor de banco de dados. Você pode realizar isso facilmente com o Entity Framework. O fragmento de código a seguir mostra uma consulta LINQ to Entities que busca um produto de número 14. Em seguida, o código muda o nome do produto para "Bean Curd" (o produto 14 originalmente tinha o nome "Tofu" no banco de dados Northwind), e então envia a alteração ao banco de dados:

```
NorthwindEntities northwindContext = new NorthwindEntities();
Product product = northwindContext.Products.Single(p => p.ProductID == 14);
product.ProductName = "Bean Curd";
northwindContext.SaveChanges();
```

A instrução-chave nesse exemplo de código é a chamada ao método *SaveChanges* do objeto *ObjectContext*. (Lembre-se de que *NorthwindEntities* herda de *ObjectContext*.) Quando você modifica as informações em um objeto entidade que foi preenchido executando uma consulta, o objeto *ObjectContext*, que gerencia a conexão utilizada para executar a consulta original, monitora as alterações que você faz nos dados. O método *SaveChanges* propaga essas alterações de volta ao banco de dados e, nos bastidores, o objeto *ObjectContext* constrói e executa uma instrução SQL UPDATE.

Se buscar e modificar vários produtos, você precisará chamar *SaveChanges* apenas uma vez, depois da modificação final. O método *SaveChanges* organiza todas as atualizações em lote, e o objeto *ObjectContext* cria uma transação de banco de dados, realizando todas as instruções SQL UPDATE dentro dessa transação. Se uma das atualizações falhar, a transação será abortada, todas as alterações feitas pelo método *SaveChanges* serão revertidas no banco de dados, e o método *SaveChanges* lançará uma exceção. Porém, se todas as atualizações forem bem-sucedidas, a transação será confirmada, e as alterações tornam-se permanentes no banco de dados. Você deve observar que, se o método *SaveChanges* falhar, somente o banco de dados é revertido; suas alterações continuarão presentes nos objetos entidade na memória. A exceção lançada quando o método *SaveChanges* falha fornece algumas informações sobre a razão da falha. Você pode tentar corrigir esse problema e chamar *SaveChanges* novamente.

A classe *ObjectContext* também disponibiliza o método *Refresh*. Com esse método, você pode repopular as coleções *EntityObject* no cache do banco de dados e descartar quaisquer alterações que você fez. Você o utiliza assim:

```
northwindContext.Refresh(RefreshMode.StoreWins, northwindContext.Products);
```

O primeiro parâmetro é um membro da enumeração *System.Data.Objects.RefreshMode*. Especificar o valor *RefreshMode.StoreWins* força os dados a serem atualizados a partir do banco de dados. O segundo parâmetro é a entidade no cache a ser atualizada.

**Dica** Alterar a monitoração é uma operação potencialmente cara para um objeto *ObjectContext* realizar. Se souber que você não modificará dados (por exemplo, se seu aplicativo gerar um relatório de leitura), você poderá desabilitar o rastreamento de mudanças para um objeto *EntityObject*, ao definir a propriedade *MergeOption* com *MergeOption.NoTracking*, como a seguir:

```
northwindContext.Suppliers.MergeOption = MergeOption.NoTracking;
```

Você pode fazer alterações em uma entidade que tem o rastreamento de mudanças desabilitado, mas essas alterações não serão gravadas quando você chamar o *SaveChanges*, e desaparecerão quando o aplicativo for encerrado.

## Tratando atualizações conflitantes

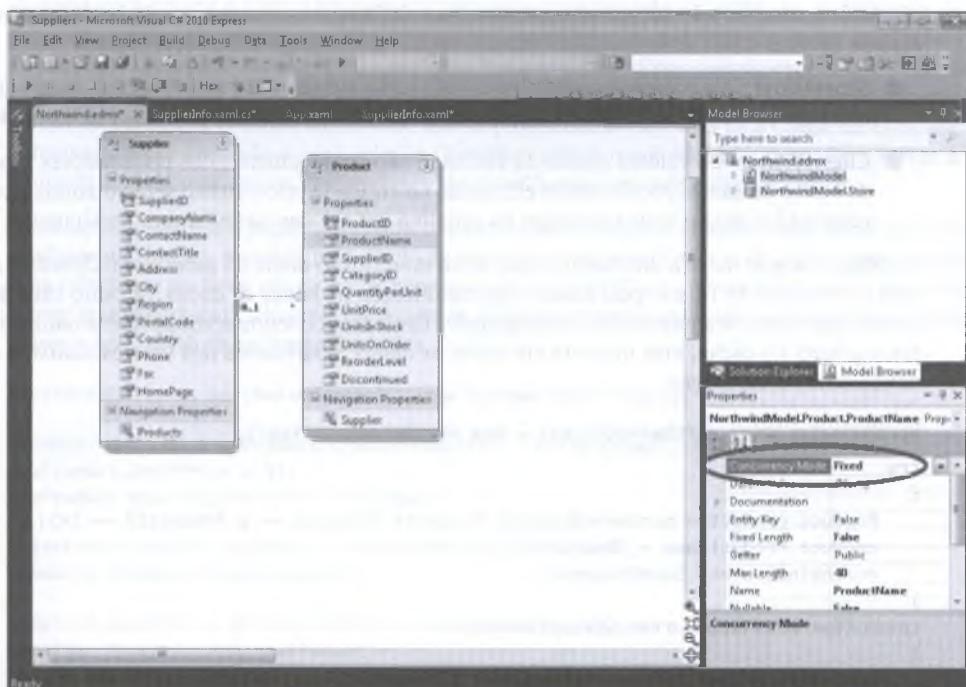
Pode haver várias razões para uma operação de atualização falhar, mas uma das causas mais comuns são conflitos que ocorrem quando dois usuários tentam atualizar os mesmos dados simultaneamente. Se pensar no que acontece quando você executa um aplicativo que utiliza o Entity Framework, você poderá ver que há bastante espaço para conflitos. Ao recuperar dados por meio de um objeto *ObjectContext*, eles são armazenados em cache na memória do seu aplicativo. Um outro usuário poderia realizar a mesma consulta e recuperar os mesmos dados. Se os dois usuários modificarem os dados e então ambos chamarem o método *SaveChanges*, um deles sobrescreverá as alterações feitas pelo outro no banco de dados. Esse fenômeno é conhecido como *perda de atualização*.

Esse fenômeno ocorre porque o Entity Framework implementa uma concorrência otimista. Em outras palavras, ao buscar dados em um banco de dados, ele não bloqueia esses dados no banco de dados.

Essa modalidade de concorrência permite que outros usuários acessem simultaneamente os mesmos dados, mas pressupõe uma baixa probabilidade de dois usuários modificarem os mesmos dados (daí, o termo *concorrência otimista*.)

O contrário da concorrência otimista é a *concorrência pessimista*. Nesse esquema, todos os dados são bloqueados no banco de dados durante a operação de busca, e nenhum outro usuário concorrente poderá acessá-lo. Essa abordagem evita a perda de quaisquer alterações, mas é um pouco extremada.

No Entity Framework, não existe suporte direto para a concorrência pessimista. Em vez disso, ele oferece um meio-termo. Cada item em uma classe *EntityObject* tem uma propriedade chamada *Concurrency Mode*. Por padrão, o *Concurrency Mode* é definido com *None*, mas é possível alterar para *Fixed* por meio do designer do Entity Framework. A imagem a seguir mostra o modelo de entidade que você construiu anteriormente. O usuário clicou no item *ProductName* na entidade *Product* e alterou a propriedade *Concurrency Mode* para *Fixed* na janela *Properties*.



Quando um aplicativo modifica o valor da propriedade *ProductName* em uma instância da classe *EntityObject Products*, o Entity Framework mantém uma cópia do valor original dessa propriedade no cache. Quando você definir o *Concurrency Mode* para uma propriedade, quando o aplicativo chamar o método *SaveChanges* do objeto *ObjectContext*, o Entity Framework usará a cópia armazenada em cache do valor original para verificar se a coluna na linha correspondente no banco de dados foi alterada por outro usuário desde que foi buscada. Se não ocorreu alteração, a linha é atualizada. Se a coluna foi modificada, o método *SaveChanges* vai parar e lançar uma exceção *OptimisticConcur-*

*rencyException*. Quando isso acontecer, todas as alterações efetuadas pelo método *SaveChanges* no banco de dados serão desfeitas, embora as alterações permaneçam no cache em seu aplicativo.

Quando uma exceção *OptimisticConcurrencyException* surgir, você conseguirá saber qual entidade ocasionou o conflito, ao examinar a propriedade *StateEntries* do objeto exceção. Essa propriedade armazena uma coleção de objetos *ObjectStateEntry*. A própria classe *ObjectStateEntry* contém algumas propriedades. As mais importantes são: a propriedade *Entity*, que contém uma referência à entidade que acarretou o conflito; a propriedade *CurrentValues*, que contém os dados modificados da entidade; e a propriedade *OriginalValues*, que contém dados da entidade, recuperados inicialmente do banco de dados.

A abordagem recomendada para resolver conflitos é utilizar o método *Refresh* para recarregar o cache a partir do banco de dados e chamar *SaveChanges* mais uma vez. O método *Refresh* preenche novamente os valores *originais* de uma entidade especificada (passada como o segundo parâmetro) com os valores atualizados do banco de dados. Se o usuário tiver feito uma grande quantidade de alterações, convém não obrigá-lo a rechaveá-las. Felizmente, o parâmetro *RefreshMode* do método *Refresh* permite tratar essa situação. A enumeração *RefreshMode* define dois valores:

- **StoreWins** Os valores *atuais* da entidade serão substituídos pelos valores atualizados, procedentes do banco de dados. As alterações efetuadas na entidade pelo usuário desaparecerão.
- **ClientWins** Os valores *atuais* da entidade não serão substituídos pelos valores procedentes do banco de dados. As alterações efetuadas na entidade pelo usuário serão mantidas no cache e propagadas para o banco de dados na próxima vez em que *SaveChanges* for chamado.

O código a seguir mostra um exemplo que tenta modificar o nome do produto no *ObjectSet Products* com o *ProductID* de 14, e depois salvar essa modificação no banco de dados. Se outro usuário já tiver modificado esses mesmos dados, o manipulador *OptimisticConcurrencyException* atualizará os valores *originais* no cache, mas manterá em cache os dados modificados nos valores *atuais*, e chamará novamente *SaveChanges*.

```
NorthwindEntities northwindContext = new NorthwindEntities();
try
{
 Product product = northwindContext.Products.Single(p => p.ProductID == 14);
 product.ProductName = "Bean Curd";
 northwindContext.SaveChanges();
}
catch (OptimisticConcurrencyException ex)
{
 northwindContext.Refresh(RefreshMode.ClientWins, northwindContext.Products);
 northwindContext.SaveChanges();
}
```

**Importante** Ao detectar o primeiro conflito, Entity Framework para e lança a exceção *OptimisticConcurrencyException*. Se você modificou várias linhas, as chamadas subsequentes ao método *SaveChanges* podem detectar outros conflitos. Além disso, há uma pequena probabilidade de que outro usuário tenha alterado os dados entre as chamadas a *Refresh* e a *SaveChanges* no manipulador de exceções *OptimisticConcurrencyException*. Em um aplicativo comercial, prepare-se também para capturar essa exceção.

## Adicionando e excluindo dados

Além de modificar os dados existentes, o Entity Framework permite que você adicione novos itens a uma coleção *ObjectSet* e remova itens de uma coleção *ObjectSet*.

Quando você utiliza o Entity Framework para gerar um modelo de entidade, a definição de cada entidade contém um método padrão, chamado *CreateXXX* (em que *XXX* é o nome da classe da entidade), que você pode utilizar para criar uma nova entidade. Esse método espera que você forneça parâmetros para cada uma das colunas obrigatórias (não nulas) no banco de dados subjacente. Para definir os valores de colunas adicionais, use as propriedades expostas pela classe da entidade. Para adicionar a nova entidade a uma coleção *ObjectSet*, utilize o método *AddObject*. Para salvar a nova entidade no banco de dados, chame o método *SaveChanges* no objeto *ObjectContext*.

O exemplo de código a seguir gera uma nova entidade *Product* e adiciona essa entidade à lista de produtos na coleção, mantida pelo objeto de contexto *NorthwindEntities*. O código também adiciona uma referência ao novo objeto ao fornecedor cujo *SupplierID* é 1. (O método *Add* é fornecido pelo Entity Framework para ajudar a preservar as relações entre as entidades.) O método *SaveChanges* insere o novo produto no banco de dados.

```
NorthwindEntities northwindContext = new NorthwindEntities();
Product newProduct = Product.CreateProduct(0, "Fried Bread", false);
newProduct.UnitPrice = 55;
newProduct.QuantityPerUnit = "10 boxes";
ObjectSet<Product> products = northwindContext.Products;
products.AddObject(newProduct);
Supplier supplier = northwindContext.Suppliers.Single(s => s.SupplierID == 1);
supplier.Products.Add(newProduct);
northwindContext.SaveChanges();
```

**Nota** Neste exemplo, o primeiro parâmetro para o método *CreateProduct* é o *ProductID*. No banco de dados Northwind, *ProductID* é uma coluna IDENTITY. Quando você chama *SaveChanges*, o SQL Server gera um valor único próprio para essa coluna e descarta o valor que você especificou.

É fácil excluir um objeto entidade de uma coleção *ObjectSet*. Chame o método *DeleteObject* e informe a entidade a ser excluída. O código a seguir exclui todos os produtos que possuem um *ProductID* maior ou igual a 79. Os produtos são removidos do banco de dados quando o método *SaveChanges* é executado.

```
NorthwindEntities northwindContext = new NorthwindEntities();

var productList = from p in northwindContext.Products
 where p.productID >= 79
 select p;

ObjectSet<Product> products = northwindContext.Products;
foreach (var product in productList)
{
 products.DeleteObject(product);
}

northwindContext.SaveChanges();
```

Tenha cuidado ao excluir linhas de tabelas que possuem relações com outras tabelas, porque essas exclusões podem ocasionar erros de integridade referencial quando você atualizar o banco de dados. Por exemplo, no banco de dados Northwind, se você tentar excluir um fornecedor que abastece produtos atualmente, a atualização falhará. Primeiro, remova todos os produtos do fornecedor. Para isso, utilize o método *Remove* da classe *Supplier*. (Como o método *Add*, o método *Remove* também é fornecido pelo Entity Framework.)

Se ocorrer um erro ao salvar as alterações após a inclusão ou exclusão de dados, o método *SaveChanges* lançará uma exceção *UpdateException*. Prepare-se para capturar essa exceção.

Você já tem bastante conhecimento para concluir o aplicativo *Suppliers*.

### Escreva o código para modificar, excluir e criar produtos

1. Retorne ao aplicativo *Suppliers* no Visual Studio 2010 e exiba o arquivo *SupplierInfo.xaml* na janela *Design View*.
2. No painel *XAML*, modifique a definição do controle *productsList* para capturar o evento *KeyDown* e chamar um método de evento chamado *productsList\_KeyDown*. (Esse é o nome padrão do método de evento.)
3. Na janela *Code and Text Editor*, adicione o seguinte código, mostrado em negrito, ao método *productsList\_KeyDown*:

```
private void productsList_KeyDown(object sender, KeyEventArgs e)
{
 switch (e.Key)
 {
 case Key.Enter: editProduct(this.productsList.SelectedItem as Product);
 break;
```

```
 case Key.Insert: addNewProduct();
 break;

 case Key.Delete: deleteProduct(this.productsList.SelectedItem as Product);
 break;
 }
}
```

Esse método examina a tecla pressionada pelo usuário. Se o usuário pressionar a tecla Enter, o código chamará o método *editProduct* e passará os detalhes do produto por parâmetro. Se o usuário pressionar a tecla Insert, o código chamará o método *addNewProduct* para criar e adicionar um novo produto à lista do fornecedor atual, e se o usuário pressionar a tecla Delete, o código chamará o método *deleteProduct* para excluir o produto. Você escreverá os métodos *editProduct*, *addNewProduct* e *deleteProduct* nas próximas etapas.

4. Retorne à janela *Design View*. No painel *XAML*, modifique a definição do controle *productsList* para capturar o evento *MouseDoubleClick* e chamar o método de evento *productsList\_MouseDoubleClick*. (Mais uma vez, esse é o nome padrão do método de evento.)
5. Na janela *Code and Text Editor*, adicione a seguinte instrução, mostrada em negrito, ao método *productsList\_MouseDoubleClick*:

```
private void productsList_MouseDoubleClick(object sender, EventArgs e)
{
 editProduct(this.productsList.SelectedItem as Product);
}
```

Esse método chama apenas o método *editProducts*. É uma conveniência para os usuários, que, obviamente, esperam editar os dados ao clicar duas vezes sobre eles.

6. Adicione o método *deleteProduct* à classe *SupplierInfo*, como a seguir:

```
private void deleteProduct(Product product)
{
 MessageBoxResult response = MessageBox.Show(
 String.Format("Delete {0}", product.ProductName),
 "Confirm", MessageBoxButtons.YesNo, MessageBoxIcon.Question,
 MessageBoxResult.No);
 if (response == MessageBoxResult.Yes)
 {
 this.northwindContext.Products.DeleteObject(product);
 saveChanges.IsEnabled = true;
 }
}
```

Esse método solicita ao usuário que confirme se realmente deseja excluir o produto atualmente selecionado. A instrução *if* chama o método *DeleteObject* da coleção *ObjectSet Products*. Por último, o método ativa o botão *saveChanges*. Em uma etapa posterior, você adicionará funcionalidade a esse botão para enviar as alterações efetuadas na coleção *ObjectSet Products* novamente para o banco de dados.

7. No menu *Project*, clique em *Add Class*. Na caixa de diálogo *Add New Items – Suppliers*, selecione o template *Window (WPF)*, digite **ProductForm.xaml** na caixa *Name* e clique em *Add*.

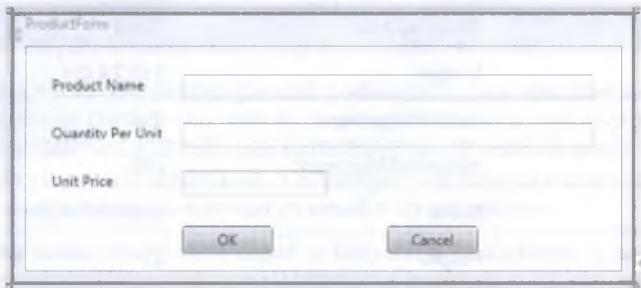
Há várias abordagens que você pode utilizar para adicionar e editar produtos. As colunas no controle *ListView* são itens de texto somente-leitura, mas você pode criar um modo de exibição em lista personalizado, que contenha caixas de texto ou outros controles para permitir a entrada do usuário. Entretanto, a estratégia mais simples é criar outro formulário, que permita ao usuário editar ou adicionar os detalhes de um produto.

8. Na janela *Design View*, clique no formulário *ProductForm*, e, na janela *Properties*, defina a propriedade *ResizeMode* como *NoResize*, a propriedade *Height* como *225*, e a propriedade *Width* como *515*.
9. Adicione três controles *Label*, três controles *TextBox* e dois controles *Button* em qualquer lugar no formulário. Na janela *Properties*, defina as propriedades desses controles com os valores apresentados na tabela a seguir.

Controle	Propriedade	Valor
label1	Content	Product Name
	Height	23
	Width	120
	Margin	17,20,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left
label2	Content	Quantity Per Unit
	Height	23
	Width	120
	Margin	17,60,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left
label3	Content	Unit Price
	Height	23
	Width	120
	Margin	17,100,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left
textBox1	Name	productName
	Height	21
	Width	340

Controle	Propriedade	Valor
	Margin	130,24,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left
textBox2	Name	quantityPerUnit
	Height	21
	Width	340
	Margin	130,64,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left
textBox3	Name	unitPrice
	Height	21
	Width	120
	Margin	130,104,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left
button1	Name	ok
	Content	OK
	Height	23
	Width	75
	Margin	130,150,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left
button2	Name	cancel
	Content	Cancel
	Height	23
	Width	75
	Margin	300,150,0,0
	VerticalAlignment	Top
	HorizontalAlignment	Left

Na janela *Design View*, o formulário deve estar parecido com este:



10. Clique duas vezes no botão *OK* para criar um manipulador de eventos para o evento *click*. Na janela *Code and Text Editor*, que exibe o arquivo *ProductForm.xaml.cs*, adicione o seguinte código, mostrado em negrito:

```

private void ok_Click(object sender, RoutedEventArgs e)
{
 if (String.IsNullOrEmpty(this.productName.Text))
 {
 MessageBox.Show("The product must have a name", "Error",
 MessageBoxButton.OK, MessageBoxIcon.Error);
 return;
 }

 decimal result;
 if (!Decimal.TryParse(this.unitPrice.Text, out result))
 {
 MessageBox.Show("The price must be a valid number", "Error",
 MessageBoxButton.OK, MessageBoxIcon.Error);
 return;
 }

 if (result < 0)
 {
 MessageBox.Show("The price must not be less than zero", "Error",
 MessageBoxButton.OK, MessageBoxIcon.Error);
 return;
 }

 this.DialogResult = true;
}

```

O aplicativo exibirá esse formulário chamando o método *ShowDialog*. Esse método apresenta o formulário como uma caixa de diálogo modal. Quando o usuário clicar em um botão no formulário, este se fechará automaticamente se o código para o evento *click* configurar a propriedade *DialogResult*. Se o usuário clicar em *OK*, esse método realizará uma validação simples das informações inseridas pelo usuário. A coluna *Quantity Per Unit* no banco de dados aceita valores *null*, assim o usuário pode deixar esse campo do formulário vazio. Se o usuário inserir um nome de produto e um preço válidos, o método configura a propriedade *DialogResult* do formulário como *true*. Esse valor é passado de volta para a chamada de método *ShowDialog*.

11. Retorne à janela *Design View* para exibir o arquivo ProductForm.xaml. Selecione o botão *Cancel* e, na janela *Properties*, configure a propriedade *IsCancel* como *true* (marque a caixa de seleção). Se o usuário clicar no botão *Cancel*, ele fechará automaticamente o formulário e retornará um valor *DialogResult* de *false* ao método *ShowDialog*.
12. Mude para a janela *Code and Text Editor* que exibe o arquivo SupplierInfo.xaml.cs. Adicione o método *addNewProduct* mostrado à classe *SupplierInfo*:

```
private void addNewProduct()
{
 ProductForm pf = new ProductForm();
 pf.Title = "New Product for " + supplier.CompanyName;
 if (pf.ShowDialog().Value)
 {
 Product newProd = new Product();
 newProd.ProductName = pf.productName.Text;
 newProd.QuantityPerUnit = pf.quantityPerUnit.Text;
 newProd.UnitPrice = Decimal.Parse(pf.unitPrice.Text);
 this.supplier.Products.Add(newProd);
 this.productsInfo.Add(newProd);
 saveChanges.IsEnabled = true;
 }
}
```

O método *addNewProduct* cria uma nova instância do formulário *ProductForm*, configura a propriedade *Title* desse formulário para conter o nome do fornecedor e então chama o método *ShowDialog* para exibir o formulário como uma caixa de diálogo modal. Se o usuário inserir alguns dados válidos e clicar no botão *OK* do formulário, o código no bloco *if* criará um novo objeto *Product* e o preencherá com as informações da instância *ProductForm*. O método então o adiciona à coleção *Products* para o fornecedor atual e também o adiciona à lista exibida no controle *ListView* do formulário. Por fim, o código ativa o botão *Save Changes*. Em um próximo passo, você adicionará código à rotina de tratamento de evento *click* para esse botão a fim de que o usuário possa salvar alterações no banco de dados.

13. Adicione o método *editProduct* mostrado à classe *SupplierInfo*:

```
private void editProduct(Product product)
{
 ProductForm pf = new ProductForm();
 pf.Title = "Edit Product Details";
 pf.productName.Text = product.ProductName;
 pf.quantityPerUnit.Text = product.QuantityPerUnit;
 pf.unitPrice.Text = product.UnitPrice.ToString();

 if (pf.ShowDialog().Value)
 {
 product.ProductName = pf.productName.Text;
 product.QuantityPerUnit = pf.quantityPerUnit.Text;
 product.UnitPrice = Decimal.Parse(pf.unitPrice.Text);
 saveChanges.IsEnabled = true;
 }
}
```

O método *editProduct* também cria uma instância do formulário *ProductForm*. Desta vez, além de configurar a propriedade *Title*, o código também preenche os campos do formulário com as informações do produto atualmente selecionado. Quando o formulário é exibido, o usuário pode editar esses valores. Se o usuário clicar no botão *OK* para fechar o formulário, o código no bloco *if* copiará os novos valores de volta para o produto atualmente selecionado antes de ativar o botão *Save Changes*. Observe que desta vez você não precisa atualizar o item atual manualmente na lista *productsInfo* porque a classe *Product* notifica automaticamente o *ListView* sobre as alterações nos dados.

14. Retorne à janela *Design View* para exibir o arquivo *SupplierInfo.xaml*. Dê um clique duplo no botão *Save Changes* para criar o método de tratamento para o evento *click*.
15. Na janela *Code and Text Editor*, adicione as seguintes instruções *using* à lista localizada no início do arquivo:

```
using System.Data;
using System.Data.Objects;
```

Esses namespaces contêm vários dos tipos utilizados no Entity Framework.

16. Localize o método *saveChanges\_Click* e adicione o código mostrado em negrito a seguir a esse método:

```
private void saveChanges_Click(object sender, RoutedEventArgs e)
{
 try
 {
 this.northwindContext.SaveChanges();
 saveChanges.IsEnabled = false;
 }
 catch (OptimisticConcurrencyException)
 {
 this.northwindContext.Refresh(RefreshMode.ClientWins,
 northwindContext.Products);
 this.northwindContext.SaveChanges();
 }
 catch (UpdateException uEx)
 {
 MessageBox.Show(uEx.InnerException.Message, "Error saving changes");
 this.northwindContext.Refresh(RefreshMode.StoreWins,
 northwindContext.Products);
 }
 catch (Exception ex)
 {
 MessageBox.Show(ex.Message, "Error saving changes");
 this.northwindContext.Refresh(RefreshMode.StoreWins,
 northwindContext.Products);
 }
}
```

Esse método chama o método *SaveChanges* do objeto *ObjectContext* para enviar todas as alterações novamente para o banco de dados. Os manipuladores de exceção capturam as exceções ocorridas. O manipulador *OptimisticConcurrencyException* usa a estratégia descrita anteriormente para renovar o cache e salvar novamente as alterações. O manipulador *UpdateException* informa o erro ao usuário e atualiza o cache a partir do banco de dados, ao especificar o parâmetro *RefreshMode.StoreWins*. (Isso causa o descarte das alterações efetuadas pelo usuário.) Observe que a maioria dos dados significativos para essa exceção é mantida na propriedade

*InnerException* da exceção (embora não seja conveniente exibir esse tipo de informação para o usuário!). Se ocorrer qualquer outro tipo de exceção, o manipulador *Exception* exibirá uma mensagem simples e atualizará o cache a partir do banco de dados.

## Teste o aplicativo Suppliers

1. No menu *Debug*, clique em *Start Without Debugging* para compilar e executar o aplicativo. Quando o formulário aparecer exibindo os produtos fornecidos pela Exotic Liquids, clique no produto 3 (Aniseed Syrup) e então pressione Enter ou clique duas vezes na linha. O formulário *Edit Product Details* deve aparecer. Altere o valor no campo *Unit Price* para **12.5** e então clique em *OK*. Verifique se o novo preço é copiado de volta para o controle *ListView*.
2. Pressione a tecla *Insert*. O formulário *New Product for Exotic Liquids* deve aparecer. Insira um nome de produto, quantidade por unidade e preço e então clique em *OK*. Verifique se o novo produto é adicionado ao controle *ListView*.

O valor na coluna *Product ID* deve ser 0. Esse valor é uma coluna *identity* no banco de dados, portanto, o SQL Server irá gerar um valor único próprio para essa coluna quando você salvar as alterações.

3. Clique em *Save Changes*. Depois que os dados são salvos, o ID para o novo produto é exibido no controle *ListView*.
4. Clique no novo produto e, então, pressione a tecla *Delete*. Na caixa de diálogo *Confirm*, clique em *Yes*. Verifique se o produto desaparece do formulário. Clique em *Save Changes* novamente e verifique se a operação é finalizada sem nenhum erro.

Sinta-se livre para fazer testes adicionando, removendo e editando produtos de outros fornecedores. Você pode fazer várias modificações antes de clicar em *Save Changes* – o método *SubmitChanges* salva todas as alterações feitas desde quando os dados foram recuperados ou salvos pela última vez.



**Dica** Se você excluir ou sobrescrever acidentalmente os dados de um produto que você quer manter, feche o aplicativo sem clicar em *Save Changes*. Observe que o aplicativo da forma que está escrito não adverte o usuário se ele tentar fechar sem antes salvar as alterações.

De outra forma, você pode adicionar um botão *Discard Changes* ao aplicativo que chama o método *Refresh* do objeto *northwindContext ObjectContext* para repreencher as tabelas do banco de dados, como mostrado nos manipuladores de exceção no exercício anterior.

5. Feche o formulário e retorne ao Visual Studio 2010.

Neste capítulo, você aprendeu a utilizar o Entity Framework para gerar um modelo de entidade para o banco de dados. Você viu como utilizar o modelo de entidade de um aplicativo WPF, ao vincular controles a coleções de entidades. Você também examinou como utilizar a LINQ to Entities para acessar dados por meio de um modelo de entidade.

- Se quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 27.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 26

Para	Faça isto
Criar classes de entidade utilizando o Entity Framework	Adicione uma nova classe ao projeto utilizando o template. Use o Entity Model Wizard para conectar-se ao banco de dados que contém as tabelas que você quer modificar e selecione as tabelas que você precisar.
Exibir dados de um objeto de entidade ou coleção em um controle WPF	Defina uma vinculação com a propriedade apropriada do controle. Se o controle exibir uma lista de objetos, configure a propriedade <i>DataContext</i> do controle como uma coleção de objetos de entidade. Se o controle exibir os dados para um único objeto, configure a propriedade <i>DataContext</i> do controle como um objeto de entidade e especifique a propriedade do objeto de entidade a exibir no atributo <i>Path</i> da vinculação.
Modificar as informações em um banco de dados utilizando o Entity Framework	Primeiro siga um dos seguintes procedimentos: <ul style="list-style-type: none"> <li>■ Para atualizar uma linha em uma tabela no banco de dados, busque os dados para a linha em um objeto de entidade e atribua os novos valores às propriedades adequadas do objeto de entidade.</li> <li>■ Para inserir uma nova linha em uma tabela no banco de dados, crie uma nova instância do objeto de entidade correspondente, por meio do método padrão <i>CreateXXX</i> gerado para essa classe de entidade (em que XXX é o nome da entidade), configure suas propriedades e então chame o método <i>AddObject</i> da coleção <i>ObjectSet</i> apropriada, especificando o novo objeto de entidade como parâmetro.</li> <li>■ Para remover uma linha de uma tabela no banco de dados, chame o método <i>DeleteObject</i> da coleção <i>ObjectSet</i> apropriada, especificando como parâmetro o objeto de entidade a ser removido.</li> </ul> Em seguida, depois de fazer todas as suas alterações, chame o método <i>SaveChanges</i> do objeto <i>ObjectContext</i> para propagar as modificações para o banco de dados.
Tratar dos conflitos ao atualizar um banco de dados utilizando o Entity Framework	Forneça uma rotina de tratamento para a exceção <i>OptimisticConcurrencyException</i> . Na rotina de tratamento de exceção, chame o método <i>Refresh</i> do objeto <i>ObjectContext</i> para recuperar os dados mais recentes no banco de dados para os valores <i>originais</i> em cache, e chame <i>SaveChanges</i> novamente.

## Parte VI

# Construindo soluções profissionais com o Visual Studio 2010

Capítulo 27	Introdução à Task Parallel Library .....	631
Capítulo 28	Realizando acesso a dados em paralelo. ....	681
Capítulo 29	Criando e utilizando um Web Service .....	715

## Capítulo 27

# Introdução à Task Parallel Library

Neste capítulo, você vai aprender a:

- Descrever os benefícios propiciados pela implementação de operações paralelas em um aplicativo.
- Explicar como a Task Parallel Library oferece uma plataforma ideal para implementar aplicativos que podem se beneficiar de vários núcleos do processador.
- Utilizar a classe *Task* para criar e executar operações paralelas em um aplicativo.
- Utilizar a classe *Parallel* para paralelizar algumas construções de programação comuns.
- Utilizar as tarefas com threads para melhorar a capacidade de resposta e a taxa de transferência em aplicativos com interface gráfica do usuário (GUI).
- Cancelar as tarefas de execução demorada e tratar das exceções levantadas pelas operações paralelas.

Já vimos como utilizar o Microsoft Visual C# para construir aplicativos que oferecem uma interface gráfica do usuário e que podem gerenciar os dados armazenados em um banco de dados. Esses são recursos comuns da maioria dos sistemas modernos. Entretanto, como o avanço tecnológico, as exigências dos usuários também evoluíram, e os aplicativos que lhes permitem executar suas operações cotidianas precisam fornecer soluções cada vez mais sofisticadas. Na última parte deste livro, você examinará alguns recursos avançados, lançados com o .NET Framework 4.0. Você verá neste capítulo principalmente como melhorar a simultaneidade (ou “concorrência”) em um aplicativo, ao utilizar a Task Parallel Library. No próximo capítulo, você verá que as extensões paralelas fornecidas no .NET Framework podem ser utilizadas em conjunto com a Language Integrated Query (LINQ) para melhorar a taxa de transferência das operações de acesso a dados. E, no último capítulo, você vai se deparar com o Windows Communication Foundation para construir soluções distribuídas que podem incorporar serviços em execução em vários computadores. Como um bônus, o apêndice (fornecido no CD) descreve como utilizar o Dynamic Language Runtime para construir aplicativos e componentes em C# que podem interoperação com os serviços construídos com outras linguagens que operam fora da estrutura fornecida pelo .NET Framework, como Python e Ruby.

De modo geral, nos capítulos anteriores deste livro, 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. Nem sempre essa será a abordagem mais eficiente para um aplicativo. Por exemplo, você aprendeu no Capítulo 23, “Obtendo a entrada do usuário”, que enquanto espera que um usuário clique em um botão em um formulário WPF (Windows Presentation Foundation), seu programa pode executar outro trabalho simultaneamente. Entretanto, se um programa *single-threaded* precisar efetuar um cálculo demorado, que usa intensamente o processador, ele não poderá responder ao usuário que estiver digitando dados em um formulário ou clicando em um item do menu. O usuário terá a impressão de que o aplicativo está

congelado. Somente após o término do cálculo, a interface do usuário começará a responder novamente. Os aplicativos que executam várias tarefas simultaneamente podem utilizar de modo mais eficiente os recursos disponíveis em um computador, podem executar mais rapidamente, e podem ser mais ágeis. Além disso, algumas tarefas individuais podem ser executadas com mais velocidade, se for possível dividi-las em caminhos paralelos de execução simultânea. No Capítulo 23, você viu que o WPF pode tirar proveito das threads para melhorar a capacidade de resposta em uma interface gráfica do usuário. Neste capítulo, você aprenderá a utilizar a Task Parallel Library para implementar uma forma mais genérica de multitarefa em seus programas, empregável em aplicativos que fazem muitos cálculos, e não somente àqueles relacionados ao gerenciamento de interfaces do usuário.

## Por que fazer multitarefa por meio de processamento paralelo

Como mencionado na introdução, há dois motivos principais pelos quais você empregaria multitarefa em um aplicativo:

- **Melhorar a capacidade 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, não é a verdadeira multitarefa, uma vez que o processador é compartilhado entre as threads, e a natureza cooperativa dessa abordagem 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 abordagem deixarão de existir. Às vezes, é difícil escrever aplicativos bem comportados, que sigam esse modelo de modo consistente.
- **Melhorar 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 tarefas que podem ser executadas em simultaneidade. Até bem recentemente, 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 Compute (HPC) Server 2008, que permite a uma empresa 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 dispendiosas para os sistemas desktop, de menor escala.

Considerando essas descrições, talvez você esteja propenso a concluir que o modo mais econômico de construir soluções multitarefa para aplicativos desktop é utilizar a abordagem de várias threads cooperativas. Entretanto, essa abordagem foi tão somente elaborada como um mecanismo para agilizar – para que os computadores com um único processador garantissem que cada tarefa recebesse uma fatia justa do processador; não é adequada às máquinas de múltiplos processadores porque não foi projetada para distribuir a carga pelos processadores e, consequentemente, não é eficiente com o aumento da escala. Na época em que as máquinas desktop com diversos processadores eram dispensáveis (e, por conseguinte, relativamente raras), isso não era um problema. Contudo, essa situação mudou, como explicaremos resumidamente.

## O surgimento do processador multinúcleo

Dez anos atrás, o preço de um bom computador pessoal estava na faixa de 500 a 1000 dólares. Atualmente, um computador pessoal aceitável ainda custa praticamente a mesma coisa, mesmo depois de 10 anos de inflação. Hoje, a especificação de um PC comum provavelmente inclui um processador com uma velocidade entre 2 e 3 GHz, 500 GB de espaço de armazenamento em disco rígido, 4 GB de RAM, gráficos de alta resolução e alta velocidade, e uma unidade de gravação de DVD. Há 10 anos, a velocidade do processador em uma máquina comum estava entre 500 MHz e 1 GHz, o um disco rígido tinha 80 GB, o Windows rodava plenamente 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 a miniaturização cada vez maior dos componentes permitia a incorporação de uma quantidade crescente 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, e dobrará a cada dois anos. (Na realidade, inicialmente, 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 mais rapidamente. Isso significava que estava previsto que os fabricantes de chips produziriam microprocessadores mais velozes e mais poderosos, praticamente de modo ininterrupto, e 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 40 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 sua pouca 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 seis anos, um processador veloz executava a 3 GHz. Atualmente, 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. Como consequência, a maioria dos processadores modernos já possui dois ou mais *núcleos de processamento*. 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 dual-core (dois núcleos) e quad-core (quarto núcleos) já são lugar-comum. Existem chips com 8, 16, 32 e 64 núcleos, e espera-se que o preço desses chips caia ainda mais daqui em diante. Sendo 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, era possível agilizar 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. 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 em estado de espera para trabalhar. Para fazer o uso mais eficiente dos processadores multicore, escreva seus aplicativos de modo a tirar proveito da multitarefa.

## Implementando multitarefa em um aplicativo

Multitarefa é a possibilidade de executar mais tarefas ao mesmo tempo, e representa um daqueles conceitos de fácil descrição, mas que, até recentemente, 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 a simultaneidade, 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 esteja executando apenas a quantidade de operações correspondente ao número de processadores disponíveis?
- Se uma operação estiver bloqueada (por exemplo, enquanto aguarda o término da E/S), 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?
- Como sincronizar o acesso aos dados compartilhados para garantir que duas ou mais operações simultâneas não danifiquem mutuamente seus dados de modo inadvertido?

Para um desenvolvedor de aplicativos, a primeira pergunta é uma questão de design. As demais perguntas dependem da infraestrutura de programação – para ajudar a dirimir essas dúvidas, a Microsoft disponibiliza a Task Parallel Library (TPL).

No Capítulo 28, “Realizando o acesso de dados paralelo”, você verá como alguns problemas baseados em consultas (queries) têm soluções naturalmente paralelas, e como utilizar o tipo da *ParallelEnumerable* da LINQ para paralelizar operações de consulta. Entretanto, ocasionalmente, é necessária uma abordagem mais imperativa para as situações mais gerais. A TPL contém uma série de tipos e operações que permitem especificar mais diretamente como você pretende dividir um aplicativo em um conjunto de tarefas paralelas.

## Tarefas, threads e *ThreadPool*

O tipo mais importante na TPL é a classe *Task*, que 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 como uma referência a um processador de um só núcleo quanto como a um único núcleo de processamento em um processador multinúcleo.

Internamente, a TPL implementa tarefas e agenda a sua execução por meio de objetos *Thread* e da classe *ThreadPool*. O multithreading e os pools de thread estão disponíveis no .NET Framework desde a versão 1.0, e você pode utilizar a classe *Thread* do namespace *System.Threading* diretamente em seu código. Contudo, a TPL disponibiliza um nível adicional de abstração para 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, geralmente esses itens são idênticos. Entretanto, em um computador com vários processadores ou com um processador multinúcleo (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.

A TPL 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. A TPL

utiliza um conjunto de threads fornecido pelo .NET Framework, chamado *ThreadPool*, e implementa um mecanismo de filas para distribuir a carga de trabalho por essas threads. 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 executada por essa thread. O *ThreadPool* implementa algumas otimizações e usa um algoritmo de “roubo de trabalho” para garantir que as threads sejam escalonadas de maneira eficiente.

 **Nota** O *ThreadPool* já estava disponível nas edições anteriores do .NET Framework, mas foi otimizado de modo significado no .NET Framework 4.0 para suportar as *Tasks*.

Observe que o número de threads criadas pelo .NET Framework para manipular 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. Consequentemente, 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 pode 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 .NET Framework segue uma estratégia iterativa, conhecida como algoritmo *hill-climbing*, para determinar dinamicamente o número perfeito de threads para a carga de trabalho atual.

O que mais importa é que tudo o que você precisa fazer em seu código é dividir seu aplicativo em tarefas, que podem ser executadas simultaneamente. O .NET Framework 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 dividir seu trabalho em muitas tarefas, porque o .NET Framework 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.

## Criando, executando e controlando tarefas

O objeto *Task* e os outros tipos existentes na TPL residem no namespace *System.Threading.Tasks*. Para criar objetos *Task*, utilize o construtor *Task*. Esse construtor é sobre carregado, mas todas as versões esperam que você forneça um delegate *Action* como parâmetro. Vimos no Capítulo 23 que um delegate *Action* faz referência a um método que não retorna um valor. Um objeto *task* utiliza esse delegate para executar um método quando ele está agendado. O exemplo a seguir gera um objeto *Task* que usa um delegate para executar o método chamado *doWork* (você também pode utilizar um método anônimo ou uma expressão lambda, como mostrado pelo código nos comentários):

```

Task task = new Task(new Action(doWork));
// Task task = new Task(delegate { this.doWork(); });
// Task task = new Task(() => { this.doWork(); });

private void doWork()
{
 // A tarefa executa este código quando iniciada

 ...
}

```

**Nota** Em muitos casos, você pode deixar o compilador deduzir o tipo de delegate *Action* e especificar o método a ser executado. Por exemplo, você pode reformular o primeiro exemplo, que acabamos de apresentar, como demonstrado a seguir:

```
Task task = new Task(doWork);
```

As regras de inferência de delegates implementadas pelo compilador se aplicam não somente ao tipo *Action*, como também a qualquer local em que um delegate possa ser utilizado. Até o final deste livro, você encontrará muitos outros exemplos.

O tipo padrão *Action* faz referência a um método que não aceita quaisquer parâmetros. Outras sobrecargas do construtor *Task* aceitam um parâmetro *Action<object>* que representa um delegate que faz referência a um método que aceita um único parâmetro *object*. Essas sobrecargas permitem passar dados para a execução do método 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 sobreescrito, e é possível especificar opcionalmente um objeto *TaskScheduler* para controlar o nível de simultaneidade e outras opções de agendamento (scheduling). É recomendável que você utilize o objeto padrão *TaskScheduler*, incorporado ao .NET Framework, ou você pode definir uma classe *TaskScheduler* personalizada, para ter mais controle sobre o modo de

enfileiramento e agendamento das tarefas. Os detalhes de como fazer isso estão além do escopo desse livro, mas para obter outras informações, consulte a descrição da classe abstrata *TaskScheduler*, na documentação da Class Library do .NET Framework fornecida com o Visual Studio.

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 ao *TaskScheduler* padrão dicas sobre como agendar e executar a tarefa se você especificar um valor da enumeração *TaskCreationOptions* no construtor de *Task*. Para obter mais informações sobre a enumeração *TaskCreationOptions*, consulte a documentação que descreve a Class Library do .NET Framework fornecida com o Visual Studio.

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.

Normalmente, o scheduler procura executar as tarefas em paralelo, sempre que possível, mas você também pode agendá-las em sequência ao criar uma *continuação*. Para criar uma continuação, 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*.

---

\* N. de R. T.: Agendador de tarefas simultâneas.

tions. O tipo *TaskContinuationOptions* é uma enumeração que contém um superconjunto de valores da enumeração *TaskCreationOptions*. Os valores adicionais disponíveis são:

- *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 “Cancelando tarefas e tratando 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 “Cancelando tarefas e tratando 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();
```

Se você costuma utilizar o mesmo conjunto de valores para *TaskCreationOptions* e o mesmo objeto *TaskScheduler*, poderá usar um objeto *TaskFactory* para criar e executar uma tarefa em uma única etapa. O construtor da classe *TaskFactory* permite especificar o agendador de tarefas (scheduler), opções de criação e de continuação de tarefas que as tarefas construídas por essa fábrica devem utilizar. A classe *TaskFactory* fornece o método *StartNew* para criar e executar um objeto *Task*. Assim como o método *Start* da classe *Task*, o método *StartNew* é sobre carregado, mas todos eles esperam uma referência a um método que a tarefa deve executar.

O código a seguir mostra um exemplo que cria e executa duas tarefas ao utilizar a mesma fábrica de tarefas:

```
TaskScheduler scheduler = TaskScheduler.Current;
TaskFactory taskFactory = new TaskFactory(scheduler, TaskCreationOptions.None,
 TaskContinuationOptions.NotOnFaulted);
Task task = taskFactory.StartNew(doWork);
Task task2 = taskFactory.StartNew(doMoreWork);
```

Mesmo que você não determine, no momento, quaisquer opções específicas de criação de tarefas e utilize o agendador de tarefas padrão, você também deve considerar o uso de um objeto *TaskFactory*; esse objeto garante consistência, e haverá menos código a ser modificado para garantir a execução de todas as tarefas do mesmo modo, caso seja necessário personalizar esse processo posteriormente. A classe *Task* expõe a *TaskFactory* padrão utilizada pela TPL por meio da propriedade estática *Factory*. Você pode utilizá-la como a seguir:

```
Task task = Task.Factory.StartNew(doWork);
```

Uma exigência comum dos aplicativos que chamam operações simultaneamente é sincronizar as tarefas. A classe *Task* disponibiliza o método *Wait*, que implementa um método simples de coordenação de tarefas. Ele permite suspender a execução da thread atual, até que a tarefa especificada seja concluída, como a seguir:

```
task2.Wait(); // Espere nesse ponto até que a task2 termine
```

Você pode esperar um conjunto de tarefas, ao utilizar os métodos estáticos *WaitAll* e *WaitAny* da classe *Task*. Os dois métodos aceitam um array *params* que contém um conjunto de objetos *Task*. O método *WaitAll* aguarda o término de todas as tarefas especificadas, e *WaitAny* para até que pelo menos uma das tarefas especificadas tenha terminado. Você os utiliza da seguinte maneira:

```
Task.WaitAll(task, task2); // Aguarda o término das duas tarefas (task e task2)
Task.WaitAny(task, task2); // Aguarda o término de uma das tarefas
```

## Utilizando a classe Task para implementar paralelismo

No próximo exercício, você utilizará a classe *Task* para parallelizar 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*, compreende um formulário WPF 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 devem ser executados em um computador equipado com um processador multinúcleo (multicore). Se você tiver apenas uma CPU de um único núcleo, não observará os mesmos efeitos. Além disso, você não deve inicializar quaisquer 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 2010 se ele ainda não estiver em execução.
2. Abra a solução GraphDemo, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 27\GraphDemo de sua pasta Documents.
3. No Solution Explorer, no projeto GraphDemo, clique duas vezes no arquivo GraphWindow.xaml para exibir o formulário na janela *Design View*.

O formulário contém os seguintes controles:

- 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*.
- Um controle *Label* chamado *duration*. O aplicativo exibe o tempo necessário para gerar e processar os dados do gráfico nesse rótulo.

4. No Solution Explorer, expanda o 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 *System.Windows.Media.Imaging.WriteableBitmap*, chamado *graphBitmap*, para processar o gráfico. As variáveis *pixelWidth* e *pixelHeight* especificam a resolução horizontal e vertical, respectivamente, do objeto *WriteableBitmap*; as variáveis *dpiX* e *dpiY* especificam a densidade horizontal e vertical, respectivamente, da imagem, em pontos por polegada:

```
public partial class GraphWindow : Window
{
 private static long availableMemorySize = 0;
 private int pixelWidth = 0;
 private int pixelHeight = 0;
 private double dpiX = 96.0;
 private double dpiY = 96.0;
 private WriteableBitmap graphBitmap = null;

}
```

5. Examine o construtor *GraphWindow*. Ele é semelhante ao seguinte:

```
public GraphWindow()
{
 InitializeComponent();

 PerformanceCounter memCounter = new PerformanceCounter("Memory", "Available
Bytes");
 availableMemorySize = Convert.ToInt64(memCounter.NextValue());

 this.pixelWidth = (int)availablePhysicalMemory / 20000;
 if (this.pixelWidth < 0 || this.pixelWidth > 15000)
 this.pixelWidth = 15000;
```

```

 this.pixelHeight = (int)availablePhysicalMemory / 40000;
 if (this.pixelHeight < 0 || this.pixelHeight > 7500)
 this.pixelHeight = 7500;
 }

```

Para evitar um código que esgota a memória disponível no computador e gera exceções *OutOfMemory*, esse aplicativo cria um objeto *PerformanceCounter* para consultar a quantidade de memória física disponível no computador. Em seguida, ele utiliza essas informações para calcular os valores adequados para as variáveis *pixelWidth* e *pixelHeight*. Quanto maior a quantidade de memória disponível no computador, tanto mais altos serão os valores gerados para *pixelWidth* e *pixelHeight* (sujeitos aos limites de 15.000 e 7.500 para cada uma dessas variáveis, respectivamente) e tanto mais abrangentes serão os benefícios do uso da TPL no decorrer dos exercícios deste capítulo. Entretanto, se você detectar que o aplicativo ainda gera exceções *OutOfMemory*, aumente os divisores (20.000 e 40.000) utilizados para gerar os valores de *pixelWidth* e *pixelHeight*, referentes à largura e à altura, respectivamente.

Se existir muita memória, os valores calculados para *pixelWidth* e *pixelHeight* poderão estourar. Nesse caso, eles conterão valores negativos e o aplicativo falhará com uma exceção mais adiante. O código do construtor verifica essa condição e define os campos *pixelWidth* e *pixelHeight* com um par de valores úteis, que permitem ao aplicativo ser executado corretamente nessa situação.

#### 6. Examine o código do método *plotButton\_Click*:

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{
 if (graphBitmap == null)
 {
 graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight, dpiX, dpiY,
PixelFormats.Gray8, null);
 }
 int bytesPerPixel = (graphBitmap.Format.BitsPerPixel + 7) / 8;
 int stride = bytesPerPixel * graphBitmap.PixelWidth;
 int dataSize = stride * graphBitmap.PixelHeight;
 byte [] data = new byte[dataSize];

 Stopwatch watch = Stopwatch.StartNew();
 generateGraphData(data);

 duration.Content = string.Format("Duration (ms): {0}",
watch.ElapsedMilliseconds);
 graphBitmap.WritePixels(
 new Int32Rect(0, 0, graphBitmap.PixelWidth, graphBitmap.PixelHeight),
 data, stride, 0);
 graphImage.Source = graphBitmap;
}

```

Este método é executado quando o usuário clica no botão *plotButton*. O código instancia o objeto *graphBitmap*, se ele ainda não tiver sido criado pelo usuário que está clicando no botão *plotButton*, e especifica que cada pixel representa um tom de cinza, com 8 bits por pixel. Este método utiliza as seguintes variáveis e métodos:

- A variável *bytesPerPixel* calcula o número de bytes necessários para armazenar cada pixel. (O tipo *WriteableBitmap* tem suporte para diversos formatos de pixel, com até 128 bits por pixel para imagens de quatro cores.)
- A variável *stride* contém a distância vertical, em bytes, entre pixels adjacentes no objeto *WriteableBitmap*.
- A variável *dataSize* calcula o número de bytes necessários para armazenar os dados para o objeto *WriteableBitmap*. Essa variável é utilizada para inicializar o array *data* com o tamanho adequado.
- O array de bytes *data* armazena os dados do gráfico.
- 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 a instancia em 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 *WriteableBitmap*. Você examinará esse método na próxima etapa.
- O método *WritePixels* da classe *WriteableBitmap* copia os dados de um array de bytes para um bitmap, para o respectivo processamento. Esse método aceita um parâmetro *Int32Rect* que especifica a área no objeto *WriteableBitmap* a ser preenchida, os dados a serem utilizados para copiar o objeto *WriteableBitmap*, a distância vertical entre os pixels adjacentes no objeto *WriteableBitmap*, e um deslocamento para o objeto *WriteableBitmap*, no qual os dados devem começar a ser escritos.

**Nota** Você pode utilizar o método *WritePixels* para substituir seletivamente as informações contidas em um objeto *WriteableBitmap*. Neste exemplo, o código substitui o conteúdo inteiro. Para obter mais informações sobre a classe *WriteableBitmap*, consulte a documentação da Class Library do .NET Framework instalada com o Visual Studio 2010.

- A propriedade *Source* de um controle *Image* especifica os dados que o controle *Image* deve processar. Este exemplo define a propriedade *Source* com o objeto *WriteableBitmap*.

7. Examine o código do método *generateGraphData*:

```
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 vez para o valor negativo.

8. Examine o método *plotXY*:

```
private void plotXY(byte[] data, int x, int y)
{
 data[x + y * pixelWidth] = 0xFF;
}
```

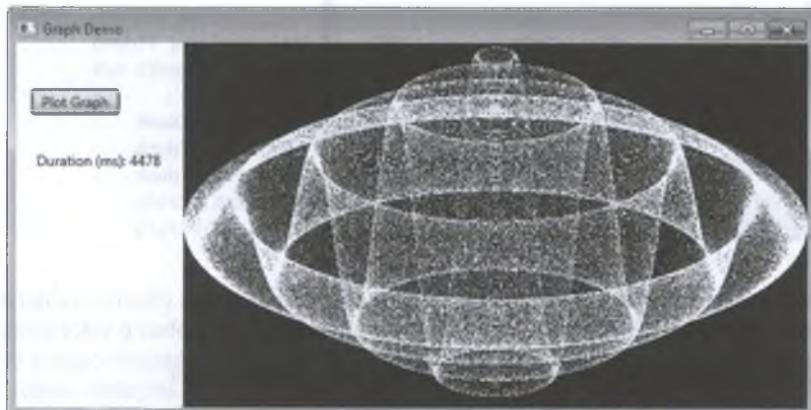
Trata-se de um método simples que define o byte adequado no array *data* correspondente às coordenadas X e Y passadas como parâmetros. O valor 0xFF indica que o pixel correspondente deve ser definido com branco quando o gráfico for processado. Os pixels sem definição serão exibidos na cor preta.

9. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.  
 10. Quando a janela *Graph Demo* for exibida, clique em *Plot Graph* e aguarde.

Seja paciente. São necessários vários segundos para que o aplicativo gere e exiba o gráfico. A imagem a seguir mostra o gráfico. Observe o valor no rótulo *Duration (ms)* na figura a seguir. Nesse caso, são necessários 4478 milissegundos (ms) para o aplicativo plotar o gráfico.



**Nota** O aplicativo foi executado em um computador com 2 GB de memória e equipado com um processador Intel® Core 2 Duo Desktop E6600 executando a 2.40 GHz. Os tempos em seu sistema podem variar com outro processador ou outra quantidade de memória. Além disso, talvez você tenha a impressão de que demora mais tempo do que aquele informado inicialmente para exibir o gráfico. Isso acontece por causa do tempo necessário para inicializar as estruturas dos dados exigidas para exibir efetivamente o gráfico, como parte do método *WritePixels* do controle *graphBitmap*, e não devido ao tempo necessário para calcular os dados do gráfico. As execuções subsequentes não apresentarão essa demora adicional.

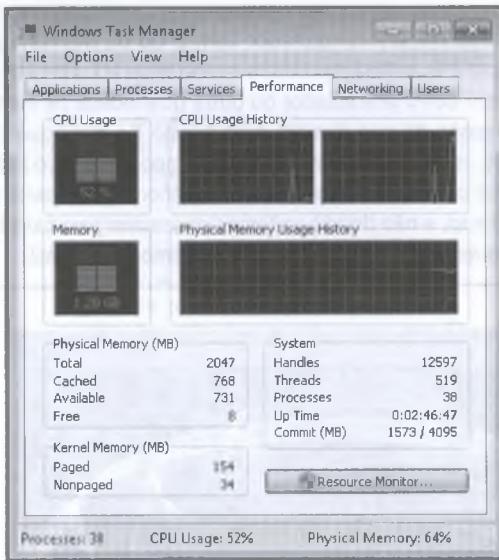


11. Clique em *Plot Graph* novamente e anote o tempo decorrido. Repita esta ação várias vezes para obter o valor médio.
12. Na área de trabalho, clique com o botão direito do mouse em uma área vazia da barra de tarefas, e, no menu pop-up, clique em *Start Task Manager*.



**Nota** No Windows Vista, o comando no menu pop-up é chamado de *Task Manager*.

13. No Windows Task Manager (Gerenciador de Tarefas do Windows), clique na guia *Performance*.
14. Retorne à janela *Graph Demo* e clique em *Plot Graph*.
15. No Windows Task Manager (Gerenciador de Tarefas do Windows), observe o valor máximo do uso da CPU enquanto o gráfico está sendo gerado. Seus resultados podem ser diferentes, mas em um processador dual-core a utilização da CPU provavelmente estará em torno de 50–55%, como mostra a imagem a seguir. Em uma máquina quad-core, a utilização da CPU provavelmente ficará abaixo dos 30%.



16. Retorne à janela *Graph Demo* e clique em *Plot Graph* novamente. Observe o valor do uso da CPU no Windows Task Manager. Repita esta ação várias vezes para obter o valor médio.
17. Feche a janela *Graph Demo* e minimize o Windows Task Manager.

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 Windows Task Manager, 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 multinúcleo (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.

### Modifique o aplicativo GraphDemo de modo a utilizar threads paralelas

1. Retorne ao Visual Studio 2010 e exiba o arquivo GraphWindow.xaml.cs na janela *Code and Text Editor* se ele ainda não estiver aberto.
2. Examine o método *generateGraphData*.

Ao observar cuidadosamente, você verá que o objetivo desse método é preencher os itens no array *data*. Ele faz uma iteração pelo array, utilizando o loop externo *for* baseado na variável de controle do loop *x*, destacado em negrito 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 fazer diversas iterações em um processador separado.

3. Modifique a definição do método *generateGraphData* de modo a aceitar dois parâmetros *int* adicionais, chamados *partitionStart* e *partitionEnd*, como mostrado em negrito a seguir:

```
private void generateGraphData(byte[] data, int partitionStart, int partitionEnd)
{
 ...
}
```

4. No método *generateGraphData*, mude o loop externo *for* para iterar entre os valores de *partitionStart* e *partitionEnd*, como mostrado em negrito a seguir:

```
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 instrução *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 a seguir em negrito, que cria um objeto *Task* usando o objeto padrão *TaskFactory*, e o coloca em execução:

```

Stopwatch watch = Stopwatch.StartNew();
// generateGraphData(data);
Task first = Task.Factory.StartNew(() => 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 em negrito a seguir:

```

Task first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 4));
Task second = Task.Factory.StartNew(() => 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 que aguarda os dois objetos *Task* terminarem seu trabalho para continuar:

```
Task.WaitAll(first, second);
```

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

10. Exiba o Windows Task Manager e clique na guia Performance, se ela ainda não estiver sendo exibida.

11. Retorne à janela *Graph Demo* e clique em *Plot Graph*. No Windows Task Manager (Gerenciador de Tarefas do Windows), observe o valor máximo do uso da CPU enquanto o gráfico está sendo gerado. 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.

12. Feche a janela *Graph Demo* e minimize o Windows Task Manager.

Dessa vez, você perceberá que a velocidade de execução do aplicativo é muito mais alta do que anteriormente. Em meu computador, o tempo caiu para 2682 milissegundos – uma redução em

torno de 40%. Além disso, você deve observar que o aplicativo utiliza mais núcleos da CPU. Em uma máquina dual-core, o uso da CPU chegou a 100%. Em um computador quad-core, a utilização da CPU não será assim tão alta. Isso ocorre porque dois dos núcleos não estarão ocupados. Para corrigir essa situação e reduzir o tempo ainda mais, adicione mais dois objetos *Task* e divida o trabalho em quatro partes no método *plotButton\_Click*, como mostrado em negrito a seguir:

```
...
Task first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8));
Task second = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 8,
pixelWidth / 4));
Task third = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 4,
pixelWidth * 3 / 8));
Task fourth = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth * 3 / 8,
pixelWidth / 2));
Task.WaitAll(first, second, third, fourth);
...

```

Se você tiver apenas um processador dual-core, também poderá experimentar essa modificação, e você deverá observar um efeito benéfico em relação ao tempo. Isso se deve basicamente às eficiências propiciadas pela TPL e aos algoritmos do .NET Framework que otimizam o modo como as threads de cada tarefa são agendadas.

## Abstraindo 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 tinha que modificar o design do aplicativo para acomodar o uso de objetos *Task*. Você também precisava adicionar código para sincronizar as operações; o aplicativo só pode processar 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.

A classe *Parallel* na TPL permite paralelizar algumas construções 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. Esses 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 ao utilizar 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++)
{
 // Executa o 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)
{
 // Executa o processamento do loop
}
```

As sobrecargas do método *Parallel.For* permitem 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>*** Você pode utilizar esse método em vez de uma instrução *foreach* do C#. Como no método *For*, o *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 na 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 ao 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 o .NET Framework 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, o .NET Framework 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, o .NET Framework gera o que ele considera o número ideal de tarefas que equilibra os recursos disponíveis quanto 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, você poderá obter 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 2010, abra a solução *GraphDemo*, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 27\GraphDemo Using the Parallel Class de sua pasta Documentos.

Esta 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 no arquivo *GraphWindow.xaml.cs* para exibir o código do formulário na janela *Code and Text Editor*.
3. Adicione a seguinte instrução *using* à lista localizada no início do arquivo:

```
using System.Threading.Tasks;
```

4. Localize o método *generateGraphData*, 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 externo *for*, 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 requer mais esforços para paralelizar devido ao tipo de *i*. (Os métodos da classe *Parallel* esperam que a variável de controle seja um inteiro.) Além disso, se existirem loops aninhados, como acontece nesse código, é recomendável paralelizar os loops externos primeiramente, e depois verificar se o desempenho do aplicativo é suficiente. Se não for, faça o que é necessário com os loops aninhados e parallelize-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 efeito máximo sobre o desempenho, enquanto os efeitos da modificação dos loops internos são mais sutis.

5. Mova o código no corpo do loop *for* e crie um novo método privado *void*, chamado *calculateData*, com esse código. O método *calculateData* deve aceitar um parâmetro inteiro 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 teste nem compile 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 a seguir:

```
private void generateGraphData(byte[] data)
{
 Parallel.For (0, pixelWidth / 2, (int 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 por meio de uma tarefa. (Cada tarefa pode fazer mais de uma iteração.) O método *Parallel.For* só terminará quando todas as tarefas por ele criadas concluirão 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 do loop atual como parâmetro. Neste exemplo, o método *calculateData* não corresponde à assinatura necessária porque ele aceita dois parâmetros: um inteiro e um array de bytes. Por esse motivo, o código utiliza uma expressão lambda para definir um método anônimo que tenha a assinatura adequada e que atue como um adaptador que chama o método *calculateData* com os parâmetros corretos.

7. No menu *Debug*, clique em *Start Without Debugging* para executar o aplicativo.
8. Exiba o Windows Task Manager e clique na guia *Performance*, se ela ainda não estiver sendo exibida.
9. Retorne à janela *Graph Demo* e clique em *Plot Graph*. No Windows Task Manager (Gerenciador de Tarefas do Windows), observe o valor máximo do uso da CPU enquanto o gráfico está sendo gerado. 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.
10. Feche a janela *Graph Demo* e minimize o Windows Task Manager.

Observe que o aplicativo funciona a uma velocidade comparável à da versão anterior, que utiliza objetos *Task* (e possivelmente, um pouco mais veloz, dependendo do número de CPUs disponíveis), e que a utilização da CPU atinge 100%.

## Quando não utilizar a classe Parallel

Saiba que, apesar das aparências e dos melhores esforços da equipe de desenvolvimento do Visual Studio 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.

Os termos mais importantes do parágrafo anterior são *vinculadas à computação* e *independentes*. Se seu código não for relacionado a alguma computação, é possível que a respectiva parallelização não melhore o desempenho. O próximo exercício mostra a necessidade de se ter cuidado ao determinar quando utilizar a construção *Parallel.Invoke* para fazer chamadas a métodos em paralelo.

### Determine quando utilizar a *Parallel.Invoke*

1. Retorne ao Visual Studio 2010 e exiba o arquivo *GraphWindow.xaml.cs* na janela *Code and Text Editor*, se ele ainda não estiver aberto.
2. Examine o método *calculateData*.

O loop *for* interno contém as seguintes instruções:

```
plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
```

Essas duas instruções definem os bytes no array *data*, correspondentes aos pontos especificados pelos dois parâmetros passados. Lembre-se de que os pontos do gráfico se refletem em torno do eixo X, de modo que o método *plotXY* é chamado para o valor positivo da coordenada X e também para o valor negativo. Essas duas instruções parecem boas candidatas para a paralelização porque não importa qual delas é executada primeiramente, e elas definem bytes distintos no array *data*.

3. Modifique essas duas instruções e encapsule-as em uma chamada ao método *Parallel.Invoke*, como mostrado a seguir. Observe que as duas chamadas estão atualmente encapsuladas em expressões lambda, que o caractere de ponto e vírgula no final da primeira chamada ao *plotXY* foi substituído por uma vírgula, e que o caractere de ponto e vírgula no final da segunda chamada ao *plotXY* foi removido porque essas instruções são agora uma lista de parâmetros:

```
Parallel.Invoke(
 () => plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2))),
 () => plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)))
);
```

4. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.
5. Na janela *Graph Demo*, clique em *Plot Graph*. Registre o tempo necessário para gerar o gráfico. Repita esta ação várias vezes para obter o valor médio.

Você deve detectar, repentinamente, que a execução do aplicativo é muito mais demorada, e que pode ser até 20 vezes mais lenta do que anteriormente.

6. Feche a janela *Graph Demo*.

Provavelmente, você deve estar se perguntando o que deu errado e por que o aplicativo ficou tão lento. A resposta está no método *plotXY*. Se você examinar novamente esse método, verá que ele é muito simples:

```
private void plotXY(byte[] data, int x, int y)
{
 data[x + y * pixelWidth] = 0xFF;
}
```

Há muito pouco nesse método que leva algum tempo para executar, e certamente ele não é um trecho de código vinculado à computação. Na realidade, ele é tão simples que a sobreulação de criar uma tarefa, executar essa tarefa em uma thread separada e aguardar o término da tarefa custa muito mais do que executar esse método diretamente. A sobreulação adicional pode contabilizar apenas alguns milissegundos, sempre que o método for chamado, mas você deve ter em mente o número de vezes que esse método é executado; a chamada ao método está localizada em um loop aninhado e é

executada milhares de vezes, de modo que todos esses pequenos custos de sobrecarga se acumulam. A regra geral é usar o *Parallel.Invoke* somente quando compensar. Reserve o *Parallel.Invoke* para as operações que efetuam muitos cálculos.

Como mencionado anteriormente neste capítulo, a outra consideração importante sobre o uso da classe *Parallel* é que as operações devem ser independentes. Por exemplo, se você tentar utilizar *Parallel.For* para paralelizar um loop no qual as iterações não são independentes, os resultados serão imprevisíveis. Para saber o que isso significa, examine o seguinte programa:

```
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, ele adicionará o valor do parâmetro à variável *accumulator*; caso contrário, ele subtrairá o valor do parâmetro. Quando o programa terminar, o resultado será exibido. Você encontrará esse aplicativo na solução *ParallelLoop*, localizada na pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 27\ParallelLoop* de sua pasta Documentos. Se você executasse esse programa, o valor emitido deveria ser -100.

Para aumentar o nível de paralelismo nesse aplicativo simples, procure 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 – 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, sempre que 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 efetuar 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); // aguardar 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 o .NET Framework 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.

## Retornando um valor de uma tarefa

Até agora, todos os exemplos que examinamos utilizam o objeto *Task* para executar um código que realiza uma parte do trabalho, mas não retorna um valor. Entretanto, talvez você também queira

executar um método que calcule um resultado. Para essa finalidade, a TPL dispõe de uma variante genérica da classe *Task*, a *Task*<*TResult*>.

Você cria e executa um objeto *Task*<*TResult*> de modo parecido com como cria um objeto *Task*. A principal diferença é que o método executado pelo objeto *Task*<*TResult*> retorna um valor, e você especifica o tipo desse valor de retorno como o parâmetro de tipo, *T*, do objeto *Task*. Por exemplo, o método *calculateValue*, mostrado no exemplo de código a seguir, retorna um valor inteiro. Para chamar esse método por meio de uma tarefa, crie um objeto *Task*<*int*> e chame o método *Start*. 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 chamador. Isso significa que você mesmo não precisa fazer qualquer sincronização, e você sabe que, quando a propriedade *Result* retornar um valor, a tarefa terá finalizado seu trabalho.

```
Task<int> calculateValueTask = new Task<int>(() => calculateValue(...));
calculateValueTask.Start(); // Chama o método calculateValue

...
int calculatedData = calculateValueTask.Result; // Bloquear até o término de calculateValueTask

...
private int calculateValue(...)
{
 int someValue;
 // Efetua cálculo e preenche someValue

 ...
 return someValue;
}
```

É evidente que você também pode utilizar o método *StartNew* de um objeto *TaskFactory* para criar um objeto *Task*<*TResult*> e iniciar a sua execução. O exemplo de código a seguir mostra como utilizar o objeto padrão *TaskFactory* para um objeto *Task*<*int*> a fim de criar e executar uma tarefa que chama o método *calculateValue*:

```
Task<int> calculateValueTask = Task<int>.Factory.StartNew(() => calculateValue(...));
...
```

Para simplificar um pouco seu código (e suportar tarefas que retornam tipos anônimos), a classe *TaskFactory* provê sobrecargas genéricas do método *StartNew* e pode inferir o tipo retornado pelo método executado por uma tarefa. Além disso, a classe *Task*<*TResult*> herda da classe *Task*. Isso significa que é possível reescrever o exemplo anterior como a seguir:

```
Task calculateValueTask = Task.Factory.StartNew(() => calculateValue(...));
...
```

O próximo exercício apresenta um exemplo mais detalhado. Nesse exercício, você vai reestruturar o aplicativo *GraphDemo* de modo a utilizar um objeto *Task*<*TResult*>. Embora esse exercício pareça um pouco acadêmico, a técnica por ele demonstrada lhe será útil em várias situações reais.

### Modifique o aplicativo GraphDemo, de modo a utilizar um objeto *Task*<*TResult*>

1. No Visual Studio 2010, abra a solução *GraphDemo*, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 27\GraphDemo Using Tasks that Return Results, de sua pasta Documentos.

Essa é uma cópia do aplicativo GraphDemo que cria um conjunto de quatro tarefas examinadas no exemplo anterior.

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. Localize o método *plotButton\_Click*. Esse é o método executado quando o usuário clica no botão *Plot Graph* no formulário. Atualmente, ele cria um conjunto de objetos *Task* para efetuar os vários cálculos necessários e gerar os dados do gráfico, e aguarda o término desses objetos *Task* para exibir os resultados no controle *Image* no formulário.
4. Abaixo do método *plotButton\_Click*, adicione um novo método, chamado *getDataForGraph*, que deve aceitar um parâmetro inteiro, chamado *dataSize*, e retornar um array de *bytes*, como mostrado no código a seguir:

```
private byte[] getDataForGraph(int dataSize)
{
}
```

Você adicionará um código a esse método para gerar os dados do gráfico em um array de *bytes* e retornar esse array para o chamador. O parâmetro *dataSize* especifica o tamanho do array.

5. Mova a instrução que cria o array de dados, do método *plotButton\_Click* para o método *getDataForGraph*, como mostrado em negrito a seguir:

```
private byte[] getDataForGraph(int dataSize)
{
 byte[] data = new byte[dataSize];
}
```

6. Mova o código que cria, executa e espera objetos *Task* que preenchem o array *data* do método *plotButton\_Click* para o método *getDataForGraph*, e adicione a instrução *return* ao final do método que passa o array *data* novamente para o chamador. O código concluído do método *getDataForGraph* deve ficar parecido com o seguinte:

```
private byte[] getDataForGraph(int dataSize)
{
 byte[] data = new byte[dataSize];
 Task first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8));
 Task second = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 8, pixelWidth / 4));
 Task third = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 4,
```

```

pixelWidth * 3 / 8));
 Task fourth = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth * 3 /
8, pixelWidth / 2));
 Task.WaitAll(first, second, third, fourth);
 return data;
}

```



**Dica** Você pode substituir o código que cria as tarefas e aguarda seu término pela seguinte construção `Parallel.Invoke`:

```

Parallel.Invoke(
 () => Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8))
 () => Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 8,
pixelWidth / 4)),
 () => Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 4,
pixelWidth * 3 / 8)),
 () => Task.Factory.StartNew(() => generateGraphData(data, pixelWidth * 3 / 8,
pixelWidth / 2))
);

```

- No método `plotButton_Click`, após a instrução que cria a variável `Stopwatch` utilizada para cronometrar as tarefas, adicione a instrução mostrada a seguir em negrito, que cria um objeto `Task<byte[]>`, chamado `getDataTask`, e utiliza esse objeto para executar o método `getDataForGraph`. Esse método retorna um array de `bytes`, de modo que o tipo da tarefa é `Task<byte[]>`. A chamada ao método `StartNew` faz referência a uma expressão lambda que chama o método `getDataForGraph` e passa a variável `dataSize` como parâmetro para esse método.

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{

 Stopwatch watch = Stopwatch.StartNew();
 Task<byte[]> getDataTask = Task<byte[]>.Factory.StartNew(() =>
getDataForGraph(dataSize));

}

```

- Após criar e iniciar o objeto `Task<byte[]>`, adicione as seguintes instruções, mostradas em negrito, que examinam a propriedade `Result` para recuperar o array de dados retornado pelo método `getDataForGraph` em uma variável local de array de bytes, chamada `data`. Lembre-se de que a propriedade `Result` bloqueia o chamador até o término da tarefa, de modo que não é necessário esperar explicitamente pelo término da tarefa.

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{

 Task<byte[]> getDataTask = Task<byte[]>.Factory.StartNew(() =>
getDataForGraph(dataSize));
 byte[] data = getDataTask.Result;

}

```



**Nota** Pode lhe parecer estranho o fato de criar uma tarefa e, logo a seguir, aguardar o seu término para fazer qualquer outra coisa, porque isso só sobrecarrega o aplicativo. Entretanto, na próxima seção, examinaremos o motivo dessa abordagem.

- Verifique se o código concluído do método `plotButton_Click` está parecido com o seguinte:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
 if (graphBitmap == null)
 {
 graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight, dpiX, dpiY,
PixelFormats.Gray8, null);
 }
 int bytesPerPixel = (graphBitmap.Format.BitsPerPixel + 7) / 8;
 int stride = bytesPerPixel * pixelWidth;
 int dataSize = stride * pixelHeight;

 Stopwatch watch = Stopwatch.StartNew();
 Task<byte[]> getDataTask = Task<byte[]>.Factory.StartNew(() =>
getaDataForGraph(dataSize));
 byte[] data = getDataTask.Result;

 duration.Content = string.Format("Duration (ms): {0}", watch.
ElapsedMilliseconds);
 graphBitmap.WritePixels(new Int32Rect(0, 0, pixelWidth, pixelHeight), data,
stride, 0);
 graphImage.Source = graphBitmap;
}
```

- No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.
- Na janela *Graph Demo*, clique em *Plot Graph*. Verifique se o gráfico é gerado como anteriormente, e se o tempo necessário é semelhante ao anterior. (O tempo informado pode ser mais lento porque o array de dados é criado, agora, pela tarefa, e anteriormente ele era criado antes do início da execução da tarefa.)
- Fechue a janela *Graph Demo*.

## Utilizando tarefas e threads de interface do usuário em conjunto

A seção “Por que fazer multitarefa por meio de processamento paralelo”, apresentada no início deste capítulo, destacou os dois motivos principais para utilizar multitarefa em um aplicativo – melhorar a taxa de transferência e a capacidade de resposta. Certamente, a TPL pode contribuir para aprimorar o desempenho (throughput), mas é necessário enfatizar que só o uso da TPL não é uma solução completa para agilizar a capacidade de resposta, principalmente em um aplicativo que fornece uma interface gráfica do usuário. No aplicativo *GraphDemo*, utilizado como base dos exercícios deste capítulo, embora o tempo necessário para gerar os dados do gráfico seja reduzido pelo uso eficiente das tarefas, o próprio aplicativo apresenta os sintomas clássicos de muitas GUIs que efetuam cálculos que exigem muito o processador – ele não é ágil em relação à entrada do usuário enquanto os

cálculos estão sendo efetuados. Por exemplo, se você executar o aplicativo *GraphDemo* do exercício anterior, clicar em *Plot Graph*, e tentar mover a janela Graph Demo ao clicar e arrastar a barra de título, perceberá que ela não se moverá até que as diversas tarefas que geram o gráfico tenham terminado e o gráfico seja exibido.

Em um aplicativo profissional, verifique se os usuários ainda podem utilizar seu aplicativo, mesmo se partes dele estiverem ocupadas com outras tarefas. É exatamente nesse momento que você deve usar as threads assim como as tarefas.

No Capítulo 23, vimos que todos os itens que formam a interface gráfica do usuário em um aplicativo WPF executam na mesma thread da interface do usuário (UI). Isso garante consistência e segurança, e impede que duas ou mais threads danifiquem as estruturas internas de dados, utilizadas pelo WPF para processar a interface do usuário. Lembre-se também de que é possível usar o objeto *Dispatcher* do WPF para enfileirar solicitações para a thread da UI, e essas solicitações podem atualizar a interface do usuário. O próximo exercício examinará novamente o objeto *Dispatcher* e mostrará como é possível empregá-lo para implementar uma solução ágil, em conjunto com tarefas que garantam o melhor desempenho disponível.

### Melhore a capacidade de resposta do aplicativo *GraphDemo*

1. Retorne ao Visual Studio 2010 e exiba o arquivo *GraphWindow.xaml.cs*, na janela *Code and Text Editor*, se ele ainda não estiver aberto.
2. Adicione um novo método, chamado *doPlotButtonWork*, abaixo do método *plotButton\_Click*. Esse método não deve aceitar quaisquer parâmetros nem retornar um resultado. Nas etapas a seguir, você moverá para esse método o código que cria e executa as tarefas que geram os dados do gráfico, e executará esse método em uma thread separada, deixando a thread da UI livre para o gerenciamento da entrada do usuário.

```
private void doPlotButtonWork()
{
}
```

3. Mova todo o código, exceto a instrução *if* que cria o objeto *graphBitmap*, do método *plotButton\_Click* para o método *doPlotButtonWork*. Observe que algumas dessas instruções tentam acessar itens da interface do usuário; você modificará essas instruções de modo a utilizar o objeto *Dispatcher* mais adiante neste exercício. Os métodos *plotButton\_Click* e *doPlotButtonWork* devem ficar parecidos com o seguinte:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
 if (graphBitmap == null)
 {
 graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight, dpiX, dpiY,
PixelFormats.Gray8, null);
 }
}
```

```

private void doPlotButtonWork()
{
 int bytesPerPixel = (graphBitmap.Format.BitsPerPixel + 7) / 8;
 int stride = bytesPerPixel * pixelWidth;
 int dataSize = stride * pixelHeight;

 Stopwatch watch = Stopwatch.StartNew();
 Task<byte[]> getDataTask = Task<byte[]>.Factory.StartNew(() =>
getaDataForGraph(dataSize));
 byte[] data = getDataTask.Result;

 duration.Content = string.Format("Duration (ms): {0}", watch.
ElapsedMilliseconds);
 graphBitmap.WritePixels(new Int32Rect(0, 0, pixelWidth, pixelHeight), data,
stride, 0);
 graphImage.Source = graphBitmap;
}

```

4. No método *plotButton\_Click*, após o bloco *if*, crie um delegate *Action* chamado *doPlotButtonWorkAction*, que faz referência ao método *doPlotButtonWork*, como mostrado em negrito a seguir:

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{

 Action doPlotButtonWorkAction = new Action(doPlotButtonWork);
}

```

5. Chame o método *BeginInvoke* no delegate *doPlotButtonWorkAction*. O método *BeginInvoke* do tipo *Action* executa o método associado ao delegate (nesse caso, o método *doPlotButtonWork*) em uma nova thread.



**Nota** O tipo *Action* também dispõe do método *Invoke*, que executa o método delegado na thread atual. Esse comportamento não é o desejado nesse caso, porque bloqueia a interface do usuário e impede que ela responda durante a execução do método.

O método *BeginInvoke* aceita parâmetros que você pode utilizar para emitir notificação quando o método terminar, assim como passar quaisquer dados para o método delegado. Neste exemplo, você não precisa receber uma notificação quando o método terminar, e o método não aceita quaisquer parâmetros, de modo que você pode especificar um valor *null* para esses parâmetros, como mostrado em negrito a seguir:

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{

 Action doPlotButtonWorkAction = new Action(doPlotButtonWork);
 doPlotButtonWorkAction.BeginInvoke(null, null);
}

```

O código copilará nesse ponto, mas se você o executar, ele não funcionará corretamente quando você clicar em *Plot Graph*. Isso ocorre porque várias instruções contidas no método *doPlotButtonWork* tentam acessar os itens da interface do usuário, e esse método não está em execução na thread da UI. Você examinou essa questão no Capítulo 23 e conheceu a solução naquela ocasião – usar o objeto *Dispatcher* para a thread da UI acessar os elementos da UI. As seguintes etapas alteram essas instruções de modo a utilizar o objeto *Dispatcher* para acessar os itens da interface do usuário a partir da thread correta.

6. Adicione a seguinte instrução *using* à lista localizada no início do arquivo:

```
using System.Windows.Threading;
```

A enumeração *DispatcherPriority* está armazenada nesse namespace. Você utilizará essa enumeração ao agendar um código para executar na thread da UI, por meio do objeto *Dispatcher*.

7. No início do método *doPlotButtonWork*, examine a instrução que inicializa a variável *bytesPerPixel*:

```
private void doPlotButtonWork()
{
 int bytesPerPixel = (graphBitmap.Format.BitsPerPixel + 7) / 8;
 ...
}
```

Essa instrução faz referência ao objeto *graphBitmap*, que pertence à thread da UI. Você só pode acessar esse objeto a partir do código em execução na thread da UI. Altere essa instrução de modo a inicializar a variável *bytesPerPixel* com zero, e adicione uma instrução para chamar o método *Invoke* do objeto *Dispatcher*, como mostrado em negrito a seguir:

```
private void doPlotButtonWork()
{
 int bytesPerPixel = 0;
 plotButton.Dispatcher.Invoke(new Action(() =>
 { bytesPerPixel = (graphBitmap.Format.BitsPerPixel + 7) / 8; }),
 DispatcherPriority.ApplicationIdle);
 ...
}
```

Vimos no Capítulo 23 que é possível acessar o objeto *Dispatcher* por meio da propriedade *Dispatcher* de qualquer elemento da UI. Esse código utiliza o botão *plotButton*. O método *Invoke* espera um delegate e uma prioridade do dispatcher opcional. Nesse caso, o delegate faz referência a uma expressão lambda. O código nessa expressão é executado na thread da UI. O parâmetro *DispatcherPriority* indica que essa instrução só deve ser executada quando o aplicativo estiver ocioso e não existir algo importante ocorrendo na interface do usuário (como o usuário clicando em um botão, digitando um texto ou movendo uma janela).

8. Examine as três últimas instruções do método *doPlotButtonWork*. Elas são semelhantes às seguintes:

```
private void doPlotButtonWork()
{

 duration.Content = string.Format("Duration (ms): {0}", watch.
ElapsedMilliseconds);
 graphBitmap.WritePixels(new Int32Rect(0, 0, pixelWidth, pixelHeight), data,
stride, 0);
 graphImage.Source = graphBitmap;
}
```

Essas instruções fazem referência aos objetos *duration*, *graphBitmap* e *graphImage*, que fazem parte da interface do usuário. Consequentemente, você deve alterar essas instruções para serem executadas na thread da UI.

9. Modifique essas instruções e execute-as ao utilizar o método *Dispatcher.Invoke*, como mostrado em negrito a seguir:

```
private void doPlotButtonWork()
{

 plotButton.Dispatcher.Invoke(new Action(() =>
 {
 duration.Content = string.Format("Duration (ms): {0}", watch.
ElapsedMilliseconds);
 graphBitmap.WritePixels(new Int32Rect(0, 0, pixelWidth, pixelHeight), data,
stride, 0);
 graphImage.Source = graphBitmap;
 }), DispatcherPriority.ApplicationIdle);
}
```

Esse código converte as instruções em uma expressão lambda encapsulada em um delegate *Action*, e depois chama esse delegate por meio do objeto *Dispatcher*.

10. No menu *Debug*, clique em *Start Without Debugging* para construir e executar o aplicativo.
11. Na janela *Graph Demo*, clique em *Plot Graph*, e antes da exibição do gráfico arraste rapidamente a janela para outro local na tela. Você perceberá que a janela responde imediatamente e não espera que o gráfico seja exibido primeiramente.

12. Feche a janela *Graph Demo*.

## Cancelando tarefas e tratando exceções

Outra exigência comum dos aplicativos que efetuam operações demoradas é a possibilidade de interromper essas operações, se necessário. Entretanto, você não deve abortar uma tarefa, simplesmente porque isso poderia deixar os dados em seu aplicativo em um estado indeterminado. Em vez disso, a TPL 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 um *token de cancelamento*, que 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 estiver definida com *true* em qualquer momento, ele reconhecerá que o aplicativo solicitou o cancelamento da tarefa. Além disso, o método reconhece o trabalho executado até então, e pode desfazer quaisquer alterações realizadas, se necessário, e depois encerrar. Como alternativa, o método pode simplesmente ignorar a solicitação e continuar a execução, se ele não quiser cancelar a tarefa.

**Dica** Você deve examinar frequentemente o token de cancelamento em uma tarefa, mas não tão frequentemente 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 um parâmetro para quaisquer métodos inicializados por tarefas geradas e executadas pelo aplicativo. 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 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, ele 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 executando o método doWork
Task myTask = Task.Factory.StartNew(() => 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)
 {
 // Conclui e encerra

 return;
 }
 // Se a tarefa não foi cancelada, continua a execução normalmente

}
}

```

Além de propiciar um alto grau de controle sobre o processamento do cancelamento, essa abordagem 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* no objeto *CancellationTokenSource*, cada tarefa reconhecerá que a propriedade *IsCancellationRequested* foi definida e poderá reagir adequadamente.

Você também pode registrar um método de retorno de chamada como 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 o próprio processamento do cancelamento, ou até mesmo durante esse processo.

```

cancellationToken.Register(doAdditionalWork);

private void doAdditionalWork()
{
 // Executa o processamento do cancelamento adicional
}

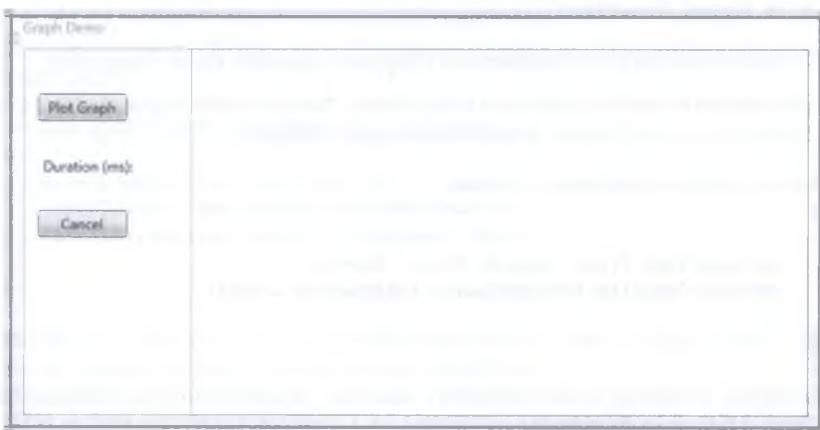
```

No próximo exercício, você adicionará a funcionalidade de cancelamento ao aplicativo GraphDemo.

## Adicione a funcionalidade de cancelamento ao aplicativo GraphDemo

1. No Visual Studio 2010, abra a solução *GraphDemo*, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 27\GraphDemo Canceling Tasks de sua pasta Documentos.  
Essa é uma cópia completa do aplicativo GraphDemo do exercício anterior, que utiliza tarefas e threads para melhorar a capacidade de resposta.
2. No Solution Explorer, no projeto *GraphDemo*, clique duas vezes em *GraphWindow.xaml* para exibir o formulário na janela *Design View*.
3. Na *Toolbox*, adicione um controle *Button* ao formulário, abaixo do rótulo *duration*. Alinhe o botão horizontalmente com o botão *plotButton*. Na janela *Properties*, altere a propriedade *Name* do novo botão para *cancelButton*, e mude a propriedade *Content* para *Cancel*.

O formulário corrigido deve ficar parecido com o da seguinte imagem.



4. Clique duas vezes no botão *Cancel* para criar um método de manipulação do evento *Click*, chamado *cancelButton\_Click*.
5. No arquivo *GraphWindow.xaml.cs*, localize o método *getDataForGraph*, que cria as tarefas utilizadas pelo aplicativo e espera o seu término. Mova a declaração das variáveis *Task* para o nível de classe para a classe *GraphWindow*, como mostrado em negrito no código a seguir, e depois modifique o método *getDataForGraph* para instanciar essas variáveis:

```
public partial class GraphWindow : Window
{

 private Task first, second, third, fourth;

 private byte[] getDataForGraph(int dataSize)
```

```

 {
 byte[] data = new byte[dataSize];
 first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8));
 second = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 8, pixelWidth / 4));
 third = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 4, pixelWidth * 3 / 8));
 fourth = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth * 3 / 8, pixelWidth / 2));
 Task.WaitAll(first, second, third, fourth);
 return data;
 }
}

```

6. Adicione a seguinte instrução *using* à lista localizada no início do arquivo:

```
using System.Threading;
```

Os tipos utilizados pelo cancelamento cooperativo residem nesse namespace.

7. Adicione um membro *CancellationTokenSource*, chamado *tokenSource*, à classe *GraphWindow*, e inicialize-o com null, como mostrado em negrito a seguir:

```

public class GraphWindow : Window
{
 ...
 private Task first, second, third, fourth;
 private CancellationTokenSource tokenSource = null;
 ...
}

```

8. Localize o método *generateGraphData* e adicione um parâmetro de *CancellationToken*, chamado *token*, à definição do método:

```

private void generateGraphData(byte[] data, int partitionStart, int partitionEnd,
CancellationToken token)
{
 ...
}

```

9. No método *generateGraphData*, no início do loop interno *for*, adicione o código mostrado em negrito a seguir para verificar se o cancelamento foi solicitado. 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)));
 }
 }
}

```

10. No método *getDataForGraph*, adicione as seguintes instruções, mostradas em negrito, que instanciam a variável *tokenSource* e recuperam o objeto *CancellationToken* em uma variável chamada *token*:

```

private byte[] getDataForGraph(int dataSize)
{
 byte[] data = new byte[dataSize];
 tokenSource = new CancellationTokenSource();
 CancellationToken token = tokenSource.Token;

}

```

11. Modifique as instruções que criam e executam as quatro tarefas e passe a variável *token* como o último parâmetro para o método *generateGraphData*:

```

first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8,
 token));
second = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 8,
 pixelWidth / 4, token));
third = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 4,
 pixelWidth * 3 / 8, token));
fourth = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth * 3 / 8,
 pixelWidth / 2, token));

```

12. No método *cancelButton\_Click*, adicione o código mostrado em negrito a seguir:

```

private void cancelButton_Click(object sender, RoutedEventArgs e)
{
 if (tokenSource != null)
 {
 tokenSource.Cancel();
 }
}

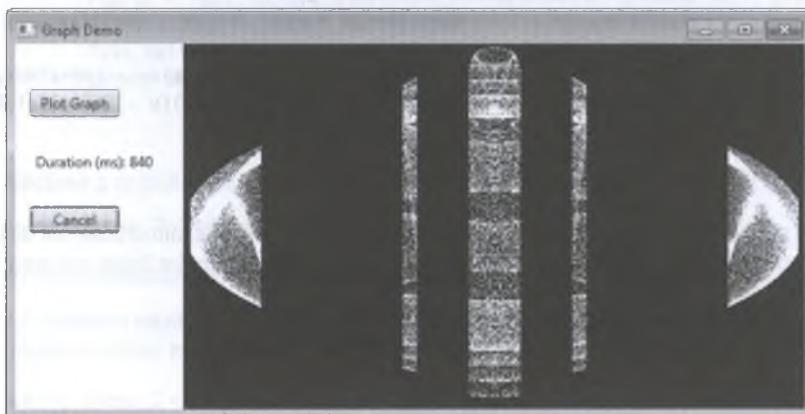
```

Esse código verifica se a variável *tokenSource* foi instanciada; em caso afirmativo, o código chama o método *Cancel* nessa variável.

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

14. Na janela GraphDemo, clique em *Plot Graph* e verifique se o gráfico é exibido como anteriormente.
15. Clique em *Plot Graph* novamente, e clique rapidamente em *Cancel*.

Se você clicar rapidamente em *Cancel* antes da geração dos dados do gráfico, esta ação instruirá o retorno dos métodos sendo executados pelas tarefas. Os dados não estarão completos, de modo que o gráfico será exibido com lacunas, como mostra a figura a seguir. (O tamanho das lacunas dependerá da rapidez com que você clicar em *Cancel*.)



16. Feche a janela GraphDemo e retorne ao Visual Studio.

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 quaisquer exceções não tratadas.
- **Canceled** A tarefa foi cancelada antes do início de sua execução, ou ela 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 você 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 antecipadamente o método *Parallel.For* ou *Parallel.ForEach*, use um objeto *ParallelLoopState*. O método que você especificar como corpo do loop deve incluir um parâmetro adicional, *ParallelLoopState*. A TPL cria um objeto *ParallelLoopState* e passa-o como esse parâmetro para o método. A TPL usa esse objeto para armazenar informações sobre cada chamada ao método. O método pode chamar o método *Stop* desse objeto para indicar que a TPL não deve tentar realizar quaisquer iterações além daquelas já iniciadas e finalizadas. O exemplo a seguir mostra o método *Parallel.For* que chama 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. De modo idêntico, 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

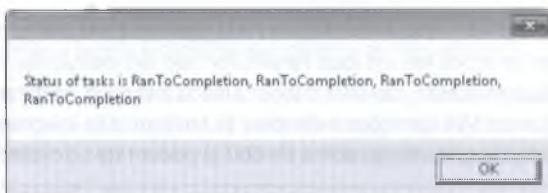
1. No Visual Studio, na janela *Code and Text Editor*, localize o método *getDataForGraph*.
2. Adicione a esse método o seguinte código, mostrado em negrito. Essas instruções geram uma string que contém o status de cada tarefa após o término da respectiva execução, e exibem uma caixa de mensagem que contém essa string.

```
private byte[] getDataForGraph(int dataSize)
{

 Task.WaitAll(first, second, third, fourth);
 String message = String.Format("Status of tasks is {0}, {1}, {2}, {3}",
 first.Status, second.Status, third.Status, fourth.Status);
 MessageBox.Show(message);

 return data;
}
```

3. No menu *Debug*, clique em *Start Without Debugging*.
4. Na janela *GraphDemo*, clique em *Plot Graph*, mas não em *Cancel*. Verifique se a seguinte caixa de mensagem é exibida, informando que o status das tarefas é *RanToCompletion* (quatro vezes), e depois clique em *OK*. Observe que o gráfico só aparece depois que você clica em *OK*.



5. Na janela *GraphDemo*, clique em *Plot Graph* novamente, e clique rapidamente em *Cancel*. Surpreendentemente, a caixa de mensagem exibida ainda informa o status de cada tarefa como *RanToCompletion*, embora o gráfico seja exibido com lacunas. Isso 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 reconhece 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.
6. Feche a janela *GraphDemo* e retorne ao Visual Studio.

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 um 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 verdadeira, 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, lançando uma exceção, ela realmente reverterá para o estado *Faulted*. Isso ocorre mesmo que a exceção seja uma *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?

A resposta está na própria tarefa. Para que uma tarefa reconheça que uma *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 um parâmetro para o método executado pela tarefa, mas, na realidade, a tarefa não examina esses parâmetros. (Elas considera que isso é assunto para ser resolvido pelo método e não se preocupa com eles.) Em vez disso, especifique o token de cancelamento ao criar a tarefa, como um parâmetro para o construtor de *Task* ou como um parâmetro para o método *StartNew* do objeto *TaskFactory* que você está utilizando para criar e executar tarefas. O código a seguir mostra um exemplo baseado no aplicativo *GraphDemo*. Observe que o parâmetro *token* é passado para o método *generateGraphData* (como anteriormente), mas também como um parâmetro separado para o método *StartNew*:

```
Task first = null;
tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;

...
first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8, 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 tratará a exceção *OperationCanceledException*, reconhecerá o cancelamento e definirá o status da tarefa com *Canceled*. Em seguida, a infraestrutura lança uma *TaskCanceledException*, que seu aplicativo deve estar preparado para capturar. Você fará exatamente isso no próximo exercício, mas, antes disso, você precisará conhecer mais detalhes sobre como as tarefas levantam exceções e como você deve tratá-las.

## Tratando 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 proble-

ma mais complexo. A questão é que cada tarefa diferente pode gerar suas próprias exceções, e você precisa encontrar um jeito de capturar e tratar as diversas exceções lançadas simultaneamente. É exatamente aí que a classe *AggregateException* entra em ação.

Uma *AggregateException* atua como um wrapper para uma coleção de exceções. Cada exceção na coleção pode ser lançada por diferentes tarefas. Em seu aplicativo, você pode capturar a exceção *AggregateException*, pode iterar sobre essa coleção e efetuar o processamento necessário. Para ajudá-lo, a classe *AggregateException* fornece o método *Handle*. O método *Handle* aceita um delegate *Func<Exception, bool>* que faz referência a um método.

O método referenciado aceita um objeto *Exception* como parâmetro e retorna um valor *booleano*. Quando você chamar o método *Handle*, o método referenciado é executado para cada exceção existente na coleção, no 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, ele retornará *false*. Quando o método *Handle* for concluído, todas as exceções não tratadas serão empacotadas em uma nova *AggregateException* e essa exceção será lançada; um manipulador de exceções externo subsequente poderá capturar essa exceção e processá-la.

No próximo exercício, você verá como capturar uma *AggregateException* e utilizá-la para tratar a exceção *TaskCanceledException* lançada quando a tarefa é cancelada.

### Reconheça o cancelamento e trate a exceção *AggregateException*

1. No Visual Studio, exiba o arquivo *GraphWindow.xaml* na janela *Design View*.
2. Na *Toolbox*, adicione um controle *Label* ao formulário, abaixo do botão *cancelButton*. Alinhe a borda esquerda do controle *Label* à borda esquerda do botão *cancelButton*.
3. Na janela *Properties*, mude a propriedade *Name* do controle *Label* para *status*, e remova o valor da propriedade *Content*.
4. Retorne à janela *Code and Text Editor* que exibe o arquivo *GraphWindow.xaml.cs* e adicione o seguinte método abaixo do método *getDataForGraph*:

```
private bool handleException(Exception e)
{
 if (e is TaskCanceledException)
 {
 plotButton.Dispatcher.Invoke(new Action(() =>
 {
 status.Content = "Tasks Canceled";
 }), DispatcherPriority.ApplicationIdle);
 return true;
 }
 else
 {
 return false;
 }
}
```

Este método examina o objeto *Exception*, passado como um parâmetro; se for um objeto *TaskCanceledException*, o método exibirá o texto “Tasks Canceled” no rótulo *status* no formulário, e retornará *true* para indicar que ele tratou a exceção; caso contrário, ele retornará *false*.

5. No método *getDataForGraph*, modifique as instruções que criam e executam as tarefas, e especifique o objeto *CancellationToken* como o segundo parâmetro para o método *StartNew*, como mostrado em negrito no código a seguir:

```
private byte[] getDataForGraph(int dataSize)
{
 byte[] data = new byte[dataSize];
 tokenSource = new CancellationTokenSource();
 CancellationToken token = tokenSource.Token;

 . . .
 first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8,
 token), token);
 second = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 8,
 pixelWidth / 4, token), token);
 third = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 4,
 pixelWidth * 3 / 8, token), token);
 fourth = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth * 3 / 8,
 pixelWidth / 2, token), token);
 Task.WaitAll(first, second, third, fourth);

 . . .
}
```

6. Adicione um bloco *try* ao redor das instruções que criam e executam as tarefas, e aguarde o respectivo término. Se a espera for bem-sucedida, exiba o texto “Tasks Completed” no rótulo *status* no formulário, usando o método *Dispatcher.Invoke*. Adicione um bloco *catch* que trata a exceção *AggregateException*. Nesse manipulador de exceções, chame o método *Handle* do objeto *AggregateException* e passe uma referência para o método *handleException*. O código mostrado em negrito a seguir destaca as alterações que você deve implementar:

```
private byte[] getDataForGraph(int dataSize)
{
 byte[] data = new byte[dataSize];
 tokenSource = new CancellationTokenSource();
 CancellationToken token = tokenSource.Token;

 try
 {
 first = Task.Factory.StartNew(() => generateGraphData(data, 0, pixelWidth / 8,
 token), token);
 second = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 8,
 pixelWidth / 4, token), token);
 third = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth / 4,
 pixelWidth * 3 / 8, token), token);
 fourth = Task.Factory.StartNew(() => generateGraphData(data, pixelWidth * 3 / 8,
 pixelWidth / 2, token), token);
 Task.WaitAll(first, second, third, fourth);
 }
```

```

 plotButton.Dispatcher.Invoke(new Action(() =>
 {
 status.Content = "Tasks Completed";
 }), DispatcherPriority.ApplicationIdle);
}
catch (AggregateException ae)
{
 ae.Handle(handleException);
}

String message = String.Format("Status of tasks is {0}, {1}, {2}, {3}",
 first.Status, second.Status, third.Status, fourth.Status);
MessageBox.Show(message);

return data;
}

```

7. No método *generateDataForGraph*, substitua a instrução *if* que examina a *IsCancellationProperty* do objeto *CancellationToken* com o código 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)
 {
 token.ThrowIfCancellationRequested();
 }
 ...
 }
 ...
}

```

8. No menu *Debug*, clique em *Start Without Debugging*.
9. Na janela Graph Demo, clique em *Plot Graph* e verifique se o status de cada tarefa é informado como *RanToCompletion*, se o gráfico é gerado, e se o rótulo *status* exibe a mensagem “Tasks Completed”.
10. Clique em *Plot Graph* novamente, e depois clique rapidamente em *Cancel*. Se você conseguir clicar rapidamente, o status de uma ou mais tarefas deverá ser informado como *Canceled*, o rótulo *status* deverá exibir o texto “Tasks Canceled”, e o gráfico será exibido com lacunas. Se você não for suficientemente rápido, repita esta etapa e tente novamente!
11. Feche a janela Graph Demo e retorne ao Visual Studio.

## Utilizando continuações com tarefas canceladas e com falhas

Para fazer um 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.OnlyOnCancelled);
task.Start();

...
private void doWork()
{
 // A tarefa executa este código quando for iniciada
 ...
}

...
private void doCancellationWork(Task task)
{
 // A tarefa executa este 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.

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 Task Parallel Library 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 do 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 você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 28.

- Se quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 27

Para	Faça isto
Criar uma tarefa e executá-la	<p>Utilize o método <i>StartNew</i> de um objeto <i>TaskFactory</i> para criar e executar uma tarefa em uma única etapa:</p> <pre>Task task = taskFactory.StartNew(doWork());</pre> <pre>***</pre> <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 para executar e chamar 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> <pre>***</pre> <pre>task.Wait();</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 = ...;</pre> <pre>***</pre> <pre>Task.WaitAll(task1, task2, task3, task4);</pre>
Especificar um método para 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>
Retornar um valor de uma tarefa	<p>Use um objeto <i>Task&lt;TResult&gt;</i> para executar um método, em que o parâmetro do tipo <i>T</i> especifica o tipo do valor de retorno do método. Use a propriedade <i>Result</i> da tarefa para esperar o término da tarefa e retornar o valor:</p> <pre>Task&lt;int&gt; calculateValueTask = new Task&lt;int&gt;(() =&gt;     calculateValue(...)); calculateValueTask.Start(); // Chama o método calculateValue</pre> <pre>***</pre> <pre>int calculatedData = calculateValueTask.Result; // Bloqueia até calculateValueTask terminar</pre>

Para	Faça isto
Fazer iterações de loop e sequências de instruções por meio de 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) {     // Realizar o processamento do loop }</pre> <p>Use o método <i>Parallel.Invoke</i> para fazer chamadas concorrentes ao método, por meio de tarefas separadas:</p> <pre>Parallel.Invoke(     doWork,     doMoreWork,     doYetMoreWork );</pre>
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 no 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.Factory.StartNew(...);      *** } catch (AggregateException ae) {     ae.Handle(new Func&lt;Exception, bool&gt; (handleException)); }  ***  private bool handleException(Exception e) {     if (e is TaskCanceledException)     {         ***         return true;     }     else     {         return false;     } }</pre>

(continua)

Para	Faça isto
Suportar o cancelamento em uma tarefa	<p>Implemente o cancelamento cooperativo, criando um objeto <code>CancellationTokenSource</code> e utilizando um parâmetro <code>CancellationToken</code> no método executado pela tarefa. No método da tarefa, chame o método <code>ThrowIfCancellationRequested</code> do parâmetro <code>CancellationToken</code> para lançar uma exceção <code>OperationCanceledException</code> e encerrar a tarefa:</p> <pre>private void generateGraphData(..., CancellationToken token) {     ...     token.ThrowIfCancellationRequested();     ... }</pre>

## Capítulo 28

# Realizando acesso a dados em paralelo

Neste capítulo, você vai aprender a:

- Utilizar a PLINQ para paralelizar as consultas LINQ demoradas.
- Utilizar as classes de coleção concorrentes paralelas para manter coleções de dados de modo seguro quanto a threads.
- Utilizar as primitivas de sincronização paralela para coordenar o acesso a dados manipulados por tarefas simultâneas.

No Capítulo 27, “Introdução à Task Parallel Library”, vimos como explorar os novos recursos do .NET Framework para efetuar operações em paralelo. Os capítulos anteriores também mostraram como é possível acessar dados de modo declarativo com a Language Integrated Query (LINQ). Uma consulta LINQ comum gera um conjunto de resultados enumeráveis, e você pode iterar sequencialmente por esse conjunto para recuperar os dados. Se a origem dos dados utilizada para gerar o conjunto de resultados for grande, fazer 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 designers da Task Parallel Library (TPL) decidiram fornecer à LINQ um recurso semelhante, e o resultado foi a Parallel LINQ, ou a PLINQ. Você estudará a PLINQ na primeira parte deste capítulo.

Contudo, nem sempre a PLINQ é a tecnologia mais adequada para ser utilizada em um aplicativo. Se você cria manualmente as próprias tarefas, certifique-se de que as threads simultâneas, que executam as tarefas, estejam coordenando as respectivas atividades corretamente. A TPL dispõe de métodos que permitem aguardar o término das tarefas, e você pode utilizá-los para coordenar tarefas em um nível muito “grosso”. Mas examine o que acontece se duas tarefas tentarem acessar e modificar os mesmos dados. Se ambas as tarefas forem executadas ao mesmo tempo, suas operações sobrepostas podem danificar os dados. Essa situação pode gerar defeitos de difícil correção, basicamente devido à sua imprevisibilidade. A partir da versão 1.0, o Microsoft .NET Framework disponibilizou primitivas para bloquear os dados e coordenar as threads, mas, para utilizá-las de modo eficiente, você deverá conhecer muito bem o modo de interação das threads. A TPL contém algumas variações dessas primitivas, e classes de coleções específicas que podem sincronizar o acesso aos dados por meio das tarefas. Essas classes aliviam 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 segunda metade deste capítulo.

## Utilizando a PLINQ para parallelizar o acesso declarativo a dados

Nos capítulos anteriores, conhecemos o poder da LINQ para recuperar dados em uma estrutura de dados enumeráveis. No .NET Framework 4.0, a LINQ foi estendida por meio da tecnologia disponível como parte da TPL, para ajudá-lo a otimizar o desempenho e parallelizar algumas operações de consulta. Essas extensões representam a PLINQ.

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. Os resultados recuperados para cada partição são combinados em um único conjunto de resultados enumeráveis ao término das tarefas. 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.

Um objetivo primordial da PLINQ é não ser invasiva o máximo possível. Se você tiver muitas consultas LINQ já existentes, não convém modificar seu código para permitir a sua execução 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 que ele disponibiliza implementações paralelas de vários operadores LINQ, como *join* e *where*. Essas novas implementações dos operadores LINQ se baseiam na TPL 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 parallelização. 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.

## Utilizando 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 na 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.



**Nota** Se os cálculos exigirem acesso a dados compartilhados, você deverá sincronizar as threads. Isso pode impor uma sobrecarga e perder os benefícios da paralelização da consulta.

### Paralelize uma consulta LINQ sobre uma coleção simples

1. No Microsoft Visual Studio 2010, abra a solução PLINQ, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 28\PLINQ em sua pasta Documentos.
2. No Solution Explorer, 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. A classe *Program* contém dois métodos, chamados *Test1* e *Test2*, que ilustram um par de 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, 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. Você adicionará essas instruções ao longo deste conjunto de exercícios.

3. Localize 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. Você adicionará uma consulta LINQ que recupera nesse array todos os números com um valor acima de 100.
4. No primeiro comentário *TO DO* nesse método, adicione a consulta LINQ mostrada em negrito a seguir\*:

```
// TO DO: Crie uma consulta LINQ que recupere todos os números acima de 100
var over100 = from n in numbers
 where TestIfTrue(n > 100)
 select n;
```

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;
}
```

\* N. de R. T.: Os comentários TO DO aparecem aqui traduzidos para facilitar a compreensão.

5. Após o segundo comentário *TO DO* no método *Test1*, adicione o seguinte código, mostrado em negrito:

```
// TO DO: Execute uma consulta LINQ e salve os resultados em um objeto List<int>
List<int> numbersOver100 = new List<int>(over100);
```

Convém lembrar que as consultas LINQ usam uma 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* no método *Test1*, adicione a seguinte instrução, mostrada em negrito:

```
// TO DO: Exiba 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 se o número de itens acima de 100 é o mesmo em todas as vezes. Quando terminar, retorne ao Microsoft Visual Studio.

9. Cada item retornado pela consulta LINQ é independente de todas as outras linhas, e 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 em negrito a seguir:

```
var over100 = from n in numbers.AsParallel()
 where TestIfTrue(n > 100)
 select n;
```

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 duração do teste. 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 obter quando se faz uma mudança pequenina em uma consulta LINQ. Entretanto, lembre-se de que você só verá resultados como esses se os cálculos efetuados pela consulta exigirem muito tempo da CPU. Eu trapaceei um pouco o processador usando *spinning*. Sem essa sobrecarga, a versão paralela da consulta é realmente mais lenta do que a versão serial. Neste exercício, você verá uma consulta LINQ que une dois arrays na memória. Dessa vez, o exercício utiliza volumes de dados mais reais, e não há necessidade de utilizar qualquer artifício para atrasar a consulta artificialmente.

## Paralelize uma consulta LINQ que une duas coleções via join

1. Na janela *Code and Text Editor*, localize a classe *CustomersInMemory*.

Essa classe contém um array público de *strings*, 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 texto que utiliza campos separados por vírgulas. O primeiro campo contém o ID do cliente, o segundo campo é o nome da empresa que o cliente representa, e os demais campos armazenam o 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 array *Customers* e *Orders*.
4. Localize o método *Test2* na classe *Program*. Nesse método, você criará uma consulta LINQ que une os arrays *Customers* e *Orders* por meio de um ID de cliente. A consulta armazenará cada linha do resultado em um objeto *OrderInfo*.
5. No bloco *try* nesse método, adicione o código mostrado a seguir em negrito depois do primeiro comentário TO DO:

```
// TO DO: Crie uma consulta LINQ que recupera os clientes e os pedidos em arrays
// Armazene 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 e 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.

6. No método *Test2*, adicione o seguinte código mostrado em negrito após a segunda instrução TO DO:

```
// TO DO: Execute a consulta LINQ e salve 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. Adicione a instrução mostrada em negrito a seguir depois da terceira instrução TO DO:

```
// TO DO: Exiba os resultados
Console.WriteLine("There are {0} orders", orderInfo.Count);
```

8. No menu *Debug*, clique em *Start Without Debugging*.

Verifique que o Test 2 recupera 830 pedidos, e observe a duração do teste. Execute o aplicativo várias vezes para obter uma duração média e retorne ao Visual Studio.

9. 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"))
};
```



**Nota** 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, o runtime execução não paralelizará a consulta e você perderá os benefícios.

10. Execute o aplicativo várias vezes novamente. Observe que o tempo necessário para o Test 2 deve ser muito mais curto que anteriormente. A PLINQ pode utilizar várias threads para otimizar operações de união, buscando simultaneamente os dados de cada parte da união.
11. Feche o aplicativo e retorne ao Visual Studio.

Esses dois exercícios simples demonstraram o poder do método de extensão *AsParallel* e da PLINQ. Contudo, a PLINQ é uma tecnologia em evolução, e é 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.

## Especificando opções para uma consulta PLINQ

O objeto *ParallelEnumerable* retornado pelo método *AsParallel* apresenta alguns métodos que você pode utilizar para influenciar no modo como uma consulta é paralelizada. Por exemplo, você pode especificar uma quantidade de tarefas que, na sua opinião, é o número ideal, e substituir quaisquer decisões tomadas durante a execução, por meio do método *WithDegreeOfParallelism*, como a seguir:

```
var orderInfoQuery =
 from c in CustomersInMemory.Customers.AsParallel().WithDegreeOfParallelism(4)
 join o in OrdersInMemory.Orders.AsParallel()
 on ...
```

O valor especificado se aplica à consulta inteira. Consequentemente, especifique *WithDegreeOfParallelism* apenas uma vez em uma consulta. No exemplo que acabamos de mostrar, o grau de paralelismo é aplicado aos objetos *Customers* e *Orders*.

Ocasionalmente, podem ocorrer algumas situações para as quais o runtime determina, por meio de uma heurística própria, que a paralelização de uma consulta ajudará muito pouco. Se você tiver certeza de que esse não é o caso, poderá utilizar o método *WithExecutionMode* da classe *ParallelQuery* e instruir o runtime a paralelizar a consulta. O código a seguir apresenta um exemplo:

```
var orderInfoQuery =
 from c in CustomersInMemory.Customers.AsParallel().WithExecutionMode(ParallelExecutionMode.ForceParallelism)
 join o in OrdersInMemory.Orders.AsParallel()
 on ...
```

Mais uma vez, você só pode utilizar *WithExecutionMode* uma única vez em uma consulta.

Ao paralelizar consultas, você pode interferir na sequência de retorno dos dados. Se a ordenação for importante, você pode especificar o método de extensão *AsOrdered* da classe *ParallelQuery*. Por exemplo, para retornar apenas os primeiros resultados de uma consulta, a preservação dessa sequência pode ser importante para garantir que a consulta retorne resultados consistentes em cada execução, como demonstra o exemplo a seguir, que utiliza o método *Take* para retornar os 10 primeiros itens de um conjunto de dados:

```
var over100 = from n in numbers.AsParallel().AsOrdered().Take(10)
 where ...
 select n;
```

Provavelmente, isso retardará a consulta, e você deverá pesar os prós e os contras entre o desempenho e a ordenação ao implementar consultas dessa maneira.

## Cancelando uma consulta PLINQ

Diferentemente das consultas LINQ comuns, uma consulta PLINQ pode ser cancelada. Para isso, especifique um objeto *CancellationToken* 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 *CancellationToken* for cancelado, a consulta será paralisada com uma exceção *OperationCanceledException*.

## Sincronizando acessos imperativos e simultâneos a dados

A TPL dispõe de uma estrutura poderosa, que permite 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, principalmente 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 da TPL. 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 conhecer como seu código é executado; caso contrário, você 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 maçante), 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 é 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 executada 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 que ele utiliza 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, ele armazena o valor de *j* no elemento do array de dados indexado por *i*. Se *i* contiver 5, então *j* receberá o valor 5, e posteriormente, 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 pode suspender a thread e agendar outra tarefa. Uma tarefa simultânea, que execute outra iteração da construção *Parallel.For*, 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, ele não funciona; tudo depende de quanto o computador se encontra ocupado e quando as diversas tarefas estão agendadas. Consequentemente, 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 por meio de todas as tarefas simultâneas que podem acessá-la. Uma maneira de sincronizar o acesso é bloquear os dados.

## Bloqueando dados

A linguagem C# fornece uma 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.

Se um objeto estiver bloqueado, o que uma thread deve fazer enquanto aguarda a liberação do bloqueio? Há duas respostas, pelo menos. A thread pode ser colocada em repouso (sleep) até o bloqueio se tornar disponível. Isso exige que o runtime execute um volume de trabalho considerável, inclusive salvar o estado da thread e enfileirar a thread para posterior execução quando o bloqueio estiver

disponível. Se o bloqueio for mantido apenas por um curto período de tempo, a sobrecarga de suspender, reagendar e reiniciar a thread pode ultrapassar o tempo necessário para que o bloqueio fique disponível, o que, por sua vez, afeta o desempenho de seu aplicativo. Uma estratégia alternativa é deixar que a thread “gire o processador” (ao estilo do método *Thread.SpinWait*, que você viu anteriormente, neste capítulo) até que o bloqueio seja liberado, quando ele poderá obter rapidamente o bloqueio e continuar. Esse mecanismo evita a sobrecarga de suspender e reiniciar a thread; contudo, se a espera for muito longa, esta ação poderá ocupar o processador e consumir recursos, além de não realizar qualquer trabalho útil, o que também impactará o desempenho de seu aplicativo.

Internamente, a instrução *lock* utiliza a classe *System.Threading.Monitor* para bloquear o objeto especificado. A classe *Monitor* segue um algoritmo inteligente para minimizar a possível sobrecarga associada ao bloqueio de um objeto e à espera pela liberação desse bloqueio. Quando uma thread solicita um bloqueio, se o mesmo objeto estiver atualmente bloqueado por outra thread, a thread que estiver aguardando “girará o processador” por um pequeno número de iterações. Se o bloqueio não for obtido durante esse período, a thread será colocada em modo de repouso e será suspensa. A thread é reiniciada quando o bloqueio é liberado. Por conseguinte, as esperas curtas respondem melhor enquanto as esperas mais prolongadas não impactam demasiadamente o desempenho de outras threads.

A palavra-chave *lock* é adequada a muitos cenários simples, mas, em algumas situações, você pode enfrentar exigências mais complexas. A TPL dispõe de algumas primitivas de sincronização adicionais para tratar dessas situações. As seções a seguir resumem algumas dessas primitivas. As primitivas foram projetadas para operar não somente com a TPL, mas com qualquer código multitarefas. Todas elas estão localizadas no namespace *System.Threading*.



**Nota** Desde a sua primeira versão, o .NET Framework tem mantido um conjunto eficiente de primitivas de sincronização. A seção a seguir descreve apenas as novas primitivas incluídas como parte da TPL. Ocorre certa sobreposição entre as novas primitivas e aquelas já fornecidas. Quando houver sobreposição de funcionalidades, use as primitivas existentes na TPL, porque 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 fora do escopo 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) no .NET Framework Developers Guide (Guia de Desenvolvedores do .NET Framework), fornecido como parte da documentação com o Visual Studio 2010.



**Importante** Não utilize as primitivas de sincronização e bloqueio para substituir um bom design ou a prática de programação adequada. O .NET Framework dispõe de vários outros mecanismos para maximizar o paralelismo, além de reduzir a possível sobrecarga associada ao bloqueio de dados. Por exemplo, se várias tarefas precisarem atualizar informações comuns em uma coleção, você poderá utilizar o armazenamento local das threads (TLS – thread-local storage) para guardar os dados utilizados pelas threads que executarem cada tarefa. Para criar uma variável no TLS, declare-a como um membro estático de classe e atribua a essa variável um prefixo com o atributo *ThreadStatic*, como a seguir:

```
[ThreadStatic]
static Hashtable privateDataForThread;
```

Embora declarada como estática, a variável *Hashtable* não é compartilhada por todas as instâncias da classe em que está definida. Em vez disso, uma cópia da variável estática é mantida no TLS. Se duas tarefas simultâneas acessarem essa variável, cada uma poderá obter uma cópia própria no TLS. Quando as tarefas terminarem, poderão retornar sua cópia dos dados estáticos e você poderá agragar esses resultados. Uma discussão detalhada sobre o TLS está fora do escopo deste livro, mas para obter mais informações, consulte a documentação fornecida com o Visual Studio 2010.

## Primitivas de sincronização na Task Parallel Library

A maioria das primitivas de sincronização assume a forma de mecanismos de bloqueio, que restringem o acesso a um recurso enquanto uma thread mantiver o bloqueio. A TPL tem suporte para diversas técnicas de bloqueio, que você pode utilizar para implementar diferentes estilos de acesso simultâneo, desde os bloqueios exclusivos simples (onde uma única thread tem acesso exclusivo a um recurso) até os semáforos (onde várias threads podem acessar simultaneamente um recurso, mas de modo controlado), até os bloqueios do tipo leitor/escritor (reader/writer) que permitem a diferentes threads compartilharem o acesso somente leitura a um recurso, além de garantir o acesso exclusivo a uma thread que precisa modificar o recurso.

### Classe *ManualResetEventSlim*

A classe *ManualResetEventSlim* oferece uma função que permite que uma ou mais threads aguardem um evento. Um objeto *ManualResetEventSlim* pode se encontrar em um de dois estados: *signaled* (true) e *unsignedaled* (false). Uma thread cria um objeto *ManualResetEventSlim* e especifica seu estado inicial. Outras threads podem esperar que o objeto *ManualResetEventSlim* seja sinalizado, ao chamar o método *Wait*. Se o objeto *ManualResetEventSlim* estiver no estado *unsignedaled*, o método *Wait* bloqueará as threads. Outra thread pode mudar o estado do objeto *ManualResetEventSlim* para *signaled* ao chamar o método *Set*. Essa ação libera todos as threads que estão aguardando no objeto *ManualResetEventSlim*, que pode, então, continuar a sua execução. O método *Reset* altera o estado de um objeto *ManualResetEventSlim* novamente para *unsignedaled*.

Enquanto aguarda um evento, uma thread "gira". Entretanto, se a espera ultrapassar determinado número de ciclos de rotação, a thread será suspensa e entrega o processador\*, de modo semelhante à classe *Monitor*, descrita anteriormente. Você pode especificar o número de ciclos de rotação que

\* N. de R. T.: Operação conhecida como *Yield*.

devem ocorrer antes da suspensão da thread no construtor para um objeto *ManualResetEventSlim*, e pode determinar quantos ciclos de rotação serão executados antes da suspensão da thread, ao consultar a propriedade *SpinCount*. Para verificar o estado de um objeto *ManualResetEventSlim*, examine a propriedade booleana *IsSet*.

O exemplo a seguir cria duas tarefas que acessam a variável de inteiro *i*. A primeira tarefa utiliza um loop *do/while* para exibir várias vezes o valor de *i* e incrementá-lo até que o valor de *i* alcance 10. A segunda tarefa aguarda que a primeira termine de atualizar *i*; em seguida, ela lê o valor de *i* e o exibe. As tarefas utilizam um objeto *ManualResetEventSlim* para coordenar suas atividades. O objeto *ManualResetEventSlim* é inicializado com o estado *unsignaled* (false) e ele especifica que uma thread aguardando nesse objeto pode girar 100 vezes antes de sua suspensão. A primeira tarefa sinaliza o objeto *ManualResetEventSlim* no final do loop. A segunda tarefa espera que o objeto *ManualResetEventSlim* seja sinalizado. Portanto, a segunda tarefa só lerá *i* quando ele atingir 10.

```
ManualResetEventSlim resetEvent = new ManualResetEventSlim(false, 100);
int i = 0;

Task t1 = Task.Factory.StartNew(() =>
{
 do
 {
 Console.WriteLine("Task t1. i is {0}", i);
 i++;
 } while (i < 10);
 resetEvent.Set();
});

Task t2 = Task.Factory.StartNew(() =>
{
 resetEvent.Wait();
 Console.WriteLine("Task t2. i is {0}", i);
});

Task.WaitAll(t1, t2);
```

A saída desse código é parecida com a seguinte:

```
Task t1. i is 0
Task t1. i is 1
Task t1. i is 2
Task t1. i is 3
Task t1. i is 4
Task t1. i is 5
Task t1. i is 6
Task t1. i is 7
Task t1. i is 8
Task t1. i is 9
Task t2. i is 10
```

## Classe *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 threads que acessam os recursos no pool chamam primeiramente 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 thread deve chamar o método *Release* no objeto *SemaphoreSlim*. Essa ação incrementa o valor do semáforo.

Se uma thread chamar o método *Wait* e o resultado da decrementação do valor do objeto *SemaphoreSlim* der um valor negativo, a thread aguardará que outra thread chame o método *Release*. Inicialmente, a thread gira, mas será suspensa se o tempo de espera for muito longo.

A classe *SemaphoreSlim* também fornece a propriedade *CurrentCount*, que pode ser utilizada para determinar a probabilidade de uma operação *Wait* obter êxito imediatamente ou se resultará em bloqueio. O exemplo a seguir demonstra como criar um objeto *SemaphoreSlim* para proteger um pool de três recursos compartilhados. As tarefas simultâneas podem chamar o método *Wait* antes de acessarem um recurso desse pool, e chamam o método *Release* quando terminam. Assim, não mais do que três tarefas poderão usar um recurso em determinado momento – a quarta tarefa ficará bloqueada até que uma das três anteriores chame o método *Release*.

```
// Objeto SemaphoreSlim compartilhado pelas threads
SemaphoreSlim semaphore = new SemaphoreSlim(3);

Task t1 = Task.Factory.StartNew(() =>
{
 semaphore.Wait();
 // Acesse um recurso no pool
 semaphore.Release();
});

Task t2 = Task.Factory.StartNew(() =>
{
 semaphore.Wait();
 // Acesse um recurso no pool
 semaphore.Release();
});

Task t3 = Task.Factory.StartNew(() =>
{
 semaphore.Wait();
 // Acesse um recurso no pool
 semaphore.Release();
});
```

```
Task t4 = Task.Factory.StartNew(() =>
{
 // Esta tarefa será bloqueada até que uma das três tarefas anteriores chame o método Release
 semaphore.Wait();
 // Acessa um recurso do pool
 semaphore.Release();
});

Task.WaitAll(t1, t2, t3, t4);
```

## Classe CountdownEvent

Imagine a classe *CountdownEvent* como se fosse um cruzamento entre o oposto de um semáforo e um evento de redefinição (reset) manual. Ao criar um objeto *CountdownEvent*, uma thread especifica um valor de estado inicial (um inteiro não negativo). Uma ou mais threads podem chamar o método *Wait* do objeto *CountdownEvent*, e, se seu valor for diferente de zero, as threads serão bloqueadas. (A thread gira inicialmente, e depois é suspensa.) O método *Wait* não decrementa o valor do objeto *CountdownEvent*; em vez disso, outras threads podem chamar o método *Signal* para reduzir o valor. Quando o valor do objeto *CountdownEvent* atinge zero, todas as threads bloqueadas são sinalizadas e poderão retomar a sua execução.

Uma thread pode redefinir o valor de um objeto *CountdownEvent* com o valor especificado em seu construtor por meio método *Reset* e pode aumentar o valor ao chamar o método *AddCount*. Para consultar o valor de um objeto *CountdownEvent* e saber se uma chamada ao método *Wait* provavelmente o bloqueará, examine a propriedade *CurrentCount*. O código a seguir cria um objeto *CountdownEvent* que deve ser sinalizado cinco vezes para que as threads bloqueadas em espera possam continuar. O *countDownWaitTask* aguarda esse objeto, e a tarefa *countDownSignalTask* o sinaliza cinco vezes.

```
CountdownEvent countDown = new CountdownEvent(5);

Task countDownWaitTask = Task.Factory.StartNew(() =>
{
 countDown.Wait();
 Console.WriteLine("CountdownEvent has been signaled 5 times");
});

Task countDownSignalTask = Task.Factory.StartNew(() =>
{
 for (int i = 0; i < 5; i++)
 {
 Console.WriteLine("Signaling CountdownEvent");
 countDown.Signal();
 }
});

Task.WaitAll(countDownWaitTask, countDownSignalTask);
```

Veja a seguir a saída desse código:

```
Signaling CountdownEvent
Signaling CountdownEvent
Signaling CountdownEvent
Signaling CountdownEvent
Signaling CountdownEvent
CountdownEvent has been signaled 5 times
```

## Classe *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.

Para ler um recurso, uma thread chama o método *EnterReadLock* de um objeto *ReaderWriterLockSlim*. Essa ação captura um bloqueio de leitura no objeto. Quando a thread termina com o recurso, ela chama o método *ExitReadLock*, que libera o bloqueio de leitura. Várias threads podem ler o mesmo recurso ao mesmo tempo, e cada thread obtém um bloqueio de leitura próprio.

Para modificar o recurso, uma thread chama o método *EnterWriteLock* do mesmo objeto *ReaderWriterLockSlim*, para obter um bloqueio de escrita. Se uma ou mais threads tiverem atualmente um bloqueio de leitura para esse objeto, o método *EnterWriteLock* bloqueará até que todos eles estejam liberados. A thread poderá, então, modificar o recurso e chamar o método *ExitWriteLock* para liberar o bloqueio de escrita. Um objeto *ReaderWriterLockSlim* tem apenas um bloqueio de escrita. Se outra thread tentar obter o bloqueio de escrita, será bloqueada até que a primeira thread libere esse bloqueio.

Para garantir que as threads de escrita não fiquem bloqueadas indefinidamente, assim que uma thread solicitar o bloqueio de escrita, todas as chamadas subsequentes ao *EnterReadLock* serão bloqueadas até que o bloqueio de escrita tenha sido obtido e liberado.

O mecanismo de bloqueio é semelhante ao utilizado pelas outras primitivas descritas nesta seção; ele gira o processador durante um número especificado de ciclos, antes de suspender a thread, se ela ainda estiver bloqueada.

O código a seguir cria um objeto *ReaderWriterLockSlim* para proteger um recurso compartilhado e depois cria três tarefas. Duas das tarefas obtêm um bloqueio de leitura sobre o objeto, e a terceira obtém um bloqueio de escrita. As duas tarefas, *readerTask1* e *readerTask2*, podem acessar simultaneamente o recurso compartilhado, mas a tarefa *writerTask* só pode acessar o recurso quando as tarefas *readerTask1* e *readerTask2* liberarem os respectivos bloqueios de leitura.

```

 ReaderWriterLockSlim readerWriterLock = new ReaderWriterLockSlim();

 Task readerTask1 = Task.Factory.StartNew(() =>
 {
 readerWriterLock.EnterReadLock();
 // Lê o recurso compartilhado
 readerWriterLock.ExitReadLock();
 });

 Task readerTask2 = Task.Factory.StartNew(() =>
 {
 readerWriterLock.EnterReadLock();
 // Lê o recurso compartilhado
 readerWriterLock.ExitReadLock();
 });

 Task writerTask = Task.Factory.StartNew(() =>
 {
 readerWriterLock.EnterWriteLock();
 // Escreve no recurso compartilhado
 readerWriterLock.ExitWriteLock();
 });

 Task.WaitAll(readerTask1, readerTask2, writerTask);

```

## Classe *Barrier*

A classe *Barrier* permite interromper temporariamente a execução de um conjunto de threads em um ponto específico, em um aplicativo, e só prosseguir quando todas as threads tiverem alcançado esse ponto. Essa classe é útil para sincronizar as threads que precisam executar uma série de operações simultâneas em etapas com dependências entre si.

Ao criar um objeto *Barrier*, uma thread especifica o número de threads no conjunto que será sincronizado. Considere esse valor um contador de threads mantido internamente, dentro da classe *Barrier*. Esse valor pode ser corrigido posteriormente, ao chamar os métodos *AddParticipant* ou *RemoveParticipant*. Ao atingir um ponto de sincronização, uma thread chama o método *SignalAndWait* do objeto *Barrier*, que decrementa o contador de threads dentro do objeto *Barrier*. Se esse contador for maior que zero, a thread será bloqueada. Somente quando o contador atingir zero todas as threads 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 threads que ela sincronizará, e a propriedade *ParticipantsRemaining*, que indica quantas threads devem chamar o método *SignalAndWait* para que a barreira seja erguida e as threads bloqueadas possam continuar operando.

Você também pode especificar um delegate no construtor *Barrier*. Esse delegate pode fazer referência a um método a ser executado quando todas as threads alcançarem a barreira. O objeto *Barrier* é passado com um parâmetro para esse método. A barreira não é levantada e as threads não são liberadas até o término desse método.

O exemplo a seguir cria um objeto *Barrier* que sincroniza três threads e fornece uma expressão lambda que é executada quando o *Barrier* é sinalizado. As tarefas *t1*, *t2* e *t3* chamam o método *SignalAndWait* nessa barreira. As três tarefas só podem prosseguir após sinalizarem a barreira e depois que o delegate especificado pelo objeto *Barrier* estiver concluído.

```
Barrier barrier = new Barrier(3, (x) =>
{
 Console.WriteLine("All tasks have reached the barrier");
});

Task t1 = Task.Factory.StartNew(() =>
{
 Console.WriteLine("Task t1 starting");
 barrier.SignalAndWait();
 Console.WriteLine("Task t1 continuing after the barrier");
});

Task t2 = Task.Factory.StartNew(() =>
{
 Console.WriteLine("Task t2 starting");
 barrier.SignalAndWait();
 Console.WriteLine("Task t2 continuing after the barrier");
});

Task t3 = Task.Factory.StartNew(() =>
{
 Console.WriteLine("Task t3 starting");
 barrier.SignalAndWait();
 Console.WriteLine("Task t3 continuing after the barrier");
});

Task.WaitAll(t1, t2, t3);
```

A saída desse código é parecida com a seguinte (a sequência na qual as tarefas iniciam e prosseguem depois que a barreira for erguida pode variar):

```
Task t1 starting
Task t2 starting
Task t3 starting
All tasks have reached the barrier
Task t3 continuing after the barrier
Task t1 continuing after the barrier
Task t2 continuing after the barrier
```

## Primitivas de cancelamento e sincronização

As classes *ManualResetEventSlim*, *SemaphoreSlim*, *CountdownEvent* e *Barrier* têm suporte ao cancelamento seguindo o modelo de cancelamento descrito no Capítulo 27. 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*.

**Nota** Se a operação de espera estiver sendo executada por uma tarefa, a exceção *OperationCanceledException* será encapsulada em uma *AggregateException*, como descrito no Capítulo 27.

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, o manipulador catch *OperationCanceledException*, será executado.

```
CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
CancellationToken cancellationToken = cancellationTokenSource.Token;

...
// Semáforo que protege um pool de 3 recursos
SemaphoreSlim semaphoreSlim = new SemaphoreSlim(3);

...
// Aguarda no semáforo, e captura a exceção 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 multitarefas é armazenar e recuperar dados em uma coleção. Por padrão, as classes de coleção padrão fornecidas no .NET Framework não são thread-safe (protegidas quanto a threads), embora você possa utilizar as primitivas de sincronização descritas na seção anterior para encapsular um 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 4.0 contém um pequeno conjunto de classes de coleção e interfaces thread-safe no namespace *System.Collections.Concurrent*, projetadas especificamente para serem utilizadas com a TPL. A tabela a seguir descreve resumidamente esses tipos.

Classe	Descrição
<code>ConcurrentBag&lt;T&gt;</code>	Essa é uma classe genérica para armazenar uma coleção desordenada de itens. Contém métodos para inserir ( <code>Add</code> ), remover ( <code>TryTake</code> ) e examinar ( <code>TryPeek</code> ) 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 <code>foreach</code> .
<code>ConcurrentDictionary&lt; TKey, TValue &gt;</code>	Essa classe implementa uma versão thread-safe da classe de coleção genérica <code>Dictionary&lt; TKey, TValue &gt;</code> , descrita no Capítulo 18, "Apresentando genéricos". Ela fornece os métodos <code>TryAdd</code> , <code>ContainsKey</code> , <code>TryGetValue</code> , <code>TryRemove</code> e <code>TryUpdate</code> , que você pode utilizar para adicionar, consultar, remover e modificar itens no dicionário.
<code>ConcurrentQueue&lt;T&gt;</code>	Essa classe apresenta uma versão thread-safe da classe genérica <code>Queue&lt;T&gt;</code> , descrita no Capítulo 18. Ela fornece os métodos <code>Enqueue</code> , <code>TryDequeue</code> e <code>TryPeek</code> , que você pode utilizar para adicionar, remover e consultar itens na fila.
<code>ConcurrentStack&lt;T&gt;</code>	Essa é uma implementação thread-safe da classe genérica <code>Stack&lt;T&gt;</code> , também descrita no Capítulo 18. Ela fornece métodos, como <code>Push</code> , <code>TryPop</code> e <code>TryPeek</code> , que você pode utilizar para enviar, remover e consultar itens na pilha.
<code>IProducerConsumerCollection&lt;T&gt;</code>	<p>Essa interface define métodos para implementar classes que apresentam um comportamento produtor/consumidor. Um produtor adiciona itens a uma coleção, e um consumidor lê (e possivelmente remove) itens da mesma coleção. Costuma ser utilizada para definir tipos que operam com a classe <code>BlockingCollection&lt;T&gt;</code>, descrita a seguir, nesta tabela.</p> <p>A interface <code>IProducerConsumerCollection</code> define métodos para adicionar (<code>TryAdd</code>) itens a uma coleção, remover (<code>TryTake</code>) itens de uma coleção e obter um enumerador (<code>GetEnumerator</code>) para que um aplicativo possa iterar sobre uma coleção. (A interface também define outros métodos e propriedades.)</p> <p>Você pode implementar essa interface e criar as próprias classes personalizadas de coleções concorrentes. As classes <code>ConcurrentBag&lt;T&gt;</code>, <code>ConcurrentQueue&lt;T&gt;</code> e <code>ConcurrentStack&lt;T&gt;</code> implementam essa interface.</p>
<code>BlockingCollection&lt;T&gt;</code>	<p>Essa classe é útil para construir aplicativos baseados em produtores e consumidores que acessam a mesma coleção. O parâmetro <code>type</code> faz referência a um tipo que implementa a interface <code>IProducerConsumerCollection&lt;T&gt;</code> e adiciona capacidades de bloqueio a essa classe.</p> <p>Elá alcança esse feito ao atuar como um adaptador thread-safe e fornece métodos, como <code>Add</code> e <code>Take</code>, que encapsulam chamadas aos métodos <code>TryAdd</code> e <code>TryTake</code> na coleção subjacente, ao utilizar um código que cria e usa as primitivas de sincronização descritas anteriormente. A classe <code>BlockingCollection&lt;T&gt;</code> também pode limitar o número de itens armazenados na coleção subjacente.</p>

O código a seguir mostra como utilizar um objeto *BlockingCollection*<T> para encapsular uma instância de um tipo definido pelo usuário, chamado *MyCollection*<T>, que implementa a interface *IProducerConsumerCollection*. Ele limita a 1.000 o número de itens contidos na coleção subjacente.

```
class MyCollection<T> : IProducerConsumerCollection<T>
{
 // Detalhes da implementação não mostrados
 ...
}

...
// Crie uma instância de MyCollection<T>,
// e encapsule-a em um objeto BlockingCollection<T>
MyCollection<int> intCollection = new MyCollection<int>();
BlockingCollection<int> collection = new BlockingCollection<int>(myCollection, 1000);
```

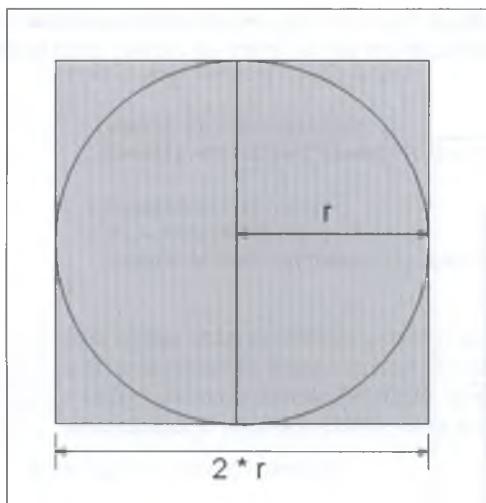
**Nota** Você também pode instanciar um objeto *BlockingCollection*<T> sem especificar uma classe de coleção. Nesse caso, o objeto *BlockingCollection*<T> cria um objeto *ConcurrentQueue*<T> internamente.

Adicionar o acesso thread-safe 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 parallelizar um conjunto de operações que exigem acesso a uma coleção compartilhada.

## Utilizando uma coleção concorrente e um bloqueio para implementar o 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*; 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 da classe de coleção concurrent 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 na imagem a seguir:



A área do quadrado,  $Q$ , é calculada desta maneira:

$$(2 * r) * (2 * r)$$

ou

$$4 * r * r$$

A área do círculo,  $C$ , é calculada assim:

$$\pi * r * r$$

Ao utilizar essas áreas, você pode calcular PI como a seguir:

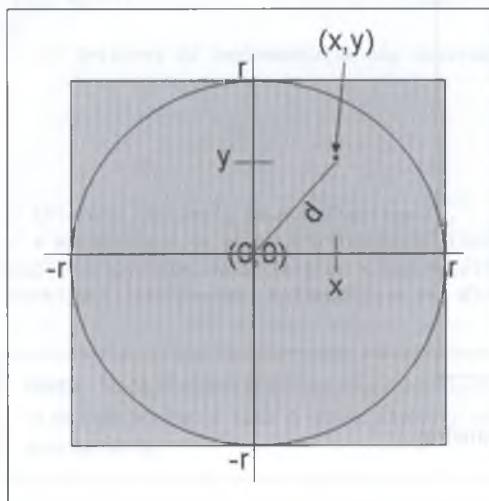
$$4 * C / Q$$

O truque é determinar o valor de  $C / Q$ . É exatamente aí que entra em ação a amostragem estatística.

Para isso, gere um conjunto de pontos aleatórios posicionados dentro do quadrado e conte 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 reside dentro do círculo? Para ajudá-lo a vislumbrar a solução, desenhe o quadrado em um pedaço de papel milimetrado, com o centro do quadrado posicionado na origem, o ponto  $(0, 0)$ . A partir de 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, então 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 precisa gerar pares de números aleatórios entre 0 e  $r$ . Essa é a abordagem 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 quaisquer outros programas ou serviços entre os exercícios, porque eles poderão interferir nos resultados visualizados.

### Calcule PI usando uma única thread

1. Inicialize o Microsoft Visual Studio 2010 se ele ainda não estiver em execução.
2. Abra a solução CalculatePI, localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 28\CalculatePI em sua pasta Documentos.
3. No Solution Explorer, 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*, parecido com este:

```
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*, que calculará PI por meio do algoritmo geométrico descrito anteriormente, neste exercício. O valor é retornado como um *double* e exibido. Em seguida, o código chama o método *ParallelPI*, que 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: Implemente 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 gerará um grande conjunto de coordenadas e calculará 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*. Quando 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 existir memória suficiente, você poderá aumentar o valor de *NUMPOINTS*. De modo semelhante, se você detectar que o aplicativo lança exceções *OutOfMemoryException* ao ser executado, você poderá reduzir esse valor.

Armazene a distância de cada ponto a partir da origem na coleção `pointsList List<double>`. 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 ela acusou dentro do círculo. Você adicionará o código que realmente efetua o cálculo ao bloco `try` nas próximas etapas.

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 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 utiliza 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 abordagem 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, para manter a CPU ocupada:

```
private static void doAdditionalProcessing()
{
 Thread.SpinWait(SPINWAITS);
}
```

*SPINWAITS* é outra constante definida no início da classe *Program*.

7. No método *SerialPI*, no bloco *try*, adicione a instrução *foreach* mostrada em negrito a seguir depois do bloco *for*:

```
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 as seguintes instruções, mostradas em negrito, ao bloco *try* depois do bloco *foreach*:

```
try
{
 for (int points = 0; points < NUMPOINTS; points++)
 {
 ...
 }

 foreach (double datum in pointsList)
 {
 ...
 }

 double pi = 4.0 * numPointsInCircle / NUMPOINTS;
 return pi;
}
```

Essas instruções calculam 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 é exibido. (Você pode ignorar os resultados do método *ParallelPI* porque você ainda não escreveu o código desse método.)



**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 óbvia que pode ser paralelizada. É exatamente 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 bloco *try* para calcular PI.
3. No bloco *try*, exclua o comentário e remova a instrução *return*. Adicione ao bloco *try* a instrução *ParallelFor* mostrada em negrito a seguir:

```
try
{
 Parallel.For (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 no loop *for* no método *SerialPI*. O corpo do loop *for* original é encapsulado em uma expressão lambda.

4. Adicione seguinte código, mostrado em negrito, ao bloco *try* depois da instrução *Parallel.For*. O código é exatamente igual às instruções correspondentes no método *SerialPI*.

```
try
{
 Parallel.For (...);

 foreach (double datum in pointsList)
 {
 if (datum <= RADIUS)
 {
 numPointsInCircle++;
 }
 }

 double pi = 4.0 * numPointsInCircle / NUMPOINTS;
 return pi;
}
```

5. No menu *Debug*, clique em *Start Without Debugging*.

O programa é executado. A imagem a seguir mostra a saída característica:



O valor calculado pelo método *SerialPI* deve ser exatamente igual ao anterior. Entretanto, 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 ele 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* realmente contiver o número esperado de itens, execute o aplicativo novamente. Você deve descobrir que ela contém menos itens do que o previsto na maioria (mas não necessariamente em todas) das execuções.

## 6. 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 genérico *List<double>*. 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 simultâneos. 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* porque elas são thread-safe. A classe genérica *ConcurrentBag<T>* nesse namespace provavelmente é a mais adequada para uso nesse exemplo.

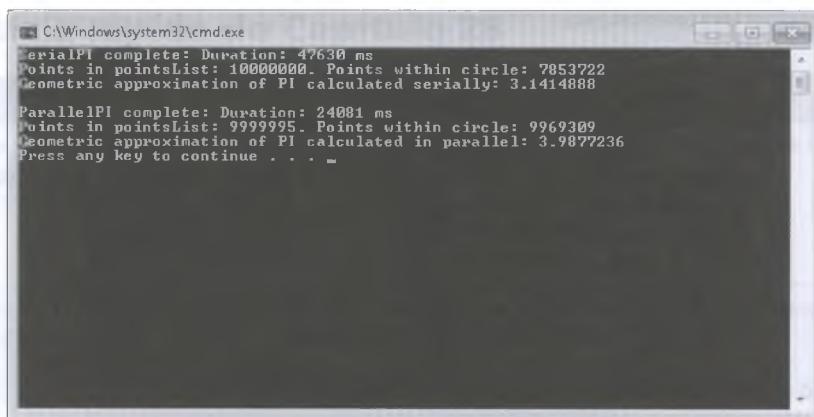
### Use uma coleção thread-safe

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. 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>*, mostrado em negrito a seguir:

```
static double ParallelPI()
{
 ConcurrentBag<double> pointsList = new ConcurrentBag <double>();
 Random random = ...;
 ...
}
```

3. 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: 47630 ms
Points in pointsList: 10000000. Points within circle: 7953722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 24081 ms
Points in pointsList: 9999995. Points within circle: 9969309
Geometric approximation of PI calculated in parallel: 3.9877236
Press any key to continue . . .
```

Dessa 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 ocupado pelo método *ParallelPI* aumentou consideravelmente. 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.

#### 4. Feche a janela do console e retorne ao Visual Studio.

Você já tem o número correto de pontos na coleção *pointsList*, mas os valores desses pontos são suspeitos. O código na construção *Parallel.For* 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 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 na expressão lambda na instrução *Parallel.For* para proteger as chamadas ao *random.Next*, por meio de uma instrução *lock*. Especifique a coleção *pointsList* como o alvo do bloqueio, como mostrado em negrito a seguir:

```
static double ParallelPI()
{
 ...
 Parallel.For(0, NUMPOINTS, (x) =>
 {
 int xCoord;
 int yCoord;
```

```

 lock(pointsList)
 {
 xCoord = random.Next(RADIUS);
 yCoord = random.Next(RADIUS);
 }

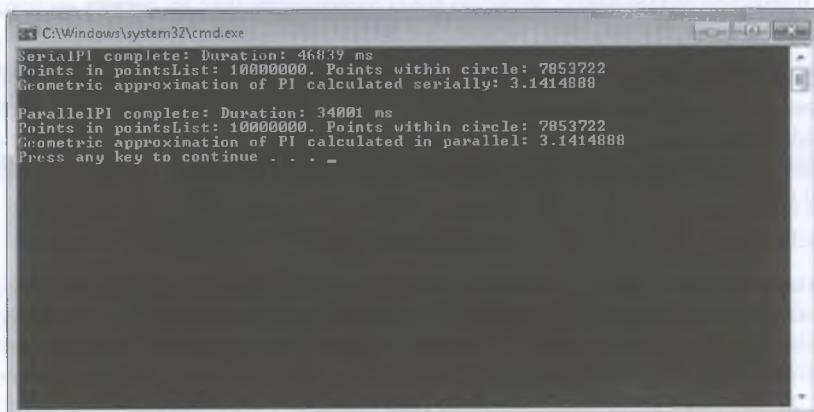
 double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
 pointsList.Add(distanceFromOrigin);
 doAdditionalProcessing();
}

...
}

```

**3.** No menu *Debug*, clique em *Start Without Debugging*.

Dessa 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.



**4.** Feche a janela do console e retorne ao Visual Studio.

Neste capítulo, 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 a TPL. Você também viu como é possível utilizar as classes de coleção concorrentes para manter coleções de dados de modo thread-safe.

- Se você quiser seguir para o próximo capítulo agora:

Mantenha o Visual Studio 2010 em execução e vá para o Capítulo 29.

- Se você quiser sair do Visual Studio 2010:

No menu *File*, clique em *Exit*. Se vir uma caixa de diálogo *Save*, clique em *Yes* e salve o projeto.

## Referência rápida do Capítulo 28

Para	Faça isto
Paralelizar uma consulta LINQ	Especifique o método de extensão <i>AsParallel</i> com a origem de dados na consulta. Por exemplo:  <pre>var over100 = from n in numbers.AsParallel()               where ...               select n;</pre>
Permitir o 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 o acesso exclusivo thread-safe a dados compartilhados	Use a instrução <i>lock</i> para assegurar o 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 instruí-las a esperar 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 durante 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	Use um objeto <i>SemaphoreSlim</i> . Especifique o número de itens no 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(); // Acesse um recurso do pool  ... semaphore.Release();</pre>

(continua)

Para	Faça isto
Fornecer acesso de escrita exclusivo a um recurso, mas acesso de leitura compartilhado	<p>Use um objeto <code>ReaderWriterLockSlim</code>. Antes de ler o recurso compartilhado, chame o método <code>EnterReadLock</code>. Quando terminar, chame o método <code>ExitReadLock</code>. Antes de escrever no recurso compartilhado, chame o método <code>EnterWriteLock</code>. Quando você terminar a operação de escrita, chame o método <code>ExitWriteLock</code>. Por exemplo:</p> <pre>ReaderWriterLockSlim readerWriterLock = new ReaderWriterLockSlim();  Task readerTask = Task.Factory.StartNew(() =&gt; {     readerWriterLock.EnterReadLock();     // Leia o recurso compartilhado     readerWriterLock.ExitReadLock(); });  Task writerTask = Task.Factory.StartNew(() =&gt; {     readerWriterLock.EnterWriteLock();     // Escreva no recurso compartilhado     readerWriterLock.ExitWriteLock(); });</pre>
Cancelar uma operação de espera de 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); ... // Espere o semáforo e lance uma OperationCanceledException se // outra thread chamar Cancel em cancellationTokenSource semaphore.Wait(cancellationToken);</pre>
Criar um objeto de coleção thread-safe	<p>Dependendo da funcionalidade necessária à coleção, use uma das classes do namespace <code>System.Collections.Concurrent</code> (<code>ConcurrentBag&lt;T&gt;</code>, <code>ConcurrentDictionary&lt;TKey, TValue&gt;</code>, <code>ConcurrentQueue&lt;T&gt;</code> ou <code>ConcurrentStack&lt;T&gt;</code>) ou crie uma classe própria que implemente a interface <code>IProducerConsumerCollection&lt;T&gt;</code> e encapsule uma instância desse tipo em um objeto <code>BlockingCollection&lt;T&gt;</code>. Por exemplo:</p> <pre>class MyCollection&lt;T&gt; : IProducerConsumerCollection&lt;T&gt; {     // Detalhes da implementação não mostrados     ... }  // Crie uma instância de MyCollection&lt;T&gt;, // e encapsule-a em um objeto BlockingCollection&lt;T&gt; MyCollection&lt;int&gt; intCollection = new MyCollection&lt;int&gt;(); BlockingCollection&lt;int&gt; collection = new BlockingCollection&lt;int&gt;(myCollection, 1000);</pre>

## Capítulo 29

# Criando e utilizando um Web service

Neste capítulo, você vai aprender a:

- Criar Web services SOAP e REST que expõem métodos Web simples.
- Exibir a descrição de um Web service SOAP utilizando o Internet Explorer.
- Projetar classes que podem ser passadas como parâmetros a um método Web e retornadas de um método Web.
- Criar proxies para Web services SOAP e REST em um aplicativo cliente.
- Chamar um método Web REST por meio do Internet Explorer.
- Chamar métodos Web a partir de um aplicativo cliente.

Os capítulos anteriores demonstraram como construir aplicativos desktop, que podem executar diversas tarefas. Entretanto, nos dias atuais, raros sistemas operam isoladamente. Uma organização realiza operações comerciais e, para isso, costuma ter aplicativos com suporte para essa funcionalidade empresarial. Uma necessidade cada vez mais comum é construir novas soluções que possam reutilizar grande parte dessa funcionalidade e proteger o investimento da organização na construção ou na compra dos componentes de software subjacentes. Esses componentes e serviços podem ser construídos por meio de diferentes tecnologias e linguagens de programação, e funcionar em uma coleção de computadores interconectados por rede. Além disso, como estamos na era da Internet, uma organização pode compor soluções que incorporem diversos serviços de terceiros. O desafio é estabelecer o modo de combinação desses componentes para que possam se comunicar e colaborar de modo harmônico.

Os Web services oferecem uma solução viável. Ao utilizá-los, é possível construir sistemas distribuídos a partir de elementos disseminados na Internet – bancos de dados, serviços comerciais e muito mais. Os componentes e serviços são hospedados por um servidor Web que recebe solicitações de um aplicativo cliente, analisa essas solicitações e envia o comando correspondente para o componente ou serviço. A resposta é redirecionada pelo servidor Web para o aplicativo cliente.

O objetivo deste capítulo é mostrar como projetar, construir e testar Web services que podem ser acessados pela Internet e integrados em aplicativos distribuídos. Você também aprenderá a construir um aplicativo cliente que utiliza os métodos expostos por um Web service.



**Nota** O propósito deste capítulo é fornecer uma introdução muito básica aos Web services e ao Microsoft Windows Communication Foundation (WCF). Se quiser informações detalhadas sobre como o WCF funciona e como construir serviços seguros utilizando o WCF, consulte um livro como o *Microsoft Windows Communication Foundation Step By Step*, publicado pela Microsoft Press, 2007.

## O que é um Web service?

Um Web service é um componente de negócio que fornece uma funcionalidade reutilizável para os clientes, ou consumidores. Um Web service pode ser pensado como um componente com acessibilidade verdadeiramente global – se tiver os direitos de acesso apropriados, você poderá utilizar um Web service de qualquer lugar do mundo contanto que seu computador esteja conectado à Internet. Os Web services utilizam um protocolo padrão, aceito e bem entendido, o Hypertext Transfer Protocol (HTTP), para transmitir dados e um formato de dados portável que é baseado em XML. O HTTP e o XML são tecnologias padronizadas que podem ser utilizadas por outros ambientes de programação fora do Microsoft .NET Framework. Com o Microsoft Visual Studio 2010, você pode construir Web services no Microsoft Visual C++, no Microsoft Visual C# ou no Microsoft Visual Basic. Entretanto, no que diz respeito ao aplicativo cliente, não importa a linguagem utilizada para criar o Web service, nem como o Web service executa suas tarefas. Aplicativos cliente em execução em um ambiente totalmente diferente, como o Java, podem utilizá-los. O contrário também é válido: você pode construir Web services em Java e escrever um aplicativo cliente em C#.

## O papel do Windows Communication Foundation

O Windows Communication Foundation, ou WCF, surgiu como parte da versão 3.0 do .NET Framework. O Visual Studio dispõe de um conjunto de templates que você pode utilizar para construir Web services por meio do WCF. Contudo, os Web services são apenas uma tecnologia que você pode empregar para criar aplicativos distribuídos para os sistemas operacionais Windows. Outros incluem Enterprise Services, .NET Framework Remoting e Microsoft Message Queue (MSMQ). Se você estiver construindo um aplicativo distribuído para Windows, que tecnologia você deve usar, e que nível de complexidade você enfrentará posteriormente para mudar de tecnologia, se necessário? O objetivo do WCF é oferecer um modelo de programação unificada para muitas dessas tecnologias, a fim de permitir a construção de aplicativos o mais independentemente possível do mecanismo subjacente utilizado para conectar serviços e aplicativos. (Observe que o WCF se aplica tanto aos serviços em funcionamento em ambientes não Web quanto à World Wide Web.) É muito difícil, se não impossível, desassociar totalmente a estrutura de programação de um aplicativo ou serviço de sua infraestrutura de comunicação, mas, na maioria das situações, o WCF permite que você chegue bem pertinho desse objetivo.

Resumindo a história: se você estiver pensando em construir aplicativos e serviços distribuídos para Windows, use o WCF. Os exercícios deste capítulo demonstrarão como fazer isso.

## Arquiteturas de Web Service

Existem duas arquiteturas comuns, utilizadas pelas organizações para implementar Web services: os serviços baseados em SOAP (Simple Object Access Protocol) e os serviços baseados no modelo REST (Representational State Transfer). As duas arquiteturas dependem do onipresente protocolo

HTTP (o protocolo utilizado pela Web para enviar e receber páginas HTML) e do esquema de endereçamento empregado pela Internet, mas elas o usam de várias maneiras. Se você estiver construindo soluções que incorporam Web services hospedados por organizações de terceiros, elas podem implementar esses Web services por meio de um desses modelos; portanto, compensa ter conhecimento sólido de ambos. As seções a seguir descrevem resumidamente essas arquiteturas.

## Web Services SOAP

Um Web service SOAP expõe funcionalidades ao utilizar o modelo tradicional de procedimentos; a grande diferença em relação a um aplicativo desktop comum é que os procedimentos são executados remotamente, no servidor Web. A perspectiva de um Web service sob o prisma de um aplicativo cliente é a de uma interface que dispõe de alguns métodos bem definidos, conhecidos como métodos Web. O aplicativo cliente envia solicitações para esses métodos Web pelos protocolos Internet padrão, passam os parâmetros em formato XML e recebem as respostas também em formato XML. Os métodos SOAP Web podem consultar e modificar dados.

### O papel do SOAP

O SOAP é o protocolo utilizado pelos aplicativos cliente para enviar solicitações e receber respostas dos Web services. O SOAP é um protocolo construído sobre o HTTP e define uma gramática XML para especificar os nomes dos métodos Web que um consumidor pode chamar em um Web service, a fim de definir os parâmetros e valores de retorno e descrever os tipos dos parâmetros e dos valores de retorno. Quando um cliente chama um Web service, ele deve especificar o método e os parâmetros utilizando essa gramática XML.

O SOAP é um padrão da indústria. Sua função é aprimorar a interoperabilidade entre plataformas. A força do SOAP é sua simplicidade e também o fato de que é baseado em outras tecnologias padrão da indústria, como HTTP e XML: a especificação do SOAP define várias coisas. As mais importantes são as seguintes:

- O formato de uma mensagem SOAP
- Como os dados devem ser codificados
- Como enviar mensagens
- Como processar respostas

As descrições dos detalhes exatos de como o SOAP funciona e do formato interno de uma mensagem SOAP estão além do escopo deste livro. É muito improvável que você precise alguma vez criar e formar mensagens SOAP manualmente porque muitas ferramentas de desenvolvimento, incluindo o Visual Studio 2010, automatizam esse processo, apresentando uma API amigável ao programador, para desenvolvedores construírem Web services e aplicativos cliente.

## O que é a linguagem de descrição de Web services?

O corpo de uma mensagem SOAP é um documento XML. Quando um aplicativo cliente invoca um método Web, o servidor Web espera que o cliente utilize um conjunto particular de tags para codificação dos parâmetros para o método. Como um cliente sabe quais tags, ou esquema XML, utilizar? A resposta é que, quando solicitado, espera-se que um Web service forneça uma descrição dele mesmo. A resposta do Web service é outro documento XML que descreve o Web service. Previsivelmente, esse documento é conhecido como descrição do Web service. O esquema XML utilizado por esse documento foi padronizado e é chamado Web Services Description Language (WSDL). Essa descrição fornece informações suficientes para que um aplicativo cliente possa construir uma solicitação SOAP em um formato que o servidor Web deve entender. Mais uma vez, os detalhes da WSDL estão além do escopo deste livro, mas o Visual Studio 2010 contém ferramentas que podem analisar sintaticamente a WSDL para um Web service de uma maneira mecânica. O Visual Studio 2010 utiliza, então, essas informações para definir uma classe proxy que um aplicativo cliente pode usar para converter chamadas de método comuns nessa classe proxy, em solicitações SOAP que o proxy envia pela Web. Essa é a abordagem que utilizaremos nos exercícios deste capítulo.

## Requisitos não funcionais dos Web services

Os esforços iniciais para definir os Web services e seus padrões associados concentraram-se nos aspectos funcionais do envio e recebimento de mensagens SOAP. Logo depois de os Web services terem se tornado uma tecnologia predominante para integrar serviços distribuídos, tornou-se evidente que existiam problemas que o SOAP e o HTTP sozinhos não podiam tratar. Essas questões dizem respeito a muitos requisitos não funcionais que são importantes em qualquer ambiente distribuído, porém, muito mais ao utilizar a Internet como a base de uma solução distribuída. Elas incluem:

- **Segurança** Como você garante que as mensagens SOAP que fluem entre um Web service e um cliente não foram interceptadas e alteradas no seu percurso pela Internet? Como pode ter certeza de que uma mensagem SOAP foi realmente enviada pelo consumidor ou Web service que afirma tê-la enviado, e não por algum site “mal-intencionado” que está tentando obter informações desonestamente? Como você pode limitar o acesso a um Web service para usuários específicos? Essas são questões de integridade, confidencialidade e autenticação de mensagens e são de fundamental interesse se você estiver criando aplicativos distribuídos que fazem uso da Internet.

No início de 1990, várias empresas que forneciam ferramentas para a criação de sistemas distribuídos formaram uma organização que mais tarde ficou conhecida como Organization for the Advancement of Structured Information Standards (OASIS). Quando as deficiências das primeiras infraestruturas de Web services tornaram-se aparentes, os membros da OASIS ponderaram esses problemas (e outras questões dos Web services) e produziram o que ficou conhecido como a especificação WS-Security, que descreve como proteger as mensagens enviadas por Web services. Os fornecedores que se inscrevem no WS-Security fornecem suas próprias implementações, que atendem a essa especificação, em geral, usando tecnologias como criptografia e certificados.

- **Política** Embora a especificação da WS-Security defina como fornecer segurança avançada, os desenvolvedores ainda precisam escrever um código para implementá-la. Os Web services criados por diferentes desenvolvedores apresentam frequentemente uma variação na precisão do mecanismo de segurança que eles resolveram implementar. Por exemplo, um Web service poderá usar apenas uma forma de criptografia relativamente frágil que pode ser facilmente quebrada. Um consumidor enviando informações altamente confidenciais para esse Web service provavelmente insistirá em um nível de segurança mais alto. Isso é uma forma de política (policy). Outros exemplos incluem a qualidade do serviço e a confiabilidade do Web service, que pode implementar vários graus de segurança, qualidade de serviço e confiabilidade, e cobrar do aplicativo cliente de acordo. O aplicativo cliente e o Web service podem negociar que nível de serviço usar com base nos requisitos e custo. Mas essa negociação requer que o cliente e o Web service tenham um entendimento comum das políticas disponíveis. A especificação WS-Policy fornece um modelo de finalidade geral e sintaxe correspondente para descrever e comunicar as políticas que o Web service implementa.
- **Roteamento e endereçamento** É útil para um Web service ser capaz de redirecionar uma solicitação de um Web service para um de vários computadores que hospedam instâncias do serviço. Por exemplo, muitos sistemas que suportam escalabilidade fazem uso de balanceamento de carga, nos quais as solicitações enviadas para um servidor Web são, na verdade, redirecionadas por esse servidor para outros computadores, a fim de distribuir a carga entre esses computadores. O servidor pode usar diversos algoritmos para tentar equilibrar a carga. O ponto importante é que esse redirecionamento seja transparente para o cliente que faz a solicitação ao Web service e o servidor que, em última instância, trata a solicitação precisa para saber onde enviar quaisquer respostas que ele gera. O redirecionamento das solicitações do Web service também é útil caso um administrador precise desligar um computador para realizar manutenção. As solicitações que teriam sido enviadas de outra forma para esse computador podem ser rerroteadas para um de seus pares. A especificação WS-Addressing descreve um framework para rotear as solicitações de Web service.

**Nota** Desenvolvedores referem-se coletivamente à WS-Security, à WS-Policy, ao WS-Addressing e a outras especificações WS como especificações WS-\*.

## Web Services REST

Diferentemente dos Web services SOAP, o modelo REST de Web services utiliza um esquema de navegação para representar os objetos de negócio e os recursos por uma rede. Por exemplo, uma organização pode fornecer acesso às informações dos empregados, expondo os detalhes de cada empregado como um único recurso, por meio de um esquema semelhante ao seguinte:

<http://northwind.com/employees/7>

O acesso a esse URL faz o Web service recuperar os dados do empregado 7. Esses dados podem ser retornados em diversos formatos, mas, por questões de portabilidade, a maioria dos formatos co-

mundos é XML (às vezes chamada de "Plain Old XML" ou POX) e JavaScript Object Notation (ou JSON). Se a organização Northwind Traders preferir utilizar POX, o resultado retornado ao consultar o URL mostrado anteriormente pode ser parecido com o seguinte:

```
<Employee>
 <EmployeeID>
 7
 </EmployeeID>
 <LastName>
 King
 </LastName>
 <FirstName>
 Robert
 </FirstName>
 <Title>
 Sales Representative
 </Title>
</Employee>
```

O segredo para elaborar uma solução baseada no modelo REST é saber dividir um modelo de negócios em um conjunto de recursos. Em alguns casos, como os empregados, isso pode ser muito simples, mas em outras situações, pode se tornar um desafio.

O modelo REST depende do aplicativo que acessa os dados que enviam o verbo HTTP adequado como parte da solicitação utilizada para acessar os dados. Por exemplo, a solicitação simples, mostrada anteriormente, deve enviar uma solicitação GET do protocolo HTTP 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. Ao utilizar o modelo REST, você pode explorar esses verbos e construir Web services que podem atualizar dados.

Diferentemente do modelo SOAP, as mensagens enviadas e recebidas por meio do modelo REST costumam ser mais compactas. Isso ocorre basicamente porque o REST não dispõe dos mesmos recursos de roteamento, política ou segurança oferecidos pelas especificações WS-\*, e você passa a depender da infraestrutura subjacente disponibilizam pelo servidor Web para proteger os Web services REST. Entretanto, essa abordagem minimalista significa que um Web service REST é geralmente muito mais eficiente do que o Web service SOAP equivalente ao transmitir e receber mensagens.

## Construindo Web services

No WCF, você pode construir Web services que adotam o modelo REST ou SOAP. O mecanismo SOAP é o de implementação mais simples dos dois esquemas, por meio do WCF, e você se concentrará nesse modelo nos primeiros exercícios deste capítulo. Mais adiante, você verá como é possível construir um Web service REST.

Neste capítulo, você criará dois Web services:

- Web service ProductInformation – Este é um Web service SOAP, que permite ao usuário calcular o custo de comprar uma quantidade específica de determinado produto no banco de dados Northwind
- Web service ProductDetails – Este é um Web service REST, que permite ao usuário consultar os detalhes de produtos no banco de dados Northwind.

## Criando o Web Service SOAP ProductInformation

No primeiro exercício, você criará o Web service ProductInformation e examinará o exemplo de código gerado pelo Visual Studio 2010 sempre que você cria um novo projeto de serviço WCF. Nos exercícios subsequentes, você definirá e implementará o método Web *HowMuchWillItCost* e testará esse método para ter certeza de que ele funciona como previsto.

**Importante** Não é possível construir Web services na Microsoft Visual C# 2010 Express. Em vez disso, use o Microsoft Visual Web Developer 2010 Express. Você pode baixar gratuitamente o Visual Web Developer 2010 Express no site da Microsoft.

### Crie o Web service SOAP e examine o exemplo de código

1. Inicie o Visual Studio 2010 se ele ainda não estiver em execução, ou inicie o Visual Web Developer 2010 Express.
2. No menu *File* do Visual Studio 2010 Professional ou Enterprise, aponte para *New* e clique em *Web Site*.
3. No menu *File* do Visual Web Developer 2010 Express, clique em *New Web Site*. Certifique-se de selecionar *Visual C#* em *Installed Templates*, no painel da esquerda.
4. Na caixa de diálogo *New Web Site*, clique no template *WCF Service*. Selecione *File System* na caixa de listagem suspensa *Location*, especifique a pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 29\ProductInformationService` sob sua pasta Documentos, e clique em *OK*.

O Visual Studio 2010 gera um site hospedado por meio do Development Web Server, fornecido com o Visual Studio e o Visual Web Developer. Você também pode hospedar Web services por meio do Microsoft Internet Information Services (IIS) se estiver disponível em seu sistema, embora os detalhes dessa operação estejam fora do escopo deste capítulo. O site Web contém pastas chamadas `App_Code` e `App_Data`, um arquivo chamado `Service.svc`, e um arquivo de configuração chamado `Web.config`. O código de um exemplo de Web service está definido na classe `Service`, armazenada no arquivo `Service.cs`, na pasta `App_Code`, e é exibido na janela `Code and Text Editor`. A classe `Service` implementa um exemplo de interface, chamada `IService`, armazenada no arquivo `IService.cs`, na pasta `App_Code`.



**Nota** O arquivo de solução para um projeto de site Web está localizado na pasta Visual Studio 2010\Projects em sua pasta Documentos, e não na pasta que contém os arquivos do site. Você pode abrir um projeto de site já existente, ao localizar e abrir o arquivo da solução adequado, ou ao usar o comando *Open Web Site* no menu *File* e ao especificar a pasta que contém os arquivos do site Web. Você também pode copiar o arquivo de solução de um site Web para uma pasta que armazena os arquivos do site, mas isso não é recomendável em um ambiente de produção por motivos de segurança.

- Clique no projeto C:\...\ProductInformationService\. Na janela *Properties*, defina a propriedade *Use dynamic ports* como *False*, e a propriedade *Port number* como 4500.



**Nota** Talvez seja necessário aguardar alguns segundos após determinar a propriedade *Use dynamic ports* como *False* para depois definir a propriedade *Port number*.

Uma porta específica a localização na qual o servidor Web ouve as solicitações recebidas dos aplicativos cliente. Por padrão, o servidor Web de Desenvolvimento escolhe uma porta aleatória para reduzir as chances de conflitos com outras portas utilizadas por outros serviços de rede em execução em seu computador. Esse recurso é útil caso você esteja construindo e testando sites (e não Web services) em um ambiente de desenvolvimento, antes de copiá-los para um servidor de produção, como o IIS. Entretanto, ao construir um Web service, compensa mais utilizar um número de porta fixo, porque os aplicativos cliente precisam se conectar a essa porta.



**Nota** Quando você fecha um site e o reabre no Visual Studio ou no Visual Web Developer, a propriedade *Use dynamic ports* costuma se reverter para *True*, e a propriedade *Port number* é definida com uma porta aleatória. Nesse caso, redefina essas propriedade com os valores descritos nesta etapa.

- No *Solution Explorer*, expanda a pasta App\_Code, se ela ainda não estiver aberta, clique com o botão direito do mouse no arquivo Service.cs e clique em *Rename*. Mude o nome do arquivo para **ProductInformation.cs**.
- Aplicando a mesma técnica, mude o nome do arquivo IService.cs para **IProductInformation.cs**.
- Clique duas vezes no arquivo IProductInformation.cs para exibi-lo na janela *Code and Text Editor*. Esse arquivo contém a definição de uma interface chamada *IService*. No início do arquivo IProductInformation.cs, você encontrará instruções *using* que fazem referência aos namespaces *System*, *System.Collections.Generic* e *System.Text* (que você conheceu anteriormente), e três outras instruções que fazem referência aos namespaces *System.ServiceModel*, *System.ServiceModel.Web* e *System.Runtime.Serialization*.

Os namespaces *System.ServiceModel* e *System.ServiceModel.Web* contêm as classes utilizadas pelo WCF para definir serviços e suas operações. O WCF utiliza as classes do namespace *System.Runtime.Serialization* para converter objetos em um fluxo de dados para transmissão pela rede (um processo conhecido como *serialização*) e para converter um fluxo de dados recebido da rede de volta em objetos (*deserialização*). Você aprenderá alguns detalhes de como o WCF serializa e desserializa objetos mais adiante neste capítulo.

O principal conteúdo do arquivo *IProductInformation*.sln são a interface *IService* e uma classe chamada *CompositeType*. A interface *IService* é prefixada com o atributo *ServiceContract* e a classe *CompositeType* é marcada com o atributo *DataContract*. Por causa da estrutura de um serviço WCF, você pode adotar uma abordagem “primeiro o contrato” para o desenvolvimento. Ao utilizá-la você define as interfaces, ou *contratos*, que o serviço implementará e, então, constrói um serviço que se adapta a esses contratos. Essa não é uma técnica nova, e já vimos exemplos dessa estratégia por todo o livro. A questão por trás do uso do desenvolvimento “primeiro o contrato” é que você pode se concentrar no design do seu serviço. Se necessário, ele pode ser rapidamente revisado para assegurar que seu design não introduza qualquer dependência de hardware ou software específica antes de você realizar boa parte do desenvolvimento; lembre-se de que em muitos casos os aplicativos cliente talvez não sejam construídos por meio do WCF e, talvez, nem mesmo estejam em execução em ambiente Windows.

O atributo *ServiceContract* marca uma interface para definir métodos que a classe que implementa o Web Service exibirá como métodos Web. Os próprios métodos são marcados com o atributo *OperationContract*. As ferramentas fornecidas com o Visual Studio 2010 utilizam esses atributos para ajudar a gerar o documento WSDL apropriado para o serviço. Quaisquer métodos na interface não marcados com o atributo *OperationContract* não serão incluídos no documento WSDL e, portanto, não estarão acessíveis aos aplicativos cliente que utilizam o Web service.

Se um método Web receber parâmetros ou retornar um valor, os dados e valores desses parâmetros devem ser convertidos em um formato que possa ser transmitido pela rede e, então, convertidos novamente em objetos – esse é o processo conhecido como *serialização* e *deserialização* mencionado anteriormente. Os vários padrões dos Web services definem mecanismos para especificar o formato serializado dos tipos de dados simples, por exemplo, números e strings, como parte da descrição WSDL de um Web service. Mas você também pode definir seus próprios tipos de dados complexos com base em classes e estruturas. Se utilizar esses tipos em um Web service, você deverá fornecer as informações de como serializá-los e desserializá-los. Se examinar a definição do método *GetDataUsingDataContract* na interface *IService*, você poderá ver que ele espera um parâmetro do tipo *CompositeType*. A classe *CompositeType* é marcada com o atributo *DataContract*, o qual especifica que a classe deve definir um tipo que pode ser serializado e desserializado como um fluxo XML, como parte de uma solicitação ou mensagem de resposta SOAP. Cada membro que você quiser incluir no fluxo serializado enviado pela rede deve ser marcado com o atributo *DataMember*.

9. Dê um clique duplo no arquivo ProductInformation.cs para exibi-lo na janela *Code and Text Editor*.

Esse arquivo contém uma classe chamada *Service* que implementa a interface *IService* e fornece os métodos *GetData* e *GetDataUsingDataContract* definidos por essa interface. Essa classe é o Web service. Quando um aplicativo cliente invoca um método Web nesse Web service, ele gera uma mensagem de solicitação SOAP e envia-a ao servidor Web que hospeda o Web service. O servidor Web cria uma instância dessa classe e executa o método correspondente. Quando o método termina, o servidor Web constrói uma mensagem SOAP de resposta, que ele reenvia ao aplicativo cliente.

10. Dê um clique duplo no arquivo Service.svc para exibi-lo na janela *Code and Text Editor*.

Esse é o arquivo do serviço para o Web service; ele é utilizado pelo ambiente do host (IIS, nesse caso) para determinar qual classe carregar quando ele recebe uma solicitação a partir de um aplicativo cliente.

A propriedade *Service* da diretiva *@ ServiceHost* especifica o nome da classe do Web service e a propriedade *CodeBehind* especifica a localização do código-fonte para essa classe.



**Dica** Se não quiser implantar o código-fonte do seu serviço WCF no servidor Web, você poderá, em vez disso, fornecer um assembly compilado. Você pode então especificar o nome e a localização desse assembly utilizando a diretiva *@ Assembly*. Para informações adicionais, procure “*@ Assembly*” na documentação fornecida com o Visual Studio 2010.

Agora que você já conhece a estrutura de um serviço WCF, pode definir a interface que especifica o contrato de serviço para o Web service ProductInformation e criar uma classe que implemente esse contrato.

## Defina o contrato do Web service ProductInformation

1. Exiba o arquivo IProductInformation.cs na janela *Code and Text Editor*.
2. Na linha do código que define a interface *IService*, clique duas vezes no nome *IService* para destacá-lo. No menu *Refactor*, clique em *Rename*. Na caixa de diálogo *Rename*, digite **IProductInformation** na caixa de texto *New name*, desmarque a caixa de seleção *Preview reference changes* e clique em *OK*.

Esta ação modifica o nome da interface de *IService* para *IProductInformation* e também muda todas as referências a *IService* para *IProductInformation* em todos os arquivos no projeto. A linha que define a interface na janela *Code and Text Editor* deve ser semelhante à seguinte:

```
public interface IProductInformation
{

}
```

3. Na interface *IProductInformation*, remova as definições dos métodos *GetData* e *GetDataUsingDataContract* e as substitua pelo método *HowMuchWillItCost* mostrado em negrito a seguir. Certifique-se de manter o atributo *OperationContract* no método Web.

```
[ServiceContract]
public interface IProductInformation
{
 [OperationContract]
 decimal HowMuchWillItCost(int productID, int howMany);
}
```

O método *HowMuchWillItCost* aceita um ID de produto e uma quantidade, e retorna um valor *decimal* que especifica quanto custará essa quantidade.

4. Remova a classe *CompositeType*, inclusive o atributo *DataContract*, do arquivo *IProductInformation.cs*. O arquivo deve conter apenas a definição da interface *IProductInformation*.

A próxima etapa é definir a classe *ProductInformation*, que implementa a interface *IProductInformation*. O método *HowMuchWillItCost* nessa classe recuperará o preço do produto no banco de dados ao fazer uma simples consulta ADO.NET.

**Nota** Os Web services que você construir neste capítulo exigem acesso ao banco de dados Northwind. Caso ainda não o tenha feito, crie esse banco de dados seguindo as etapas descritas na seção "Criando o banco de dados", no Capítulo 25, "Consultando informações em um banco de dados".

### Implemente a interface *IProductInformation*

1. Exiba o código do arquivo *ProductInformation.cs* na janela *Code and Text Editor*.
2. Adicione as seguintes instruções *using* à lista existente no início do arquivo:

```
using System.Data;
using System.Data.SqlClient;
```

Vimos no Capítulo 25 que esses namespaces contêm os tipos necessários para acessar um banco de dados Microsoft SQL Server e consultar dados.

3. Na linha do código que define a classe *Service*, clique duas vezes no nome *Service* para destaca-lo. No menu *Refactor*, clique em *Rename*. Na caixa de diálogo *Rename*, digite **ProductInformation** na caixa de texto *New name* e clique em *OK*.

Como no exercício anterior, esta ação muda o nome da classe, de *Service* para *ProductInformation*, e também muda todas as referências a *Service* para *ProductInformation* em todos os

arquivos no projeto. A linha que define a classe na janela *Code and Text Editor* deve ser parecida com esta:

```
public class ProductInformation : IProductInformation
{
 ...
}
```

4. Remova os métodos *GetData* e *GetDataUsingDataContract* da classe *ProductInformation*.
5. Adicione o método *HowMuchWillItCost* à classe *ProductInformation* mostrada em negrito a seguir:

```
public class ProductInformation : IProductInformation
{
 public decimal HowMuchWillItCost(int productID, int howMany)
 {
 SqlConnection dataConnection = new SqlConnection();
 decimal totalCost = 0;

 try
 {
 SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
 builder.DataSource = ".\\SQLEXPRESS";
 builder.InitialCatalog = "Northwind";
 builder.IntegratedSecurity = true;
 dataConnection.ConnectionString = builder.ConnectionString;
 dataConnection.Open();

 SqlCommand dataCommand = new SqlCommand();
 dataCommand.Connection = dataConnection;
 dataCommand.CommandType = CommandType.Text;
 dataCommand.CommandText = "SELECT UnitPrice FROM Products WHERE ProductID
= @ProductID";

 SqlParameter productIDParameter = new SqlParameter("@ProductID",
SqlDbType.Int);
 productIDParameter.Value = productID;
 dataCommand.Parameters.Add(productIDParameter);

 decimal? price = dataCommand.ExecuteScalar() as decimal?;
 if (price.HasValue)
 {
 totalCost = price.Value * howMany;
 }
 }
 finally
 {
 dataConnection.Close();
 }

 return totalCost;
 }
}
```

Esse método se conecta o banco de dados e executa uma consulta ADO.NET para recuperar o preço do produto correspondente ao ID de produto informado no banco de dados Northwind. Se o preço retornado não for nulo, o método calculará o custo total da solicitação e o retornará; caso contrário, o método retornará o valor 0. O código é semelhante ao mostrado no Capítulo 25, mas utiliza o método *ExecuteScalar* para ler o valor da coluna *UnitPrice* no banco de dados. O método *ExecuteScalar* dispõe de um mecanismo muito eficiente para executar consultas que retornam um único valor escalar (e é muito mais eficaz do que abrir um cursor e ler os dados no cursor). O valor retornado por *ExecuteScalar* é um objeto, e você deve fazer um casting desse objeto para o tipo adequado antes de utilizá-lo.

**Importante** Esse método não valida os parâmetros de entrada. Por exemplo, você pode especificar um valor negativo para o parâmetro *howMany*. Em um Web service de produção, você capturaria erros como este, os registraria em um log, e retornaria uma exceção. Entretanto, transmitir os motivos significativos de uma exceção para um aplicativo cliente tem implicações sobre a segurança em um serviço WCF. Os detalhes estão além do escopo deste livro. Para obter mais informações, consulte *Microsoft Windows Communication Foundation Step by Step*.

Antes de utilizar o Web service, atualize a configuração no arquivo Service.svc de modo a fazer referência à classe *ProductInformation* do arquivo ProductInformation.cs. O servidor Web utiliza as informações contidas no arquivo Web.config criado com o projeto para armazenar informações sobre como publicar o serviço e disponibilizá-lo para os aplicativos cliente. Você deve modificar o arquivo Web.config e adicionar os detalhes do Web service.

## Configure o Web service

1. No *Solution Explorer*, clique duas vezes no arquivo Service.svc para exibi-lo na janela *Code and Text Editor*. Atualize os atributos *Service* e *CodeBehind* da diretiva *ServiceHost*, como mostrado em negrito a seguir:

```
<%@ ServiceHost Language="C#" Debug="true" Service="ProductInformation"
CodeBehind="~/App_Code/ProductInformation.cs" %>
```

2. No *Solution Explorer*, clique duas vezes no arquivo Web.config. Na janela *Code and Text Editor*, localize o elemento *<system.serviceModel>*. Use esse elemento para especificar a configuração de um serviço WCF. Esse elemento contém atualmente um elemento *<behaviors>*, que você pode ignorar por enquanto.
3. No arquivo Web.config, adicione o elemento *<services>* e os elementos filhos mostrados em negrito a seguir ao elemento *<system.serviceModel>*, antes do elemento *<behaviors>*:

```
<system.serviceModel>
 <services>
 <service name="ProductInformation">
 <endpoint address="" binding="wsHttpBinding" contract="IProductInformation"/>
 </service>
```

```
</services>
<behaviors>
 ...
</behaviors>
</system.serviceModel>
```

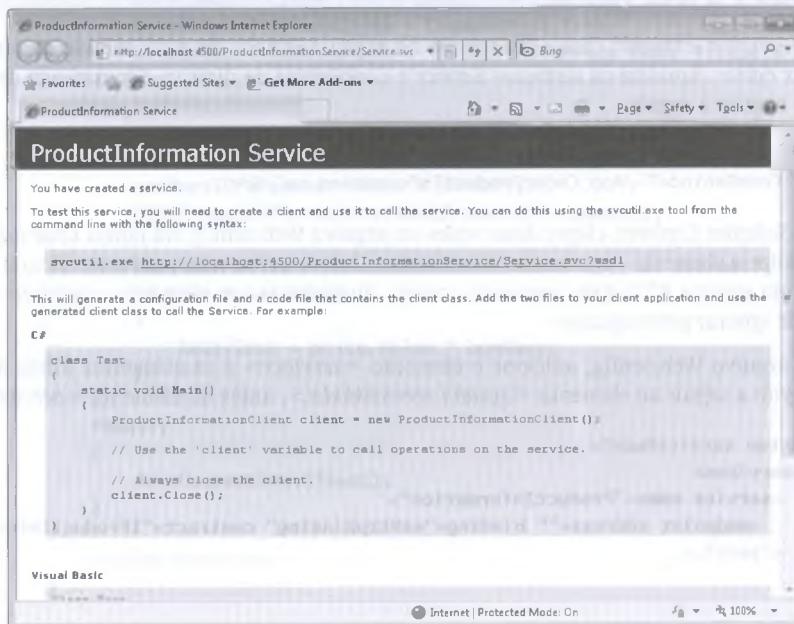
Essa configuração especifica o nome da classe que implementa o Web service (*ProductInformation*). O WCF utiliza o conceito de pontos de extremidade (end points) para associar um endereço de rede a um Web service específico. Se você estiver hospedando um Web service usando o IIS ou o Development Web Server, deixe a propriedade *address* de seu ponto de extremidade em branco, porque esses servidores escutam as solicitações recebidas em um endereço especificado pelas próprias informações de configuração. Você pode construir seus próprios aplicativos host personalizados, caso não queira utilizar o IIS ou o Development Server. Nessas situações, você deve especificar um endereço para o serviço como parte da definição do ponto de extremidade. O parâmetro *binding* indica o protocolo de rede utilizado pelo servidor para receber solicitações e transmitir respostas.

Para obter mais informações sobre pontos de extremidade, hosts personalizados e vinculações, consulte o *Microsoft Windows Communication Foundation Step by Step*, publicado pela Microsoft Press.

4. No menu *File*, clique em *Save All*.

5. No *Solution Explorer*, clique com o botão direito do mouse em *Service.svc* e clique em *View in Browser*.

O Internet Explorer se inicia e exibe a página a seguir, confirmando que você criou e implementou com sucesso o Web Service e forneceu informações úteis sobre como criar um aplicativo cliente simples que pode acessar o Web service.





**Nota** Se clicar no link mostrado na página Web (<http://localhost:4500/ProductInformationService/Service.svc?wsdl>), o Internet Explorer exibirá uma página contendo a descrição WSDL do Web service. Esse é um longo e complicado fragmento de XML, mas o Visual Studio 2010 pode pegar as informações nessa descrição e utilizá-las para gerar uma classe que um aplicativo cliente pode utilizar para se comunicar com o Web service.

6. Feche o Internet Explorer e retorne ao Visual Studio 2010.

## Web Services SOAP, clientes e proxies

Um Web service SOAP usa o protocolo SOAP para transmitir dados entre um aplicativo cliente e um serviço. O SOAP utiliza XML para formatar os dados que estão sendo transmitidos, os quais trafegam sobre o protocolo HTTP empregado pelos servidores e navegadores Web. É isso o que torna os Web services tão poderosos – o SOAP, o HTTP e o XML são bem conhecidos (teoricamente) e são assuntos de diversos comitês de padrões. Um aplicativo cliente que “fala” SOAP pode se comunicar com um Web service. Portanto, como um cliente “fala” SOAP? Existem duas maneiras: a difícil e a fácil.

### Falando SOAP: a maneira difícil

Na maneira difícil, o aplicativo cliente executa vários passos. Ele deve fazer o seguinte:

1. Determinar o URL do Web service que executa o método Web.
2. Executar uma consulta Web Services Description Language (WSDL) utilizando o URL para obter uma descrição dos métodos Web disponíveis, os parâmetros empregados e os valores retornados. Já vimos como fazer isso, através do Internet Explorer, no exercício anterior.
3. Analisar sintaticamente o documento WSDL, converter cada operação em uma solicitação Web e serializar cada parâmetro no formato descrito pelo documento WSDL.
4. Enviar a solicitação, junto com os dados serializados, para o URL utilizando HTTP.
5. Esperar o Web service responder.
6. Usando os formatos especificados pelo documento WSDL, desfazer a serialização dos dados retornados pelo Web service em valores significativos para que seu aplicativo possa, assim, processar.

É muito trabalho apenas para invocar um método, e, além disso, é muito propenso a erros.

### Falando SOAP: a maneira fácil

A má notícia é que a maneira fácil de utilizar o SOAP não é muito diferente da maneira difícil. A boa notícia é que o processo pode ser automatizado porque é muito mecânico. Como mencionado anteriormente, muitos fornecedores, inclusive a Microsoft, disponibilizam ferramentas que podem

gerar uma classe proxy com base em uma descrição WSDL. O proxy oculta a complexidade do uso do SOAP e expõe uma interface programática simples baseada em métodos publicados pelo Web service. O aplicativo cliente chama os métodos Web invocando os métodos com o mesmo nome no proxy. O proxy converte essas chamadas de método locais em solicitações SOAP e as envia para o Web service. O proxy espera pela resposta, desfaz a serialização dos dados e, então, a retorna para o cliente exatamente como o retorno de qualquer chamada de método simples. Essa é a abordagem que você utilizará nos exercícios desta seção.

## Consumindo o Web Service SOAP ProductInformation

Você criou uma chamada ao Web service SOAP, que expõe um método Web chamado *HowMuchWillItCost* para calcular o custo da aquisição de  $n$  itens do produto  $x$  da Northwind Traders. Nos exercícios a seguir, você utilizará esse Web service e criará um aplicativo que consome esse método.

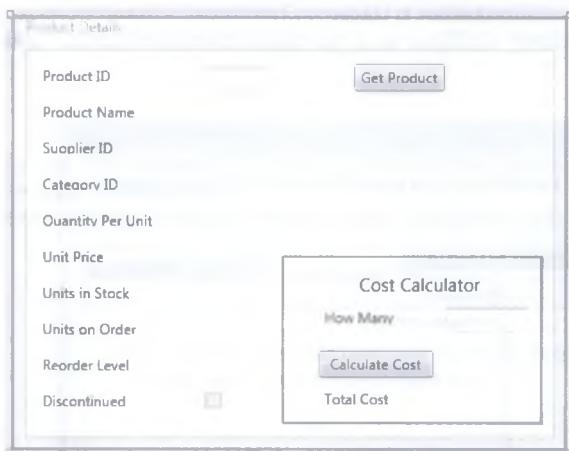
### Abra o aplicativo cliente do Web service

1. Inicialize outra instância do Visual Studio 2010. Essa etapa é importante. O Development Server utilizado para hospedar o Web service vai parar se você fechar o projeto do Web service ProductInformationService, ou seja, você não conseguirá acessá-lo a partir do cliente. (Uma abordagem alternativa, se você estiver executando o Visual Studio 2010 e não o Visual Web Developer 2010 Express, é criar o aplicativo cliente como um projeto na mesma solução que o Web service.) Quando você hospeda um Web service em um ambiente de produção, utilizando o IIS, esse problema não se manifesta porque o IIS opera independentemente do Visual Studio 2010.



**Importante** Se você tem usado o Visual Web Developer 2010 Express para fazer os exercícios dessa parte do livro, inicialize o Visual C# 2010 Express em vez de uma segunda instância do Visual Web Developer 2010 Express. (Deixe o Visual Web Developer 2010 Express em execução.)

2. Na segunda instância do Microsoft Visual Studio 2010, abra a solução ProductClient localizada na pasta \Microsoft Press\Visual CSharp Step By Step\Chapter 29\ProductClient em sua pasta Documentos.
3. No *Solution Explorer*, clique duas vezes no arquivo ProductClient.xaml para exibir o formulário na janela *Design View*. O formulário é parecido com este:



O formulário permite que o usuário especifique um ID de produto e recupere os detalhes do produto no banco de dados Northwind. (Você implementará essa funcionalidade em um exercício mais adiante, ao utilizar um Web service REST.) O usuário também pode informar uma quantidade e recuperar um preço de compra dessa quantidade do produto. Atualmente, os botões no formulário não fazem coisa alguma. Nas etapas a seguir, você adicionará o código necessário para chamar o método *HowMuchWillItCost* no Web service *ProductInformation* para obter o custo e depois exibi-lo.

### Adicione um código para chamar o Web service no aplicativo cliente

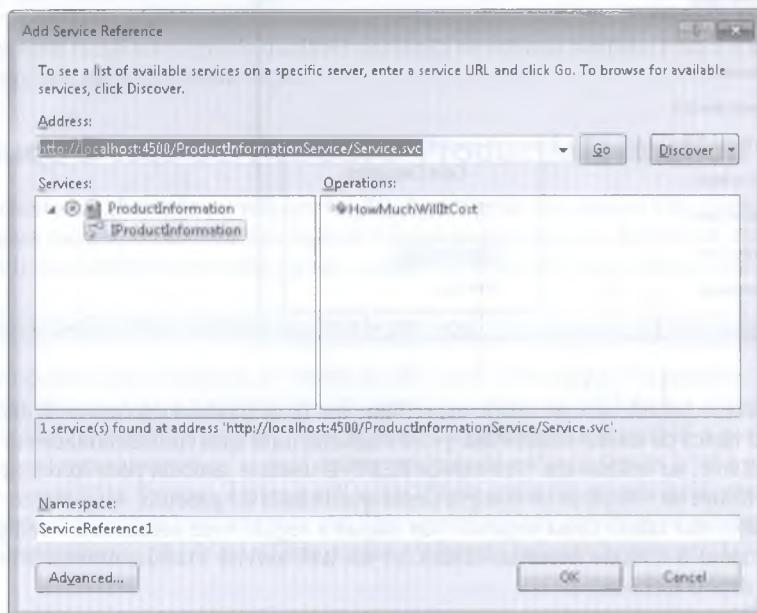
1. No menu *Project*, clique em *Add Service Reference*.

É exibida a caixa de diálogo *Add Service Reference*. Nela, você pode procurar os Web services e examinar os métodos Web oferecidos.

2. Na caixa de texto *Address*, digite **http://localhost:4500/ProductInformationService/Service.svc** e clique em *Go*.

O serviço *ProductInformation* aparece na caixa *Services*.

3. Expanda o serviço *ProductInformation* e clique na interface *IProductInformation* exibida. Na caixa de listagem *Operations*, verifique se a operação *HowMuchWillItCost* aparece, como na imagem a seguir:



4. Mude o valor existente na caixa de texto *Namespace* para **ProductInformationService** e clique em *OK*.

Uma nova pasta, chamada *Service References*, aparece no *Solution Explorer*. Essa pasta contém um item chamado *ProductInformationService*.

5. Clique no botão *Show All Files* na barra de ferramentas do *Solution Explorer*. Expanda a pasta *ProductInformationService* e, em seguida, a pasta *Reference.svcmap*. Clique duas vezes no arquivo *Reference.cs* e examine seu conteúdo na janela *Code and Text Editor*.

Esse arquivo contém várias classes e interfaces, inclusive uma classe chamada *ProductInformationClient* contida em um namespace chamado *ProductClient.ProductInformationService*. A *ProductInformationClient* é uma classe proxy gerada pelo Visual Studio 2010 a partir da descrição em WSDL do Web service *ProductInformation*. Ela abrange alguns construtores e um método chamado *HowMuchWillItCost*. O código a seguir mostra alguns destaques desse arquivo, formatados para torná-lo um pouco mais legível:

```
namespace ProductClient.ProductInformationService {

 [System.ServiceModel.ServiceContractAttribute(...)]
 public interface IProductInformation {
```

```

[System.ServiceModel.OperationContractAttribute(...)]
 decimal HowMuchWillItCost(int productID, int howMany);
}

public partial class ProductInformationClient :
System.ServiceModel.ClientBase<ProductClient.ProductInformationService,
IProductInformation>,
ProductClient.ProductInformationService.IProductInformation {

 public ProductInformationClient() {
 }

 public ProductInformationClient(string endpointConfigurationName) :
 base(endpointConfigurationName) {
 }

 public ProductInformationClient(string endpointConfigurationName, string
remoteAddress) :
 base(endpointConfigurationName, remoteAddress) {
 }

 public ProductInformationClient(string endpointConfigurationName,
 System.ServiceModel.EndpointAddress remoteAddress) :
 base(endpointConfigurationName, remoteAddress) {
 }

 public ProductInformationClient(System.ServiceModel.Channels.Binding binding,
 System.ServiceModel.EndpointAddress remoteAddress) :
 base(binding, remoteAddress) {
 }

 public decimal HowMuchWillItCost(int productID, int howMany) {
 return base.Channel.HowMuchWillItCost(productID, howMany);
 }
}
}

```

A interface *IProductInformation* é parecida com aquela que você definiu no Web service, mas alguns dos atributos especificam parâmetros adicionais. (O objetivo desses parâmetros está além do escopo deste capítulo.) A classe *ProductInformation* implementa essa interface, além de herdar da classe genérica *ClientBase*. A classe *ClientBase* do namespace *System.ServiceModel* fornece a funcionalidade da comunicação básica, necessária a um aplicativo cliente para se comunicar com um Web service. O parâmetro de tipo especifica a interface que a classe implementa. A classe *ClientBase* fornece a propriedade *Channel*, que encapsula uma conexão HTTP com um Web service. Os diversos construtores para a classe *ProductInformationClient* configuram o canal para conectá-lo com o ponto de extremidade no qual o Web service está ouvindo.

O aplicativo cliente pode instanciar a classe *ProductInformationClient*, especificando o ponto de extremidade ao qual se conectar, e depois chama o método *HowMuchWillItCost*. Quando isso acontece, o canal na classe subjacente *ClientBase* empacota as informações fornecidas como

parâmetros em uma mensagem SOAP que ele transmite para o Web service. Quando o Web service responde, as informações retornadas são desempacotadas da resposta do SOAP e passadas novamente para o aplicativo cliente. Assim, o aplicativo cliente pode chamar um método em um Web service da mesma maneira como chamaria um método local.



**Nota** Você deve ter observado que a interface está marcada com *ServiceContractAttribute*, e não apenas com *ServiceContract*, e a operação está marcada com *OperationContractAttribute*, em vez de *OperationContract*. Na verdade, todos os atributos levam o sufixo *Attribute* em seus nomes. O compilador do C# reconhece essa convenção de nomeação e, consequentemente, permite que você omita o sufixo *Attribute* em seu código.

6. Exiba o formulário ProductClient.xaml na janela *Design View*. Clique duas vezes no botão *Calculate Cost* para gerar o método manipulador de evento, *calcCost\_Click*, para esse botão.
7. Na janela *Code and Text Editor*, adicione as seguintes instruções *using* à lista posicionada no início do arquivo ProductClient.xaml.cs:

```
using ProductClient.ProductInformationService;
using System.ServiceModel;
```

8. No método *calcCost\_Click*, adicione o seguinte código, mostrado em negrito:

```
private void calcCost_Click(object sender, RoutedEventArgs e)
{
 ProductInformationClient proxy = new ProductInformationClient();
}
```

Essa instrução cria uma instância da classe *ProductInformationClient* que seu código usará para chamar o método Web *HowMuchWillItCost*.

9. Adicione o código mostrado em negrito a seguir ao método *calcCost\_Click*. Esse código extrai o ID do produto e o número necessário na caixa de texto *How Many* no formulário, executa o método *HowMuchWillItCost* Web por meio do objeto *proxy* e exibe o resultado no rótulo *totalCost*.

```
private void calcCost_Click(object sender, RoutedEventArgs e)
{
 ProductInformationClient proxy = new ProductInformationClient();
 try
 {
 int prodID = Int32.Parse(productID.Text);
 int numberRequired = Int32.Parse(howMany.Text);
 decimal cost = proxy.HowMuchWillItCost(prodID, numberRequired);
 totalCost.Content = String.Format("{0:C}", cost);
 }
 catch (Exception ex)
 {
 MessageBox.Show("Error obtaining cost: " + ex.Message,
 "Error", MessageBoxButton.OK, MessageBoxIcon.Error);
 }
}
```

```
 }
 finally
 {
 if (proxy.State == CommunicationState.Faulted)
 proxy.Abort();
 else
 proxy.Close();
 }
}
```

Provavelmente, você já sabe que as redes são imprevisíveis, e isso se aplica em dobro à Internet. O bloco *try/catch* garante que o aplicativo cliente capture as exceções de rede que possam ocorrer. Também é possível que o usuário não digite um inteiro válido na caixa de texto *ProductID*, no formulário. O bloco *try/catch* também trata dessa exceção.

O bloco *finally* examina o estado do objeto proxy. Se ocorreu uma exceção no Web service (que tenha sido causada pelo usuário, ao fornecer um ID de produto inexistente, por exemplo), o proxy estará no estado *Faulted*. Nesse caso, o bloco *finally* chama o método *Abort* do proxy para confirmar a exceção e encerrar a conexão; caso contrário, ele chama o método *Close*. Os métodos *Abort* e *Close* fecham o canal de comunicação com o Web service e liberam os recursos associados a essa instância do objeto *ProductInformationClient*.

## Teste o aplicativo

1. No menu *Debug*, clique em *Start Without Debugging*.
2. Quando o formulário Product Details aparecer, digite **3** na caixa de texto *Product ID*, digite **5** na caixa de texto *How Many* e clique em *Calculate Cost*.

Após um curto intervalo, enquanto o cliente instancia o proxy e constrói uma solicitação SOAP que contém o ID do produto, o proxy enviará a solicitação para o Web service. O Web service desserializa a solicitação SOAP para extrair o ID do produto, lê o preço unitário do produto no banco de dados, calcula o custo total, encapsula-o como XML em uma mensagem de resposta SOAP e envia essa mensagem de resposta de volta para o proxy. O proxy desserializa os dados XML e os passa para seu código no método *calcCost\_Click*. O custo de 5 unidades do produto 3 aparece no formulário (50 unidades monetárias).

 **Dica** Se você receber uma exceção com a mensagem "Error obtaining cost: There was no endpoint listening at http://localhost:4500/ProductInformationService/Service.svc that could accept the message", provavelmente o Development Server parou de funcionar. (Ele será desligado se ficar inativo durante algum tempo.) Para reinicializá-lo, alterne para a instância do Web service *ProductInformation*, no Visual Studio 2010, clique com o botão direito do mouse em *Service.svc* no *Solution Explorer*, e clique em *View in Browser*. Feche o Internet Explorer quando ele for exibido.

3. Experimente digitar os IDs de outros produtos. Observe que, se você inserir um ID de um produto inexistente, o Web service retornará o valor 0 para o custo total.
4. Quando você terminar, feche o formulário e retorne ao Visual Studio.

## Criando o Web service REST ProductDetails

Na seção anterior, você construiu e utilizou um Web service SOAP para implementar um pequeno fragmento de funcionalidade por procedimentos. No próximo conjunto de exercícios, você construirá o Web service *ProductDetails*, que permite ao usuário recuperar os detalhes do produto. Esse tipo de Web service é de navegação, de modo que você o implementará por meio do modelo REST. Para começar, crie um contrato de dados para transmitir objetos *Product* na rede.

É possível acessar um Web service REST em um aplicativo cliente de forma semelhante ao modo como você acessa o Web service SOAP – por meio de um objeto proxy que oculta a complexidade do envio de uma mensagem pela rede no aplicativo cliente. Entretanto, no Visual Studio, não há suporte atualmente para gerar automaticamente classes proxy para Web services REST; sendo assim, você criará a classe proxy manualmente. Além disso, não é necessariamente uma prática consagrada duplicar código, como os contratos de serviços por meio de Web services e clientes, porque isso pode dificultar a manutenção. Por esses motivos, você adotará uma abordagem um pouco diferente para construir o Web service.

### Crie o contrato de dados para o Web service REST

1. No Visual Studio 2010 Standard ou Visual Studio 2010 Professional, execute as seguintes tarefas para criar um novo projeto de biblioteca de classes:
  - 1.1. Na instância do Visual Studio que você utilizou para editar o aplicativo cliente, no menu *File*, aponte para *New*, e clique em *Project*.
  - 1.2. Na caixa de diálogo *New Project*, no painel da esquerda, em *Visual C#*, clique em *Windows*.
  - 1.3. No painel central, selecione o template *Class Library*.
  - 1.4. Na caixa de texto *Name*, digite **ProductDetailsContracts**.
  - 1.5. Na caixa de texto *Location*, especifique a pasta *|Microsoft Press|Visual CSharp Step By Step|Chapter 29* em sua pasta Documentos.
  - 1.6. Clique em *OK*.
2. No Microsoft Visual C# 2010 Express, execute as seguintes tarefas para criar um novo projeto de biblioteca de classes:
  - 2.1. Inicializa o Visual C# 2010 Express se ele ainda não estiver em execução.
  - 2.2. No menu *File*, clique em *New Project*.
  - 2.3. Na caixa de diálogo *New Project*, no painel central, selecione o template *Class Library*.

- 2.4. Na caixa de texto *Name*, digite **ProductDetailsContracts**.
  - 2.5. Clique em *OK*.
  - 2.6. No menu *File*, clique em *Save ProductDetailsContracts*.
  - 2.7. Na caixa de diálogo *Save Project*, especifique na caixa de texto *Location* a pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 29* em sua pasta Documentos.
  - 2.8. Clique em *Save*.
3. No menu *Project*, clique em *Add Reference*.
  4. Na caixa de diálogo *Add Reference*, clique na guia *.NET*. Selecione os assemblies *System.Data.Linq*, *System.ServiceModel*, *System.ServiceModel.Web* e *System.Runtime.Serialization* e clique em *OK*.
  5. No *Solution Explorer*, clique com o botão direito do mouse no arquivo *Class1.cs* e clique em *Rename*. Mude o nome do arquivo para **Product.cs**. Quando solicitado, deixe o Visual Studio mudar todas as referências a *Class1* para *Product*.
  6. Clique duas vezes no arquivo *Product.cs* para exibi-lo na janela *Code and Text Editor*, se ele ainda não estiver aberto.
  7. No arquivo *Product.cs*, adicione as seguintes instruções *using* à lista posicionada no início do arquivo:

```
using System.Runtime.Serialization;
using System.Data.Linq.Mapping;
```

8. Atribua um prefixo à classe *Product* com os atributos *Table* e *DataContract*, como mostrado em negrito a seguir:

```
[Table (Name="Products")]
[DataContract]
public class Product
{
}
```

Você usará LINQ to SQL para recuperar os dados no banco de dados Northwind. Vimos no Capítulo 25 que o atributo *Table* marca a classe como uma classe *Entity*. O nome da tabela é *Products* no banco de dados Northwind.

9. Adicione as propriedades mostradas em negrito a seguir à classe *Product*. Certifique-se de preфиксar cada propriedade com os atributos *Column* e *DataMember*. Observe que algumas dessas propriedades permitem valores nulos.

```
[DataContract]
public class Product
{
 [Column]
 [DataMember]
 public int ProductID { get; set; }

 [Column]
```

```

[DataMember]
public string ProductName { get; set; }

[Column]
[DataMember]
public int? SupplierID { get; set; }

[Column]
[DataMember]
public int? CategoryID { get; set; }

[Column]
[DataMember]
public string QuantityPerUnit { get; set; }

[Column]
[DataMember]
public decimal? UnitPrice { get; set; }

[Column]
[DataMember]
public short? UnitsInStock { get; set; }

[Column]
[DataMember]
public short? UnitsOnOrder { get; set; }

[Column]
[DataMember]
public short? ReorderLevel { get; set; }

[Column]
[DataMember]
public bool Discontinued { get; set; }
}

```

A próxima etapa é definir o contrato de serviço para o Web service ProductDetails.

### Crie o contrato de serviço para o Web service REST

1. No menu *Project*, clique em *Add Class*.
2. Na caixa de diálogo *Add Class*, no painel central, selecione o template *Class*. Na caixa de texto *Name*, digite **IProductDetails.cs** e clique em *Add*.
3. Na janela *Code and Text Editor*, que exibe o arquivo IProductDetails.cs, adicione as seguintes instruções *using* à lista existente no início do arquivo:

```

using System.ServiceModel;
using System.ServiceModel.Web;

```

4. Transforme a classe *IProductDetails* em uma interface pública, e atribua um prefixo com o atributo *ServiceContract*, como mostrado em negrito a seguir:

```
[ServiceContract]
public interface IProductDetails
{
}
```

5. Adicione a definição do método *GetProduct* mostrada em negrito a seguir à interface *IProductDetails*:

```
[ServiceContract]
public interface IProductDetails
{
 [OperationContract]
 [WebGet(UriTemplate = "products/{productID}")]
 Product GetProduct(string productID);
}
```

O método *GetProduct* aceita um ID de produto e retorna um objeto *Product* para o produto desse ID. O atributo *OperationContract* indica que esse método deve ser exposto como um método Web. (Se você omitir o atributo *OperationContract*, o método não ficará acessível aos aplicativos cliente.) O atributo *WebGet* indica que essa é uma operação de recuperação lógica, e o parâmetro *UriTemplate* especifica o formato do URL que você informa para chamar essa operação, em relação ao endereço base do Web service. Nesse caso, você pode especificar o seguinte URL para recuperar o produto com o *productID* 7:

<http://host/service/products/7>

Os termos *host* e *service* representam o endereço de seu servidor Web e o nome do Web service. O elemento do *UriTemplate* entre chaves indica os dados passados como parâmetro para o método *GetProduct*. O identificador entre chaves deve corresponder ao nome do parâmetro.

6. No menu *Build*, clique em *Build Solution* e verifique se a biblioteca de classes é compilada sem quaisquer erros. O projeto cria um assembly chamado *ProductDetailsContracts.dll*.

Após construir um assembly que define o contrato de dados e o contrato de serviço para o Web service, você pode construir o próprio Web service.

## Crie o Web service REST

1. Abra outra instância do Visual Studio ou Visual Web Developer Express.

 **Importante** Não use a instância utilizada para criar o Web service SOAP, porque essa cópia do Visual Studio deve permanecer em execução para que o Development Web Server que está hospedando o Web service SOAP continue aberto.

2. No Visual Studio 2010 Professional ou Enterprise, no menu *File*, aponte para *New* e clique em *Web Site*.
3. No Visual Web Developer 2010 Express, no menu *File*, clique em *New Web Site*.
4. Na caixa de diálogo *New Web Site*, clique no template *WCF Service*. Selecione *File System* na caixa de listagem suspensa *Location*, especifique a pasta *\Microsoft Press\Visual CSharp Step By Step\Chapter 29\ProductDetailsService* em sua pasta Documentos, e clique em *OK*.
5. Clique no projeto *C:\...\ProductDetailsService*. Na janela *Properties*, defina a propriedade *Use dynamic ports* como *False*, e a propriedade *Port number* como *4600*.



**Nota** É importante especificar uma porta diferente daquela do Web service *ProductInformationService*; caso contrário, os dois Web services entrarão em conflito.

6. No menu *Website*, clique em *Add Reference*. Na caixa de diálogo *Add Reference*, clique na guia *Browse*. Na barra de ferramentas, clique no botão *Up One Level*, vá até a pasta *ProductDetails-Contracts\ProductDetailsContracts\bin\Debug*, selecione o assembly *ProductDetailsContracts.dll*, e clique em *OK*.
7. No *Solution Explorer*, expanda a pasta *App\_Code*, se ela ainda não estiver aberta, clique com o botão direito do mouse no arquivo *Service.cs*, e clique em *Rename*. Mude o nome do arquivo para *ProductDetails.cs*.
8. Na pasta *App\_Code*, exclua o arquivo *IService.cs*. Esse arquivo não é necessário ao Web service.
9. Clique duas vezes no arquivo *ProductDetails.cs* para exibi-lo na janela *Code and Text Editor*.
10. Adicione as seguintes instruções *using* à lista localizada no início do arquivo:

```
using System.Data.Linq;
using System.Data.SqlClient;
using ProductDetailsContracts;
```

11. Modifique a definição da classe *Service*, mude o nome para *ProductDetails*, e especifique que ela implementa a interface *IProductDetails*, como mostrado em negrito a seguir. Remova os métodos *GetData* e *GetDataUsingDataContract* da classe *ProductDetails*:

```
public class ProductDetails : IProductDetails
{
}
```

12. Adicione o método *GetProduct* mostrado em negrito a seguir à classe *ProductDetails*:

```
public class ProductDetails : IProductDetails
{
 public Product GetProduct(string productID)
```

```
{
 int ID = Int32.Parse(productId);
 SqlConnectionStringBuilder builder =
 new SqlConnectionStringBuilder();
 builder.DataSource = ".\\SQLEXPRESS";
 builder.InitialCatalog = "Northwind";
 builder.IntegratedSecurity = true;
 DataContext productsContext =
 new DataContext(builder.ConnectionString);

 Product product = (from p in productsContext.GetTable<Product>()
 where p.ProductID == ID
 select p).First();

 return product;
}
}
```

O ID de produto é passado para o método como uma string, de modo que a primeira instrução o converte em um inteiro e armazena o resultado na variável *ID*. Em seguida, o código cria um objeto *DataContext* que se conecta com o banco de dados Northwind. A consulta LINQ recupera todas as linhas que têm um ID de produto correspondente ao valor contido na variável *ID*. Deve existir no máximo um produto correspondente. Geralmente, você deverá iterar pelos resultados de uma consulta LINQ to SQL para pesquisar uma linha de cada vez, mas, se existir apenas uma única linha, você poderá utilizar o método de extensão *First* para recuperar os dados imediatamente. O objeto *Product* recuperado pela consulta é retornado como o resultado do método.

A próxima etapa é configurar o Web service REST para fornecer a string de conexão utilizada pelo assembly *ProductDetailsContract* para se conectar com o banco de dados; depois, especifique o protocolo e o ponto de extremidade que os aplicativos clientes podem utilizar para se comunicar com o Web service.

## Configure o Web service

1. No *Solution Explorer*, clique duas vezes no arquivo *Web.config* para exibi-lo na janela *Code and Text Editor*.
2. Adicione o elemento *<services>* e os elementos filhos, mostrados em negrito a seguir, ao elemento *<system.serviceModel>*, antes do elemento *<behaviors>*. Além disso, adicione o elemento *<endpointBehaviors>*, também mostrado em negrito, como um filho do elemento *<behaviors>*. Observe que é necessário qualificar totalmente o nome da interface que fornece o contrato de serviço junto ao namespace *ProductDetailsContracts*.

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 <compilation debug="false" targetFramework="4.0" />
 </system.web>
 <system.serviceModel>
 <services>
```

```

<service name="ProductDetails">
 <endpoint address="" binding="webHttpBinding"
 contract="ProductDetailsContracts.IProductDetails"
 behaviorConfiguration="WebBehavior"/>
 </service>
</services>
<behaviors>
 <endpointBehaviors>
 <behavior name="WebBehavior">
 <webHttp/>
 </behavior>
 </endpointBehaviors>
 <serviceBehaviors>
 <behavior>
 <!-- To avoid disclosing metadata information, set the value below
 to false and remove the metadata endpoint above before deployment -->
 <serviceMetadata httpGetEnabled="true"/>
 <!-- To receive exception details in faults for debugging
 purposes, set the value below to true. Set to false before deployment to
 avoid disclosing exception information -->
 <serviceDebug includeExceptionDetailInFaults="false"/>
 </behavior>
 </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

Esse Web service usa uma vinculação diferente do Web service ProductInformation— *webHttpBinding*. A vinculação *webHttpBinding* e o comportamento *WebBehavior* indicam que o Web service espera que as solicitações sejam enviadas de acordo com o estilo REST, codificadas no URL, e que ele deve retornar as mensagens de resposta como XML (POX) simples.

3. No *Solution Explorer*, clique duas vezes no arquivo Service.svc para exibi-lo na janela *Code and Text Editor*. Atualize os elementos *Service* e *CodeBehind* de modo a fazer referência à classe *ProductsDetails* no arquivo ProductDetails.cs, como mostrado em negrito a seguir:

```
<%@ ServiceHost Language="C#" Debug="true" Service="ProductDetails"%
CodeBehind="~/App_Code/ProductDetails.cs" %>
```

4. No menu *Build*, clique em *Build Web Site*.
5. No *Solution Explorer*, clique com o botão direito do mouse em Service.svc, e clique em *View in Browser*. O Internet Explorer aparece e exibe a página do serviço ProductDetails.
6. Na barra de endereço, especifique o seguinte URL e pressione Enter:

```
http://localhost:4600/ProductDetailsService/Service.svc/products/5
```

Esse URL chama o método *GetProduct* do Web service ProductDetails e especifica o produto 5. O método *GetProduct* busca os dados do produto 5 no banco de dados Northwind e retorna as informações como um objeto *Product*, serializadas como XML. O Internet Explorer deve exibir a representação desse produto em XML.



7. Feche o Internet Explorer.

## Consumindo o Web service REST *ProductDetails*

Você viu como é fácil chamar um Web service REST em um navegador da Web simplesmente especificando um URL correto. Para chamar métodos em um Web service REST a partir de um aplicativo, você pode construir uma classe proxy semelhante à utilizada por um aplicativo cliente que se conecta a um Web service SOAP. Como mencionado anteriormente, o Visual Studio não dispõe da funcionalidade que pode gerar uma classe proxy para um Web service REST. Felizmente, não é difícil criar manualmente uma classe proxy REST simples; você pode usar a mesma classe genérica *ClientBase* utilizada por uma classe proxy SOAP.

No último exercício, você voltará ao aplicativo *ProductClient* e adicionará a funcionalidade para chamar o método *GetProduct* do Web service REST.

### Chame o Web service REST no aplicativo cliente

1. Retorne à instância do Visual Studio ou Visual C# Express que você usou para criar o contrato de serviço para o Web service REST.
2. Abra a solução *ProductClient* localizada na pasta `\Microsoft Press\Visual CSharp Step By Step\Chapter 29\ProductClient` em sua pasta Documentos. Esse é o aplicativo cliente que você utilizou para testar o Web service SOAP anteriormente neste capítulo.

3. No menu *Project*, clique em *Add Reference*. Na caixa de diálogo *Add Reference*, clique na guia *Browse*. Na barra de ferramentas, clique no botão *Up One Level* duas vezes, vá até a pasta *ProductDetailsContracts\ProductDetailsContracts\bin\Debug*, selecione o assembly *ProductDetailsContracts*, e clique em *OK*.
4. No menu *Project*, clique em *Add Reference* novamente. Na caixa de diálogo *Add Reference*, clique na guia *.NET*. Selecione o assembly *System.Data.Linq*, e clique em *OK*.
5. No menu *Project*, clique em *Add Class*. Na caixa de diálogo *Add New Item – ProductClient*, no painel central, clique no template *Class*. Na caixa de texto *Name*, digite **ProductClientProxy.cs**, e clique em *Add*.
6. Na janela *Code and Text Editor*, que exibe o arquivo *ProductClientProxy.cs*, adicione as seguintes instruções *using* à lista localizada no início do arquivo:

```
using System.ServiceModel;
using ProductDetailsContracts;
```

7. Modifique a definição da classe *ProductClientProxy* de modo que ela herde da classe genérica *ClientBase* e implemente a interface *IProductDetails*. Especifique a interface *IProductDetails* como o parâmetro de tipo para a classe *ClientBase*. A classe *ProductClientProxy* deve ficar parecida com o exemplo de código a seguir:

```
class ProductClientProxy : ClientBase<IProductDetails>, IProductDetails
{
}
```

8. Adicione o método *GetProduct* mostrado em negrito a seguir à classe *ProductClientProxy*. Esse método segue o mesmo padrão utilizado pelo proxy SOAP mostrado anteriormente neste capítulo; ele encaminha a solicitação do cliente para o canal de comunicação.

```
class ProductClientProxy : ClientBase<IProductDetails>, IProductDetails
{
 public Product GetProduct(string productID)
 {
 return this.Channel.GetProduct(productID);
 }
}
```

9. Exiba o arquivo *ProductClient.xaml* na janela *Design View*.
10. Clique duas vezes no botão *Get Product* para gerar o método manipulador de evento, *getProduct\_Click*, para esse botão.
11. Na janela *Code and Text Editor*, adicione a seguinte instrução *using* à lista localizada no início do arquivo *ProductClient.xaml.cs*:

```
using ProductDetailsContracts;
```

12. No método *getProduct\_Click*, adicione o seguinte código, mostrado em negrito:

```
private void getProduct_Click(object sender, RoutedEventArgs e)
{
 ProductClientProxy proxy = new ProductClientProxy();
 try
 {
 Product product = proxy.GetProduct(productID.Text);
 productName.Content = product.ProductName;
 supplierID.Content = product.SupplierID.Value;
 categoryID.Content = product.CategoryID.Value;
 quantityPerUnit.Content = product.QuantityPerUnit;
 unitPrice.Content = String.Format("{0:C}", product.UnitPrice.Value);
 unitsInStock.Content = product.UnitsInStock.Value;
 unitsOnOrder.Content = product.UnitsOnOrder.Value;
 reorderLevel.Content = product.ReorderLevel.Value;
 discontinued.IsChecked = product.Discontinued;
 }
 catch (Exception ex)
 {
 MessageBox.Show("Error fetching product details: " + ex.Message,
 "Error", MessageBoxButton.OK, MessageBoxImage.Error);
 }
 finally
 {
 if (proxy.State == CommunicationState.Faulted)
 {
 proxy.Abort();
 }
 else
 {
 proxy.Close();
 }
 }
}
```

Esse código cria uma instância da classe *ProductClientProxy* e a utiliza para chamar o método *GetProduct* do Web service REST. Os dados do objeto *Product* retornados são exibidos nos rótulos existentes no formulário.

13. No *Solution Explorer*, clique duas vezes no arquivo *app.config*. Esse é o arquivo de configuração do aplicativo, gerado automaticamente quando você criou o proxy do Web service SOAP, em um exercício anterior. Esse arquivo contém um elemento *<system.serviceModel>* que descreve o ponto de extremidade para o Web service SOAP, inclusive o URL ao qual o aplicativo deve se conectar.
14. Localize o elemento *<client>* e adicione o elemento *<endpoint>* mostrado em negrito a seguir acima da seção *<endpoint>* já existente:

```
<client>
<endpoint address="http://localhost:4600/ProductDetailsService/Service.svc"
 binding="webHttpBinding" contract="ProductDetailsContracts.
IPrductDetails">
 <behaviorConfiguration="WebBehavior">
</endpoint>
```

```

<endpoint address="http://localhost:4500/ProductInformationService/Service.svc"
 binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_IProductInformation"
 contract="ProductInformationService.IProductInformation"
 name="WSHttpBinding_IProductInformation">
 <identity>
 <userPrincipalName value="YourComputer\YourName" />
 </identity>
</endpoint>
</client>

```

15. Após a tag de fechamento `</client>`, adicione a seção `<behaviors>` mostrada em negrito a seguir:

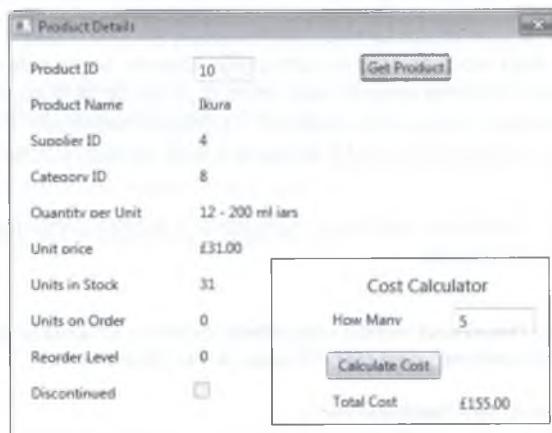
```

<client>
 /**
</client>
<behaviors>
 <endpointBehaviors>
 <behavior name="WebBehavior">
 <webHttp />
 </behavior>
 </endpointBehaviors>
</behaviors>

```

Esse código define o comportamento *WebBehavior* referenciado pelo ponto de extremidade do cliente. Ele especifique que o cliente deve se conectar ao Web service ao utilizar o comportamento *webHttp* esperado pelo Web service REST.

16. No menu *Debug*, clique em *Start Without Debugging*.
17. Quando o formulário *Product Details* aparecer, digite **10** na caixa de texto *Product ID*; digite **5** na caixa de texto *How Many*; e clique em *Calculate Cost*. O custo total deve ser exibido (155 unidades monetárias). Isso confirma que o Web service SOAP ainda está funcionando.
18. Clique em *Get Product*. Os detalhes de *Ikura* devem aparecer nos rótulos existentes no formulário, como na imagem a seguir:



19. Experimente outros IDs de produto. Se você especificar um ID de produto inexistente, o Web service retornará uma exceção "Bad Request".
20. Feche o formulário Product Details quando você terminar.

Neste capítulo, vimos como utilizar o Visual Studio para construir dois estilos diferentes de Web service: SOAP e REST. Vimos também como construir aplicativos cliente que podem consumir esses estilos diferente de Web service.

Você finalizou todos os exercícios deste livro. Esperamos que você já esteja se entendendo muito bem com a linguagem C# e saiba utilizar o Visual Studio 2010 para construir aplicativos profissionais. Entretanto, a história ainda não acabou. Você saltou apenas o primeiro obstáculo, mas os melhores programadores na linguagem C# aprendem com a experiência contínua, e você só poderá obter essa experiência ao construir aplicativos em C#. Ao fazer isso, você descobrirá novas maneiras de utilizar a linguagem C# e os diversos recursos disponíveis no Visual Studio 2010 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 escrevemos a primeira edição deste livro, a linguagem C# lançava a sintaxe e a semântica necessárias para construir aplicativos que utilizavam o .NET Framework 1.0. Foram implementados alguns aprimoramentos no Visual Studio e no .NET Framework 1.1, em 2003, e, mais tarde, em 2005, surgia 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. Agora, 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 que será que a próxima versão da linguagem C# nos oferecerá? Fique antenado!

## Referência rápida do Capítulo 29

Para	Faça isto
Criar um Web service SOAP	Utilize o template WCF Service. Defina um contrato de serviço que especifique os métodos Web expostos pelo Web service criando uma interface com o atributo <i>ServiceContract</i> . Marque cada método com o atributo <i>OperationContract</i> . Crie uma classe que implementa essa interface.
Criar um Web service REST	Use o modelo WCF Service. Defina um contrato de serviço que especifique os métodos Web expostos pelo Web service, criando uma interface com o atributo <i>ServiceContract</i> . Marque cada método com o atributo <i>OperationContract</i> e com o atributo <i>WebGet</i> , que especifica o template URI para chamar o método. Crie uma classe que implemente essa interface.  Configure o serviço para usar <i>webHttpBinding</i> , e especifique o comportamento <i>webHttp</i> para o ponto de extremidade do serviço.
Exibir a descrição de um Web service SOAP	Clique com o botão direito do mouse no arquivo .svc no <i>Solution Explorer</i> e clique em <i>View in Browser</i> . O Internet Explorer executa, vai para o URL do Web Service e exibe uma página que descreve como criar um aplicativo cliente que pode acessar o Web service. Clique no link <i>WSDL</i> para exibir a descrição WSDL do Web service.
Passar dados complexos como parâmetros de métodos Web e retornar valores	Defina uma classe para armazenar os dados e marque-a com o atributo <i>DataContract</i> . Verifique se cada item de dados é acessível como um campo público ou por meio de uma propriedade pública que fornece acesso <i>get</i> e <i>set</i> . Verifique se a classe tem um construtor padrão (que pode ser vazio).
Criar uma classe proxy para um Web service SOAP em um aplicativo cliente	No menu <i>Project</i> , clique em <i>Add Service Reference</i> . Digite o URL do Web service na caixa de endereço <i>Address</i> , na parte superior da caixa de diálogo, e clique em <i>Go</i> . Especifique o namespace da classe proxy e clique em <i>OK</i> .
Criar uma classe proxy para um Web service REST em um aplicativo cliente	Crie uma classe que herde da classe genérica <i>ClientBase</i> e especifique a interface que define o contrato de serviço como o parâmetro de tipo. Implemente essa interface e use a propriedade <i>Channel</i> herdada da classe <i>ClientBase</i> para enviar solicitações ao Web service.
Chamar um método Web	Crie uma instância da classe proxy. Chame o método Web por meio da classe proxy.

# Interoperabilidade com linguagens dinâmicas

Neste apêndice, você vai aprender a:

- Explicar o objetivo do Dynamic Language Runtime.
- Usar a palavra-chave *dynamic* para fazer referência a objetos implementados em linguagens dinâmicas, e chamar métodos nesses objetos.

A interoperabilidade entre códigos escritos em linguagens gerenciadas tem sido um recurso importante no Microsoft .NET Framework desde o seu surgimento. A ideia é a possibilidade de construir um componente em sua linguagem preferida, compilá-lo em um assembly, referenciar o assembly em seu aplicativo e acessar o componente no código em seu aplicativo. Seu aplicativo pode ser construído em uma linguagem diferente daquela do componente, mas isso não é importante. Todos os compiladores de cada linguagem gerenciada (Visual C#, Visual Basic, Visual C++, Visual F#, e outras) convertem o código escrito nessas linguagens em outra linguagem, chamada MSIL, ou Microsoft Intermediate Language. Quando você executa um aplicativo, o runtime do .NET Framework converte o código MSIL em instruções de máquina e depois os executa. O resultado é que, na realidade, o .NET Framework não conhece nem se preocupa com a linguagem utilizada originalmente. Se preferir, você pode escrever seus aplicativos em MSIL em não em C#, embora isso seria uma grande vergonha!

Entretanto, nem todas as linguagens dos computadores modernos são compiladas. Existem muitas linguagens de script interpretado atualmente em uso. Dois dos exemplos mais comuns, que têm surgido fora do domínio da Microsoft, são Ruby e Python. Nas primeiras versões do .NET Framework, nunca foi fácil incorporar código escrito nessas linguagens aos aplicativos gerenciados, e a consequência era, frequentemente, o surgimento de aplicativos de difícil entendimento e manutenção. O .NET Framework 4.0 solucionou esse problema com o Dynamic Language Runtime, que é o assunto deste apêndice resumido.



**Nota** Este apêndice pressupõe que você conheça Ruby ou Python, e seu objetivo não é ensinar o uso dessas linguagens. Além disso, o apêndice não contém quaisquer exercícios. Para executar o código apresentado neste capítulo, faça o download e instale as compilações mais recentes do IronRuby ou do IronPython a partir do site da CodePlex, em <http://www.codeplex.com>. O IronPython e o IronRuby são implementações completas das linguagens Python e Ruby, que contêm extensões que as permitem instanciar os objetos definidos no .NET Framework. Elas são totalmente compatíveis com as versões de código-fonte aberto mais recentes dessas linguagens, e você pode utilizá-las para executar scripts Python e Ruby inalterados.

## O que é o Dynamic Language Runtime?

C# é uma linguagem fortemente tipada. Ao criar uma variável, você especifica o tipo dessa variável, e só pode chamar os métodos e acessar os membros definidos por esse tipo. Se você tentar chamar um método não implementado pelo tipo em questão, seu código não será compilado. Isso é bom porque captura uma grande quantidade de possíveis erros antecipadamente, antes mesmo de você executar seu código.

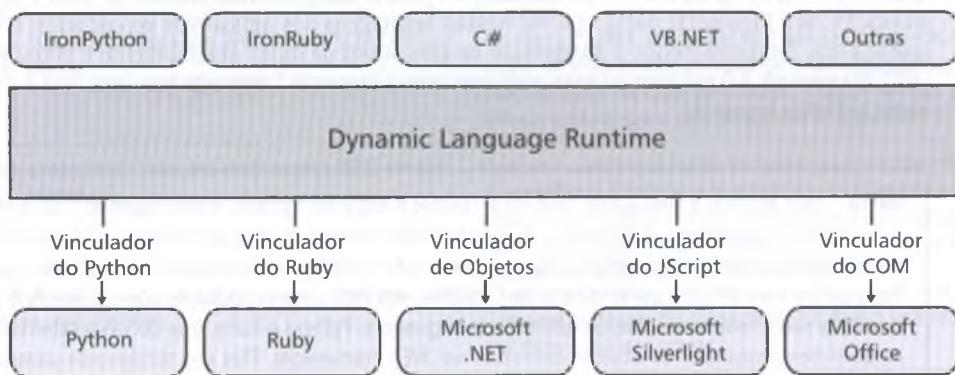
Entretanto, essa forte tipagem se torna um problema se você quiser criar objetos definidos por linguagens como Ruby e Python que são interpretadas e não compiladas. É muito difícil, se não impossível, que o compilador C# verifique se algum membro que você acessa em seu código C# realmente existe nesses objetos. Além disso, se você chamar um método em um objeto Ruby ou Python, o compilador C# não conseguirá verificar se você passou o número correto de parâmetros e se cada parâmetro tem o tipo adequado.

Há outra questão. Os tipos definidos pelo C# e pelo .NET Framework têm uma representação interna muito diferente daquela utilizada pelas linguagens Ruby e Python. Portanto, se você chamar um método Ruby que retorna um inteiro, por exemplo, de alguma maneira esse inteiro deverá ser convertido da representação utilizada pela linguagem Ruby naquela que o C# espera. Ocorre um problema semelhante se você passar um inteiro como um parâmetro de um aplicativo C# para um método Ruby; o inteiro deve ser convertido da representação do C# para a da linguagem Ruby.

O processo de converter dados entre formatos é conhecido como *marshaling*, e é um problema antiquíssimo, conhecido pelos desenvolvedores que já compilaram aplicativos que chamam componentes COM. A solução é utilizar uma camada intermediária. No .NET Framework 4.0, essa camada intermediária é chamada de Dynamic Language Runtime, ou DLR.

Além do recurso de *marshaling* de dados entre linguagens, o DLR também dispõe de vários serviços oferecidos pelo compilador, ao utilizar uma linguagem fortemente tipada. Por exemplo, quando você chama um método em um objeto Ruby ou Python, o DLR verifica se essa chamada ao método é válida.

O DLR não está associado a um conjunto específico de linguagens; ele implementa um arquitetura baseada em vinculadores de linguagens (language binders), como mostra a imagem a seguir:



Um vinculador de linguagens é um componente que se encaixa no DLR e reconhece como chamar os métodos em uma linguagem especificada, e como fazer o marshaling e cancelar o marshaling de dados entre o formato esperado pela linguagem e o .NET Framework. O vinculador também permite determinado volume de verificação, como avaliar se um objeto realmente expõe um método que está sendo chamado, e se os parâmetros e tipos de retorno são válidos.

O .NET Framework 4.0 fornece vinculadores para IronPython, IronRuby, COM (que você pode utilizar para acessar os componentes COM, como os do Microsoft Office), e Jscript, além do próprio .NET Framework. Além disso, o .NET Framework 4.0 permite que você escreva seus próprios vinculadores para outras linguagens por meio dos tipos e das interfaces existentes no namespace *System.Dynamic*. (Os detalhes de como fazer isso estão fora do escopo deste apêndice.) Além disso, IronPython e IronRuby podem utilizar o DLR para acessar objetos construídos em outras tecnologias e linguagens.

O DLR realiza seu trabalho em tempo de execução. Isso significa que qualquer verificação de tipos de objetos citados através do DLR é prorrogada até que seu aplicativo seja executado. Como indicar em um aplicativo C# que essa verificação de tipos de um objeto deve ser adiada dessa maneira? A resposta está na palavra-chave *dynamic*.

## A palavra-chave *dynamic*

A palavra-chave *dynamic* é uma novidade no C# 4.0. Você a utiliza exatamente da mesma maneira que um tipo. Por exemplo, a seguinte instrução cria uma variável chamada *rubyObject* que utiliza o tipo *dynamic*:

```
dynamic rubyObject;
```

Na realidade, não existe esse negócio de tipo *dinâmico* na linguagem C#. Tudo o que essa instrução faz é criar uma variável do tipo *object*, mas com a verificação de tipo adiada até a execução. Você pode atribuir um valor a essa variável e chamar métodos por meio dela. Durante a execução, o DLR usa o vinculador adequado para validar seu código, instancia os objetos e chama os métodos. Os detalhes internos do DLR estão sujeitos a mudanças, de modo que uma discussão sobre como tudo isso funciona está além do escopo deste apêndice. Basta dizer que o DLR sabe como chamar um vinculador para criar objetos, chamar métodos e empacotar (marshaling) e desempacotar (unmarshaling) dados.

Existe mais uma pequena advertência. Como a verificação de tipos é prorrogada até a execução, o Visual Studio IntelliSense não pode ajudá-lo, fornecendo os nomes dos membros expostos por meio de uma referência ao objeto dinâmico. Se você tentar chamar um método inválido ou fazer uma referência a uma campo inexistente em um objeto dinâmico, você não saberá o resultado até o momento da execução, quando ele lançará uma exceção *RuntimeBinderException*.

## Exemplo: IronPython

O exemplo a seguir mostra um script Python, chamado CustomerDB.py. Essa classe contém quatro itens:

- Uma classe chamada *Customer*. Essa classe tem três campos que contêm o ID, nome e telefone de um cliente. O construtor inicializa esses campos com valores passados como parâmetros. O método `_str_` formata os dados na classe como uma string, para que ele possa devolvê-la como resultado.
- Uma classe chamada *CustomerDB*. Essa classe tem um dicionário chamado *customerDatabase*. O método `storeCustomer` adiciona um cliente a esse dicionário, e o método `getCustomer` recupera um cliente quando o ID do cliente é informado. O método `_str_` itera pelos clientes no dicionário e os formata como uma string. Para simplificar, nenhum desses métodos contém qualquer modalidade de verificação de erros.
- Uma função chamada `GetNewCustomer`. Este é um método predefinido que constrói um objeto *Customer* por meio dos parâmetros passados e depois retorna esse objeto.
- Uma função chamada `GetCustomerDB`. Esse é outro método predefinido que constrói um objeto *CustomerDB* e retorna esse objeto.

```
class Customer:
 def __init__(self, id, name, telephone):
 self.custID = id
 self.custName = name
 self.custTelephone = telephone

 def __str__(self):
 return str.format("ID: {0}\tName: {1}\tTelephone: {2}",
 self.custID, self.custName, self.custTelephone)

class CustomerDB:
 def __init__(self):
 self.customerDatabase = {}

 def storeCustomer(self, customer):
 self.customerDatabase[customer.custID] = customer

 def getCustomer(self, id):
 return self.customerDatabase[id]

 def __str__(self):
 list = "Customers\n"
 for id, cust in self.customerDatabase.iteritems():
 list += str.format("{0}", cust) + "\n"
 return list

 def GetNewCustomer(id, name, telephone):
 return Customer(id, name, telephone)

def GetCustomerDB():
 return CustomerDB()
```

O exemplo de código a seguir mostra um simples aplicativo de console em C# que testa esses itens. Você encontrará esse aplicativo na pasta \Microsoft Press\Visual CSharp Step By Step\Appendix\PythonInteroperability em sua pasta Documentos. Ele faz referência aos assemblies IronPython, que propiciam a vinculação da linguagem para o Python. Esses assemblies são incluídos com o download do IronPython e não fazem parte da Biblioteca de Classes do .NET Framework.



**Nota** Este exemplo de aplicativo foi construído na versão do IronPython utilizada quando o livro foi encaminhado para impressão. Se você tiver uma compilação mais recente do IronPython, substitua as referências aos assemblies do IronPython e Microsoft.Scripting contidos neste aplicativo por aquelas fornecidas por sua instalação do IronPython.

O método estático *CreateRuntime* da classe *Python* cria uma instância do runtime do Python. O método *UseFile* do runtime do Python abre um script contendo o código Python e torna acessível o código contido nesse script.



**Nota** Neste exemplo, o script *CustomerDB.py* está localizado na pasta Appendix, mas o executável é compilado na pasta Appendix\PythonInteroperability\PythonInteroperability\bin\Debug, que considera o caminho para o script *CustomerDB.py* mostrado no parâmetro para o método *UseFile*.

Observe nesse código que as variáveis *pythonCustomer* e *pythonCustomerDB* fazem referência a tipos Python, portanto, são declaradas como *dynamic*. A variável *python*, utilizada para chamar as funções *GetNewCustomer* e *GetCustomerDB*, também é declarada como *dynamic*. Na verdade, o tipo retornado pelo método *UseFile* é um objeto *Microsoft.Scripting.Hosting.ScriptScope*. Entretanto, se você declarar a variável *python* usando o tipo *ScriptScope*, o código não será compilado porque, certamente, o compilador informará que o tipo *ScriptScope* não contém definições para os métodos *GetNewCustomer* e *GetCustomerDB*. A especificação de *dynamic* instrui o compilador a adiar sua verificação até a execução do DLR, momento em que a variável *python* faz referência a uma instância do script Python, que contém essas funções.

O código chama a função Python de *GetNewCustomer* para criar um novo objeto *Customer* com os detalhes para Fred. Em seguida, ele chama *GetCustomerDB* para criar um objeto *CustomerDB*, e depois chama o método *storeCustomer* para adicionar Fred ao dicionário no objeto *CustomerDB*. O código cria outro objeto *Customer* para um cliente chamado Sid, e também adiciona esse cliente ao objeto *CustomerDB*. Por último, o código exibe o objeto *CustomerDB*. O método *Console.WriteLine* espera uma representação de string do objeto *CustomerDB*. Consequentemente, o runtime do Python chama o método *\_str\_* para gerar essa representação, e a instrução *WriteLine* exibe uma lista dos clientes encontrados no dicionário do Python.

```
using System;
using IronPython.Hosting;

namespace PythonInteroperability
{
 class Program
 {
 static void Main(string[] args)
 {
 // Criando objetos IronPython
 Console.WriteLine("Testing Python");
 dynamic python =
 Python.CreateRuntime().UseFile(@"..\..\..\..\CustomerDB.py");
```

```

 dynamic pythonCustomer = python.GetNewCustomer(100, "Fred", "888");
 dynamic pythonCustomerDB = python.GetCustomerDB();
 pythonCustomerDB.storeCustomer(pythonCustomer);
 pythonCustomer = python.GetNewCustomer(101, "Sid", "999");
 pythonCustomerDB.storeCustomer(pythonCustomer);
 Console.WriteLine("{0}", pythonCustomerDB);
 }
}
}

```

A imagem a seguir mostra a saída gerada por este aplicativo:



## Exemplo: IronRuby

Para finalizar, o código a seguir mostra um script Ruby, chamado CustomerDB.rb, que contém as classes e funções que apresentam uma funcionalidade semelhante à contida no script Python, demonstrada anteriormente.

**Nota** O método `to_s` em uma classe Ruby retorna uma representação de string de um objeto, exatamente como o método `_str_` em uma classe Python.

```

class Customer
 attr_reader :custID
 attr_accessor :custName
 attr_accessor :custTelephone

 def initialize(id, name, telephone)
 @custID = id
 @custName = name
 @custTelephone = telephone
 end

 def to_s
 return "ID: #{custID}\tName: #{custName}\tTelephone: #{custTelephone}"
 end
end

```

```

class CustomerDB
 attr_reader :customerDatabase

 def initialize
 @customerDatabase = {}
 end

 def storeCustomer(customer)
 @customerDatabase[customer.custID] = customer
 end

 def getCustomer(id)
 return @customerDatabase[id]
 end

 def to_s
 list = "Customers\n"
 @customerDatabase.each {
 |key, value|
 list = list + "#{value}" + "\n"
 }
 return list
 end
end

def GetNewCustomer(id, name, telephone)
 return Customer.new(id, name, telephone)
end

def GetCustomerDB
 return CustomerDB.new
end

```

O seguinte programa em C# utiliza esse script Ruby para criar dois objetos *Customer*, armazená-los em um objeto *CustomerDB* e depois imprimir o conteúdo do objeto *CustomerDB*. Ele funciona da mesma maneira que o aplicativo de interoperabilidade Python descrito na seção anterior, e utiliza o tipo *dynamic* para definir as variáveis para o script Ruby e os objetos Ruby. Você encontrará esse aplicativo na pasta \Microsoft Press\Visual CSharp Step By Step\Appendix\RubyInteroperability em sua pasta Documentos. Ele faz referência aos assemblies IronRuby, que fornecem a vinculação da linguagem para o Ruby. Esses assemblies fazem parte do download do IronRuby.

 **Nota** Este exemplo de aplicativo foi construído na versão do IronRuby utilizada quando o livro foi encaminhado para impressão. Se você tiver uma compilação mais recente do IronRuby, substitua as referências aos assemblies IronRuby, IronRuby.Libraries e Microsoft.Scripting contidos neste aplicativo por aquelas fornecidas por sua instalação do IronRuby.

```

using System;
using IronRuby;

namespace RubyInteroperability
{
 class Program
 {

```

```

static void Main(string[] args)
{
 // Creating IronRuby objects
 Console.WriteLine("Testing Ruby");
 dynamic ruby =
 Ruby.CreateRuntime().UseFile(@"..\..\..\..\CustomerDB.rb");
 dynamic rubyCustomer = ruby.GetNewCustomer(100, "Fred", "888");
 dynamic rubyCustomerDB = ruby.GetCustomerDB();
 rubyCustomerDB.storeCustomer(rubyCustomer);
 rubyCustomer = ruby.GetNewCustomer(101, "Sid", "999");
 rubyCustomerDB.storeCustomer(rubyCustomer);
 Console.WriteLine("{0}", rubyCustomerDB);
 Console.WriteLine();
}
}
}

```

A imagem a seguir mostra a saída deste programa:



## Resumo

Este apêndice apresentou uma introdução sucinta ao uso do DLR para integrar um código escrito com linguagens de script, como Ruby e Python, a um aplicativo em C#. O DLR oferece um modelo extensível, com suporte para qualquer linguagem ou tecnologia que dispõe de um vinculador. Para escrever um vinculador, utilize os tipos existentes no namespace System.Dynamic.

A linguagem C# contém o tipo *dynamic*. Quando você declara uma variável como dinâmica, a verificação de tipos do C# é desabilitada para essa variável. O DLR faz a verificação de tipos no momento de sua execução, despacha chamadas a métodos e empacota (faz o *marshaling*) os dados.

# Índice

## Símbolos

- = atribuição composta, operador, 124, 364, 376  
+ = atribuição composta, operador, 124, 363, 375  
? modificador, 189, 202-203, 206  
--, operador, 76, 457  
\*, operador, 68, 201-202  
\*=, operador, 124  
/, operador, 124  
%/, operador, 69, 124  
+++, operador, 75, 457

## A

abordagem multitarefas, 632-633  
*about*, métodos de evento, 520-522  
About Box, template de janelas, 520  
*abstract* palavra-chave, 302, 308, 309  
acessibilidade  
  de campos e métodos, 164-165  
  de propriedades, 333  
acesso protegido, 274  
acesso simultâneo a dados obrigatórios, 688-712  
  protegido quanto a threads, 702-714  
acesso sincronizado, 691  
*Action*, delegates, 636-637  
  chamando, 664  
  criando, 540  
*Action*, tipo, 662  
adaptadores de método, 371  
*Add*, método, 240, 246, 249, 619  
<*Add New Event*>, comando, 508  
Add Window, comando, 489  
*add\_Click*, método, 504  
*AddCount*, método, 696  
*AddExtension*, propriedade, 528  
*AddObject*, método, 628  
*AddParticipant*, método, 698  
*addValues*, método, 82, 84  
adição, operador, 68  
  precedência de, 73, 109  
adição composta, operador, 140  
ADO.NET, 567  
  conectando-se a bancos de dados com, 595-596  
  consultando bancos de dados com, 567-580, 595-596  
  LINQ to SQL e, 581  
ADO.NET, biblioteca de classes, 567  
ADO.NET Entity Data Model, template, 598, 601, 628  
ADO.NET Entity Framework, 598-615

*AggregateException*, classe, 674-676  
  Handle, método, 674  
*AggregateException*, exceções, 679  
*AggregateException*, rotina de tratamento, 674-676  
alças de redimensionamento, 53  
algoritmo de aritmética de inteiros, 134  
algoritmo geométrico, 702-704  
algoritmo hill-climbing, 636  
ampliando conversões, 466  
AND (&), operador, 348  
AND (E) condicional, operador, precedência e associatividade de, 109  
APIs, 332  
aplicativos  
  capacidade de resposta de, 530-539  
  construindo, 58  
  executando, 58  
  multitarefa em, 634-660  
  paralelização em, 635  
aplicativos de banco de dados  
  interface do usuário para, 606-611  
  recuperando informações, 611-614  
  vinculações de dados, estabelecendo, 611-614  
aplicativos de console  
  criados pelo Visual Studio, arquivos, 40-41  
  criando, 40-46, 58  
  definição de, 35  
  referências a assemblies em, 48  
aplicativos gráficos  
  criando, 49-58  
  modos de exibição de, 49  
aplicativos WPF  
  atualizando e recarregando formulários, 503  
  capacidade de resposta, melhorando, 530-539  
  *Closing*, eventos, 506-508  
  código, visualizando, 508  
  construindo, 476-490  
  controles, adicionando, 490-502  
  controles, redefinindo com valores padrão, 498-502  
  controles de menu, 509-540  
  criando, 475-477, 508  
  estilo de controles, 483-489  
  eventos, manipulando, 502-508, 508  
formulários, adicionando, 489  
funcionalidade de, 489-492  
*Grid*, painéis, 478  
imagens de fundo, adicionando, 481-483  
informações de banco de dados, exibindo em, 606-611  
painéis de layout, 478  
pontos âncora de controles, 479-480  
propriedades, alterando dinamicamente, 498-502  
propriedades, definindo, 508  
propriedades de texto, 489  
regras de validação, 542  
rotina demorada de tratamento de eventos, simulando, 530-531  
segurança de threads, 534  
XAML, definição de, 477  
App.config (configuração de aplicativo), arquivo, 40, 605  
  strings de conexão, armazenando em, 604, 605  
App.xaml, arquivos, código em, 56  
*Application*, objetos, 489  
*Application.xaml.cs*, arquivos, 56  
*ApplicationException*, exceções, 549  
*ArgumentException*, classe, 252  
*ArgumentException*, exceções, 237, 257  
*argumentList*, 83  
argumentos  
  em métodos, 84  
  modificando, 191-194  
  nomeados, ambiguidades com, 98-103  
  omitindo, 98  
  passando para métodos, 191  
  posicionais, 98  
argumentos de array, 252-258  
argumentos nomeados, 98  
  ambiguidades com, 98-103  
*ArgumentOutOfRangeException*, classe, 153  
aritmética de inteiros, checked e unchecked, 150-152, 158  
aritmética de ponto flutuante, 151  
armazenamento local de threads (TLS), 693  
arquiteturas de Web services, 716  
arquivo-fonte em C#, (Program.cs), 40  
arquivos, fechando, 128  
arquivos de projeto, 40

- arquivos de solução  
de nível superior, 40  
nomes de arquivo, 65
- ArrayList*, classe, 240-241, 249  
número de elementos em, 241
- arrays, 223-238  
aplicativo de jogo de cartas, 231-238  
associativos, 244  
células em, 230  
comprimento de, 250  
copiando, 229-230  
de chaves, 244, 245  
de comprimento zero, 255  
de objetos, 239  
de valores, 244  
de variáveis int, 239  
implicitamente tipado, 226-227  
inicializando elementos de, 250  
inserindo elementos, 240  
iterando por, 227-229, 250  
multidimensional, 230-231  
*params*, arrays, 251-252  
redimensionando, 240  
removendo elementos de, 240  
tamanho de, 224-225  
*versus* coleções, 246
- arrays de parâmetros, 251, 253-254  
declarando, 253-254  
escrevendo, 256-258  
objeto tipo, 255, 261  
*versus* parâmetros opcionais, 258-261
- arrays multidimensionais, 230-231  
*params*, palavra-chave e, 253
- árvores binárias  
caminhamento, 416  
classificando dados, 391  
construindo com o uso de genéricos, 393  
criando classes genéricas, 403  
enumeradores, 415  
*Comparable*, interface, 394  
inserindo um nó, 394  
nó, 390  
referência, 390  
subárvores, 390  
teoria de, 390  
*TreeEnumerator*, classe, 415
- as*, operador, 200-201, 268
- AsOrdered*, método, 687
- AsParallel*, método, 682, 713  
especificando, 684
- aspas duplas ("), 120
- aspas simples ('), 120
- assemblies, 393  
definição de, 48  
namespaces e, 48  
usos de, 40
- AssemblyInfo.cs*, arquivos, 40
- assinantes, 374  
assinaturas de métodos, 269
- Association*, atributo, 587
- associatividade, 74  
de atribuição, operador, 74  
de operadores booleanos, 108-109
- asterisco (\*), operador, 68
- ataques de injeção de SQL, 575
- atributo TargetType de marca Style*, 486-488
- atributos, classe, 555
- atributos de projeto, adicionando, 40
- B**
- BackgroundWorker*, classe, 536
- Bancos de dados  
adicionando e excluindo dados, 619-626, 628  
atualizando dados, 615-628  
buscando e exibindo dados, 575-576, 583-585  
concedendo acesso a, 599-600  
conexões simultâneas, 579  
consultando, 573-575  
consultando com LINQ to SQL, 581-596  
consultando com ADO.NET, 567-580  
criando, 568-570  
dados bloqueados, 576  
desconectando-se de, 577-579  
erros de acesso, 571  
erros de integridade referencial, 620  
estabelecendo conexão com, 570-572, 595-596  
mapeamento de tipos de dados, 582  
mapeando camada, 597  
modelos de dados de entidade, 597  
novos, criando, 583  
salvando alterações em, 616-618, 626-628  
salvando informações do usuário, 594  
solicitando informações ao usuário, 573-575, 594  
valores nulos em, 579-580, 582, 595-596
- Windows Authentication  
acesso, 572, 573
- barra de status, exibindo status de operação de salvamento em, 537-539
- barra diagonal (/), operador, 68
- barra invertida (\), 120
- Barrier*, classe, 698-699
- Barrier*, construtores, especificando delegates para, 699
- Barrier*, objetos, 713
- base*, palavra-chave, 266, 271
- BeginInvoke*, método, 662
- BellRingers, projeto, 476-508  
GUI de aplicativo, 476
- bibliotecas de classes, 393
- Bin*, pasta, 45
- Binding*, elementos, para associar propriedades de controle entre si, 545
- Binding*, objetos, 558
- BindingValidationRules*, elementos, 564  
Elementos filho, 548
- BindingExpression*, classe  
*HasError*, propriedade, 561-562, 564  
*UpdateSource*, método, 561
- BindingExpression*, objetos, 564  
criando, 561
- BindingOperations*, classe GetBinding, método, 558
- BlackHole*, método, 255
- BlockingCollection*<T>, classe, 701
- BlockingCollection*<T>, objetos, 702
- bloqueando, 691-693. Consulte também primitivas de sincronização  
serializando chamadas a métodos com, 711-712  
sobreulação de, 693
- bloqueios de instruções, 110-111  
chaves em, 130
- bloqueios de leitura, 693, 697
- bloqueios exclusivos, 693
- Bool*, palavra-chave, 121-122
- bool*, tipo de dado, 64, 106
- boxing, 197-198
- break*, instruções, 117  
em instruções *switch*, 119  
fall-through, impedindo, 118  
para sair de loops, 131
- Breakpoints ícone, 135
- Build Solution, comando, 43, 44
- busca, 575-577  
adiada, 585-586, 590  
com objetos *SqlDataReader*, 595-596  
imediata, 590
- Button*, classe, 377
- Button.Resources*, propriedade, 483-484
- ButtonBase*, classe, 377
- C**
- C#  
compilando código, 43  
diferenciação de maiúsculas e minúsculas, 41  
estilo do layout, 60  
IntelliSense e, 41  
interoperabilidade COM, 96

- papel no .NET, 35  
 pares de caracteres combinados, 42  
**C#, palavras-chave.** Consulte também palavras-chave  
 IntelliSense, listas de, 41  
 .NET, equivalentes, 211  
**C#, projeto, arquivos,** 40  
 cache WPF, renovando, 611  
 caixa de texto de resultado, 149  
 caixas de diálogo comuns, 528-530  
 modais, 528  
*SaveFileDialog*, classe, 528-530  
 caixas de diálogo modais, 528  
 caixas de listagem suspensas de membro de classe, 66  
 caixas de texto  
   exibindo itens em, 66-67  
   limpando, 133  
*calculateClick*, método, 84-85  
*camelCase*, 62, 165  
   em nomes de métodos, 80  
 campos, 86-87, 161  
   compartilhados, 175-176  
   convenções de nome, 165  
   definição de, 163  
   estáticos e não estáticos, 175-176  
   herança de, 266-267  
   inicializando, 165, 171, 172  
   públicos, 164-165, 274  
 campos estáticos, 175-176  
   `const`, palavra-chave para, 176  
   declarando, 181  
   escrevendo, 177  
 campos estáticos privados, escrevendo, 177  
 campos privados, 164-165, 274, 330  
   adicionando, 171  
*CanBeNull*, parâmetro, 582  
 cancelamento, 664-677  
   de consultas PLINQ, 688  
   primitivas de sincronização e, 700  
 cancelamento cooperativo, 664-669  
*Canceled*, estado de tarefa, 670, 673  
*CancelEventArgs*, classe, 507  
*CancellationToken*, objetos, 665  
   especificando, 675, 688  
*ThrowIfCancellationRequested*, método, 672-673  
*Cancellationtokensource*, objetos Cancel, método, 700  
 capacidade de resposta, aplicativo  
   melhorando, 632. Consulte também multitarefa  
   melhorando com objeto *Dispatcher*, 661-664  
   threads e, 530-539, 539  
 caractere de escape (\), 120  
 caractere de nova linha ('\n'), 127  
 caracteres, lendo fluxos de, 127  
 caracteres de ponto e vírgula  
   em instruções *do*, 131  
   em instruções *for*, 130  
   regras de sintaxe para, 59  
 carga de trabalho, número ideal de threads para, 636  
*case*, palavra-chave, 117  
*case*, rótulos, 117  
   fall-through e, 118  
   regras de utilização, 118  
*catch*, rotina de tratamentos, 142  
   escrevendo, 149, 158  
   múltiplos, 144-145  
   palavra-chave, 142  
   sequência de execução, 146  
   sintaxe de, 143  
 células em arrays, 230  
 chamadas a método  
   examinando, 84-85  
   memória necessária para, 195  
 paralelizando, 650  
 parâmetros opcionais *versus* listas de parâmetros, 259-261  
 parênteses em, 83  
 serializando, 711-712  
 sintaxe de, 83-85  
*Change Data Source*, caixa de diálogo, 601-602  
*char*, tipo de dado, 64, 574  
 chaves  
   em definições de classes, 162  
   para agrupar instruções, 110-111, 125, 130  
 chaves primárias, tabelas de banco de dados, 582  
*checked*, expressões, 151-152  
*checked*, instruções, 150  
*checked*, palavra-chave, 158,  
*Choose Data Source*, caixa de diálogo, 601-602  
*Circle*, classe, 162-163  
   NumCircles campo, 175-176  
*Class*, atributo, 477  
*class*, palavra-chave, 162, 181  
 classes, 161  
   acessibilidade de campos e métodos, 164-174  
   anônimas, 179-180  
   atributos de, 555  
   classificação e, 161-162  
   com várias interfaces, 289  
   combinações de palavras-chave de método, 308  
   construtores para, 165-166. Consulte também construtores  
   convenções de nome, 165  
   corpo de, 163  
   de coleção, 238-249, 700-702  
   declarando, 181  
   definindo, 162-164  
   derivadas, 264-266  
   em assemblies, 48  
   encapsulamento em, 162  
   estáticas, 176-177, 280  
   fazendo referência por meio de interfaces, 288-289  
   genéricas, 390-402  
   herdando de interfaces, 287-288  
   instâncias de, atribuindo, 163  
   interfaces, implementando, 293-298  
   modelando entidades com, 545-547  
   novas, adicionando, 275  
   parciais, 168  
   seladas, 264, 303-309  
   testando, 298-301  
*versus* estruturas, 213-214, 220-222  
 classes abstratas, 264, 285, 301-303  
   criando, 304-306, 309  
 classes base, 264-266. Consulte também herança  
   impedindo o uso da classe como, 303-304, 309  
   membros de classe protegidos de, 274  
 classes de coleção, 238-249  
*ArrayList*, 240-241  
 implementação de jogo de cartas, 246-249  
 protegido quanto a threads (thread-safe), 700-702  
*Queue*, 242  
*SortedList*, 245  
*Stack*, 242-243  
 classes de conversão, 610  
   criando, 555-557  
   para vinculação de dados, 554  
 classes de entidades  
   código para, 604  
   convenções de nome, 588  
   criando, com o Entity Framework, 628  
   definindo, 581-583, 591-596  
*EntityCollection<Product>*, propriedade, 609  
 geração de, 603  
 herança de, 604  
 modificando propriedades de, 603-604  
 relações de tabelas, definindo em, 586-587  
 tabelas de banco de dados, relacionamentos entre, 583-584  
 classes de provedor de dados para o ADO.NET, 571

- classes derivadas, 264-266. *Consulte também* herança  
 construtores da classe base, chamando, 266-267, 283  
 criando, 283  
 classes seladas, 264, 303-309  
 criando, 309  
 classificação, 161-162  
 herança e, 263-264  
 classificando dados com árvores binárias, 391  
*clear\_Click*, método, 503  
*clearName\_Click*, método, 525  
*Click*, eventos, 57, 377  
*Click*, rotina de tratamento de evento, 503-506  
 para itens de menu, 517-519  
*Clone*, método, 230  
 CLS (Common Language Specification), 62  
*Code and Text Editor*, painel, 39  
 palavras-chave em, 61  
 código. *Consulte também* fluxo de execução  
 em formulários WPF, visualizando, 508  
 experimentando, 142-149  
 inseguro, 201-202  
 protegido contra exceções, 321-324  
 refatorando, 92, 302  
 tratamento de erros, isolando, 142  
 vinculado à computação, 653-655  
 código compilado, 40, 43, 46  
 referências a, 40  
 código-fonte, 40  
 códigos de caracteres, 134  
 colchetes em declarações de array, 223  
 coleções  
 contando número de linhas, 438  
 de itens desordenados, 701  
 enumerando elementos, 413-421  
 enumeráveis, 413  
*GetEnumerator*, métodos, 414  
*IEnumerable*, interface, 414  
 iteradores, 421  
 iterando por, 250, 682-687  
*join*, operador, 439  
 limitando número de itens em, 701-702  
 número de itens em, 250  
 produtores e consumidores de, 701  
 protegido quanto a threads, 710-711  
*versus* arrays, 246  
 coleta de lixo, 188-189, 312  
 chamando, 315, 324  
 destrutores, 312-314  
 garantias de, 314-315  
 intervalo de, 315  
 coletor de lixo, funcionalidade de, 315-316  
*Collect*, método, 315  
*Collection Editor: Items*, caixa de diálogo, 512, 513  
*Colors*, enumeração, 300  
*Column*, atributo, 582, 595-596  
 Comentando um bloco de código, 446  
 comentários, 43  
 comentários de várias linhas, 43  
 comentários do compilador e, 43  
 resolução de chamada de método, 98-103, 258-260  
*Command*, classe, 574  
*Command*, objetos, 574  
*CommandText*, propriedade, 574, 595-596  
*Common Dialog*, classes, 126  
 Common Language Specification (CLS), 62  
 comparando dois objetos, 409  
 comparando strings, 410  
*Compare*, método, 116, 409  
*CompareTo*, método, 394  
 compilando código, 43, 46  
*Complex*, objetos, 464  
*Component Object*  
 Interoperabilidade de modelo (COM) e do C#, 96  
*CompositeType*, classe, 723  
 concorrência otimista, 616-617  
 concorrência pessimista, 617  
*Concurrency Mode*, propriedade, 617  
*ConcurrentBag<T>*, classe, 701, 710-711  
 sobrecarga de, 711  
*ConcurrentDictionary< TKey, TValue >*, classe, 701  
*ConcurrentQueue<T>*, classe, 701  
*ConcurrentStack<T>*, classe, 701  
 conexões de banco de dados  
 abrindo, 572  
 criando, 601-602  
 fechando, 577-579, 585  
 lógica para, 604  
 pool de conexões, 579  
 conjuntos de dados, particionando, 682  
*Connection*, classe, 571  
*Connection*, propriedade, 595-596  
*Connection Properties*, caixa de diálogo, 601-602  
*ConnectionString*, propriedade, 572, 595-596  
*Console*, classe, 41  
*Console.WriteLine*, método, 90  
*Console.WriteLine*, método, 218, 251, 256  
 chamando, 169-170  
 Console Application, ícone, 37, 38  
 Console Application, template, 39  
*Const*, palavra-chave, 176  
 construtores, 165-166  
 chamando, 181  
 código de menu de atalho em, 526-527  
 declarando, 181  
 definição de, 55  
 escrevendo, 169-172  
 inicializando campos com, 171  
 inicializando objetos com, 311  
 padrão, 165-166, 167  
 para estruturas, 215, 217  
 privada, 166  
 sequência de definição, 167  
 sobrecarregando, 166-167  
 construtores da classe base, chamando, 266-267, 283  
 construtores não padrão, 166  
 construtores padrão, 165-167  
 em classes estáticas, 176  
 escrevendo, 170  
 estruturas e, 213, 216-217  
 consultando dados, 427-449  
 consultas de banco de dados  
 adiadas, 589-590  
 ADO.NET para, 595-596  
 avaliação imediata de, 585-586  
 iterando por, 584  
 LINQ to Entities para, 614-615  
 LINQ to SQL para, 595-596  
*Consulte também* aplicativos WPF; controles WPF; formulários WPF  
 consumidores, 701  
*Content*, propriedade, 52, 491, 501  
*ContextMenu*, elementos, 524  
 adicionando, 540  
*ContextMenu*, propriedade, 527  
 continuações, 638-640, 677, 678  
*Continue*, botão (barra de ferramentas Debug), 95  
*continue*, instruções, 132  
*ContinueCom*, método, 638, 677, 678  
 contravariância, 409  
 controles  
 adicionando a formulários, 491-493, 508  
 alinhando, 492  
*Content*, propriedade, 501  
 controles WPF, 490-491  
 estilo de, 483-489, 496-498  
 exibindo, 72  
 foco, validação e, 541, 550, 559  
 indicadores de alinhamento, 53  
*IsChecked*, propriedade, 508

- Name*, propriedade, 484  
ordem z de, 483  
Pontos de ancoragem, 479  
Propriedades, definindo, 481, 508  
Propriedades, modificando, 52  
Propriedades de layout de, 493-496  
Propriedades de texto, 489  
redefinindo para valores padrão, 498-502  
redimensionando, 53  
removendo de formulários, 491  
repositionando, 51  
*Resources*, elementos, 484  
*TargetType*, atributo, 486-488  
ToolTip, propriedade de, 564  
Controles botão  
  adicionando, 53  
  ancorando, 479  
Click, rotina de tratamento de eventos, 503-506  
comportamentos de passar o mouse, 488-489  
*Width* e *Height*, propriedades, 481  
Controles botão de opção, 501  
  adicionando, 493  
  inicializando, 508  
  mutuamente exclusivos, 493, 508  
Controles caixa de texto  
  adicionando, 53, 487, 491-492  
  menu de atalho para, 524-525  
  vinculando a propriedades de classe, 547-550  
Controles check box (caixa de seleção), 490  
  adicionando, 492  
  inicializando, 508  
  *IsChecked*, propriedade, 505  
Controles combo box (caixa de combinação), 490  
  adicionando, 492  
  preenchendo, 508  
Controles de imagem, adicionando, 481-483  
Controles de painel  
  *DockPanel*, controles, 511, 513, 540  
  *Grid*, painéis, 478, 479  
  ordem z, 483  
  painéis de layout, 478  
  *StackPanels*, 478, 492, 508  
  *WrapPanels*, 478  
Controles label (rótulos)  
  adicionando, 51, 491  
  propriedades, modificando, 52  
Controles list box (caixa de listagem), 491  
  adicionando, 493  
  preenchendo, 508  
Controles WPF. Consulte também  
  controles  
    como itens de menu, 516  
    exibindo dados de entidade em, 628  
    menus de atalho para, 524  
    vinculando a origens de dados, 612  
Convenções de nomeação  
  para campos e métodos, 165  
  para classes de entidades, 588  
  para identificadores, 269-270  
  para identificadores não públicos, 165  
  para interfaces, 287  
  para variáveis de array, 224  
    pública/privada, 330  
Convert, método, 610  
ConvertBack, método, 555, 556  
Convertendo dados, 199-203, 207  
Copiando  
  variáveis de estrutura, 219  
  variáveis tipo-referência e tipo-valor, 221  
Copy, método, 230  
CopyTo, método, 229  
Count, método, 435  
Count, propriedade, 241, 250  
CountdownEvent, objetos, 713  
covariância, 408  
CREATE TABLE, instruções, 581  
Created, estado de tarefa, 670  
CreateDatabase, método, 583  
CreateXXX, método, 619, 628  
.csproj, sufixo, 65  
CurrentCount, propriedade, 695  
cursos (conjunto atual de linhas), 576  
cursos firehose, 576  
curto-círcuito, 108
- ## D
- Dados  
  agregando, 433  
  bloqueando, 691-693  
  consultando, 427-449  
  contando número de linhas, 438  
Count, método, 435  
Distinct, método, 438  
encapsulamento de, 162  
filtrando, 432  
group, operador, 438  
GroupBy, método, 434  
Grouping, 433  
Max, método, 435  
Min, método, 435  
OrderBy, 434  
orderby, operador, 438  
OrderByDescending, 434  
selecionando, 430  
ThenByDescending, 434  
unindo, 436  
validação de, 541-564  
DataContext, classes, 583-584  
  acessando tabelas de banco de dados com, 594-596  
  personalizada, 591-592  
DataContext, objetos, 583-585  
  criando, 595-596  
DataContext, propriedade, 609, 628  
  de controles pai, 612  
DataContract, atributo, 723  
DataLoadOptions, classe  
  LoadWith, método, 590  
DataMember, atributo, 723  
Datas, comparando, 112-115, 116  
DataTemplate, 608  
dateCompare, método, 113, 114  
DatePicker, controles  
  adicionando, 492  
  menu de atalho padrão para, 525  
DateTime, tipo de dado, 113, 116  
DateTimePicker, controles, 490  
  SelectedDate, propriedade, 113  
DbType, parâmetro, 582  
Debug, barra de ferramentas, 93  
  exibindo, 93, 104  
Debug, pasta, 45  
decimal, tipo de dado, 63  
default, palavra-chave, 117  
DefaultExt, propriedade, 528  
definindo pares de operadores, 458  
Definite Assignment Rule, 64  
Delegate, classe, 538  
Delegados, 361  
  anexando a eventos, 377  
  cenários para utilizar, 362  
  chamando, 364  
  chamando automaticamente, 374  
  declarando, 363  
  definindo, 363  
DoWorkEventHandler, 536  
expressões lambda, 370  
formas combinadas, 363  
inicializando com um único método específico, 363  
utilizando, 365  
vantagens, 364  
DeleteObject, método, 621, 628  
Delimitadores, 120  
Depurador  
  examinando métodos com, 93-95  
  variáveis, verificando valores em, 94-95  
Dequeue, método, 386  
Desempenho  
  de coleções concorrentes  
    classes, 702  
melhorando com PLINQ, 682-687  
suspendendo e retomando threads e, 692

*Design View*, janela, 51  
 Formulários WPF em, 477  
 informações armazenadas em cache em, 611  
 trabalhando em, 51-54  
 desigualdade (!=), operador, 106  
 deslocamento à esquerda (<<), operador, 348  
 desserialização, 723  
 destrutores  
   chamando, 324  
*Dispose*, método, chamando a partir de, 320-321  
 escrevendo, 312-314, 324  
 intervalo de execução, 315  
   recomendações sobre, 316  
`detach.sql`, script, 599  
 dicionários, criando, 701  
*Dictionary*, classe, 388  
*Dictionary< TKey, TValue >*, classe de coleções, versão protegida quanto a threads, 701  
*DictionaryEntry*, classe, 244, 245  
 disparadores, 488  
*Dispatcher*, objetos, 537-539  
   capacidade de resposta, melhorando com, 661, 663-664  
   *Invoke*, método, 538  
*Dispatcher.Invoke*, método, 664  
*DispatcherPriority*, enumeração, 539, 663  
*Dispose*, método, 319  
   chamando a partir de destrutores, 320-321  
*Distinct*, método, 438  
*DivideByZeroException*, exceções, 155  
 divisão, operador, 68  
   precedência de, 73  
 divisão de inteiros, 71  
`.dll`, extensão de nome de arquivo 48  
`do`, instruções, 131-140  
   examinando, 135-139  
   sintaxe de, 131  
*DockPanel* controles, adicionando, 511, 540  
*Document Outline*, janela, 71-72  
 documentando código, 43  
 documentos, definição de, 509  
 Documentos, pasta, 38  
`double`, tipo de dado, 63  
`double.Parse`, método, 90  
*DoWork* evento, inscrevendo-se para, 536  
*DoWorkEventHandler*, delegates, 536  
*Drawing*, projeto, 292-294  
*drawingCanvas\_MouseLeftButtonDown*, método, 299  
*drawingCanvas\_MouseRightButtonDown*, método, 300

*DrawingPad*, janela, classe, 299  
 duplicação de código, 301-302  
*DynamicResource*, palavra-chave, 486  
**E**  
 edição de texto, menu de atalho para, 524  
*ElementName*, marca, 563-564  
 elementos de array  
   acessando, 227, 250  
   tipos de, 224  
 elementos *Set* da classe *Control*, 488  
*Else*, palavra-chave, 109, 110  
 encapsulamento, 162, 178  
   campos públicos, 329  
   regra de ouro, 328  
   violações de, 274  
*Enqueue*, método, 386  
 Enterprise Services, 716  
*EnterReadLock*, método, 697  
*EnterWriteLock*, método, 697  
 entidades  
   adicionando, 619  
   excluindo, 620  
   modelando, 161  
*entity*, objetos  
   exibindo dados em, 628  
   modificando, 615  
   vinculando propriedades de controles a, 598-615  
 Entity Data Model Wizard, 601-603  
 Entity Framework, 597  
   adicionando e excluindo dados com, 619-620, 628  
   atualizando dados com, 615-628  
   buscando e exibindo dados, 611-614  
   classes de entidades, criando, 628  
   concorrência otimista, 616-617  
   interfaces de usuário de aplicativo, criando, 606-611  
*LoadProperty< T >*, método, 613  
 mapeando camada e, 597  
 recuperando dados em tabelas com, 613  
 vinculação de dados, utilizando com, 598-615  
*EntityCollection< Product >*, propriedade, 609, 612  
*EntityRef< TEntity >*, tipos, 587-588  
*EntitySet< Product >*, tipos, 588  
*EntitySet< TEntity >*, tipos, 587-588  
 entrada do usuário  
   capacidade de resposta a, 660-661  
   pressionamentos de teclas, 620-621  
 entrando em métodos, 93-95  
*enum*, palavra-chave, 205, 222  
*enum*, tipos, 205-210

enumerações, 205-210  
   convertendo em strings, 554  
   declarando, 205-206, 208-210, 222  
   nomes literais, 206  
   sintaxe de, 205-206  
   tipo subjacente de, 208  
   utilizando, 206-207  
   valores inteiros para, 207  
   valores literais, 207  
   versões nullable de, 206  
 enumeradores  
   *Current*, propriedade, 414  
   implementando manualmente, 415  
   iteradores, 421  
   *MoveNext*, método, 414  
   *Reset*, método, 414  
   *yield*, palavra-chave, 422  
 enumerando coleções, 413  
*enumerator*, objetos, 414  
*Equal*, método, 463  
*Equals*, método, 464  
*Error List*, janela, 44  
*errorStyle*, estilo, 557  
 erros  
   descrições de texto de, 143  
   de integridade referencial, 620  
   do compilador, 44-45  
   exceções. *Consulte* exceções  
   lidando com, 141  
   marcação de, 44  
 escalabilidade, melhorando, 632.  
*Consulte também* multitarefa  
 escopo  
   aplicando, 85-88  
   de classe, definindo, 86-87  
   de estilos, 485  
   de instruções *for*, 130-131  
   de recursos estáticos, 486  
   definindo, 86  
   local, definindo, 86  
 escrevendo em recursos, 697-698  
 espaço em branco, 60  
 estilos  
   de controles de formulários WPF, 483-489, 496-498  
   escopo de, 485  
 estruturas, 210-222  
   arrays de, 226  
   campos de instância em, 213-214  
   campos privados em, 212  
   declarando, 212  
   de dados recursivas, 390  
   de programação extensíveis, construindo, 285  
   herdando de interfaces, 287-288  
   hierarquia de heranças para, 264  
   Inicialização de, 215-219  
   natureza selada de, 303

- operadores para, 212  
tipos de, 210-211  
utilizando, 216-219  
*versus* classes, 213-214, 220-222
- EventArgs*, argumento, 378  
eventos, 374-376  
anexando delegates, 377  
argumento do remetente, 378  
assinantes, 374  
cancelando inscrição, 376  
declarando, 374  
de interface do usuário WPF, 377  
disparando, 376  
esperando, 693, 713  
*EventArgs*, argumento, 378  
eventos de menu, manipulando, 516-524  
inscrições, 375  
interface do usuário do WPF, 377  
origens, 374  
utilizando um único método, 378  
verificações de null, 376  
*versus* disparadores, 488
- Example*, classe, 321  
exceções, 141  
*AggregateException*, 679  
*ApplicationException*, 549  
*ArgumentException*, 237, 257  
 capturando, 142-149, 158  
 capturando tudo, 155, 156, 158  
*DivideByZeroException*, 155  
 examinando, 143-144  
 exibindo código causador, 148  
 fluxo de execução e, 143, 145, 156  
*FormatException*, 142, 143  
 hierarquias de herança, 145  
*InvalidCastException*, 199  
*InvalidOperationException*, 154, 533, 585  
 lançando, 153-158  
 manipulando, 142  
*NotImplementedException*, 234, 295  
*NullReferenceException*, 376  
*OperationCanceledException*, 673, 700  
*OptimisticConcurrency-Exception*, 618, 619  
*OutOfMemoryException*, 196, 231  
*OverflowException*, 143, 150, 152  
 para regras de validação, detectando, 548-551  
 Sequência de execução de rotina de tratamento, 146  
*SqlException*, 571-572, 594  
*UpdateException*, 620  
 exceções não tratadas, 143-144, 155  
 capturando, 156-157  
 relatando, 147
- Exception*, hierarquia de herança, 145  
 capturando, 158,  
*ExceptionValidationRule*, elementos, 548  
 execução  
 multitarefa, 634-660  
 processamento paralelo, 632-633, 640-649, 708-710  
 execução single-threaded, 631. Consulte também multithreading  
*ExecuteReader*, método, 575  
 chamando, 595-596  
 sobrecarregando de, 580  
*Exit*, comando, manipulador de eventos *Click* para, 518  
*ExitReadLock*, método, 697  
*ExitWriteLock*, método, 697  
 expressões, comparando valores de, 121-122  
 expressões booleanas  
 criando, 121-122  
 declarando, 105-106  
 em instruções *if*, 110  
 em instruções *while*, 125  
 expressões lambda  
 como adaptadores, 371  
 corpo, 373  
 e delegates, 370-374  
 formulários, 372  
 métodos anônimos, 373  
 para delegates anônimos, 533, 539  
 parâmetros de método especificados como, 585  
 sintaxe, 372  
 variáveis, 373  
 expressões *unchecked*, 151  
 Extensible Application Markup Language (XAML), 51-52, 477  
*Extract Method*, comando, 92
- F**
- F*, tipo, sufixo, 66  
 F10, tecla, 94  
 F11, tecla, 94  
 F5, tecla, 95  
 falhas. Consulte erros; exceções  
 fall-through, interrompendo, 118  
*Faulted*, estado de tarefa, 670, 673  
*Fechando*, rotina de tratamento de evento, 506-508  
 fila *freachable*, 316  
 filas, 385  
 criando, 701  
*FileInfo*, classe, 126  
 OpenText, método, 127  
*fileName*, parâmetro, 532  
*fileName*, propriedade, 528  
 finalização, 316  
 sequência de, 315, 316  
 finalização de método, notificação de, 662  
*Finalize*, método, gerado pelo compilador, 314  
*finally*, blocos, 156-158  
 fluxo de execução e, 157  
 instruções de encerramento de conexão de banco de dados em, 577  
 métodos de descarte em, 317-318  
*first-in, first-out* (FIFO)  
 mecanismos, 242  
*float*, tipo de dado, 63  
 fluxo de execução, exceções e, 143, 145, 156  
 foco de controles, validação e, 541, 550, 559  
*FontFamily*, propriedade, 489  
*FontSize*, propriedade, 52, 489  
*FontWeight*, propriedade, 489  
*for*, instruções, 129-131, 140  
 escopo de, 130-131  
 iterando sobre arrays com, 228  
 omitindo partes de, 129-130  
 sintaxe de, 129  
*foreach*, instruções, 228, 413  
 interando sobre arrays com, 236  
 interando sobre arrays de comprimento zero com, 255  
 interando pelo array *paramList*, 257  
 interando sobre consultas de banco de dados com, 584  
 interando sobre tabelas de banco de dados com, 595  
 para consultas de banco de dados, 585, 588-590  
 forma de prefixação, operadores, 76-77  
 forma de sufixação, operadores, 76-77  
*Format*, método, 218  
*FormatException*, exceções, 142, 143  
*FormatException*, rotina de tratamento catch, 142-145, 152  
 formulários.  
 alças de redimensionamento, 53  
 formulários WPF, 489  
 código, exibindo, 55  
*Document Outline*, janela, 71-72  
 exibindo, 522, 624  
 instâncias de, 522  
 menus de atalho, desassociando de, 527  
 XAML em, 51-52  
*Func<T>*, tipo genérico, 538

**G**

*GC*, classe  
*Collect*, método, 315  
*SuppressFinalize*, método, 321  
Generate Method Stub Wizard, 89-92, 104  
genéricos, 387-412  
árvore binária, 390  
árvore binária, construindo, 393  
criando, 390-402  
objetivo, 385  
parâmetros de tipo, 388  
parâmetros de vários tipos, 388  
restrições, 390  
*versus* classes genéricas, 389  
gerador de números pseudoaleatórios, 225  
gerenciamento de recursos, 316-321  
conexões de bancos de dados, 577  
liberando recursos, 324  
multitarefa e, 632  
*get*, blocos, 330, 331

para consultas de banco de dados, 587-588

*GetBinding*, método, 558

*GetEnumerator*, método, 414

*GetInt32*, método, 576

*GetPosition*, método, 299

*GetString*, método, 576

*GetTable< TEntity >*, método, 584

*GetXXX*, métodos, 576, 595-596

girando, 683, 692

threads, 693

*goto*, instruções, 119

*Grid*, controles, em formulários WPF, 72

*Grid*, painéis, 478

controles, posicionando, 479

em aplicativos WPF, 478

*GridView*, controles, características de exibição, 610

*group*, operador, 438

*GroupBox*, controles, 501

adicionando, 492

*GroupBy*, método, 434

**H**

*Handle*, método, 674, 679

*HasError*, propriedade, 561

testando, 562

*Hashtable*, classe, 247

*Hashtable*, objeto, *SortedList*, objetos de coleção em, 247

*HasValue*, propriedade, 190

*Header*, atributo, 512

herança, 263-264

acesso protegido, 274

classes, atribuindo, 267-268

classes abstratas e, 301-303

construtores de classe base, chamando, 266-267, 283  
hierarquia de classes, criando, 274-279  
implementando, 306-308  
implicitamente pública, 265  
itens de menu, 515  
métodos override, declarando, 271-272  
métodos virtuais, declarando, 270-271, 283  
*new*, método, declarando, 269-270  
utilizando, 264-279  
hierarquia, 145  
hierarquias de classe, definindo, 274-279  
High Performance Compute (HPC) Server 2008, 632  
*HorizontalAlignment*, propriedade, 479-480  
Hypertext Transfer Protocol (HTTP), 716

**I**

*IColor*, interface, 292-293

implementando, 293-298

*IComparable*, interface, 287, 394

identificadores, 60-61

escopo de, 86

nomeando, 269-270

reservados, 60

sobrecregados, 87-97

identificadores não públicos, 165

identificadores públicos, convenções de nomeação para, 165

*IDisposable*, interface, 319

*IDraw*, interface, 292-293

implementando, 293-298

*IEnumerable*, interface, 414, 581

implementando, 419

*IEnumerable*, objetos, unindo, 686

*if*, instruções, 109-116, 121-122

em cascata, 111-112, 116

escrevendo, 112-115

expressões booleanas em, 110

instruções de bloqueio, 110-111

sintaxe, 109-110

igualdade (*==*), operador, 106

precedência e associatividade de, 109

*Image.Source*, propriedade, 482

Implement Interface Explicitly, comando, 294-295

implementações de propriedades

virtuais, 336

implementações originais (de métodos), 271-273

incremento (*++*), operador, 76, 124,

457

indexadores, 347-354

chamando, 358

definindo, 350

em contexto combinado de leitura/gravação, 351

em interfaces, 354

em um aplicativo Windows, 355

escrevendo, 356

exemplo com e sem, 347

implementações virtuais, 354

métodos de acesso, 351

operadores com ints, 348

sintaxe, 347

sintaxe de implementação explícita de interface, 355

*versus arrays*, 352

índices de array, 227, 250

tipos inteiros para, 233

informações sobre erros, exibindo, 550-551, 564

inicialização

de campos, 165, 171, 172

de classes derivadas, 266-267

de estruturas, 215-216

de variáveis de array, 225-226

inicializadores de coleção, 246

inicializadores de objetos, 342

*InitialDirectory*, propriedade, 528

*InitializeComponent*, método, 55

*INotifyPropertyChanged*, interface, 604

*INotifyPropertyChanging*, interface, 604

*Insert*, método, 240, 398

instância do usuário do SQL Server Express, 599

desassociando de, 599-600

instâncias

de classes, atribuindo, 163

de formulários WPF, 522

instâncias de array

copiando, 229-230

criando, 224-225, 250

*instnwnd.sql*, script, 570, 581, 599

instruções, 59-60

executando iterações de, 140.

*Consulte também* semântica de instruções em loop, 59

sintaxe, 59

instruções de atribuição, 123

para classes anônimas, 180

instruções de blocos unchecked, 151

instruções de declaração, 62-63

instruções de iteração, 123

instruções de loop, 140

continuando, 132

*do*, 131-139

*for*, 129-131

saindo de, 131

*while*, 124-128, 129-131

- instruções *if* em cascata, 111-112, 116  
*Int*, argumentos, somando, 256-258  
*int*, parâmetros, passando, 186  
*int*, tipo, 63  
  tamanho fixo de, 150  
*int*, tipo de variável, 63  
*int*, valores  
  mínimos, procurando, 252-253  
  operações aritméticas em, 70-73  
*int.MaxValue*, propriedade, 150  
*int.MinValue*, propriedade, 150  
*int.Parse*, método, 72, 142, 148, 211  
*int?*, tipo, 190  
*Int32.Parse*, método, 69  
inteiros, convertendo valores de string em, 72, 77-78  
IntelliSense, 41-42  
  dicas, percorrendo, 42  
  ícones, 42, 43  
*interface*, palavra-chave, 286, 304, 309  
*interface*, propriedades, 336  
interfaces, 285-301  
  combinações, 308  
  contravariantes, 409  
  convenções de nome, 287  
  covariantes, 407  
  declarando, 309  
  definindo, 286-287, 292-293  
  explicitamente implementadas, 289-291  
  fazendo referência a classes por meio de, 288-289  
  herdando de, 285, 287-288  
  implementando, 287-288, 293-298, 309  
  palavra-chave de método parciais, 168  
  regras de utilização, 287  
  restrições de, 291  
  várias, 289  
interfaces de programação de aplicativos, 362  
interfaces do usuário, Microsoft  
  diretrizes para, 510  
interfaces genéricas  
  contravariantes, 409  
  covariantes, 407  
  variância, 405-412  
interoperabilidade entre plataformas, 717  
*InvalidOperationException*, exceções, 199  
*InvalidOperationException*, exceções, 154, 533, 585  
*InvalidOperationException*, rotina de tratamento catch, 155  
*Invoke*, método, 538-540  
  chamando, 537  
*IProducerConsumerCollection<T>*, classe, 701  
*is*, operador, 200-201  
*IsChecked*, propriedade, 505, 508  
  condição de nulidade, 536  
*IsDBNull*, método, 580, 595-596  
*IsPrimaryKey*, parâmetro, 582  
*IsSynchronizedWithCurrentItem*, propriedade, 608, 609  
*IsThreeState*, propriedade (controle *CheckBox*), 490  
*ItemsSource*, propriedade, 609  
*ItemTemplate*, propriedade, 609  
itens de menu  
  *Click*, eventos, 517-519  
  controles WPF como, 516  
  estilização de texto, 515  
  itens about, 520-522  
  itens filhos, 513  
  nomeando, 513, 517  
  teclas de acesso para, 512  
  tipos de, 515-516  
iteradores, 421  
*IValueConverter*, interface, 554-555  
  *Convert*, método, 556  
  *ConvertBack*, método, 556-557
- J**
- janelas de prompt de comando, abrindo, 570  
JavaScript Object Notation (JSON), 720  
*Join*, método, 436  
  parâmetros, 436  
JSON (JavaScript Object Notation), 720  
junções, 436, 586-590  
  de origens de dados, 686
- K**
- Key*, propriedade, 245  
Knuth, Donald E., 390
- L**
- Lambda Calculus, 372  
Language Integrated Query (LINQ), 427  
  *All*, método, 439  
  *Any*, método, 439  
  avaliação adiada, 444  
  *BinaryTree*, objetos, 439  
  definindo uma coleção enumerável, 444  
  filtrando dados, 432  
  *Intersect*, método, 439  
  *Join*, método, 436  
  junções por igualdade, 439  
  juntando dados, 436  
  métodos de extensão, 444
- métodos genéricos *versus* não genéricos, 447  
operadores de consulta, 437  
*OrderBy*, método, 433  
selecionando dados, 430  
*Select*, método, 430  
*Skip*, método, 439  
*Take*, método, 439  
*Union*, método, 439  
utilizando, 428  
*Where*, método, 433  
last-in, first-out (LIFO) mecanismos, 242-243  
Lei de Moore, 633  
lendo recursos, 697-698  
*Length*, propriedade, 227-228, 250  
LINQ, consultas, 681  
  paralelizando, 683-687, 713  
LINQ. Consulte Language Integrated Query (LINQ)  
LINQ to Entities, 598  
  consultando dados com, 605-606  
LINQ to SQL, 567, 581-596  
  busca adiada, 585-586, 590-591  
  consultando bancos de dados com, 583-585, 591-596  
DataContext, classe, personalizada, 591-592  
extensões para, 597  
junções, 586-590  
mapeamento de tipos de dados, 593  
novos bancos de dados e tabelas, criando, 583  
relações de tabelas, 586-590  
*List<Object>*, objetos, 410  
*List<T>*, classe de coleções genérica, 410  
*ListView*, controles  
  opções de exibição, 610  
  para aceitar entrada do usuário, 622  
*View*, elemento, 609  
*LoadProperty<T>*, método, 613  
*LoadWith*, método, 590-591  
*Locals*, janela, 136-138  
*lock*, instruções, 691-693, 713  
*lock*, palavra-chave, 691  
*long*, tipo de dado, 63  
loops  
  iterações independentes, 656  
  paralelizando, 650-653, 655, 679  
*LostFocus*, evento, 541
- M**
- Main*, método, 40  
  para aplicativos de console, 41  
  para aplicativos gráficos, 55-56  
*MainWindow*, classe, 55

- MainWindow*, construtores, código de menu de atalho em, 526-527  
*MainWindow.xaml.cs*, arquivo, código para, 54-55  
*ManualResetEventSlim*, classe, 693-714  
*ManualResetEventSlim*, objetos, 713  
*Margin*, propriedade, 52, 479-480, 511  
 MARS (multiple active result sets), 577  
*Math*, classe, 163
  - Sqrt*, método, 173, 175*Math.PI*, campo, 163  
*MathsOperators*, código de programa, 71-73  
 mecanismos de bloqueio de primitivas de sincronização, 695-697  
 membros de classe protegidos, acesso a, 274  
 memória
  - alocação para novos objetos, 311-312
  - organização de, 194-196
  - para arrays, 223
  - para *Hashtables*, 244
  - para tipos de classe, 183
  - para tipos-valor, 183
  - para variáveis de tipos de classe, 163
  - recuperando, 311. *Consulte também* coleta de lixo
 memória de pilha, 195-196, 210, 701
  - empilhando, desempilhando e consultando itens em, 701
  - estruturas em, 210
 memória física. *Consulte também* memória
  - consultando quantidade de, 642
 memória heap, 195-196, 311, 312
  - alocações de, 311
  - devolvendo memória para, 312*Menu*, controles, 509, 510
  - adicionando, 511, 540*MenuItem*, elementos, 512, 513
  - aninhada, 515
  - Header*, atributo, 512*MenuItem*, objetos, 540  
*MenuItem\_Click*, métodos, 517  
 menus, 509-510
  - barras de separação em, 513, 540
  - criando, 510-516, 540
  - DockPanel*, controles, adicionando a, 511
  - em cascata, 515
 menus de atalho, 524-527
  - adicinando em código, 526-527
  - associando a formulários e controles, 526-527, 540
  - criando, 524-528, 540
 criando dinamicamente, 540
  - desassociando de formulários WPF, 527
  - para controles caixa de texto, 524-525
  - para controles *DatePicker*, 525
 menus de contexto. *Consulte* menus de atalho
  - menus pop-up. *Consulte* menus de atalho*MergeOption*, propriedade, 616  
*Message*, propriedade, 143, 149  
 Message Passing Interface (MPI), 632  
*MessageBox.Show*, instrução, 57  
*methodName*, 80, 83
  - método *Next* de *SystemRandom*, 225
  - método *Split* da classe *String*, 685
  - método *Sqrt* da classe *System.Math*, 173
 métodos, 161
  - abstratos, 302-303, 309
  - anônimos, 373
  - argumentos, 84. *Consulte também* argumentos
    - arrays de parâmetros e, 259, 261
    - chamando, 83, 85, 104
    - combinações de palavras-chave para, 308
    - compartilhando informações entre, 86-87
    - comprimento de, 83
    - construtores, 166-167
    - corpos de, 79
    - criando, 79-85
    - de acesso, *get* e *set*, 330
    - declarando, 80-81, 104
    - de classe, 176
    - de retorno de chamada, registrando, 666
    - em interfaces, 286-287
    - encapsulamento de, 162
    - entrando e saindo de, 93-95, 104
    - escopo de, 86-87
    - escrevendo, 88-95
    - estáticos (não de instância), 174. *Consulte também* métodos estáticos
    - estáticos públicos, escrevendo, 178
    - examinando, 82-83
    - geração de assistentes de, 89-92
    - global, 80
    - implementações de, 271-273
    - instruções em, 59
    - nomeando, 79, 165
    - ocultando, 269-270
    - override, 271-272
    - parâmetros opcionais para, 97-98, 100-104, 258
    - polimórficos, regras de utilização, 272
    - retornando dados de, 81-83, 104
    - saindo, 81
    - selados, 303-304
    - sobrecregados, 41
    - sobrecregando, 87-88, 251
    - sobrescrevendo, 304
    - tipos de retorno, 80, 104
    - utilitários, 174
    - valores codificados para, 500
    - virtuais, 270-271, 272-273
    - Web, 717
 métodos de conversão, 554-555
    - criando, 555-557
 métodos de descarte, 317
    - escrevendo, 324
    - protegido contra exceções, 317-318, 321-324
 métodos de evento, 503
    - escrevendo, 508
    - nomeando, 504
    - para itens de menu, 540
    - removendo, 504
 métodos de extensão, 279-283
    - criando, 280-282
    - Single*, método, 585
    - sintaxe de, 280
 métodos de instância
    - definição de, 172
    - escrevendo e chamando, 172-174
 métodos estáticos, 174-180
    - chamando, 181
    - declarando, 181
    - escrevendo, 178
    - métodos de extensão, 280
 métodos genéricos, 402-405
    - parâmetros, 403
    - restrições, 403
 métodos sobrecregados, 41, 87-88
    - métodos virtuais
      - declarando, 270-271, 283
      - polimorfismo e, 272-273
 Microsoft .NET Framework. *Consulte* .NET Framework
    - Microsoft.Win32*, namespace, 528
    - Microsoft Message Queue (MSMQ), 716
    - Microsoft SQL Server 2008 Express, 567. *Consulte também* SQL Server
    - Microsoft Visual C#. *Consulte* C#
    - Microsoft Windows Presentation Foundation. *Consulte* aplicativos WPF
    - Min*, método, 252, 253
    - modelos de dados de entidade, 597
      - gerando, 600-604
    - modificador ponto de interrogação (?) para valores nullable, 189
    - modos de exibição de código, 49
    - modos design, 49
    - Monitor*, classe, 692
    - Moore, Gordon E., 633

- MouseButtonEventArgs*, parâmetro, 299  
*MoveNext*, método, 414  
 MPI (Message Passing Interface), 632  
 MSMQ (Microsoft Message Queue), 716  
 multiple active result sets (MARS), 577  
 multiplicação, operador, 68  
   precedência de, 73, 109  
 multitarefa  
   considerações sobre, 634-635  
   definição de, 634  
   implementando, 634-660  
   motivos para, 632-633  
 multithreading, 635
- N**
- Name*, parâmetro, 582  
*Name*, propriedade, 53, 484  
 namespaces, 46-49  
   assemblies e, 48  
   incorporando ao escopo, 47  
 Namespaces de nível superior, 47  
 .NET, common language  
   runtime, 362  
 .NET Framework, biblioteca de classes  
   classes em, 48  
   namespaces em, 47  
 .NET Framework Remoting, 716  
 .NET Framework, 362  
   algoritmo de busca, 636  
   LINQ, extensões, 682  
   multithreading, 635  
   parallelismo, determinando, 636, 649-651, 656, 671  
   pools de threads, 635-636  
   primitivas de sincronização, 692  
   *TaskScheduler*, objeto, 637  
*new*, método, declarando, 269-270  
*new*, operador, funcionalidade de, 311-312  
*New*, palavra-chave, 163, 250, 270, 308  
   para classes anônimas, 179  
   para construtores, 181  
   para instâncias de array, 224  
*<New Event Handler>*, comando, 504, 517, 525  
*New Project*, caixa de diálogo, 37, 38  
 nomes de arquivo, asteriscos no lugar de, 44  
 nomes totalmente qualificados, 47  
 Northwind, banco de dados, 568  
   criando, 568-570  
   desassociando de instância do usuário, 599-600  
 Orders, tabela, 591-594  
   redefinindo, 570  
 Suppliers, aplicativo, 607, 614-616, 627-628  
 Suppliers, tabela, 586  
 Northwind Traders, 568  
 NOT (~), operador, 348  
 NOT, operador (!), 106  
 notação de ponto, 166  
 notação húngara, 62  
 notação octal, convertendo números em, 132-135  
 notificação de método  
   termino, 662  
*NotImplementedException*, exceções, 234, 295  
*NotOnCanceled*, opção, 639  
*NotOnFaulted*, opção, 639  
*NotOnRanToCompletion*, opção, 639  
*Null*, valores, 188-191, 202-203  
   em bancos de dados, 579, 580, 595-596  
   em colunas de tabela de banco de dados, 582  
*NullReferenceException*, exceções, 376  
*NumCircles*, campo, 175-176  
 números, convertendo em strings, 132-135  
 números complexos, 460
- O**
- OASIS (Organization for the Advancement of Structured Information Standards), 718  
*obj*, pasta, 45  
*object*, palavra-chave, 197  
*object*, tipo, 91, 239  
*Object.Finalize*, método, substituindo, 313-314  
*ObjectContext*, classe, 604  
   *Refresh*, método, 616  
*ObjectContext*, objetos  
   cache de dados, 615  
   rastreamento de mudanças, 616  
*objectCount*, campo, 177  
*ObjectQuery<T>*, objetos, consultas de banco de dados baseadas em, 614-615  
*ObjectSet*, coleções  
   *AddObject*, método, 628  
   *DeleteObject*, método, 628  
   excluindo entidades de, 620  
*ObjectSet<T>*, classe de coleções, 608  
*ObjectStateEntry*, classe, 618  
 objetos  
   acesso a membros, 312  
   alcançáveis e inalcançáveis, 316  
   atribuindo, 267-268  
   bloqueando, 691-693  
   criando, 163, 169-172, 311-312  
   definição de, 164  
   destruição de, 312, 314-315  
   desvantagens, 385  
   duração de, 311-316  
   fazendo referência por meio de interfaces, 288  
   inicializando por meio de propriedades, 340  
   memória para, 195  
   na memória, atualizando, 615-616  
   referências a, 312  
   vinculando a propriedades de, 563-564  
   vinculados, referências a, 558  
 objetos de exceção, 153  
   examinando, 143-144, 674-675  
 escondendo métodos, 269-270  
*ok\_Click*, método, 57  
*okayClick*, método, 377  
*OnlyOnCanceled*, opção, 639  
*OnlyOnFaulted*, opção, 639  
*OnlyOnRanToCompletion*, opção, 639  
*Open*, caixa de diálogo, 126  
*Open*, método, chamando, 595-596  
*Open File*, caixa de diálogo, 530  
*OpenFileDialog*, classe, 126, 528  
*openFileDialog*, objetos, 126  
*openFileDialogOk*, método, 126  
*OpenText*, método, 127  
 operações aritméticas, 68-75  
   resultados, tipo, 69  
 operações de atualização, atualizações conflitantes no banco de dados, 616-619, 628  
   executando, 615-616  
 operações de espera  
   *CurrentCount* propriedade, 695  
   tokens de cancelamento para, 700  
 operações de salvamento, atualizações da barra de status em, 537-539  
 operações demoradas  
   cancelando, 664-677  
   capacidade de resposta, melhorando com objeto *Dispatcher*, 661-664  
   dividindo em tarefas paralelas, 646-649  
   medindo o tempo de processamento, 644-646  
   paralelizando com classe *Parallel*, 651-653  
 operações independentes, 655-656  
 operações paralelas  
   agendando, 688  
   desempenho imprevisível de, 688-691  
 operações relacionais, 433  
 operações single-threaded, 704-708  
 operador de atribuição (=), 63, 106, 123-130  
   precedência e associatividade de, 74, 109

operadores, 451-472  
 --, 76, 457  
 -=, 124, 364, 376  
 +, 451  
 ++, 75, 76, 124, 457  
 +=, 124, 363, 375  
 \*=, 124  
 /=, 124  
 adição composta, 140  
 AND (&), 348  
 as, 200-201, 268  
 associatividade e, 74, 451  
 asterisco (\*), 68, 201-202  
 barra diagonal (/), 68  
 binários, 451  
 comparando em estruturas e classes, 458  
 conceitos básicos, 451-456  
 conversão definida por usuário, 467  
 curto-circuito, 108  
 de adição, 68, 73, 109  
 de atribuição composta, 123-124, 456  
 de consulta, 437  
 de grupo, 438  
 desigualdade (!=), 106  
 deslocamento à esquerda (<<), 348  
 de ponto (), 312, 452  
 de subtração composta, 140  
 divisão, 68, 73  
 em nível de bit, 349  
 estáticos, 453  
 formas de pós-correção, 76-77  
 formas de prefixo, 76-77  
 igualdade (==), 106, 109, 463  
 implementando, 459-465  
 incremento (++), 75, 76, 124, 457  
 interoperabilidade de linguagens, 456  
*is*, 200-201  
*join*, 439  
 lógico AND (&&), 107, 121-122  
 lógico OR (||), 107, 121-122  
 multiplicação, 68, 73, 109, 451  
 multiplicidade, 452  
*new*, 311-312  
 NOT (~), 348  
 NOT (!), 106  
 números complexos, 460  
 operandos, 452  
 OR (|), 348  
*orderby*, 438  
 pares de, 458  
 precedência, 73-74, 451  
 públicos, 453  
 resto (módulo), , 69  
 restrições, 452

símbolo de porcentagem (%), 69, 124  
 simétricos, 454, 468  
 simulando [], 452  
 sobrecregando, 452  
 tipos de dados e, 69-70  
 XOR (^), 348  
 operadores aritméticos  
 checked e unchecked, 151  
 precedência, 73-74  
 usando, 70-73  
 operadores booleanos, 106-109  
 curto-circuito, 108  
 precedência e associatividade, 108-109  
 operadores de conversão, 466, 467  
 escrevendo, 469  
 operadores de decremento, 457  
 -, operador, 76  
 ++, operador, 124  
 operadores lógicos  
 curto-circuito, 108  
 operador lógico AND (&&), 107, 121-122  
 operador lógico OR (||), 107, 121-122  
 operadores lógicos condicionais, 107-109  
 curto-circuito, 108  
 operadores primários, precedência e associatividade de, 108  
 operadores relacionais, 106  
 precedência e associatividade de, 109  
 operadores unários, 76, 451  
 precedência e associatividade de, 108  
 operandos, 68  
*OperationCanceledException*, exceções, 673, 700  
*OperationContract*, atributo, 723  
*OptimisticConcurrencyException*, exceções, 618, 619  
*OptimisticConcurrencyException*, rotina de tratamento, 626, 628  
 OR (|), operador, 348  
 OR (OU) condicional, operador, precedência e associatividade de, 109  
 ordem z de controles, 483  
*OrderBy*, método, 433  
*orderby*, operador, 438  
*OrderByDescending*, método, 434  
 Organization for the Advancement of Structured Information Standards (OASIS), 718  
 origem, arquivos, visualizando, 39  
 origens de dados, unindo, 686  
 origens de eventos, 374

*OtherKey*, parâmetro, 587  
*out*, modificador, arrays de parâmetros e, 254  
*out*, palavra-chave, 192-193, 408  
*out*, parâmetros, 191-194  
*OutOfMemoryException*, exceções, 196  
 arrays multidimensionais e, 231  
*Output*, ícone, 135, 136  
*Output*, janela (Visual Studio 2010), 43  
*OverflowException*, exceções, 143, 150, 152  
*OverflowException*, rotina de tratamento, 152  
*override*, métodos, declarando, 271-272  
*override*, palavra-chave, 271, 272, 304, 308  
*OverwritePrompt*, propriedade, 528

**P**

painéis de layout, 478  
 ordem z de controles, 483  
 palavras-chave, 60-61  
*abstract*, 302, 308, 309  
*base*, 266, 271  
*bool*, 121-122  
*case*, 117  
*catch*, 142  
*checked*, 158  
*class*, 162, 181  
*const*, 176  
*default*, 117  
 de método  
 combinações, 308  
*DynamicResource*, 486  
*else*, 109, 110  
*enum*, 205, 222  
 equivalentes .NET, 211  
*get e set*, 330  
 IntelliSense, listas de, 41  
*interface*, 286, 304, 309  
*lock*, 691  
*new*, 163, 250, 270, 308  
*object*, 197  
*out*, 192-193, 408  
*override*, 271, 272, 304, 308  
*params*, 251, 253, 254  
*partial*, 168  
*private*, 164, 177, 274, 308  
*protected*, 274, 308  
*public*, 164, 274, 308  
*ref*, 191  
*return*, 81  
*sealed*, 303, 304, 308, 309  
*set*, 330  
*static*, 175, 177, 181  
*StaticResource*, 486  
*string*, 184

- struct*, 212, 222  
*this*, 171-172, 178, 280  
*try*, 142  
*unchecked*, 150-151  
*unsafe*, 201-202  
*var*, 76-77, 180  
*virtual*, 271, 272, 283, 304, 308  
*void*, 80, 81, 83  
*yield*, 422  
 paralelização de consultas LINQ, 682-688  
**Parallel**, classe  
 abstraindo tarefas com, 649-656  
 para operações independentes, 653, 655-656  
*Parallel.ForEach<T>*, método, 650  
 quando utilizar, 653  
*Parallel.For*, construção, 689-690  
*Parallel.For*, método, 649, 650, 652-653, 679  
 cancelando, 671  
 quando utilizar, 656  
*Parallel.ForEach*, método, 679  
 cancelando, 671  
 quando utilizar, 656  
*Parallel.Invoke*, método, 650, 659  
 quando utilizar, 653-656  
 Parallel LINQ, 681-687  
*ParallelEnumerable*, objetos, 687  
*ParallelLoopState*, objetos, 650, 671  
**ParallelQuery**, classe  
*AsOrdered*, método, 687  
*WithCancellation*, método, 688, 713  
*WithExecutionMode*, método, 687  
*ParallelQuery*, objetos, 682, 686  
*parameterList*, 80  
 parâmetros  
 aliases para argumentos, 191-192  
 nomeados, 104  
 nomeando, 91  
 opcionais, definindo, 97-98  
 passando, 98  
 tipos de, especificando, 80  
 tipos de método, 184  
 tipos-referência, 184-188, 191  
 valores padrão para, 97-98  
 parâmetros de referência  
*out* e *ref*, modificadores para, 194  
 utilizando, 185-188  
 parâmetros de tipo, 388  
*out*, palavra-chave, 408  
 Parâmetros de valor  
*out* e *ref*, modificadores para, 194  
 utilizando, 185-188  
 parâmetros nomeados, 104  
 passando, 98  
 parâmetros opcionais, 96-97  
 ambiguidades com, 98-103  
 definindo, 97-98, 100-101, 104  
*versus* arrays de parâmetros, 258-261  
*params*, objeto [], 255  
*params*, palavra-chave, 251, 253  
 sobrecarregando métodos e, 254  
*params* métodos, prioridade de, 254  
 parênteses  
 em expressões booleanas, 107, 125  
 em instruções *if*, 110  
 precedência de operadores e, 73  
 pares de caracteres combinados, 42  
 pares de tecla/valor, 388  
 como tipos anônimos, 246  
 em *Hashtables*, 244  
 em *SortedLists*, 245  
*Parse*, método, 85, 133  
*partial*, estruturas, 168  
*partial*, palavra-chave, 168  
 particionando dados, 682  
*ParticipantCount*, propriedade, 698  
*ParticipantsRemaining*, propriedade, 698  
*PascalCase*, esquema de nomes, 165  
*Pass.Value*, método, 186-187  
*Password*, parâmetro, 573  
*Path*, marca, 563-564  
 Plain Old XML (POX), 720  
 PLINQ (Parallel LINQ), 681  
 melhorando desempenho com, 682-687  
 PLINQ, consultas  
 cancelamento de, 713  
 opções de paralelismo para, 687-688  
 polimorfismo  
 métodos virtuais e, 272-273  
 testando, 278  
 ponteiros, 200-202  
 pontos de âncora de controles, 479-480  
 pontos de entrada do programa, 40  
 pool de conexões, 579  
 pools de recursos, controle de acesso, 695-696  
 pools de threads, 635  
 POX (Plain Old XML), 720  
 precedência, 451  
 controlando, 73-74  
 de operadores booleanos, 108-109  
 substituindo, 77-78  
*Press any key to continue* (Pressione qualquer tecla para continuar), aviso, 45  
 pressionamentos de teclas, examinando, 620-621  
 primitivas de sincronização  
 cancelamento e, 700  
 na TPL, 693-699  
*private*, métodos, 164-165  
*private*, palavra-chave, 164, 177, 274, 308  
*private*, qualificador, 90  
*PrivilegeLevel*, enumeração, 553  
 adicionando, 552  
 privilégios de Administrador, para exercícios, 567-569  
 problemas de conflito entre nomes, 46  
 processadores  
 dual-core (núcleo duplo), 634  
 girando (spinning), 683  
 multicore, 633-634  
 quad-core, 634  
 processamento paralelo, 708-710  
 benefícios de, 632-633  
 implementando com classe *Task*, 640-649  
 produtores, 701  
*Program*, classe, 40  
*Program.cs*, arquivo, 40  
*ProgressChanged*, evento, 536  
 projetos, procurando em, 66  
*Properties*, janela, 481  
 exibindo, 52  
*Properties*, pasta, 40  
*propriedade Concurrency Mode* da classe *EntityObject*, 617  
 propriedades, 329-346  
 acessibilidade, 333  
 aplicativos Windows, 337  
 automáticas, 338-339, 342  
 contexto de gravação, 331  
 contexto de leitura, 331  
 contexto de leitura/gravação, 331  
 declarações, 330  
 estáticas, 332  
*get*, bloco, 329  
*get* e *set*, palavras-chave, 330  
 implementações explícitas, 337  
 implementações virtuais, 336  
 inicializadores de objetos, 342  
 inicializando objetos, 340  
 interface, 336  
 motivos para definir, 338-339  
 privado, 333  
 protegido, 333  
 público, 330, 333  
 restrições, 334  
 segurança, 333  
*set*, bloco, 329  
 sintaxe de declaração, 329  
 somente gravação, 332  
 somente leitura, 332  
 uso adequado, 335  
 utilizando, 331  
 vinculando a propriedades de controle, 557-558, 563-564  
 vinculando a propriedades de objeto, 563-564

propriedades de limite,  
*BindingExpression*, objeto de, 564  
 propriedades de projeto, definindo,  
 150  
*protected*, palavra-chave, 274, 308  
 provedores de dados, 567-568  
*public*, métodos, 164-165  
*public*, palavra-chave, 164, 274,  
 308  
*public*, propriedades, 330

**Q**

*Queue*, classe, 242  
*Queue*, tipo de dado, 386  
*Queue*<*T*>, classe, versão protegida  
 quanto a threads, 701  
*Quick Find*, comando, 66

**R**

radio button. Ver controles botão de  
 opção  
*RanToCompletion*, estado de tarefa,  
 670  
 rastreamento de mudanças, 616  
*Read*, método, 575  
*reader.ReadLine*, método, 127  
*ReaderWriterLockSlim*, classe,  
 697-714  
*ReadLine*, método, 90  
*readonly*, campos, 232  
 recursos  
   escrevendo em, 697-698  
   lendo, 697-698  
 recursos compartilhados.  
   acesso exclusivo a, 713  
 recursos de janela, adicionando a  
 menus de atalho, 523  
 recursos estáticos, regras de escopo,  
 486  
*ref*, modificador, arrays de parâmetros  
 e, 254  
*ref*, palavra-chave, 191  
*ref*, parâmetros, 191-194  
   passando argumentos para,  
 202-203  
 refatorando código, 92, 302  
*References*, pasta, 40  
 referências, adicionando, 48  
 referências pendentes, 314  
*Refresh*, método, 616, 628  
   chamando, 618  
*RefreshMode*, enumeração, 618  
*Register*, método, 666  
 regras de sintaxe, 59  
   para identificadores, 60  
   para instruções, 59  
 regras de validação, 542  
   adicionando, 543-550  
   especificando, 548

exceções para, detectando,  
 550-551  
 relatório de problemas, configurando,  
 147  
*Release*, método, 713  
*Release*, pasta, 46  
*Remove*, método, 240  
*RemoveParticipant*, método, 698  
 Representational State Transfer  
 (REST), 716, 720  
*Reset*, método, 498-502  
   chamando, 518  
*Resources*, elementos, 484  
 REST, modelo, 716, 720  
 resto (módulo), operador, 69  
   validade de, 70  
 restrições, com genéricos, 390  
 restringindo conversões, 467  
*Result*, propriedade, 678  
*result =*, cláusula, 83  
 resultados, sequência de retorno de,  
 687  
*return*, instruções, 81-82, 173  
   fall-through, evitando com, 118  
*return*, palavra-chave, 81  
*returnType*, 80  
 rotina de tratamento de evento para  
 eventos about, 520  
   de execução prolongada, simulando,  
 530-531  
   em aplicativos WPF, 502-508  
   para ações de menu, 540  
   para eventos Closing, 506-508  
   para eventos de salvamento,  
 519-520  
   para novos eventos, 517-518  
 testando, 522-523  
*RoutedEventArgs*, object, 377  
*RoutedEventHandler*, 377  
*Run*, método, 88  
*Run as administrator*, comando,  
 568  
*Run To Cursor*, comando, 93, 135  
*Running*, estado de tarefa, 670  
 runtime, paralelização de consultas,  
 687  
*RunWorkerAsync*, método, 536  
*RunWorkerCompleted*, evento, 536

**S**

saindo de métodos, 93-95  
 save, rotina de tratamento de evento,  
 519  
*Save File*, caixa de diálogo, 528, 529  
*SaveChanges*, método, 619  
   chamando, 615-616, 628  
   falha de, 616  
*saveChanges\_Click*, método, 626  
*SaveFileDialog*, classe, 528-530, 540

ScreenTips  
   no depurador, 94  
   para variáveis, 63  
*sealed*, palavra-chave, 303, 304, 308,  
 309  
 segurança, codificando usuário  
   nomes e senhas e, 573  
*Select*, método, 430  
   parâmetros de tipo, 431  
*SelectedDate*, propriedade, 113  
   condição de nulidade, 536  
 semáforos, 693  
 semântica, 59  
*SemaphoreSlim*, classe, 695-714  
*SemaphoreSlim*, objetos, 713  
*Separator*, elementos, 513, 540  
 sequências de instruções, executando,  
 679  
 serialização, 723  
   de chamadas a método, 711-712  
*ServiceContract*, atributo, 723  
*Set*, elementos, 488  
*set*, métodos de acesso, 330, 331  
   para consultas de banco de dados,  
 587-588  
*set*, palavra-chave, 330  
*Shape*, classe, 305  
*Shift+F11 (Step Out)*, 94  
*Show All Files*, comando, (Solution  
 Explorer), 45  
*ShowDialog*, método, 522, 528, 624  
*showDoubleValue*, método, 68  
*showFloatValue*, método, 66  
*showIntValue*, método, 67  
*showResult*, método, 82, 85  
*SignalAndWait*, método, 698  
 símbolo arroba (@), 574  
 símbolo de porcentagem (%), operador, 69  
 Simple Object Access Protocol.  
   Consulte SOAP (Simple Object Access  
 Protocol), 716  
 sinal de igualdade (=), operador, 74.  
   Consulte também operador de atribuição (=)  
 sinal de mais (+), operador, 68  
 sinal de subtração (-), operador, 68  
 sincronização de threads, 698, 713  
*Single*, método, 585  
 sintaxe de declaração de propriedade,  
 329  
*Sleep*, método, 531  
*.sln*, sufixo, 65  
 SOAP (Simple Object Access Protocol),  
 716-720  
   falando, métodos, 729  
   função, 717  
   segurança, 718  
 Web services, 717

- sobrecarregando, 251  
 ambíguo, 254  
 construtores, 166-167  
 parâmetros opcionais e, 96-97  
 sobrescrevendo métodos, 271  
     métodos selados e, 303-304  
**Solicitações a objeto Dispatcher**, 537-539  
**Solução de nível superior**, arquivos, 40  
**Solution Explorer**, acessando código em, 66  
**Solution Explorer**, painel, 39  
*SortedList*, classe, 245  
*SortedList*, objetos de coleção em HashTables, 247  
*Source*, propriedade, 643  
*SpinWait*, operações, 683  
*Split*, método, 685  
**SQL Configuration Manager**, ferramenta, 569  
**SQL SELECT**, instruções, 578, 585  
**SQL Server**  
     iniciando, 569  
     registrando no, 569  
     vários conjuntos de resultados ativos, 577  
**SQL Server**, autenticação, 573  
**SQL Server**, banco de dados. *Consulte também* bancos de dados  
     concedendo acesso a, 599-600  
**SQL Server Express**, instância do usuário, 599  
**SQL UPDATE**, comandos, 615-616  
*Sqlcmd*, utilitário, 569  
*SqlCommand*, objetos, criando, 574, 595-596  
*SqlConnection*, objetos, criando, 571, 595-596  
*SqlConnectionStringBuilder*, classe, 572  
*SqlConnectionStringBuilder*, objetos, 572, 594  
*SqlDataReader*, classe, 575, 576  
*SqlDataReader*, objetos, 575  
     busca de data com, 595-596  
     criando, 595-596  
     fechando, 577  
     lendo dados com, 576  
*SqlException*, exceções, 571-572, 594  
*SqlParameter*, objetos, 574-575  
*Sqrt*, método, 173, 174  
     declaração de, 175  
*Stack*, classe, 242-243  
*Stack<T>*, classe, versão protegida quanto a threads, 701  
*StackPanel*, controles, 478, 508  
     adicionando, 492  
*Start*, método, 533, 637  
*Start Debugging*, comando, 45  
*Start Without Debugging*, comando, 45  
*StartNew*, método, 657, 678  
*StartupUri*, propriedade, 56-57, 489  
*StateEntries*, propriedade, 618  
*static*, palavra-chave, 175, 177, 181  
*StaticResource*, palavra-chave, 486  
*Status*, propriedade, 670  
*StatusBar* controles, adicionando, 537  
*Step Into*, botão (barra de ferramenta Debug), 93-95  
*Step Out*, botão (barra de ferramenta Debug), 94-95  
*Step Over*, botão (barra de ferramenta Debug), 94-95  
*StopWatch*, tipo, 643  
*Storage*, parâmetro, 587  
*StreamWriter*, objetos, criando, 519  
*string*, palavra-chave, 184  
*String.Format*, método, 505, 610  
*StringBuilder*, objetos, 505, 506  
**strings**  
     convertendo em enumerações, 554  
     convertendo enumerações em, 206-207  
     definição de, 66  
     dividindo em arrays, 685  
     formatando argumentos como, 218  
     incluindo em outras strings, 124  
     strings de formatação, 92  
**strings de conexão**, 591, 594-596  
     armazenando, 604  
     construindo, 572  
     para construtor *DataContext*, 583-584  
**strings de formatação**, 92  
**strings de texto**. *Consulte também* strings  
     convertendo em inteiros, 72  
*struct*, palavra-chave, 212, 222  
*StructsAndEnums*, namespace, 208  
*Style*, propriedade, 484  
*<Style.Triggers>*, elemento, 488  
*sublinhado*, regras de sintaxe para, 60, 62  
*subtração*, operador, 68  
     precedência de, 73  
*switch*, instruções, 116-122  
     break, instruções em, 119  
     escrevendo, 119-122  
     regras de fall-through, 118-119  
     regras de utilização, 118-119  
     sintaxe, 117  
*System.Array* classe, 227  
*System.Collections*, namespace, 238  
*System.Collections.Concurrent*, namespace, 700  
*System.Collections.Generic*, namespace, 409  
*System.Collections.IEnumerable*, interface, 413  
*System.ComponentModel*, namespace, 536  
*System.Data*, namespace, 571  
*System.Data.Linq*, assembly, 591-592  
*System.Data.Objects*.*DataClasses*.*EntityObject*, classe, 604  
*System.Data.Objects*.*DataClasses*.*StructuralObject*, classe, 604  
*System.Data.SqlClient*, namespace, 571  
*System.GC.Collect*, método, 315, 324  
*System.IComparable*, interface, 394  
*System.Object*, classe, 197  
     classes derivadas de, 265-266  
*System.Random*, classe, 225  
*System.Runtime.Serialization*, namespace, 723  
*System.ServiceModel*, namespace, 723  
*System.ServiceModel.Web*, namespace, 723  
*System.Threading*, namespace, 635  
     primitivas de sincronização em, 692  
*System.Threading.CancellationToken*, parâmetro, 665  
*System.Threading.Monitor*, classe, 692  
*System.Threading.Tasks*, namespace, 636, 649  
*System.ValueType*, classe, 264  
*System.Windows*, namespace, 475  
*System.Windows.Data*, namespace, 555  
*SystemException*, herança

**T**

- tabelas de banco de dados  
     chaves primárias, 582  
     classes de entidades, relações entre, 583-584  
*Column*, atributo, 582  
     consultando, 573-575  
     excluindo linhas em, 620, 628  
     juntando, 586-590  
     modelos de dados de entidade para, 600-604  
     modificando informações em, 628  
     novas, criando, 583  
     recuperando dados em, 611-614  
     recuperando linhas individuais, 585  
     relações muitas para um, 587-588

- relações um para muitos, 588-590  
*Table*, atributo, 582  
 tipo subjacente de colunas, 582  
 valores nulos em, 582  
*Table*, atributo, 582, 595-596  
*Table*, coleções, 585, 590  
 criando, 595-596  
*Table< TEntity >*, coleções como  
 membros públicos, 591  
*Table< TEntity >*, tipos, 584  
 tamanho da letra, uso em nomes de  
 identificador, 62  
 tarefas, 635-636  
   abortando, 664  
   abstraindo, 649-656  
   agendando, 638-639  
   cancelando, 664-677  
   continuações de, 638-640, 677,  
     678  
   coordenando, 681  
   criando, executando, controlando,  
     636-640, 678  
   esperando, 648, 678  
   manipulando exceções, 673-676,  
     679  
   paralelas, 632, 679  
   retornando valores de, 656-660,  
     678  
   sincronizando, 640, 647-649  
   status de, 670, 672, 676  
   threads da interface do usuário e,  
     660-664  
   tokens de cancelamento, 665  
 tarefas de execução demorada  
   executando em várias threads,  
     531-534  
   simulando, 530-531  
 tarefas simultâneas  
   desempenho imprevisível de,  
     688-691  
   sincronizando acesso a recursos,  
     691  
*TargetType*, atributo, 486-488  
*Task*, classe, 635  
   parallelismo, implementando com,  
     640-649  
   *Wait*, método, 640  
   *WaitAll*, método, 678  
*Task*, construtores, 636-637  
   sobrecargas de, 637  
*Task*, objetos  
   *ContinueWith*, método, 638  
   criando, 636-637, 648, 678  
   diversos, 635  
   executando, 637-638  
   *Start*, método, 678  
   *Status*, propriedade, 670  
   *Wait*, método, 678  
 Task Parallel Library. Consulte TPL  
   (Task Parallel Library)  
*Task< byte[] >*, objetos, criando, 659  
*Task< TResult >*, objetos, 657-660,  
   678  
*TaskContinuationOptions*, tipo,  
   638-639, 677  
*TaskCreationOptions*, enumeração,  
   638  
*TaskFactory*, classe, 639-640  
*TaskFactory*, objetos, 639-640  
   *StartNew*, método, 657, 678  
*Tasks*, 539  
*TaskScheduler*, classe, 638  
*TaskScheduler*, objetos, 637  
 teclas de acesso para itens de menu,  
   512  
*TEntity*, parâmetro de tipo, 584  
 teoria das árvores binárias, 390  
*TestIfTrue*, método, 683  
*Text*, propriedade, definindo, 66-67  
*TextReader*, classe, 127  
   método de descarte de, 317  
 "The name 'Console' does not exist in  
 the current context", erro, 47  
*ThenBy*, método, 434  
*ThenByDescending*, método, 434  
*this*, palavra-chave, 171-172, 178,  
   280  
   com indexadores, 350  
*ThisKey*, parâmetro, 587  
*Thread*, classe, 531  
   *Start*, método, 533  
*Thread*, objetos, 635  
   criando novos, 533  
   fazendo referência a métodos em,  
     540  
*Thread.Sleep*, método, 656  
*Thread.Sleep*, método, 692  
*ThreadPool*, classe, 635  
 threads, 635-636  
   agendando, 635-636  
   bloqueando, 695-696  
   bloqueando dados, 691-693  
   de segundo plano, 534-536  
   definição de, 315, 531  
   diversos, 531-532  
   em repouso, 691-692  
   escrevendo para recursos, 697  
   esperando eventos, 693-694  
   girando o processador, 692  
   interrompendo execução de,  
     698-699  
   lendo recursos, 697  
   número ideal de, 636  
   paralelas, 646-649  
   pools de recursos, acessando,  
     695-696  
 restrições de acesso a objetos, 534  
 sincronizando, 683, 698, 713  
 suspendendo, 693-694  
 wrapper para, 536  
 threads de interface do usuário  
   copiando dados de, 534-537  
   executando métodos em nome de  
   outras threads, 537-539  
   tarefas e, 660-664  
 threads de segundo plano  
   acesso a controles, 540  
   copiando dados em, 534-536  
   executando operações em, 540  
   para operações de execução prolongada, 531-534  
 threads simultâneas, 632. Consulte  
   também multitarefa; threads  
*thread-safe*, 700-702  
*ThreadStatic*, atributo, 693  
*Throw*, instruções, 158,  
   escrevendo, 154  
   fall-through, impedindo com, 118  
*ThrowIfCancellationRequested*, método, 672-673  
 Ticket Ordering, aplicativo  
   converter classe e métodos, criando, 555-557  
   examinando, 543-544  
   exibindo número de tíquetes, 544-546  
   Ligando controle caixa de texto a  
   propriedade de classe, 547-550  
   nível de privilégio e número de  
   tíquetes, validando, 552-554  
*TickerOrder*, classe com lógica de  
   validação, 546-547  
*til (~)*, modificador, 313, 324  
 tipos, estendendo, 280  
 tipos anônimos em arrays, 226-227,  
   229  
 tipos construídos, 389  
 tipos de classe, copiando, 183-188  
 tipos de dados  
   *bool*, 64, 106  
   *char*, 64, 574  
   *DateTime*, 113, 116  
   de enumerações, 208  
   *decimal*, 63  
   *double*, 63  
   *float*, 63  
   IntelliSense, listas de, 41  
   *long*, 63  
   mapeamento, 582  
   operadores e, 69-70  
   protegido quanto a threads, 710  
   *Queue*, 386  
 tipos de dados primitivos, 63-68  
   exibindo valores de, 64-65

- switch*, instruções em, 118  
 tamanho fixo de, 150  
 utilizando em código, 65-66  
 tipos de estrutura, declarando, 222  
 tipos de string, 64, 184, 506  
 tipos inteiros, enumerações baseadas em, 208  
 tipos nullable, 188-191  
   propriedades de, 190-191  
*Value*, propriedade, 190-191  
 tipos-referência, 183  
   arrays. Consulte arrays  
   destrutores para, 313  
   heap, criação em, 195  
*Object*, classe, 197  
*Title*, propriedade, 53, 528  
*TKey*, 389  
*ToArray*, método para recuperar dados, 585-586, 590  
 tokens de cancelamento, 665  
   criando, 665-666  
   especificando, 700, 713  
   examinando, 673  
   para operações de espera, 700  
*ToList*, método para recuperar dados, 585-586, 590-591  
 Toolbox  
   All Controls, seção, 51  
   Common WPF Controls, seção, 51  
   exibindo, 51  
*ToolTip*, propriedade, mensagens de erro como, 550-551, 564  
*ToString*, método, 73, 207, 217  
   de estruturas, 210-211  
   implementação de, 270-271  
*TPL* (Task Parallel Library), 635  
   agendando threads, 635-636  
   classes de coleção e interfaces protegidas quanto a threads (thread-safe), 700  
   estratégia de cancelamento, 664-677  
*Parallel*, classe, 649-656  
 primitivas de sincronização em, 693-699  
*Task*, classe, 635. Consulte também Task, classe  
   técnicas de bloqueio, 693  
 tratamento de exceções, 142  
   para tarefas, 673-676  
*TResult*, parâmetro de tipo, 431  
*try*, blocos, 142  
   escrevendo, 148  
*try*, palavra-chave, 142  
*try/catch*, blocos de instrução, escrevendo, 146-150  
*TSource*, parâmetro de tipo, 431  
*TValue*, 389
- "Type 'typename' already defines a member called X with the same parameter types" (O tipo 'nome\_do\_tipo' já define um membro chamado X com os mesmos tipos de parâmetros), erro, 97
- U**
- unboxing, 198-200  
*unchecked*, palavra-chave, 150-151  
*unsafe*, palavra-chave, 201-202  
*UpdateException*, exceções, 620  
*UpdateException*, rotina de tratamento, 626-627  
*UpdateSource*, método, 561  
   chamando, 564  
*UpdateSourceTrigger*, propriedade, 560  
   deferindo validação com, 564  
*Use dynamic ports*, propriedade, 722  
 user data, validação de, 541-564  
*User ID*, parâmetro, 573  
*using*, diretivas, 318  
*using*, instruções, 47, 48  
   encerramento de conexão de dados, instruções em, 578  
   escrevendo, 321-324  
   para gerenciamento de recursos, 318-320  
   sintaxe de, 318
- V**
- validação  
   com vinculação de dados, 543-564  
   controle por programação de, 564  
   de dados, 541-564  
   de entrada, 541-542  
   explícita, 560-564  
   temporização de, 550, 559-564  
   testando, 558-559, 562-564  
*Validação.HasError*, propriedade  
   detectando mudanças em, 564  
   disparador para, 550  
*ValidaçãoRules*, elementos, 548  
*ValidateNames*, propriedade, 528  
 valores  
   boxing, 197-198  
   comparando, 121-122  
   retornando de tarefas, 657-660  
   unboxing, 198-200  
 valores de string  
   concatenando, 69, 72  
   convertendo em inteiros, 77-78,  
   convertendo em valores *int*, 133  
 valores nullable, 154  
*Value*, propriedade, 190, 191  
*Value*, tipos, 202-203  
   copiando, 183-188
- destruição de, 311  
 estruturas, 210-222  
 nullable, 189-190  
 numerações, 205-210  
 pilha, criação em, 195  
*value*, variável tipo-valor, 331  
   copiando, 202-203, 221  
*ValueConversion*, atributo, 555  
*Var*, palavra-chave, 76-77  
   para variáveis implicitamente tipadas, 180  
*Variant*, tipo, 76-77  
 variáveis, 61-63  
   atribuindo valores a, 63  
   convenções de nomeação, 62  
   copiando conteúdo em tipos-referência, 185  
   de tipos de classe, 163  
   declarando, 77-78  
   decrementando, 75-78, 124, 140  
   em métodos, 85  
   escopo de, 85  
   estáticas, 176  
   implicitamente tipadas, 46-78, 180  
   incrementando, 75-78, 124, 140  
   inicializando, 85  
   inicializando para o mesmo valor, 77-78  
   não atribuídas, 64, 105  
   nomeando, 61, 62  
   qualificando como parâmetros, 171-172  
*ScreenTips* em, 63  
 tipos de, inferindo, 76-77  
 valor de, alterando, 77-78  
 valores atribuídos a, 76-77  
*value* tipos, 202-203, 221, 331  
 verificando valores no depurador, 94-95
- variáveis Booleanas, declarando, 121-122
- variáveis de array  
   convenções de nome, 224  
   declarando, 223-224, 250  
   inicializando, 225-226
- variáveis de estrutura  
   copiando, 219  
   declarando, 214, 222  
   inicializando, 222  
   versões nullable de, 214
- variáveis de referência, 312  
   copiando, 202-203, 221  
   inicializando, 188-189  
   valores null, 189
- variáveis de string, armazenando dados em, 133
- variáveis do tipo enumerado, 206  
   atribuindo a valores, 222

- convertendo em strings, 206  
declarando, 222  
operações matemáticas em, 209-210  
variáveis locais, 86  
  exibindo informações sobre, 136  
variáveis nullable  
  atribuindo expressões a, 190  
  atualizando, 191  
  testando, 189  
verificação cruzada de dados, 541-542  
verificação de estouro, 150, 151  
verificação de tipos, herança e, 267  
versões de uma classe genérica, específica de tipos, 389  
*VerticalAlignment*, propriedade, 479-480  
*View Code*, comando, 65, 508  
vinculação de dados  
  buscando e exibindo dados com, 611-615  
  classes de conversão, 554-557  
  dados existentes, atualizando com, 615-616  
  Entity Framework, utilizando com, 611-614  
  modificando data com, 615-628  
  para validação, 543-559  
  vinculando caminhos, 551  
  vinculando controles a propriedades de classe, 547  
  vinculando controles WPF a origens de dados, 612  
  vinculando origens, 550, 563-564  
  vinculando propriedades de controle a propriedades de controle, 545, 557-558, 563-564  
  vinculando propriedades de controle a propriedades de objeto, 563-564  
vinculando origens, 550  
  especificando, 563-564, 609  
*virtual*, palavra-chave, 271, 272, 283, 304, 308  
Visual C# 2010 Express, 36  
  aplicativos de console, criando, 38-40  
  aplicativos gráficos, criando, 50  
  configurações padrão do ambiente de desenvolvimento, 37  
  inicializando, 36  
  local de salvamento, especificando, 571  
Visual Studio 2010  
  ambiente de programação, 35-40  
  arquivos criados por, 40-41  
  barra de ferramentas, 39  
  barra de menu, 39  
  *Code and Text Editor*, painel, 39  
  Codificando exibição de erro, 44  
  código gerado automaticamente, 54-55  
  configurações padrão do ambiente de desenvolvimento, 36  
  Entity Framework, 597. *Consulte também Entity Framework*  
  *Error List*, janela, 44  
  inicializando, 36  
  *Output*, janela, 43  
  *Solution Explorer*, painel, 39  
Visual Studio 2010 Professional, 36.  
  aplicativos de console, criando, 37-38  
  aplicativos gráficos, criando, 49  
Visual Studio 2010 Standard, 36  
  aplicativos de console, criando, 37-38  
  aplicativos gráficos, criando, 49  
*Visual Studio Just-In-Time Debugger*, caixa de diálogo, 147  
*void*, palavra-chave, 80, 81, 83
- W**
- Wait*, método, 640, 678, 693, 713  
*WaitAll*, método, 640, 678  
*WaitAny*, método, 640  
*WaitingToRun*, estado de tarefa, 670  
WCF (Windows Communication Foundation), 716  
Web services, 715-748  
  arquiteturas, 716  
  balanceamento de carga, 719  
  chamando, 743  
  construindo, 720  
  consumindo, 743  
  criando usando REST, 736  
  criando usando SOAP, 721  
  definição, 716  
  endereçamento, 719  
  métodos Web, 717  
  política, 719  
  Representational State Transfer (REST), 716, 719  
  requisitos não funcionais, 718  
  roteando, 719
- segurança, 718  
*Service.svc*, arquivo, 727  
Simple Object Access Protocol (SOAP), 716, 717  
SOAP versus REST, 720  
*Web.config*, arquivo, 727  
Web Services Description Language (WSDL), 718  
Windows Communication Foundation, 716  
WS-Addressing, especificação, 719  
WS-Policy, especificação, 719  
WS-Security, especificação, 718  
*Where*, método, 433  
*While*, instruções, 124-128, 140  
  escrevendo, 125-128  
  sintaxe de, 124-125  
  término de, 125  
*Window.Resources*, elemento, 485-486, 547  
*Window\_Closing*, método, 506-508  
*Window\_Loaded*, método, 611  
Windows, caixa de diálogo comuns, 528-530  
Windows Authentication para acesso a bancos de dados, 572, 573  
Windows Communication Foundation (WCF), 716  
Windows Forms, 49  
Windows Forms Application, template, 49  
*Windows Open*, caixa de diálogo, exibindo, 126  
Windows Presentation Foundation (WPF), 49  
*WithCancellation*, método, 688, 713  
*WithDegreeOfParallelism*, método, 687  
*WithExecutionMode*, método, 687  
WPF, janelas, compilando, 484  
WPF Application, template, 49, 477, 508  
*WrapPanels*, 478  
*WrappedInt*, classe, 188  
*WrappedInt*, objetos, passando como argumentos, 186-188  
*WrappedInt variables*, declarando, 187  
write locks, 693, 697  
*WriteableBitmap*, classe, 643  
*WriteableBitmap*, tipo, 643

*WriteLine*, método, 41, 251

sobrecargas de, 256

sobrecregando, 87-88

*WS-\**, especificações, 719

*WS-Addressing*, especificação, 719

WSDL (Web Services Description

Language), 718. *Consulte também*

SOAP (Simple Object Access Pro-

ocol)

*WS-Policy*, especificação, 719

*WS-Security*, especificação, 718

## X

XAML (Extensible Application Markup  
Language) em formulários WPF,

51-52, 477

XML, 716

XML, declaração de namespace, 547

XML, namespaces, 477

*Xmllns*, atributos, 477

XOR (^), operador, 348

## Y

*Yield*, palavra-chave, 422

## Z

*ZIndex*, propriedade, 483

IMPRESSÃO:



Santa Maria - RS - Fone/Fax: (55) 3220.4500  
[www.pallotti.com.br](http://www.pallotti.com.br)

# Microsoft® Visual C# 2010 Passo a Passo

Microsoft®

Mc  
Graw  
Hill



grupo A

Conhecimento que transforma.

[www.grupoaeitoras.com.br](http://www.grupoaeitoras.com.br)  
0800 703 3444

grupo A

Conhecimento que transforma.

A Artmed cresceu e agora é Grupo A. Uma empresa que engloba várias editoras e diversas plataformas de distribuição de informação técnica, científica e profissional. Uma corporação que disponibiliza o conteúdo que você precisa onde, quando e como for necessário.

Microsoft®

# Visual C#® 2010

## Passo a Passo



Seu guia prático e passo a passo para aprender o Visual C# 2010

Aprenda a construir aplicativos com o Visual C# 2010 e o Microsoft .NET Framework 4.0 – um passo de cada vez. Destinado a desenvolvedores com conhecimento básico de programação, este guia fornece a orientação prática necessária e os exemplos adequados para criar componentes C# e aplicativos baseados em Windows®.

Descubra como:

- Declarar variáveis, escrever instruções, criar operadores e chamar métodos
- Criar seus primeiros aplicativos usando Windows Presentation Foundation
- Construir a IU e validar a entrada de dados
- Gerenciar erros e tratar exceções
- Usar a coleta de lixo para gerenciar recursos
- Usar genéricos, construir novos tipos e criar componentes reutilizáveis
- Consultar e manipular dados com LINQ e ADO.NET
- Examinar o suporte à multitarefa via Task Parallel Library
- Criar Web services com o Windows Communication Foundation

O autor

John Sharp é chefe de tecnologia na Content Master, parte do CM Group Ltd, empresa de desenvolvimento e consultoria técnica. Especialista em desenvolvimento de aplicativos com Microsoft .NET Framework e em questões de interoperabilidade, John já produziu diversos tutoriais, artigos e apresentações sobre sistemas distribuídos, Web services e linguagem C#.

Recursos do CD (em inglês):

- Exercícios e exemplos de código
- eBook totalmente pesquisável

Para informações sobre os requisitos do sistema, veja a Introdução.



TI/PROGRAMAÇÃO

ISBN 978-85-7780-849-6



[www.grupoaeitoras.com.br](http://www.grupoaeitoras.com.br)



Microsoft®  
Visual Studio®

**Microsoft**

**grupo**   
Conhecimento que transforma.