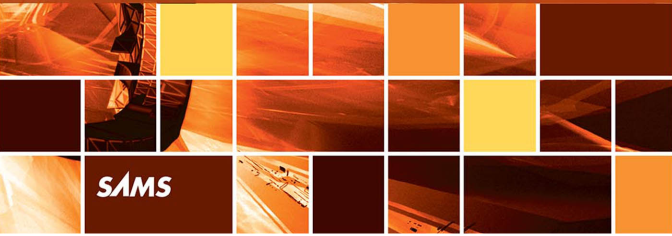
Alessandro Del Sole

Foreword by Anthony D. Green,
Program Manager, Visual Basic, Microsoft

Visual Basic® 2015

UNLEASHED



SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Alessandro Del Sole

Visual Basic® 2015

UNLEASHED

SAMS | 800 East 96th Street, Indianapolis, Indiana 46240 USA

Visual Basic® 2015 Unleashed

Copyright © 2016 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33450-4

ISBN-10: 0-672-33450-X

Library of Congress Control Number: 2015906633

Printed in the United States of America

First Printing July 2015

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Editor-in-Chief
Greg Wiegand

Acquisitions Editor
Joan Murray

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Senior Project Editor
Betsy Gratner

Indexer
Tim Wright

Proofreader
Katie Matejka

Technical Editors
Anthony D. Green
Lucian Wischik

Publishing Coordinator
Cindy Teeters

Cover Designer
Mark Shirar

Senior Compositor
Gloria Schurick

Contents at a Glance

Introduction 1

Part I Learning the Basics of VB

1 Introducing .NET 2015 3
2 The Visual Studio 2015 IDE for Visual Basic 17
3 The Anatomy of a Visual Basic Project 71
4 Data Types and Expressions 99
5 Debugging Visual Basic 2015 Applications 199
6 Errors, Exceptions, and Code Refactoring 231

Part II Object-Oriented Programming with Visual Basic 2015

7 Class Fundamentals 267
8 Managing an Object's Lifetime 313
9 Organizing Types Within Namespaces 327
10 Modules 347
11 Structures and Enumerations 351
12 Inheritance 369
13 Interfaces 393
14 Generics and Nullable Types 413
15 Delegates and Events 427
16 Working with Collections and Iterations 441
17 Working with Objects: Visual Tools and Code Sharing 473

Part III Advanced Language Features

18 Manipulating Files and Streams 513
19 The `My` Namespace 537
20 Advanced Language Features 571

Part IV Data Access with ADO.NET and LINQ

21 Introducing ADO.NET and DataSets 599
22 Introducing LINQ 609
23 LINQ to Objects 615
24 LINQ to SQL 645
25 LINQ to DataSets 679

26 Introducing ADO.NET Entity Framework 687
27 Manipulating XML Documents with LINQ and XML Literals 729

Part V Building Windows Desktop Applications
28 Creating WPF Applications 745
29 WPF Common Controls 779
30 Brushes, Styles, Templates, and Animations in WPF 811
31 Manipulating Media and Documents 851
32 Introducing Data-Binding 869
33 Localizing Applications 901

Part VI Building Web and Mobile Applications
34 Building and Publishing ASP.NET Web Applications 909
35 Building and Deploying Applications for Microsoft Azure 951
36 Building Universal Apps for Windows 10 973

Part VII Networking and Exposing Data Through Networks
37 Creating and Consuming WCF Services 1013
38 Implementing and Consuming OData Services 1035

Part VIII Advanced .NET Framework with VB 2015
39 Serialization 1073
40 Processes and Multithreading 1095
41 Parallel Programming and Parallel LINQ 1105
42 Asynchronous Programming 1139
43 Working with Assemblies 1177
44 Reflection 1191
45 Coding Attributes 1217
46 Platform Invokes and Interoperability with the COM Architecture 1229
47 Documenting Source Code with XML Comments 1245

Part IX Applications Deployment
48 Understanding the Global Assembly Cache 1261
49 Setup and Deployment Projects with InstallShield for Visual Studio 1269
50 Deploying Applications with ClickOnce 1287

Part X Code Analysis with VB 2015

51 Code Analysis: The .NET Compiler Platform and Tools 1305

A Useful Resources and Tools for Visual Basic 2015 1363

Index 1367

Online-Only Chapters

52 Advanced IDE Features

53 Testing Code with Unit Tests and Test-Driven Development

NOTE

In order to accommodate maximum page count for a print book and still be the exhaustive reference on Visual Basic, Chapters 52 and 53 are only available online. To access them, register your book at www.informit.com/title/9780672334504. Click the *Register Your Product* link. You will be prompted to sign in or create an account. When asked for the ISBN, enter 9780672334504. This *print book* ISBN must be entered even if you have a digital copy of the book. From there, click *Access Bonus Content* in the “Registered Products” section of your account page.

Table of Contents

Introduction 1

Code Samples and Software Requirements ... 2

Part I Learning the Basics of VB

1 Introducing .NET 2015 3

.NET 2015: A New Vision for Development ... 3
The .NET Framework 4.6 for Desktop ... 5
Introducing .NET Core 5 ... 10
Programming Languages in Visual Studio 2015 ... 12
What's New with Compilers ... 13
The Windows Software Development Kit ... 15
Summary ... 15

2 The Visual Studio 2015 IDE for Visual Basic 17

What's New in Visual Studio 2015 ... 17
Status Bar and Start Page ... 18
Working with Projects and Solutions ... 20
Working with Tool Windows ... 35
The My Project Window ... 42
Compiling Projects ... 48
Debugging Overview ... 58
Browsing the Visual Basic and .NET Documentation ... 65
Quick Launch Tool ... 67
Showing the Hierarchy of Method Calls ... 69
Summary ... 70

3 The Anatomy of a Visual Basic Project 71

Brief Overview of Types and Members ... 71
Visual Basic 2015 Reserved Keywords ... 80
Understanding Project Files ... 82
Understanding References ... 92
Summary ... 97

4 Data Types and Expressions 99

Introducing the Common Type System ... 99
Understanding Value Types ... 103
Understanding Reference Types ... 113

Differences Between Value Types and Reference Types 116
Converting Between Value Types and Reference Types 121
Understanding Conversion Operators 129
Working with .NET Fundamental Types 135
Common Operators 172
Iterations, Loops, and Conditional Code Blocks 188
Summary 197

5 Debugging Visual Basic 2015 Applications 199

Preparing an Example 199
Debugging Instrumentation 200
Inspecting Object Details with Debugger Visualizers 215
Debugging in Code 216
Summary 230

6 Errors, Exceptions, and Code Refactoring 231

Introducing Exceptions 231
Handling Exceptions 233
Refactoring Your Code: Light Bulbs and Quick Actions 249
Summary 266

Part II Object-Oriented Programming with Visual Basic 2015

7 Class Fundamentals 267

Declaring Classes 267
Storing Information with Properties 271
Types and Members Visibility: Scope 276
Executing Actions with Methods 278
Organizing Code with Partial Classes 290
Splitting Method Definitions with Partial Methods 293
Instantiating Objects with Constructors 295
Shared Members 301
Common Language Specification 306
Summary 311

8 Managing an Object's Lifetime 313

Understanding Memory Allocation 313
Understanding Garbage Collection 314
Understanding the `Finalize` Method 315
Understanding `Dispose` and the `IDisposable` Interface 317
Advanced Garbage Collection 323
Summary 326

9 Organizing Types Within Namespaces 327

Understanding Namespaces 327
Organizing Types Within Namespaces 328
Global Namespaces and the `Global` Keyword 342
Summary 345

10 Modules 347

Modules Overview 347
Differences Between Modules and Classes 349
Summary 350

11 Structures and Enumerations 351

Understanding Structures 351
Overloading Operators 356
Structures and Common Language Specification 360
Grouping Constants with Enumerations 361
Summary 367

12 Inheritance 369

Applying Inheritance 370
Introducing Polymorphism 375
Overriding Members 377
Conditioning Inheritance 380
Accessing Base Classes Members 383
Constructors' Inheritance 387
Shadowing 388
Overriding Shared Members 389
Practical Inheritance: Building Custom Exceptions 390
Summary 392

13 Interfaces 393

Defining Interfaces 393
Implementing and Accessing Interfaces 394
Partial Interfaces 399
Interfaces and Polymorphism 399
Interfaces Inheritance 400
Defining CLS-Compliant Interfaces 402
Most Common .NET Interfaces 402
Summary 412

14 Generics and Nullable Types 413

Introducing Generics 413
Creating and Consuming Generics 414
Introducing Nullable Types 422
Summary 425

15 Delegates and Events 427

Understanding Delegates 427
Handling Events 431
Offering Events to the External World 433
Summary 439

16 Working with Collections and Iterators 441

Understanding Collections Architecture 442
Working with Nongeneric Collections 442
Working with Generic Collections 451
The Null-Conditional Operator and Collections 461
Building Custom Collections 462
Concurrent Collections 463
Immutable Collections 463
Iterators 464
Summary 472

17 Working with Objects: Visual Tools and Code Sharing 473

Visual Studio Class Designer 473
Class View Window 484
Creating Types with Generate from Usage 485
Creating Portable Classes 491
Shared Projects 503
Summary 512

Part III Advanced Language Features

18 Manipulating Files and Streams 513

Manipulating Directories and Pathnames 513
Handling Exceptions for Directories and Pathnames 519
Manipulating Files 519
Introducing Streams 524
Summary 535

19 The `My` Namespace 537

Introducing the `My` Namespace 537
`My.Application` 538
`My.Computer` 542
`My.Settings` 550
`My.Resources` 556
`My.User` 560
`My.WebServices` 562
Extending `My` 562
`My` in Different Applications 566
Summary 570

20 Advanced Language Features 571

Local Type Inference 571
Array Literals 575
Extension Methods 577
Anonymous Types 585
Relaxed Delegates 586
Lambda Expressions 587
Generic Variance 596
Summary 598

Part IV Data Access with ADO.NET and LINQ

21 Introducing ADO.NET and DataSets 599

System Requirements 599
Introducing ADO.NET 600
Introducing DataSets 603
Summary 608

22 Introducing LINQ 609

What Is LINQ? 609
LINQ Examples 611
Language Support 612
Understanding Providers 613
Overview of LINQ Architecture 613
Summary 614

23 LINQ to Objects 615

Introducing LINQ to Objects 615
Querying in Memory Objects 616

Introducing Standard Query Operators ... 625
Summary ... 644

24 LINQ to SQL 645

Introducing LINQ to SQL ... 646
Querying Data with LINQ to SQL ... 658
Insert/Update/Delete Operations with LINQ ... 663
Advanced LINQ to SQL ... 672
Summary ... 677

25 LINQ to DataSets 679

Querying DataSets with LINQ ... 679
LINQ to DataSets' Extension Methods ... 682
Summary ... 685

26 Introducing ADO.NET Entity Framework 687

Introducing Entity Framework ... 687
Understanding Entity Data Models ... 688
Insert/Update/Delete Operations for Entities ... 703
Querying EDMs with LINQ to Entities ... 710
Querying EDMs with Entity SQL ... 712
Mapping Stored Procedures ... 712
Introducing the Code First Approach ... 716
Summary ... 727

27 Manipulating XML Documents with LINQ and XML Literals 729

Introducing LINQ to XML ... 730
Writing XML Markup in VB with XML Literals ... 735
Summary ... 743

Part V Building Windows Desktop Applications

28 Creating WPF Applications 745

What Is WPF? ... 746
Introducing the WPF Architecture ... 747
Building WPF Applications with Visual Studio 2015 ... 749
Understanding the eXtensible Application Markup Language ... 751
Understanding Visual Tree and Logical Tree ... 757
Handling Events in WPF ... 759
Arranging Controls with Panels ... 762
Managing Windows ... 770

Introducing the Application Object 772
Brief Overview of WPF Browser Applications 774
Live Visual Tree 777
Summary 778

29 WPF Common Controls 779

Introducing WPF Controls Features 779
Understanding the `ContentControl` 780
Understanding Common Controls 781
Using Common Dialogs 809
Summary 810

30 Brushes, Styles, Templates, and Animations in WPF 811

Introducing Brushes 812
Introducing Styles 828
Introducing Control Templates 833
Introducing Transformations 836
Introducing Animations 840
Summary 849

31 Manipulating Media and Documents 851

Viewing Images 851
Playing Media 853
Manipulating Documents 857
Viewing XPS Documents 866
Summary 867

32 Introducing Data-Binding 869

Introducing the Data-Binding in WPF 869
Discussing the Drag and Drop Data-Binding 876
Summary 900

33 Localizing Applications 901

Introducing .NET Localization 902
Introducing the Multilingual App Toolkit 902
Windows Forms Localization 903
WPF Localization 906
Summary 908

Part VI Building Web and Mobile Applications

34 Building and Publishing ASP.NET Web Applications 909

Introducing the ASP.NET Model 910
Web Forms and Master Pages 914
ASP.NET Controls 917
Handling Events 919
Understanding State Management 920
Creating a Web Application with VB 2015 with Data Access and Pages 923
Publishing ASP.NET Web Applications 940
Summary 949

35 Building and Deploying Applications for Microsoft Azure 951

Overview of the Microsoft Azure Platform 951
Registering for Microsoft Azure 953
Downloading and Installing Tools for Visual Studio 954
Introducing the Management Portal 954
Creating a SQL Azure Database 955
Creating an ASP.NET Application for the Cloud 961
Deploying Applications to Microsoft Azure 964
A Step Further: Web Roles and Worker Roles 967
Additional Tools 970
Summary 972

36 Building Universal Apps for Windows 10 973

Introducing Universal Windows Apps 974
Introducing the Universal Windows Platform 975
Registering with the Windows Store 976
Installing the Developer Tools 976
Creating Apps with Visual Basic 977
Customizing the Application Manifest 998
Starting and Debugging Universal Windows Apps 1002
Creating and Testing App Packages 1009
Submitting Apps to the Store 1010
Summary 1011

Part VII Networking and Exposing Data Through Networks

37 Creating and Consuming WCF Services 1013

Introducing Windows Communication Foundation 1014
Implementing WCF Services 1015

Consuming WCF Services ... 1024
Handling Exceptions in WCF ... 1030
Hosting WCF Services in Internet Information Services and Microsoft Azure ... 1031
Configuring Services with the Configuration Editor ... 1033
Summary ... 1034

38 Implementing and Consuming OData Services 1035

What Are OData Services? ... 1035
Creating an OData Endpoint ... 1036
Consuming OData Services ... 1060
Implementing and Consuming Functions ... 1068
Summary ... 1071

Part VIII Advanced .NET Framework with VB 2015

39 Serialization 1073

Objects Serialization ... 1074
XML Serialization ... 1081
Custom Serialization ... 1083
Serialization with XAML ... 1086
Serialization in Windows Communication Foundation ... 1088
Serialization in the ADO.NET Entity Framework ... 1091
Summary ... 1092

40 Processes and Multithreading 1095

Managing Processes ... 1096
Introducing Multithreading ... 1098
Understanding the .NET Thread Pool ... 1099
Threads Synchronization ... 1101
Summary ... 1104

41 Parallel Programming and Parallel LINQ 1105

Introducing Parallel Computing ... 1106
Understanding and Using Tasks ... 1107
Parallel Loops ... 1116
Debugging Tools for Parallel Tasks ... 1122
Concurrent Collections ... 1124
Introducing Parallel LINQ ... 1128
Summary ... 1137

42 Asynchronous Programming 1139

Overview of Asynchrony....1140
The Old-Fashioned Way: Event-Based Asynchrony....1140
The Old-Fashioned Way: The Asynchronous Programming Model....1142
The Modern Way: The `Async` Pattern....1143
Getting Started with `Async/Await`....1147
Exception Handling in `Async`....1161
Implementing Task-Based Asynchrony....1161
Cancellation and Progress....1165
Asynchronous Lambda Expressions....1170
Asynchronous I/O File Operations in .NET 4.6....1171
Debugging Tasks....1175
Summary....1176

43 Working with Assemblies 1177

Assembly Overview....1177
Understanding Application Domains....1179
Security Model in .NET 4.6....1182
Summary....1189

44 Reflection 1191

Introducing Reflection....1191
Understanding Assemblies' Metadata....1192
Getting Assembly Information....1194
Reflecting Types....1196
Invoking Code Dynamically....1204
Generating Code at Runtime with `Reflection.Emit`....1206
Caller Information....1212
Summary....1215

45 Coding Attributes 1217

Applying Attributes....1217
Coding Custom Attributes....1220
Reflecting Attributes....1225
Summary....1226

46 Platform Invokes and Interoperability with the COM Architecture 1229

Importing and Using COM Objects....1230
Catching Exceptions....1233
Releasing COM Objects....1233
Calling COM Objects from WPF....1234

P/Invokes and Unmanaged Code 1237
References to the Win32 API Calls 1244
Summary 1244

47 Documenting Source Code with XML Comments 1245

Understanding XML Comments 1246
Implementing XML Comments 1248
Generating Compiled Help Files 1258
Summary 1259

Part IX Applications Deployment

48 Understanding the Global Assembly Cache 1261

The Dll Hell Problem 1261
The Global Assembly Cache 1263
Summary 1267

49 Setup and Deployment Projects with InstallShield for Visual Studio 1269

Windows Installer Overview 1270
Introducing InstallShield 1271
Creating a Setup Project 1272
Configuring the Setup Project 1283
Building and Deploying the Windows Installer Package 1285
Summary 1285

50 Deploying Applications with ClickOnce 1287

Introducing ClickOnce 1287
Deploying Applications with ClickOnce 1289
Configuring ClickOnce 1294
Security Considerations 1299
Programmatically Accessing ClickOnce 1301
Registration-Free COM 1302
Summary 1304

Part X Code Analysis with VB 2015

51 Code Analysis: The .NET Compiler Platform and Tools 1305

Live Code Analysis with the .NET Compiler Platform 1306
Calculating Code Metrics 1349
Diagnostic Tools and IntelliTrace 1351
Code Clone Detection 1359
Summary 1361

A Useful Resources and Tools for Visual Basic 2015 1363

Visual Basic Resources in MSDN 1363
Useful Developer Tools for Visual Basic 1364

Index 1367

Online-Only Chapters

52 Advanced IDE Features

53 Testing Code with Unit Tests and Test-Driven Development

NOTE

In order to accommodate maximum page count for a print book and still be the exhaustive reference on Visual Basic, Chapters 52 and 53 are only available online. To access them, register your book at www.informit.com/title/9780672334504. Click the *Register Your Product* link. You will be prompted to sign in or create an account. When asked for the ISBN, enter 9780672334504. This *print book* ISBN must be entered even if you have a digital copy of the book. From there, click *Access Bonus Content* in the “Registered Products” section of your account page.

Foreword

Back in 2013 Alessandro Del Sole reached out to the Visual Basic team to let us know that the VB Tips & Tricks user group in Italy had reached a 15-year milestone and that what would make it even more special would be to have a team member come out and celebrate with them. Being asked by the leader of one of our longest-running user groups boasting a membership of 40,000+ strong, it was a no-brainer. I knew I had to go. So I hopped on a 12-hour flight to Milan to give a 1-hour talk, a 10-minute speech, turned around, and flew back home (another 12 hours)—and it was totally worth it!

I joined the Visual Basic team in 2010 and in the entire time that I've known him since then, Alessandro has been an invaluable member of the VB community. He has consistently exemplified the qualities of an MVP, demonstrating technical leadership in the community, subject matter expertise, and receiving ongoing nominations and recognition by his peers as MVP of the Year.

Being familiar with Alessandro's VB books, I've always admired the comprehensiveness of his writing style. So many books approach development from just the language, or just a few libraries, but Alessandro covers the end-to-end—from language to library to IDE, in keeping with the Visual Basic spirit. And that style continues in this new edition. Here in Redmond, we're all very proud of the tremendous value we've added for VB developers in Visual Studio 2015, including a new ecosystem of Roslyn-powered diagnostic analyzers, refactoring (for the first time), great productivity language features, and a brand new experience for developing universal Windows 10 apps that run on PCs, Windows Phone, Xbox One, Microsoft Band, and HoloLens! True to form, Alessandro has taken the time to revisit everything new; each of his chapters in this edition highlights those enhancements, leaving nothing out. And personally, as the PM for the Roslyn APIs and a VB language designer for the last five years, I was especially thrilled to see him take up the topic of authoring code analysis tools with Roslyn (with his usual technical fervor) in the "Code Analysis" chapter.

So if you're looking for one-stop shopping to get the big picture (and get developing) in .NET and Visual Studio for VB developers in 2015, then this is the book for you. You'll be glued to it for a week integrating all the little enhancements into your day-to-day. And then after you've caught your breath, you'll keep coming back to it again and again as you explore whole new technologies over time. Much like those 24 hours of flying back in 2013, this book is totally worth it!

Anthony D. Green

Program Manager, Visual Basic, Microsoft

About the Author

Alessandro Del Sole, a Microsoft Most Valuable Professional (MVP) for .NET and Visual Basic since 2008, is well known throughout the global VB community. He is a community leader on the Italian Visual Basic Tips and Tricks website (http://www.visual-basic.it), which serves more than 46,000 VB developers, as well as a frequent contributor to the MSDN Visual Studio Developer Center. He has been awarded MVP of the Year five times (2009, 2010, 2011, 2012, 2014) and enjoys writing articles on .NET development both in English and Italian. He also writes blog posts and produces instructional videos as well as Windows Store apps. You can find him online in forums and you can follow him on Twitter at @progalex.

Dedication

To my mom: Life without you is not the same. You still live inside me through all of your lessons in life. I miss you.

To my dad, a brave great man. I am still learning from you what being a man means and I hope to be like you in my life.

To my girlfriend, Angelica, a wonderful ray of light in my life. Thanks for being there every day.

Acknowledgments

First, I would like to thank Joan Murray, Betsy Gratner, Kitty Wilson, Mark Renfrow, and everyone else at Sams Publishing for trusting me enough to write the third edition of this book about Visual Basic. Writing books like this is hard work not only for the author but also for all the people involved in the reviews and in the production process. Working with these guys made the process much more pleasant. Thank you!

Very special thanks to Lucian Wischik and Anthony D. Green from the Managed Languages Team at Microsoft, who have been the technical editors for this book. They did an incredible job walking through every single sentence and every single line of code. As the people who best know the Visual Basic language in the world, their suggestions and corrections were invaluable to me and definitely contributed to creating excellent and precise content. Thank you so much.

Great thanks also to the guys from the Italian subsidiary of Microsoft, Roberto Andreoli, Lorenzo Barbieri, Erica Barone, and Matteo Pagani, for their continuous support and encouragement for my activities. My deep thanks and appreciation for their passionate work with communities and with MVPs to Alessandro Teglia, Cristina Gonzalez Herrero (my MVP lead), and Marjorie di Clemente.

I would like to thank my everyday friends who are always ready to encourage me even if they are not developers and will never read my books. Most importantly, these people always support me when I need their help. So my deep thanks to Nadia Paloschi, Roberto Bianchi, Alessandro Ardovini, Michela Santini, Leonardo Amici, and Karin Meier. You are in my heart.

As a community leader in the Italian Visual Basic Tips and Tricks community (www.visual-basic.it), I would like to thank all those guys who are the stimulus for making things better every day; all those people who visited my blog at least once or who read even one article of mine; and all those people who visit our website and follow us on forums, videos, articles, and blogs. Special thanks to my MVP colleagues Diego Cattaruzza, Antonio Catucci, Renato Marzaro, and Raffaele Rialdi for their great support and valuable suggestions. Thanks to Marco Notari for his continuous support and encouragement.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email:	consumer@samspublishing.com
Mail:	Sams Publishing ATTN: Reader Feedback 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Introduction

A new era is coming for Microsoft. From Windows 10, to embracing open source, to opening to third-party platforms and operating systems, to the cloud first/mobile first vision, to the release of Visual Studio 2015, it is really an exciting time to be a software developer working with Microsoft products and technologies. From a developer perspective, Visual Studio 2015 marks a very important milestone because it is the state of the art in representing Redmond's new vision and because it is the most productive version ever. Both the Visual Basic and C# compilers have been open sourced, together with a tiny, modular subset of the .NET Framework called .NET Core, which is intended for cross-platform development. There are many new features, many new tools, and many new development opportunities with Visual Studio 2015 and the Visual Basic language that certainly open up amazing new scenarios but that also require some changes in how to approach building applications, especially with regard to mobile devices. As an experienced developer working with Visual Basic for many years and as a community person who daily connects with developers worldwide, I know very well what programmers using Visual Basic need to face in the real world, what they expect from developer tools and learning resources, and what they need to get the most out of their code and skills in order to build high-quality, rich applications and to get appropriate information to look at the future.

This book has two main goals: the first goal is to walk through the Visual Basic programming language deeply, explaining all the available language features, object-oriented programming, common patterns, and everything you need to know to really master the language. The second goal is to show what you can do with Visual Basic in practice; for instance, you can build Windows applications for the desktop, and you can also build apps for Windows 10 (which is a brand-new topic), as well as applications for the web, the cloud, and other platforms. Describing the VB language and what you can do with it is a tradition retaken from previous editions; but technology evolves, and so does the *Visual Basic Unleashed* book. With updated content and chapters that describe in details all the new language features, this new edition also focuses on the latest tools and platforms. As a couple of significant examples, *Visual Basic 2015 Unleashed* explains how to build universal Windows apps for Windows 10 and how to leverage the new compiler APIs from the .NET Compiler Platform to write custom domain-specific live code analysis rules. In addition, the book explains how to get the maximum out of the Visual Studio's development environment so that you can be more productive than ever. But there is a lot more; this book embraces all the possible development areas available to Visual Basic today. A new era is coming for Microsoft, and it's coming for you, too.

Code Samples and Software Requirements

Good explanations often require effective code examples. The companion source code for this book can be downloaded from www.informit.com/title/9780672334504. Code samples are organized by chapter so that it is easy to find the code you need. In order to load, compile, and test the source code, you need Microsoft Visual Studio 2015. If you are not an MSDN subscriber or you did not purchase one of the paid editions, you can download Visual Studio 2015 Community, which is available for free and is enough to run the sample code. You can download Visual Studio 2015 Community (as well as a trial of the Enterprise edition) from https://www.visualstudio.com/downloads/visual-studio-2015-downloads-vs. You are also encouraged to download and install the Visual Studio 2015 Software Development Kit (SDK), which is a requirement in some chapters and is available from the same location as the free download.

Code-Continuation Arrows

When a line of code is too long to fit on the page of the printed book, a code-continuation arrow (➥) appears to mark the continuation. Here is an example:

```
            somePeople.Add(New Person With {.FirstName = "First Name: " &
➥i.ToString,
```

This page intentionally left blank

CHAPTER 42

Asynchronous Programming

IN THIS CHAPTER

- ▶ Overview of Asynchrony
- ▶ The Old-Fashioned Way: Event-Based Asynchrony
- ▶ The Old-Fashioned Way: The Asynchronous Programming Model
- ▶ The Modern Way: The `Async` Pattern
- ▶ Getting Started with `Async/Await`
- ▶ Exception Handling in `Async`
- ▶ Implementing Task-Based Asynchrony
- ▶ Cancellation and Progress
- ▶ Asynchronous Lambda Expressions
- ▶ Asynchronous I/O File Operations in .NET 4.6
- ▶ Debugging Tasks

When you go to the restaurant, there are waiters and waitresses ready to serve your table. A waiter takes your order, brings the order to the kitchen, goes to serve another table, and then comes back to your table to bring you your meals. While waiting for you to finish, the waiter does similar operations for other patrons. So, the waiter does not stand at your table from when you arrive until you finish your meal before going to serve another table; if he did, the restaurant would need to hire one waiter per table to avoid the block of their activity, which is not practical. If you compare this real-world description with computer programming, the waiter is some code in your application. If this code must perform a long-running operation and you write such a code in the user interface (UI) thread or, more generally, in one thread, your code will act like a waiter that waits from the start to the end of the meal on a table and cannot do anything else in the meantime, thereby blocking the application activity. This is what happens when you write code in a synchronous approach; synchronous code performs one task at a time and the next task starts only when the previous one completes. To avoid blocking the application, you can use multithreading and instances of the `Thread` class, described in Chapter 40, "Processes and Multithreading." With multithreading, you can write code that performs a long-running operation on a separate thread and keep the application responsive. Threading is a way to write asynchronous code that developers have been using for a long time, along with two patterns: the Event-based Asynchronous Pattern and the Asynchronous Programming Model. But actually, asynchrony does not necessarily mean running on a background thread. Instead, it means that a task is executed in different moments. Threading is one way to achieve asynchrony, but it is quite complex and not always the best choice. For this reason, back in .NET 4.5 Microsoft introduced new libraries and new keywords to the Visual Basic and Visual C# languages to make asynchronous calls easy. In this chapter, you get an overview of both the Event-based Asynchronous Pattern and the Asynchronous Programming Model; then you learn about the Asynchronous Pattern, which is without a doubt one of the most important features in Visual Basic language. You will see how easily you can now write modern and responsive applications via asynchronous code.

Overview of Asynchrony

Modern applications often need to perform complex computations or access resources through a network. Complex computations can become very long, a network resource might not be available, or the application might not scale well on the server. If the code that performs this kind of operation is running in the same thread as the caller, the thread gets blocked until all operations complete. If such a thread is the UI thread, the user interface becomes unresponsive and can no longer accept the user input until all operations have been completed. This type of approach is called *synchronous* because only one operation at a time is executed until all the processes are completed.

Having an unresponsive user interface is not acceptable in modern applications, so this is the place where asynchrony comes in. Asynchrony enables you to execute some pieces of code in a different thread or context, so that the caller thread never blocks. If this is the UI thread, the user interface remains responsive even if other operations are running. The other thread (or context) then tells the caller thread that an operation completed, regardless of the successful or unsuccessful result. The .NET Framework has been offering, for a long time, two thread-based approaches to asynchrony called *Event-based Asynchrony* and *Asynchronous Programming Model* in which you launch operations in a different thread and get notification of their completion via delegates. As you saw in Chapter 41, "Parallel Programming and Parallel LINQ," the .NET Framework 4.0 introduced the Task Parallel Library and the concept of parallelism. TPL makes it easier to create applications capable of scaling long-running operations across all the available processors. TPL also makes applications faster and more responsive while executing complex tasks concurrently, but this all about concurrency, which is not the same as asynchrony. In this chapter, you first learn about the Event-based Asynchrony and the Asynchronous Programming Model to get started with the important concepts; then you start putting your hands on the possibilities offered by the .NET Framework 4.6. By doing so, it will be easier for you to compare the old way to the new way and understand why you should migrate your exiting code to use the new patterns.

The Old-Fashioned Way: Event-Based Asynchrony

More often than not, applications need to perform multiple tasks at one time, while still remaining responsive to user interaction. One of the possibilities offered by the .NET Framework since the early days is the *Event-based Asynchronous Pattern* (*EAP*). A class that adheres to this pattern implements a number of methods whose names terminate with the

Async suffix and that execute some work on a different thread. Such methods mirror their synchronous counterparts, which instead block the caller thread. Also, for each of these asynchronous methods, there is an event whose name terminates with the Completed suffix and that is raised when the asynchronous operation completes. This way, the caller gets notification of the completion. Because the user might want to cancel an asynchronous operation at a certain point, classes adhering to the EAP must also implement methods whose names terminate with CancelAsync, each related to one of the asynchronous methods that actually performs the requested work. When such work is completed, a delegate will handle the operation result before control is sent back to the caller; this delegate is also known as *callback*. This pattern also requires classes to support cancellation and progress reporting. To understand how EAP works, let's consider a simple example based on the System.Net.WebClient class, which enables you to access networks from client applications. Consider the following code:

```
Sub Main()
    Dim client As New System.Net.WebClient
    AddHandler client.DownloadStringCompleted,
               AddressOf client_DownloadStringCompleted

    client.DownloadStringAsync(New Uri("http://msdn.microsoft.com"))
End Sub
```

A new instance of the WebClient class is created. To receive notification of completion, you must subscribe the DownloadStringCompleted event (assuming you will download a string, but other methods and related events are available) and supply a delegate that will be invoked when the event is raised. After you have subscribed the event, you can then invoke the desired method; in the current example, it's the WebClient.DownloadStringAsync method that downloads contents from the specified URL as a string. If you write other lines of code after the invocation of DownloadStringAsync, these are not necessarily executed after the download operation has completed as it would instead happen in synchronous code. So, if you need to manipulate the result of an asynchronous operation, you must do it inside the callback, which is the delegate invoked after the completion event is raised. The following code provides an example:

```
Private Sub client_DownloadStringCompleted(sender As Object,
                                           e As DownloadStringCompletedEventArgs)
    If e.Error Is Nothing Then
        Console.WriteLine(XDocument.Parse(e.Result).ToString)
        Console.WriteLine("Done")
    End If
End Sub
```

As you can see, the DownloadStringCompletedEventArgs class contains information about the result of the asynchronous operation. Usually, a specific class inherits from System.EventArgs and stores the result of an asynchronous operation, one per asynchronous method. You can check for errors, and if everything is successful, you can then work with the e.Result property that contains the actual result of the task. Classes that adhere to the EAP also enable you to report the progress of an asynchronous operation by exposing a `ProgressChanged` event. Continuing the previous example, the `WebClient` class exposes an event called `ProgressChanged` and a class called `DownloadProgressChangedEventArgs` that stores information about the operation progress. To handle such an event, you must first subscribe it like this:

```
AddHandler client.DownloadProgressChanged,
           AddressOf client_DownloadProgressChanged
```

You then handle the `ProgressChanged` event to report progress:

```
Private Sub client_DownloadProgressChanged(sender As Object,
                                           e As DownloadProgressChangedEventArgs)
    Console.WriteLine(e.ProgressPercentage)
    'Use e.BytesReceived for the number of bytes received in progress
    'Use e.TotalBytesToReceive to get the total bytes to be downloaded
End Sub
```

You can eventually use lambda expressions and statement lambdas as anonymous delegates, as demonstrated in the following code:

```
Private Sub Download()
    Dim client As New WebClient
    AddHandler client.DownloadStringCompleted,
               Sub(sender, e)
                   If e.Error Is Nothing Then
                       Console.WriteLine(XDocument.
                                         Parse(e.Result).
                                         ToString)
                   End If
               End Sub

    client.DownloadStringAsync(New Uri("http://msdn.microsoft.com"))
End Sub
```

The EAP has been very popular among developers for years because the way you write code is similar to how you handle events of the user interface. This certainly makes the asynchronous approach simpler. Later in this chapter, when comparing EAP to the new `Async` pattern, you will better understand why the old way can lead to confusion and become very complex to handle.

The Old-Fashioned Way: The Asynchronous Programming Model

The Asynchronous Programming Model (APM) is still based on threading. In this model, an operation is launched on a separated thread via a method whose name starts with `Begin` (e.g., `BeginWrite`). A method like this must accept, among its parameters, an argument of type IAsyncResult. This is a special type used to store the result and the state of an asynchronous operation. The most important members of this interface are two properties: AsyncState (of type Object), which represents the result of the operation under the form of either a primitive or a composite type, and IsCompleted (of type Boolean), which returns if the operation actually completed. As another parameter, these methods must receive a delegate that will be executed when the asynchronous operation is completed. Within this delegate, you will be able to analyze the result of the asynchronous operation, but you will also need to explicitly end the asynchronous operation by invoking a method whose name starts with End (e.g., EndWrite). Some classes in the .NET Framework are built to be APM-ready, such as Stream and its derived classes. So, to demonstrate how APM works, a good example can be based on the FileStream class. The following code demonstrates how to write some bytes to a stream asynchronously and how the callback receives information from the caller with IAsyncResult.

```
Private Sub OpenStreamAsync()
    Dim someBytes(1000) As Byte
    Dim randomGenerator As New Random()
   'Generate a random sequence of bytes
    randomGenerator.NextBytes(someBytes)

    Using fs As New FileStream("Somedata.dat", FileMode.Create, FileAccess.Write)
        Dim result As IAsyncResult =
            fs.BeginWrite(someBytes, 0, someBytes.Length,
                          AddressOf fs_EndWrite, fs)
    End Using

End Sub

Private Sub fs_EndWrite(result As IAsyncResult)
    Dim stream As FileStream = CType(result.AsyncState, FileStream)
    stream.EndWrite(result)
    'Additional work goes here...
End Sub
```

The IAsyncResult.AsyncState property contains the actual data sent from the caller and must be explicitly converted into the type that you need to work with; in this case, the stream. The reason is that you also must explicitly invoke the EndWrite method that finalizes the asynchronous operation. You can also pass custom objects as the IAsyncResult argument for the callback, to pass more complex and detailed information that you might need to elaborate when the task completes.

The Modern Way: The Async Pattern

Visual Basic 2012 introduced a new pattern that solves some problems related to threading and enables you to write better and cleaner code. It does this with two keywords: Async and Await. Async is a modifier you use to mark methods that run asynchronous operations. `Await` is an operator that gets a placeholder for the result of an asynchronous operation and waits for the result, which will be sent back at a later time while other operations are executed. This enables you to keep the caller thread responsive. For a first understanding of how this pattern works, let's take a look at the following function that downloads the content of a website as a string, returning the result as an `XDocument` that can be manipulated with LINQ:

```
Function DownloadSite() As XDocument
    Dim client As New System.Net.WebClient
    Dim content As String =
        client.DownloadString("http://www.microsoft.com")

    Dim document As XDocument = XDocument.Parse(content)
    Return document
End Function
```

This code is pretty easy because it creates an instance of the `WebClient` class, then downloads the content of the specified website, and finally returns the XML document converted through `XDocument.Parse`. This code is synchronous, meaning that the caller thread will remain blocked until all the operations in the method body are completed. If the caller thread is the UI thread, the user interface will remain blocked. Figure 42.1 shows a graphical representation of how a synchronous call works.

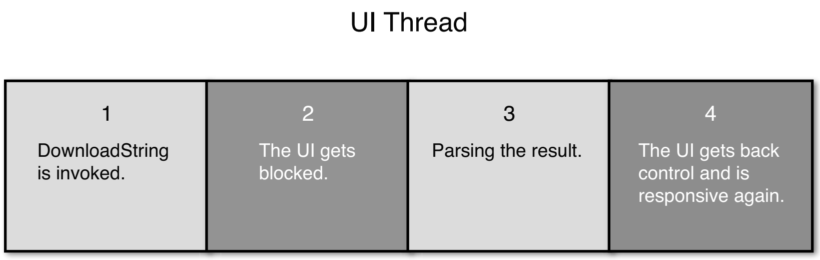

UI Thread
1
DownloadString is invoked.
2
The UI gets blocked.
3
Parsing the result.
4
The UI gets back control and is responsive again.

FIGURE 42.1 Representation of a synchronous call.

This is how you can rewrite the previous code using the `Async` pattern:

```
Async Function DownloadSiteAsync() As Task(Of XDocument)
    Dim client As New System.Net.WebClient
    Dim content As String =
        Await client.DownloadStringTaskAsync("http://www.microsoft.com")

    Dim document As XDocument = XDocument.Parse(content)

    Return document
End Function
```

Await AS A RESERVED KEYWORD

The Await keyword is not a reserved word everywhere in the code. It is a reserved word when it appears inside a method or lambda marked with the Async modifier and only if it appears after that modifier. In all other cases, it is not a reserved word.

This code is asynchronous, so it will never block the caller thread because not all the code is executed at the same time. Figure 42.2 shows a representation of an asynchronous call.

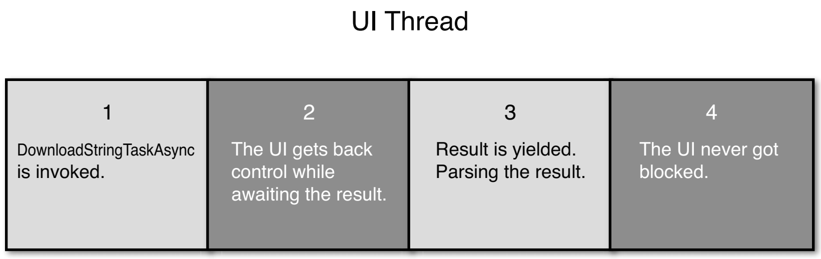

UI Thread
1
DownloadStringTaskAsync is invoked.
2
The UI gets back control while awaiting the result.
3
Result is yielded. Parsing the result.
4
The UI never got blocked.

FIGURE 42.2 Representation of an asynchronous call.

THE STORY OF THREADS WITH Async/Await

The Async pattern relies on the concept of Task described in the previous chapter. For this reason, asynchronous code written with Async/Await does not necessarily run on a separate thread. In fact, it is represented by an instance of the Task class. Because one thread can run multiple Task instances, it is normal that asynchronous code can run in the same caller thread, such as the UI thread.

The following is a list of important considerations that will be discussed in the next section:

- A method that runs asynchronous code must be marked with the Async modifier. When the compiler encounters this modifier, it expects an Await expression inside the method body.
- Methods marked with Async are also referred to as *asynchronous methods*.
- By convention, names of asynchronous methods must end with the Async suffix.
- Asynchronous methods must return a Task (if they return no value) or a Task(Of T), where T is the type that you would return with synchronous methods. See the previous XDocument example.
- Asynchronous methods support the standard access modifiers (such as Private, Public, and so on), but they cannot be iterator methods at the same time, so the Iterator modifier cannot be used along with Async.

- The `Main` method of an application can never be asynchronous. Notice that if you mark the `Main` method as asynchronous and write asynchronous calls in its body, the background compiler will not report any warnings or exceptions. It will report an error when you compile or try to run the code.
- Any method that returns a `Task` or `Task(Of T)` can be used along with `Await` ("awaitable").
- The `Await` expression puts a placeholder for the result of the invoked task. The result will be returned later at some time, making the application remain responsive. However, the next code will actually run when the result has been returned. This is because the compiler ideally splits the method into two parts; the second part is nothing but a callback that is executed when the awaited task notifies the caller of its completion.
- Although you should return a `Task` or `Task(Of T)`, the compiler automatically infers the task-based type even if you return the original type. The second code snippet in the previous example demonstrates how the `Return` statement returns `XDocument` but the compiler automatically returns `Task(Of XDocument)`.

Although these first important considerations might sound confusing, you will soon appreciate the benefits of using the `Async` pattern. It enables you to avoid multithreading and explicit callbacks, enabling you to write much easier and cleaner code. In the next section, you get started with the `Async` pattern with a practical example so that all the concepts described so far will be explained better.

Where Do I Use `Async`?

The `Async` libraries are in the .NET Framework 4.6, so you can use the pattern in whatever technology uses .NET 4.6. WPF, Windows Forms, ASP.NET, and even Windows Store apps. These are all technologies that can leverage the power of these libraries.

In other words, you have no limits in using the new pattern and should always use this new way to asynchrony.

`Async` AND WINDOWS 8.X STORE APPS

There is another important reason beyond the availability of the `Async` pattern in .NET languages now: developing Windows 8.x Store Apps. Windows 8.x Apps require you to write most of the code asynchronously, and using old techniques would make developing apps really difficult. Instead, with the `Async` pattern, coding for Windows 8.x is much faster, easier, and cleaner. Because the user interface of Windows 8.x apps must always be responsive with no exceptions, using the `Async` pattern is very common. Also, the unification of the programming model with the Windows Phone 8.1 platform made the `Async` pattern natively available in these kinds of apps. This is another reason to read this chapter with particular attention.

When and Why to Use Async/Await and Comparisons with the TPL

Using the Async pattern enables you to write applications that are more responsive and that perform better, but you will not use it everywhere. In fact, there are specific situations in which Async has benefits. As a general rule, Async's main purpose is to keep the UI responsive while tasks running in the UI thread might become potentially blocking. You use Async in the following scenarios:

- Potentially blocking tasks running in the user interface thread
- Image processing
- I/O operations (disk, networking, web access)
- Working with sockets

Using Async and Await differs from parallel programming because the purpose is not to have pieces of code that run concurrently; instead, the purpose is keeping the user interface responsive. In parallel programming, you have code that is CPU-consuming, so you use Task instances to split the execution of your code into multiple units of work. Thus, most of the code is executed at the same time by using a multicore architecture. In Async, instead, you do not have CPU-consuming code. You might have potentially blocking tasks, though, so your goal is to keep the UI thread free. This is possible because the result of an Await expression is delayed and control is yielded to the caller while waiting.

Getting Started with Async/Await

In this section you see the Async pattern with an example based on retrieving information from the Internet. You will build a WPF application that downloads RSS feeds information from the Web, simulating a long-running process over a network. You first, though, create an application that works synchronously; then you see how to implement the Event-based Asynchronous Pattern described at the beginning of this chapter. Finally, you learn how things change in a third example built using the new Async and Await keywords.

The Synchronous Approach

Create a new WPF project with Visual Basic 2015 and .NET Framework 4.6. The application will consume the Visual Basic RSS feed exposed by the Microsoft's Channel9 website, with particular regard to the list of published videos. Each item in the feed has a large number of properties, but for the sake of simplicity only the most important will be presented in the application's UI. So, the first thing you need to do is create a class that represents a single video described in the feed. Listing 42.1 demonstrates how to implement a class called Video.

LISTING 42.1 Representing a Single Video

```
Public Class Video
    Public Property Title As String
    Public Property Url As String
    Public Property Thumbnail As String
    Public Property DateRecorded As String
    Public Property Speaker As String

    Public Shared FeedUrl As String = _
           "http://channel9.msdn.com/Tags/visual+basic/RSS"
End Class
```

Notice that all the properties are of type `String` just to represent values as they exactly come from the feed. Also, a shared field contains the feed URL. Now open the MainWindow.xaml file, to prepare the application's user interface. The goal is to show the videos' thumbnails and summary information and to provide the ability to click a thumbnail to open the video in its original location. The `ListBox` control is a good choice to display a collection of items. This will be placed inside the default `Grid` panel. Each item in the `ListBox` will be presented via a custom template made of a Border and a `StackPanel` that contains an `Image` control (for the video thumbnail) and a number of `TextBlock` controls that are bound to properties of the `Video` class. Listing 42.2 shows the full code for the main window.

LISTING 42.2 Implementing the Application's User Interface

```
<Window x:Class="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>
        <ListBox Name="VideoBox" ItemsSource="{Binding}"
                 ScrollViewer.HorizontalScrollBarVisibility="Disabled">
            <ListBox.ItemsPanel>
                <ItemsPanelTemplate>
                    <WrapPanel VirtualizingPanel.IsVirtualizing="True"/>
                </ItemsPanelTemplate>
            </ListBox.ItemsPanel>
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="Black" Margin="5"
                            BorderThickness="2" Tag={Binding Url}
                            MouseLeftButtonUp="Border_MouseLeftButtonUp_1"
                            Width="200" Height="220">
                        <StackPanel>
                            <Image Source="{Binding Thumbnail}"
                                   Width="160" Height="120" />
                            <TextBlock Text="{Binding Title}" TextWrapping="Wrap"
                                       Grid.Row="1"/>
                            <TextBlock Text="{Binding DateRecorded}" Grid.Row="2"/>
                            <TextBlock Text="{Binding Speaker}" Grid.Row="3"/>
                        </StackPanel>
                    </Border>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>
```

It is worth mentioning that the code replaces the default items container (a `VirtualizingStackPanel`) with a `WrapPanel` container so that items are not forced to be presented on one line horizontally. This requires disabling the horizontal scrollbar on the `ListBox` (`ScrollViewer.HorizontalScrollBarVisibility="Disabled"`) and changing the `ListBox.ItemsPanel` content with the `WrapPanel`. Also notice how the `Border.Tag` property is bound to the `Url` property of the `Video` class. This enables you to store the video's URL and click the Border at runtime to open the video in its original location. Now switch to the code-behind file. The first thing you must do is add a number of `Imports` directives, some for importing XML namespaces needed to map information from the RSS feed and some for working with additional .NET classes:

```
Imports System.Net
Imports <xmlns:media="http://search.yahoo.com/mrss/">
Imports <xmlns:dc="http://purl.org/dc/elements/1.1/">
```

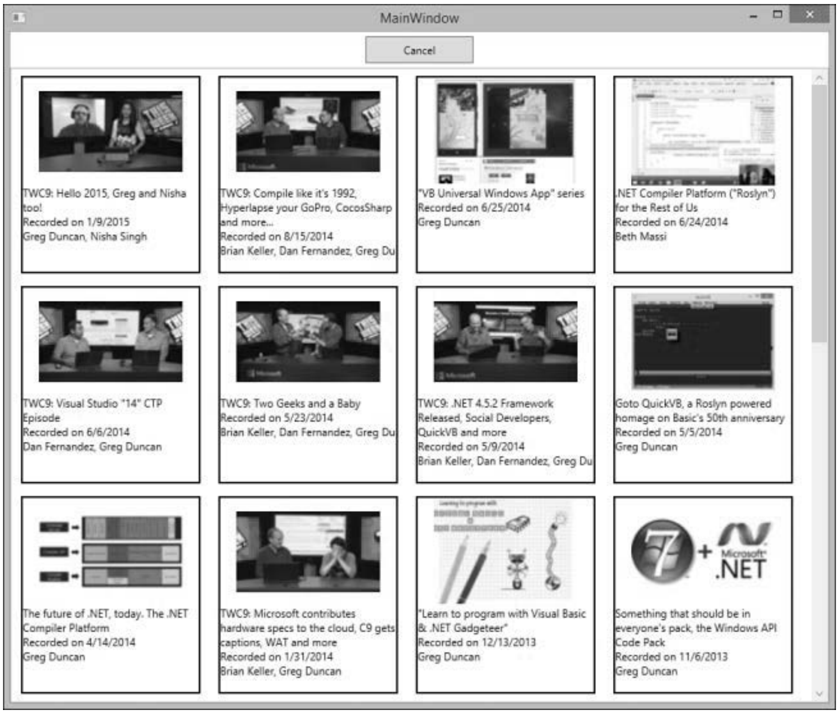
The next step is implementing a method that queries the RSS feed returning the list of videos. In this first implementation, you will use a synchronous approach, which will block the user interface when the application is running:

```
Private Function QueryVideos() As IEnumerable(Of Video)
    Dim client As New WebClient

    Dim data As String = client.DownloadString(New Uri(Video.FeedUrl))

    Dim doc As XDocument = XDocument.Parse(data)
        Dim query = From video In doc...<item>
                    Select New Video With {
                        .Title = video.<title>.Value,
                        .Speaker = video.<dc:creator>.Value,
                        .Url = video.<link>.Value,
                        .Thumbnail = video...<media:thumbnail>.
                        FirstOrDefault?.@url,
                        .DateRecorded = String.Concat("Recorded on ",
                         Date.Parse(video.<pubDate>.Value,
                         Globalization.CultureInfo.InvariantCulture).
                         ToShortDateString)}
    Return query
End Function
```

The code is simple. An instance of the `WebClient` class, which provides simplified access to networked resources, is created and the invocation of its `DownloadString` method downloads the entire content of the feed under the form of a `String` object. Notice that this is the point at which the user interface gets blocked. In fact, it will need to wait for `DownloadString` to complete the operation before returning to be responsive. After the feed has been downloaded, it is converted into an object of type `XDocument` and a LINQ query enables you to retrieve all the needed information (refer to Chapter 27, "Manipulating XML Documents with LINQ and XML Literals," for further information on LINQ to XML). Finally, a method called `LoadVideos` will run the query and assign the result to the Window's `DataContext`; such a method will be invoked at startup. You can change this type of implementation, but it will be more useful later when making comparisons with the asynchronous implementation. The following code demonstrates this, plus the event handler for the `MouseLeftButtonUp` event of the `Border` control, where you launch the video in its original web page:

```
Private Sub LoadVideos()
    Me.DataContext = QueryVideos()
End Sub

Private Sub MainWindow_Loaded(sender As Object,
                              e As RoutedEventArgs) Handles Me.Loaded
    LoadVideos()
End Sub

Private Sub Border_MouseLeftButtonUp_1(sender As Object,
                                        e As MouseButtonEventArgs)
    'Tag is of type Object so an explicit conversion to String is required
    Dim instance = CType(sender, Border)
    Process.Start(CStr(instance.Tag))
End Sub
```

You can now run the application. Figure 42.3 shows the result of the query over the video feed.



TWC9: Hello 2015, Greg and Nisha too!
Recorded on 1/9/2015
Greg Duncan, Nisha Singh

TWC9: Compile like it's 1992, Hyperlapse your GoPro, CocosSharp and more...
Recorded on 8/15/2014
Brian Keller, Dan Fernandez, Greg Du

"VB Universal Windows App" series
Recorded on 6/25/2014
Greg Duncan

.NET Compiler Platform ("Roslyn") for the Rest of Us
Recorded on 6/24/2014
Beth Massi

TWC9: Visual Studio "14" CTP Episode
Recorded on 6/6/2014
Dan Fernandez, Greg Duncan

TWC9: Two Geeks and a Baby
Recorded on 5/23/2014
Brian Keller, Dan Fernandez, Greg Du

TWC9: .NET 4.5.2 Framework Released, Social Developers, QuickVB and more
Recorded on 5/9/2014
Brian Keller, Dan Fernandez, Greg Du

Goto QuickVB, a Roslyn powered homage on Basic's 50th anniversary
Recorded on 5/5/2014
Greg Duncan

The future of .NET, today. The .NET Compiler Platform
Recorded on 4/14/2014
Greg Duncan

TWC9: Microsoft contributes hardware specs to the cloud, C9 gets captions, WAT and more
Recorded on 1/31/2014
Brian Keller, Greg Duncan

"Learn to program with Visual Basic & .NET Gadgeteer"
Recorded on 12/13/2013
Greg Duncan

Something that should be in everyone's pack, the Windows API Code Pack
Recorded on 11/6/2013
Greg Duncan

FIGURE 42.3 Loading an RSS feed the synchronous way.

The application works as expected, but the real problem with this approach is that the user cannot interact with the interface while the query is running. The reason is that the query's code is running in the UI thread, so the user interface is busy with the query and does not accept any interaction. This can be easily demonstrated by attempting to move the window while the query is running because you will not be able to move it elsewhere. This has other implications: you cannot refresh controls that display the status of the task because they would be refreshed only when the query completes. Also, you cannot enable users to cancel the operation because you would need a button that the user would never be able to click.

Event-Based Asynchrony and Callbacks

A much better approach is moving the long-running operation into a separate thread, so that the UI can remain responsive while the other thread executes the operation. Lots of classes in the .NET Framework, especially those whose job is interacting with the Web and with networks-expose event-based asynchrony through methods that launch and execute an operation on a separate thread and raise an event when it is completed, passing the result to the caller via a callback. The `WebClient` class has an asynchronous counterpart of `DownloadString`, called `DownloadStringAsync`, that you can use to execute the code on a separate thread and wait for the query result via a callback. The following code demonstrates how to accomplish this (do not worry if you notice something wrong because an explanation is provided in moments):

```
Private Function QueryVideos() As IEnumerable(Of Video)
    Dim client As New WebClient
    Dim query As IEnumerable(Of Video)

    AddHandler client.DownloadStringCompleted, Sub(sender, e)
                                                   If e.Error IsNot Nothing Then
                                                       'Error handling logic here..
                                                   End If

                                                   Dim doc = _
                                                       XDocument.Parse(e.Result)
                                                   Dim query = From video
                                                               In doc...<item>
                                                               Select _
                                                               New Video With {
                                                     .Title =
                                                     video.<title>.Value,
                                                     .Speaker =
                                                     video.<dc:creator>.
                                                     Value,
                                                     .Url = video.<link>.Value,
                                                     .Thumbnail =
                                                     video...<media:thumbnail>.
                                                     FirstOrDefault?.@url,
                                                     .DateRecorded =
                                                     String.Concat("Recorded on ",
                                                     Date.Parse(video.
                                                     <pubDate>.Value,
                                                     Globalization.CultureInfo.
                                                    InvariantCulture).
                                                    ToShortDateString)}
                                               End Sub

    client.DownloadStringAsync(New Uri(Video.FeedUrl))
    Return query
End Function
```

The code specifies a statement lambda as an event handler for the `DownloadString-Completed` event, instead of declaring a separate delegate and pointing to this via an `AddressOf` clause. The `e` object is of type `DownloadStringCompletedEventArgs` and contains the result of the operation. The problem in this code is that the `Return` statement does not work because it is attempting to return a result that has not been produced yet. On the other side, you cannot write something like this:

```
Private Function QueryVideos() As IEnumerable(Of Video)
    Dim client As New WebClient
    Dim query As IEnumerable(Of Video)

    AddHandler client.DownloadStringCompleted, Sub(sender, e)
                                                   If e.Error IsNot Nothing Then
                                                       'Error handling logic here..
                                                   End If

                                                   Dim doc = _
                                                       XDocument.Parse(e.Result)
                                                   Dim query = From ...
                                                   Return query

                                               End Sub

    client.DownloadStringAsync(New Uri(Video.FeedUrl))
End Function
```

This code does not work because you cannot return a result from a `Sub` and because it should be returned from the outer method, not the inner. In conclusion, `Return` statements do not work well with event-based asynchrony. The solution at this point is returning the result via a callback and an `Action(Of T)` object. So the appropriate implementation of the `QueryVideos` method in this approach is the following:

```
Private Sub QueryVideos(listOfVideos As Action(Of IEnumerable(Of Video),
                                        Exception))
    Dim client As New WebClient
    AddHandler client.DownloadStringCompleted, Sub(sender, e)
                                                   If e.Error IsNot Nothing Then
                                                       listOfVideos(Nothing,
                                                                    e.Error)
                                                       Return
                                                   End If

                                                   Dim doc = _
                                                       XDocument.Parse(e.Result)
                                                   Dim query =
                                                       From video In doc...<item>
                                                       Select New Video With {
                                                       .Title =
                                                       video.<title>.Value,
                                                       .Speaker =
                                                       video.<dc:creator>.
                                                       Value,
                                                       .Url = video.<link>.Value,
                                                       .Thumbnail =
                                                       video...<media:thumbnail>.
                                                       FirstOrDefault?.@url,
                                                       .DateRecorded =
                                                       String.Concat("Recorded on ",
                                                       Date.Parse(video.
                                                       <pubDate>.Value,
                                                       Globalization.CultureInfo.
                                                       InvariantCulture).
                                                       ToShortDateString)}
                                                   listOfVideos(query, Nothing)
                                               End Sub

    Try
        client.DownloadStringAsync(New Uri(Video.FeedUrl))
    Catch ex As Exception
        listOfVideos(Nothing, ex)
    End Try
End Sub
```

The previous code does the following:

1. Holds the list of videos from the RSS feed in an `Action(Of IEnumerable(Of Video), Exception)` object. The `Exception` instance here is useful to determine whether an error occurred during the query execution.
2. If the query completes successfully, the query result is passed to the `Action` object (that is, the callback).
3. If an exception occurs during the query execution (see the first `If` block inside the statement lambda), the callback receives `Nothing` as the first parameter because the collection of items was not retrieved successfully and the exception instance as the second parameter.
4. If an exception occurs immediately when the web request is made, the callback still receives `Nothing` and the exception instance. This is at the `Try..Catch` block level.

So using a callback here has been necessary for two reasons: sending the query result back to the caller correctly and handling two exception scenarios. But you are not done yet. In fact, you have to completely rewrite the `LoadVideos` method to hold the result of the callback and determine whether the operation completed successfully before assigning the query result to the Window's `DataContext`. The following code demonstrates this:

```
Private Sub LoadVideos()
    Dim action As Action(Of IEnumerable(Of Video),
                         Exception) = Nothing
    action =
        Sub(videos, ex)
```

```
            If ex IsNot Nothing Then
                MessageBox.Show(ex.Message)
                Return
            End If

            If (videos.Any) Then
                Me.DataContext = videos
            Else
                QueryVideos(action)
            End If
        End Sub
    QueryVideos(action)
End Sub
```

As you can see, the code is not easy, unless you are an expert. There is an invocation to the previous implementation of `QueryVideos`, passing the instance of the callback. When the result is sent back, the statement lambda first checks for exceptions and, if not, takes the query result as the data source. If you now run the application again, you will get the same result shown in Figure 42.3; however, this time the user interface is responsive and the user can interact with it. But reaching this objective had costs. You had to completely rewrite method implementations and write code that is complex and difficult to read and to extend. So, the multithreading in this situation has not been very helpful. This is the point in which the `Async`/`Await` pattern comes in to make things simple.

Asynchrony with `Async/Await`

The `Async`/`Await` pattern has the goal of simplifying the way developers write asynchronous code. You will learn a lot about the underlying infrastructure, but before digging into that, it is important for you to see how your code can be much cleaner and readable. Let's start by modifying the `QueryVideos` method to make some important considerations:

```
Private Async Function QueryVideosAsync() As  _
        Task(Of IEnumerable(Of Video))
    Dim client As New WebClient

    Dim data = Await client.DownloadStringTaskAsync(New Uri(Video.FeedUrl))

    Dim doc = XDocument.Parse(data)

    Dim query = From video In doc...<item>
                Select New Video With {
                    .Title = video.<title>.Value,
                    .Speaker = video.<dc:creator>.Value,
                    .Url = video.<link>.Value,
                    .Thumbnail = video...<media:thumbnail>.
                    FirstOrDefault?.@url,
                    .DateRecorded = String.Concat("Recorded on ",
                    Date.Parse(video.<pubDate>.Value,
                        Globalization.CultureInfo.InvariantCulture).
                    ToShortDateString)}

    Return query
End Function
```

Asynchronous methods must be decorated with the `Async` modifier. When the compiler encounters this modifier, it expects that the method body contains one or more `Await` statements. If not, it reports a warning saying that the method will be treated as synchronous, suggesting that the `Async` modifier should be removed. `Async` methods must return an object of type `Task`. If the method returns a value (`Function`), then it must return a `Task(Of T)` where `T` is the actual result type. Otherwise, if the method returns no value, both following syntaxes are allowed:

```
Async Function TestAsync() As Task
   'You can avoid Return statements, the compiler assumes returning no values
End Function

Async Sub TestAsync()
'...
End Sub
```

The difference between the two implementations is that the first one can be called inside another method with `Await`, but the second one cannot (because it does not need to be awaited). A typical example of the second syntax is about event handlers: they can be asynchronous and can use `Await`, but no other method will wait for their result. By convention, the suffix of asynchronous methods is the `Async` literal. This is why `QueryVideos` has been renamed into `QueryVideosAsync`. An exception is represented by asynchronous methods already existing in previous versions of the .NET Framework, based on the EAP, whose name already ends with `Async`. In this case `Async` is replaced with `TaskAsync`. For instance (as you discover in moments), the `DownloadStringAsync` method in the `WebClient` class has a new counterpart called `DownloadStringTaskAsync`. Any method that returns a `Task` or `Task(Of T)` can be used with `Await`. With `Await`, a task is started but the control flow is immediately returned to the caller. The result of the task will not be returned immediately, but later and only when the task completes. But because the control flow immediately returns to the caller, the caller remains responsive. `Await` can be thought as of a placeholder for the task's result, which will be available after the awaited task completes. In the previous `QueryVideosAsync` method, `Await` starts the `WebClient.DownloadStringTaskAsync` method and literally waits for its result but, while waiting, the control flow does not move to `DownloadStringAsyncTask`, while it remains in the caller. Because in the current example the code is running in the UI thread, the user interface remains responsive because the requested task is being executed asynchronously.

In other words, what `Await` actually does is sign up the rest of the method as a callback on the task, returning immediately. When the task that is being awaited completes, it will invoke the callback and will resume the execution from the exact point it was left.

After the operation has been completed, the rest of the code can elaborate the result. With this kind of approach, your method looks much simpler, like the first synchronous version, but it is running asynchronously with only three edits (the `Async` modifier, `Task(Of T)` as the return type, and the `Await` operator).

WHY THE `Task` TYPE?

In Chapter 41 you learned a lot about the `Task` class and saw how this can run CPU-intensive work on a separate thread, but it also can represent an I/O operation such as a network request. For this reason, the `Task` class is the natural choice as the result type for asynchronous operations using `Async/Await`.

Continuing considerations on the previous method, take a look at the final `Return` statement. It is returning an `IEnumerable(Of Video)`, but actually the method's signature requires returning a `Task(Of IEnumerable(Of Video))`. This is possible because the compiler automatically makes `Return` statements to return a `Task`-based version of their result even if they do not. As a result, you will not get confused because you will write the same code but the compiler will take care of converting the return type into the appropriate type. This also makes migration of synchronous code to asynchronous easier. Technically speaking, the compiler synthesizes a new `Task(Of T)` object at the first `Await` in the method. This `Task(Of T)` object is returned to the caller at the first `Await`. Later on, when it encounters a `Return` statement, the compiler causes that already-existing `Task(Of T)` object to transition from a "not yet completed" state into the "completed with result" state. Continuing the migration of the code example to the `Async/Await` pattern, you now need a few edits to the `LoadVideos` method. The following code demonstrates this:

```
Private Async Function LoadVideosAsync() As Task
    Me.DataContext = Await QueryVideosAsync()
End Sub
```

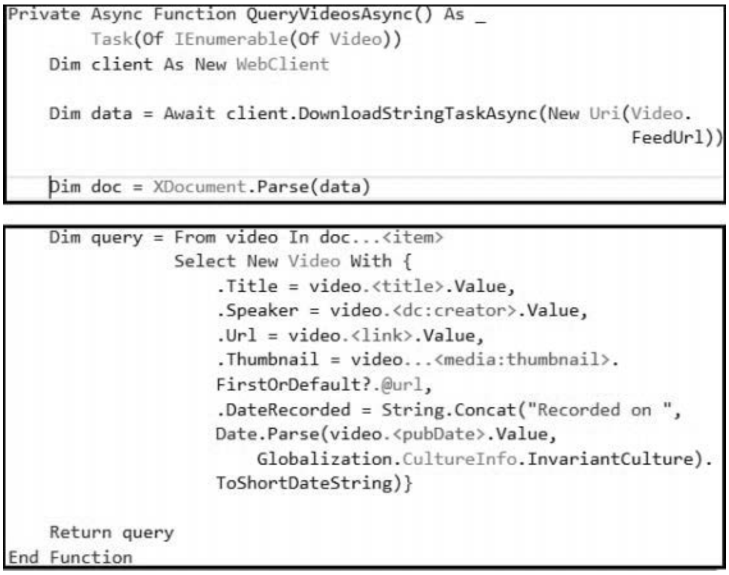
The method is now called `LoadVideoAsync` and marked with the `Async` modifier. The reason is that it contains an `Await` expression that invokes the `QueryVideosAsync` method. The result of this invocation is taken as the main window's data source. Finally, you have to edit the `MainWindow_Loaded` event handler and make it asynchronous like this:

```
Private Async Sub MainWindow_Loaded(sender As Object,
                                    e As RoutedEventArgs) Handles Me.Loaded
    Await LoadVideosAsync()
End Sub
```

If you now run the application, you will still get a responsive user interface that you can interact with while the long-running task (the query) is executing, but you have achieved this by modifying existing code with very few edits.

How `Async` and `Await` Work Behind the Scenes

Behind the scenes of the ease of the `Async` pattern, the compiler does incredible work to make the magic possible. When you make an asynchronous call by using `Await`, that invocation starts a new instance of the `Task` class. As you know from Chapter 41, one thread can contain multiple `Task` instances. So you might have the asynchronous operation running in the same thread but on a new `Task`. Internally, it's as if the compiler could split an asynchronous method in two parts, a method and its callback. If you consider the `QueryVideosAsync` shown previously, you could imagine a method defined until the invocation of `Await`. The next part of the method is moved into a callback that is invoked after the awaited operation is completed. This has two benefits. The first benefit is that it ensures that code that needs to manipulate the result of an awaited operation will work with the actual result, which has been returned after completion of the task (this is in fact the moment in which the callback is invoked). Second, such callback is invoked in the same calling thread, which avoids the need of managing threads manually or using the `Dispatcher` class in technologies like WPF or Silverlight. Figure 42.4 gives you an ideal representation of how the asynchronous method has been split.

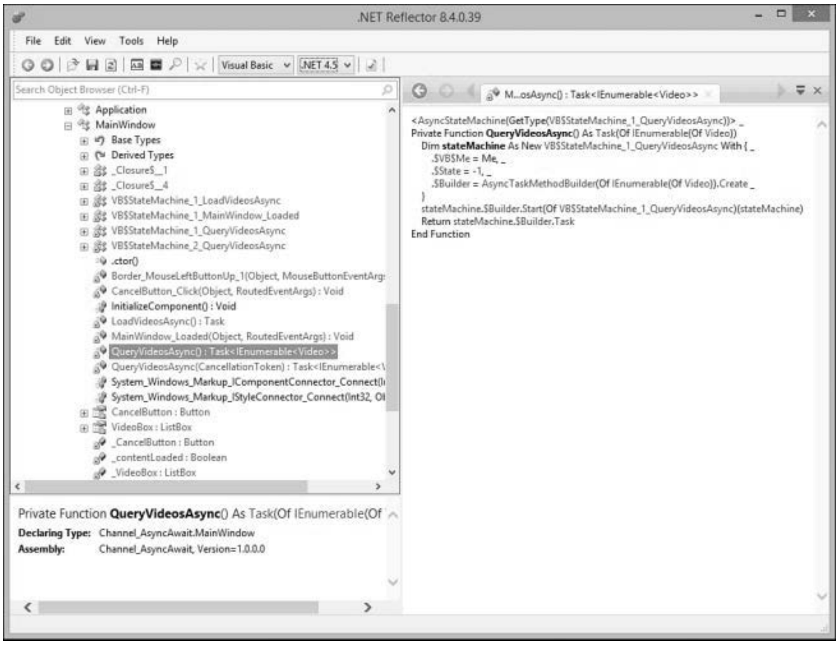


```
Private Async Function QueryVideosAsync() As _
        Task(Of IEnumerable(Of Video))
    Dim client As New WebClient

    Dim data = Await client.DownloadStringTaskAsync(New Uri(Video.
                                                    FeedUrl))

    Dim doc = XDocument.Parse(data)
```

```
    Dim query = From video In doc...<item>
                Select New Video With {
                    .Title = video.<title>.Value,
                    .Speaker = video.<dc:creator>.Value,
                    .Url = video.<link>.Value,
                    .Thumbnail = video...<media:thumbnail>.
                    FirstOrDefault?.@url,
                    .DateRecorded = String.Concat("Recorded on ",
                    Date.Parse(video.<pubDate>.Value,
                        Globalization.CultureInfo.InvariantCulture).
                    ToShortDateString)}

    Return query
End Function
```

FIGURE 42.4 An asynchronous method is split into two ideal parts; the second is a callback.

Beyond considerations like the ones about threading, it is interesting to analyze the code the compiler generated to make asynchrony so efficient in Visual Basic 2015. For this exercise, you need a decompiler tool such as .NET Reflector from Red-Gate, which is available as a free trial from https://www.red-gate.com/products/dotnet-development/reflector/. If you open the compiled .exe file with a tool like this, you can see that the implementation of asynchronous methods is completely different from the one you wrote and that the compiler generated several structures that implement a state machine that supports asynchrony. Figure 42.5 shows the `QueryVideosAsync` real implementation.


.NET Reflector 8.4.0.39
File Edit View Tools Help
Visual Basic
.NET 4.5
Search Object Browser (Ctrl-F)
M...osAsync() : Task<IEnumerable<Video>>

```
Application
MainWindow
  Base Types
  Derived Types
  _Closure$__1
  _Closure$__4
  VB$StateMachine_1_LoadVideosAsync
  VB$StateMachine_1_MainWindow_Loaded
  VB$StateMachine_1_QueryVideosAsync
  VB$StateMachine_2_QueryVideosAsync
  .ctor()
  Border_MouseLeftButtonUp_1(Object, MouseButtonEventArg:
  CancelButton_Click(Object, RoutedEventArgs) : Void
  InitializeComponent() : Void
  LoadVideosAsync() : Task
  MainWindow_Loaded(Object, RoutedEventArgs) : Void
  QueryVideosAsync() : Task<IEnumerable<Video>>
  QueryVideosAsync(CancellationToken) : Task<IEnumerable<\
  System_Windows_Markup_IComponentConnector_Connect(I
  System_Windows_Markup_IStyleConnector_Connect(Int32, Ol
  CancelButton : Button
  VideoBox : ListBox
  _CancelButton : Button
  _contentLoaded : Boolean
  _VideoBox : ListBox
```

```
Private Function QueryVideosAsync() As Task(Of IEnumerable(Of
Declaring Type:  Channel_AsyncAwait.MainWindow
Assembly:        Channel_AsyncAwait, Version=1.0.0.0
```

```
<AsyncStateMachine(GetType(VB$StateMachine_1_QueryVideosAsync))> _
Private Function QueryVideosAsync() As Task(Of IEnumerable(Of Video))
   Dim stateMachine As New VB$StateMachine_1_QueryVideosAsync With { _
      .$VB$Me = Me, _
      .$State = -1, _
      .$Builder = AsyncTaskMethodBuilder(Of IEnumerable(Of Video)).Create _
   }
   stateMachine.$Builder.Start(Of VB$StateMachine_1_QueryVideosAsync)(stateMachine)
   Return stateMachine.$Builder.Task
End Function
```

FIGURE 42.5 Investigating the actual generated code with .NET Reflector.

For your convenience, the following is the auto-generated code for `QueryVideosAsync`:

```
<AsyncStateMachine(GetType(VB$StateMachine_1_QueryVideosAsync))> _
Private Function QueryVideosAsync() As Task(Of IEnumerable(Of Video))
    Dim stateMachine As New VB$StateMachine_1_QueryVideosAsync With { _
        .$VB$Me = Me, _
        .$State = -1, _
        .$Builder = AsyncTaskMethodBuilder(Of IEnumerable(Of Video)).Create _
    }
    stateMachine.$Builder.
    Start(Of VB$StateMachine_1_QueryVideosAsync)(stateMachine)
    Return stateMachine.$Builder.Task
End Function
```

You do not need to know the code in detail because this implementation is purely internal; however, the real code relies on the `AsyncTaskMethodBuilder` class, which creates an instance of an asynchronous method and requires specifying a state machine that controls the execution of the asynchronous task (which is returned once completed). For each asynchronous method, the compiler generated an object representing the state machine. For instance, the compiler generated an object called `VB$StateMachine_1_QueryVideosAsync` that represents the state machine that controls the execution of the `QueryVideosAsync` method. Listing 42.3 contains the code of the aforementioned structure.

LISTING 42.3 Internal Implementation of a State Machine for `Async` Methods

```
<CompilerGenerated> _
Private NotInheritable Class VB$StateMachine_1_QueryVideosAsync
    Implements IAsyncStateMachine
    ' Methods
    Public Sub New()
    <CompilerGenerated> _
    Friend Sub MoveNext() Implements IAsyncStateMachine.MoveNext
    <DebuggerNonUserCode> _
    Private Sub SetStateMachine(stateMachine As IAsyncStateMachine) _
            Implements IAsyncStateMachine.SetStateMachine

    ' Fields
    Friend $A0 As TaskAwaiter(Of String)
    Public $Builder As AsyncTaskMethodBuilder(Of IEnumerable(Of Video))
    Public $State As Integer
    Friend $VB$Me As MainWindow
    Friend $VB$ResumableLocal_client$0 As WebClient
    Friend $VB$ResumableLocal_data$1 As String
    Friend $VB$ResumableLocal_doc$2 As XDocument
    Friend $VB$ResumableLocal_query$3 As IEnumerable(Of Video)
End Class
```

The code in Listing 42.3 is certainly complex, and you are not required to know how it works under the hood, but focus for a moment on the `MoveNext` method. This method is responsible of the asynchronous execution of tasks; depending on the state of the task, it resumes the execution at the appropriate point. You can see how the compiler translates the `Await` keyword into an instance of the `TaskAwaiter` structure, which is assigned with the result of the invocation to the `Task.GetAwaiter` method (both are for compiler-use only). If you compare the result of this analysis with the ease of usage of the `Async` pattern, it is obvious that the compiler does tremendous work to translate that simplicity into a very efficient asynchronous mechanism.

Documentation and Examples of the `Async` Pattern

Microsoft offers a lot of useful resources to developers who want to start coding the new way. The following list summarizes several resources that you are strongly encouraged to visit to make your learning of `Async` complete:

- **Visual Studio Asynchronous Programming:** The official developer center for `Async`. Here you can find documentation, downloads, instructional videos, and more on language specifications. It is available at http://msdn.microsoft.com/en-us/vstudio/async.aspx.

- **101 Async Samples:** An online page that contains a huge number of code examples based on Async for both Visual Basic and Visual C#. You can find samples at http://www.wischik.com/lu/AsyncSilverlight/AsyncSamples.html.
- **Sample code:** Available on the MSDN Code Gallery (http://code.msdn.microsoft.com).

Do not leave out of your bookmarks the root page of the .NET Framework 4.5 and 4.6 documentation (http://msdn.microsoft.com/en-us/library/w0x726c2).

Exception Handling in `Async`

Another great benefit of using the `Async` pattern is that exception handling is done the usual way. In fact, if an awaited method throws an exception, this can be naturally handled within a `Try..Catch..Finally` block. The following code provides an example:

```
Private Async Sub DownloadSomethingAsync()
    Dim client As New System.Net.WebClient
    Try
        Dim result = Await client.
            DownloadStringTaskAsync("http://msdn.com/vbasic")
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    Finally
        Console.WriteLine("Operation completed.")
    End Try
End Sub
```

As you can see, there is no difference in handling exceptions inside asynchronous methods compared to classic synchronous methods. This makes code migration easier.

Implementing Task-Based Asynchrony

As you remember from Chapter 41, the `Task` class provides methods and other members that enable you to execute CPU-intensive work, by splitting code across all the available processors so that most of the code is executed concurrently, when possible. Such members of the `Task` class return instances of the `Task` class itself, and therefore can be used along with `Await`. This possibility has some advantages:

- You can execute synchronous code on a separate thread more easily.
- You can run multiple tasks concurrently and wait for them to complete before making further manipulations.
- You can use `Await` with CPU-consuming code.

This approach is known as Task-Based Asynchrony, and in this section you learn how to get the most out of it.

Switching Threads

In Chapter 40 you learned how to write code that can run on a separate thread, how to create new threads manually, and how to use the Thread Pool managed by the .NET Framework. With this approach, you run a portion of synchronous code in a separate thread, thus keeping the caller thread-free from an intensive and potentially blocking work. In the .NET Framework 4.6, you have additional alternatives to reach the same objective but writing simpler code. The `Task.Run` method enables you to run a new task asynchronously, queuing such a task into a thread in the Thread Pool. The result is returned as `Task` handle for the intensive work, so that you can use `Await` to wait for the result. `Task.Run` takes as the first argument a delegate that defines the work that will be executed in the background thread. Such a delegate can be represented either by a method that you point to via the `AddressOf` clause or by lambdas. In the latter case, the delegate can be a `System.Action` represented by a statement lambda or a `System.Func(Of T)` represented by a lambda expression. The following example demonstrates how synchronous code is easily executed in a separate thread by invoking `Task.Run`:

```
Private Async Sub RunIntensiveWorkAsync()
    'This runs on the UI thread
    Console.WriteLine("Starting...")

    'This runs on a Thread Pool thread
    Dim result As Integer = Await Task.Run(Function()
                                               Dim workResult As Integer = _
                                                   SimulateIntensiveWork()
                                               Return workResult
                                           End Function)


    'This runs again on the UI thread
    Console.WriteLine("Finished")
    Console.ReadLine()
End Sub

Private Function SimulateIntensiveWork() As Integer
    Dim delay As Integer = 5000
    Threading.Thread.Sleep(delay)
    Return delay
End Function
```

While the result of `Task.Run` is being awaited, the control is immediately returned to the user interface, which remains responsive in the Console window. All the `Console.WriteLine` and `Console.ReadLine` statements are executed on the UI thread, whereas the simulated CPU-consuming code runs on the separate thread. `Task.Run` schedules a new task exactly as `Task.Factory.StartNew` does; you saw this method in Chapter 41. So, this code has the same effect as using `Task.Run`:

```
Dim result As Integer = Await Task.Factory.StartNew(Function()
                                                       Dim workResult _
                                                       As Integer = _
                                                           SimulateIntensiveWork()
                                                       Return workResult
                                                   End Function)
```

In summary, `Task.Run` lets you easily execute intensive computations on a separate thread, taking all the benefits of `Await`.

Using Combinators

The `Task` class has other interesting usages, such as managing concurrent operations. This is possible because of two methods, `Task.WhenAll` and `Task.WhenAny`, also known as *combinators*. `Task.WhenAll` creates a task that will complete when all the supplied tasks complete; `Task.WhenAny` creates a task that will complete when at least one of the supplied tasks completes. For example, imagine you want to download multiple RSS feeds information from a website. Instead of using `Await` against individual tasks to complete, you can use `Task.WhenAll` to continue only after all tasks have completed. The following code provides an example of concurrent download of RSS feeds from the Microsoft Channel 9 feed used previously:

```
Private Async Sub DownloadAllFeedsAsync()
    Dim feeds As New List(Of Uri) From
        {New Uri("http://channel9.msdn.com/Tags/windows+8/RSS"),
        New Uri("http://channel9.msdn.com/Tags/windows+phone"),
        New Uri("http://channel9.msdn.com/Tags/visual+basic/RSS")}

    'This task completes when all of the requests complete
    Dim feedCompleted As IEnumerable(Of String) = _
                                           Await Task.
                                           WhenAll(From feed In feeds
                                           Select New System.Net.WebClient().
                                           DownloadStringTaskAsync(feed))

    'Additional work here...
End Sub
```

This code creates a collection of tasks by sending a `DownloadStringTaskAsync` request for each feed address in the list of feeds. The task completes (and thus the result of awaiting `WhenAll` is returned) only when all three feeds have been downloaded, meaning that the complete download result will not be available if only one or two feeds have been downloaded. `WhenAny` works differently because it creates a task that completes when any of the tasks in a collection of tasks completes. The following code demonstrates how to rewrite the previous example using `WhenAny`:

```
Private Async Sub DownloadFeedsAsync()
    Dim feeds As New List(Of Uri) From
        {New Uri("http://channel9.msdn.com/Tags/windows+8/RSS"),
        New Uri("http://channel9.msdn.com/Tags/windows+phone"),
        New Uri("http://channel9.msdn.com/Tags/visual+basic/RSS")}

    'This task completes when any of the requests complete
    Dim feedCompleted As Task(Of String) = Await Task.WhenAny(From feed In feeds
                                                 Select New System.Net.WebClient().
                                                 DownloadStringTaskAsync(feed))
    'Additional work here...
End Sub
```

In this case a single result will be yielded because the task will be completed when any of the tasks completes. You can also wait for a list of tasks defined as explicit asynchronous methods, like in the following example:

```
Public Async Sub WhenAnyRedundancyAsync()

    Dim messages As New List(Of Task(Of String)) From
        {
            GetMessage1Async(),
            GetMessage2Async(),
            GetMessage3Async()
        }
    Dim message = Await Task.WhenAny(messages)
    Console.WriteLine(message.Result)
    Console.ReadLine()
End Sub

Public Async Function GetMessage1Async() As Task(Of String)
    Await Task.Delay(700)
    Return "Hi VB guys!"
End Function

Public Async Function GetMessage2Async() As Task(Of String)
    Await Task.Delay(600)
    Return "Hi C# guys!"
End Function

Public Async Function GetMessage3Async() As Task(Of String)
    Await Task.Delay(500)
    Return "Hi F# guys!"
End Function
```

Here you have three asynchronous methods, each returning a string. The code builds a list of tasks including each asynchronous method in the list. `Task.WhenAny` receives the instance of the collection of tasks as an argument and completes when one of the three methods completes. In this example, you are also seeing for the first time the `Task.Delay` method. This is the asynchronous equivalent of `Thread.Sleep`, but while the latter blocks the thread for the specified number of milliseconds, with `Task.Delay` the thread remains responsive.

ADDITIONAL SAMPLES ON `WhenAny`

The 101 `Async` Samples include a couple of interesting examples of different usages of `WhenAny`, such as interleaving one request at a time and limiting the number of concurrent downloads. You find them under the Combinators node of the samples page mentioned at the beginning of the chapter.

Cancellation and Progress

Because the `Async` pattern relies on the `Task` class, implementing cancellation is something similar to what you have already studied back in Chapter 41, thus the `CancellationTokenSource` and `CancellationToken` classes are used. In this section you see how to implement cancellation both for asynchronous methods and for `Task.Run` operations. Next, you learn how to report the progress of an operation, which is common in asynchronous programming and improves the user experience.

Implementing Cancellation

Let's look at the WPF sample application created in the section "Getting Started with `Async`/`Await`." Imagine you want to give users the ability of cancelling the download of the RSS feed from the Microsoft Channel9 website. First, make a slight modification to the user interface so that the main `Grid` is divided into two rows, and in the first row add a `Button` like this:

```
<Grid.RowDefinitions>
    <RowDefinition Height="40"/>
    <RowDefinition/>
</Grid.RowDefinitions>
<Button Width="120" Height="30" Name="CancelButton"
        Content="Cancel"/>
```

Do not forget to add the `Grid.Row="1"` property assignment for the `ListBox` control. Double-click the new button so that you can quickly access the code editor. Declare a new `CancellationTokenSource` object that will listen for cancellation requests. The event handler for the new button's `Click` event will invoke the `Cancel` method on the instance of the `CancellationTokenSource`:

```
Private tokenSource As CancellationTokenSource

Private Sub CancelButton_Click(sender As Object, e As RoutedEventArgs) _
    Handles CancelButton.Click
    'If Me.tokenSource IsNot Nothing Then
    '    Me.tokenSource.Cancel()
    'End If
    Me.tokenSource?.Cancel()
End Sub
```

The user can now request cancellation by clicking this button. Next, you need to make a couple of edits to the `QueryVideosAsync` method created previously. The first edit is making this method receive a `CancellationToken` object as an argument. This object will handle cancellation requests during the method execution. The second edit requires replacing the `WebClient` class with a new class called `HttpClient`. The reason for this change is that the `WebClient`'s asynchronous methods no longer support cancellation as in the first previews of the `Async` library, although asynchronous methods in `System.Net.Http.HttpClient` do. Among the others, this class exposes a method called `GetAsync` that retrieves contents from the specified URL and receives the cancellation token as the second argument. The result is returned under the form of a `System.Net.Http.HttpResponseMessage` class. As the name implies, this class represents an HTTP response message including the status of the operation and the retrieved data. The data is represented by a property called `Content`, which exposes methods to convert data into a stream (`ReadAsStreamAsync`), into an array of bytes (`ReadAsByteArrayAsync`), and into a string (`ReadAsStringAsync`). Other than changing the code to use `HttpClient` and to receive the cancellation token, you only need to handle the `OperationCanceledException`, which is raised after the cancellation request is received by the asynchronous method. The following code demonstrates the `QueryVideosAsync` method:

```
'The following implementation with HttpClient supports Cancellation
Private Async Function QueryVideosAsync(token As CancellationToken) As  _
        Task(Of IEnumerable(Of Video))
    Try
        Dim client As New HttpClient

        'Get the feed content as an HttpResponseMessage
        Dim data = Await client.GetAsync(New Uri(Video.FeedUrl), token)

        'Parse the content into a String
        Dim actualData = Await data.Content.ReadAsStringAsync

        Dim doc = XDocument.Parse(actualData)

        Dim query = From video In doc...<item>
                    Select New Video With {
                    .Title = video.<title>.Value,
                    .Speaker = video.<dc:creator>.Value,
                    .Url = video.<link>.Value,
                        .Thumbnail = video...<media:thumbnail>.
                            FirstOrDefault?.@url,
                        .DateRecorded = String.Concat("Recorded on ",
                            Date.Parse(video.<pubDate>.Value,
                            Globalization.CultureInfo.InvariantCulture).
                            ToShortDateString)}

        Return query
    Catch ex As OperationCanceledException
        MessageBox.Show("Operation was canceled by the user.")
        Return Nothing
    Catch ex As Exception
        MessageBox.Show(ex.Message)
        Return Nothing
    End Try
End Function
```

The very last edit to the application is changing the `LoadVideosAsync` method to launch the query passing a cancellation token:

```
Private Async Function LoadVideosAsync() As Task
    Me.tokenSource = New CancellationTokenSource

    Me.DataContext = Await QueryVideosAsync(Me.tokenSource.Token)
End Function
```

If you now run the application, not only will the user interface remain responsive, but you will be also able to click the Cancel button to stop the query execution. Notice that in a synchronous approach, implementing cancellation has no benefits. In fact, if on one side writing code to support cancellation is legal, on the other side the user would never have a chance to click a button because the UI thread would be blocked until the completion of the task. Similarly, you can add cancellation to tasks running in a separate thread and started with `Task.Run`. By continuing the example shown previously about this method, you can first rewrite the `SimulateIntensiveWork` method as follows:

```
Private Function SimulateIntensiveWork(token As CancellationToken) _
         As Integer
    Dim delay As Integer = 5000
    Threading.Thread.Sleep(delay)

        token.ThrowIfCancellationRequested()

    Return delay
End Function
```

You should be familiar with this approach because it has been discussed in Chapter 41. The method receives the cancellation token and checks for cancellation requests. If any exist, it throws an `OperationCanceledException`. Next, you add support for cancellation by passing an instance of the `CancellationTokenSource` class to the method invocation inside `Task.Run`:

```
Private cancellationToken As CancellationTokenSource

Private Async Sub RunIntensiveWorkAsync()
    cancellationToken = New CancellationTokenSource
    'This runs on the UI thread
    Console.WriteLine("Starting...")

    Try
        'This runs on a Thread Pool thread
        Dim result As Integer = Await Task.Run(Function()
                                                   Dim workResult As Integer = _
                                                   SimulateIntensiveWork( _
                                                   cancellationToken.Token)
                                                   Return workResult
                                               End Function)

        'This runs again on the UI thread
        Console.WriteLine("Finished")
    Catch ex As OperationCanceledException
        Console.WriteLine("Canceled by the user.")
    Catch ex As Exception

    End Try
    Console.ReadLine()
End Sub
```

To request cancellation, you should call the `cancellationToken.Cancel` method. At that point, the request is intercepted and an `OperationCanceledException` is thrown.

Reporting Progress

Reporting the progress of an asynchronous method execution is a common requirement. There is a pattern that you can use and that makes things easier. This pattern relies on the `System.IProgress(Of T)` interface and the `System.Progress(Of T)` class, which expose a `ProgressChanged` event that must be raised when the asynchronous operation is in progress. To provide an example that is easy to understand, imagine you still want to download the content of some feeds from the Microsoft Channel9 website and refresh the progress every time a site has been downloaded completely. The current example is based on a Console application. Consider the following code:

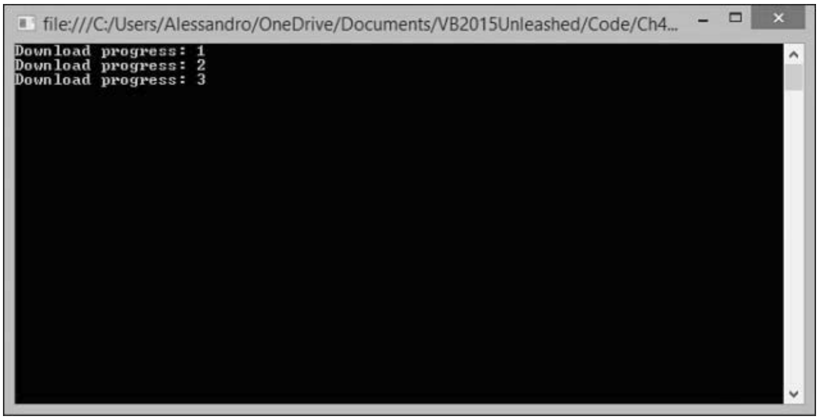
```
Private progress As Progress(Of Integer)
Private counter As Integer = 0
```

```
Sub Main()
    Try
        progress = New Progress(Of Integer)
        AddHandler progress.ProgressChanged, Sub(sender, e)
                                                 Console.
                                                 WriteLine _
                                                 ("Download progress: " & _
                                                 CStr(e))
                                             End Sub

        DownloadAllFeedsAsync(progress)

    Catch ex As Exception
        Console.WriteLine(ex.Message)
    Finally
        Console.ReadLine()
    End Try
End Sub
```

You first declare an object of type `Progress(Of Integer)` and a counter. The first object will receive the progress value when the `ProgressChanged` event is raised. In this case, the code uses the `Integer` type to pass a simple number, but you can pass more complex information with a different or custom type. Then the code specifies a handler for the `ProgressChanged` event, with type inference for the lambda's parameters. `Sender` is always `Object`, whereas `e` is of the same type as the generic type you assigned to `Progress`. So, in this case it is of type `Integer`. Here you are working in a Console application and are thus displaying the value as a text message. But in real-world applications, this is the value that you could assign to a `ProgressBar` control to report the progress in the user interface. The instance of the `Progress` class must be passed to the asynchronous method that performs the required tasks. The `Progress` class has just one method called `Report`; you invoke it after an `Await` invocation. The following code demonstrates how to report the progress of downloading a number of feeds:

```
Private Async Sub DownloadAllFeedsAsync(currentProgress As IProgress(Of Integer))
    Dim client As New System.Net.WebClient

    Dim feeds As New List(Of Uri) From
        {New Uri("http://channel9.msdn.com/Tags/windows+8/RSS"),
         New Uri("http://channel9.msdn.com/Tags/windows+phone"),
         New Uri("http://channel9.msdn.com/Tags/visual+basic/RSS")}
    For Each URL In feeds
        Await client.DownloadStringTaskAsync(URL)
        counter += 1
        If currentProgress IsNot Nothing Then currentProgress.Report(counter)
    Next
End Sub
```

`Report` receives as an argument an object of the same type that you assigned as the generic argument of the `Progress` class declaration in this case a counter of type `Integer` that is incremented every time a feed is downloaded. If you run this code, every time a feed is returned, the progress is also shown in the user interface, as demonstrated in Figure 42.6.


file:///C:/Users/Alessandro/OneDrive/Documents/VB2015Unleashed/Code/Ch4...

```
Download progress: 1
Download progress: 2
Download progress: 3
```

FIGURE 42.6 Reporting the progress of an asynchronous method.

This pattern makes reporting the progress of a task and in real-world applications, such as WPF and Windows apps, easy. It also makes updating controls like the `ProgressBar` incredibly simple by assigning to such controls the value stored in the instance of the `Progress` class.

Asynchronous Lambda Expressions

Methods can be asynchronous, but so can lambda expressions. To be asynchronous, a lambda must have the `Async` modifier and must return `Task` or `Task(Of T)`, but it cannot accept `ByRef` arguments and cannot be an iterator function. A lambda can be asynchronous when its code uses the `Await` operator to wait for a `Task` result. An example of asynchronous lambdas is with event handlers. For instance, you might need to wait for the result of a task when an event is raised, as in the following code snippet that handles a button's `Click`:

```
AddHandler Me.Button1.Click, Async Sub(sender, e)
                                 Await DoSomeWorkAsync
                             End Sub
```

You do not need an asynchronous lambda if the work you are going to execute does not return a `Task`. Another typical usage of asynchronous lambdas is with `Task.Run`. The following code shows the same example described when introducing `Task.Run`, but now the lambda that starts the intensive work is marked with `Async` and the method that actually performs intensive computations returns a `Task` so that it can be awaited:

```
Private Async Sub RunIntensiveWorkAsync()
    cancellationToken = New CancellationTokenSource
    'This runs on the UI thread
    Console.WriteLine("Starting...")

    Try
        'This runs on a Thread Pool thread
        Dim result As Integer = Await Task.Run(Async Function()
                                                   Dim workResult As Integer = _
                                                    Await _
                                                    SimulateIntensiveWorkAsync()
                                                   Return workResult
                                               End Function)
        'This runs again on the UI thread
        Console.WriteLine("Finished")
    Catch ex As OperationCanceledException
        Console.WriteLine("Canceled by the user.")
    Catch ex As Exception

    End Try
    Console.ReadLine()
End Sub

Private Async Function SimulateIntensiveWorkAsync() As Task(Of Integer)
    Dim delay As Integer = 1000
    Await Task.Delay(delay)
    Return delay
End Function
```

This code simulates CPU-intensive work inside an asynchronous method. However, this is not best practice and should be avoided when possible. Here it is shown for demonstration purposes only. For additional tips about asynchronous methods, visit http://channel9.msdn.com/Series/Three-Essential-Tips-for-Async/Async-Library-Methods-Shouldn-t-Lie.

Asynchronous I/O File Operations in .NET 4.6

Before .NET Framework 4.5, you could perform asynchronous operations over files and streams by using the Asynchronous Programming Model and methods such as `Stream.BeginRead` and `Stream.EndRead`. This kind of approach can be good, but it has the limitations described in the section "Getting Started with `Async`/`Await`" in this chapter. With .NET Framework 4.5 and after, asynchronous I/O operations can be simplified by using the `Async` pattern and by implementing asynchronous versions of methods that work with files and stream to avoid blocking the main thread. Such methods are exposed by the

`Stream`, `FileStream`, `MemoryStream`, `TextReader`, and `TextWriter` classes that you saw in action back in Chapter 18, "Manipulating Files and Streams." Table 42.1 summarizes the available asynchronous methods.

TABLE 42.1 Asynchronous Methods for Stream Classes

Method	Description	Return Type
`ReadAsync`	Reads a sequence of bytes from a stream and advances the position by the number of bytes read, with an asynchronous approach	`Task(Of Integer)`
`WriteAsync`	Writes a sequence of bytes to a stream, with an asynchronous approach	`Task`
`FlushAsync`	Clears buffers associated with the stream sending buffered data to the stream, using asynchrony	`Task`
`CopyToAsync`	Asynchronously copies a number of bytes from a stream to another	`Task`
`ReadLineAsync`	Reads a line of characters using asynchrony and returns a string	`Task(Of String)`
`ReadToEndAsync`	Asynchronously reads all characters from the current position to the end of the stream, and returns one string	`Task(Of String)`

To see some of these methods in action, create a new WPF project. The user interface of this sample application will have to look like Figure 42.7.

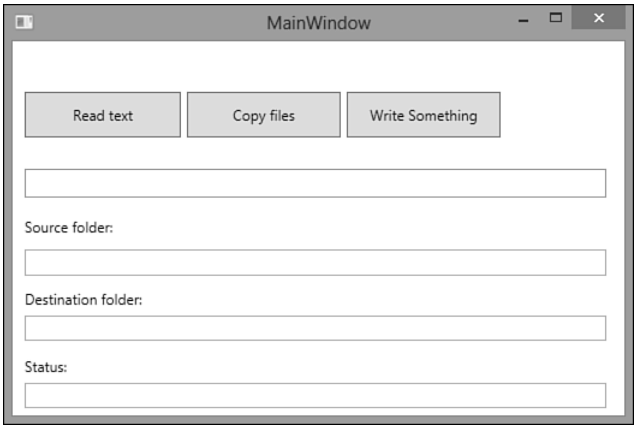


FIGURE 42.7 The user interface of the new sample application.

That said, add the following controls in the designer:

1. Three buttons, named `ReadTextButton`, `CopyButton`, and `WriteButton`. Then, set their `Content` properties with `Read text`, `Copy files`, and `Write Something`, respectively.
2. Four `TextBox` controls, named `ReadTextBox`, `SourceTextBox`, `DestinationTextBox`, and `StatusTextBox`.
3. Three `TextBlock` controls. You do not need to specify a name, but make sure their `Text` property is set with Source folder, Destination folder, and Status, respectively.

The first example uses the `StreamReader` class to read a text file asynchronously. The event handler for the `ReadTextButton.Click` event looks like this:

```
Private Async Sub ReadTextButton_Click(sender As Object,
                                       e As RoutedEventArgs) _
                                       Handles ReadTextButton.Click
    Using reader As New StreamReader("TextFile1.txt")
        Me.ReadTextBox.Text = Await reader.ReadToEndAsync
    End Using

End Sub
```

You mark the event handler with `Async`, and because this method will not be awaited by any other methods, it does not need to return a `Task`. Therefore, it can be defined as a `Sub`. Notice how you use `Await` together with the `ReadToEndAsync` method, while the rest of the implementation is made the usual way. The next example is about copying streams asynchronously. The following code shows the implementation of the `CopyButton.Click` event handler:

```
Private Async Sub CopyButton_Click(sender As Object,
                                   e As RoutedEventArgs) _
                                   Handles CopyButton.Click
    If Me.SourceTextBox.Text = "" Then
        MessageBox.Show("Please specify the source folder")
        Exit Sub
    End If

    If Me.DestinationTextBox.Text = "" Then
        MessageBox.Show("Please specify the target folder")
        Exit Sub
    End If

    For Each fileName As String In Directory.
        EnumerateFiles(Me.SourceTextBox.Text)
        Using SourceStream As FileStream = File.Open(fileName, FileMode.Open)
            Using DestinationStream As FileStream =
                File.Create(String.Concat(Me.DestinationTextBox.Text,
                                          fileName.
                                          Substring(fileName.LastIndexOf("\"c))))
                Await SourceStream.CopyToAsync(DestinationStream)
                Me.StatusTextBox.Text = "Copied " + DestinationStream.Name
            End Using
        End Using
    Next

End Sub
```

In particular, the code enumerates the content of the source folder and for each file it opens a stream for reading and another one for writing into the target folder. `Await` enables you to execute asynchronously the operation with the asynchronous method called `CopyToAsync`. It is worth mentioning that, with this approach, you can refresh the user interface with useful information, like showing the name of the last copied file. In a synchronous approach, this would not be possible because the UI would be blocked until the completion of the operation. The last example demonstrates how to write some text into a file asynchronously. This is the event handler for the `WriteButton.Click` event:

```
Private Async Sub WriteButton_Click(sender As Object,
                              e As RoutedEventArgs) Handles WriteButton.Click
    Dim uniencoding As UnicodeEncoding = New UnicodeEncoding()
    Dim filename As String =
        "c:\temp\AsyncWriteDemo.txt"

    Dim result As Byte() = uniencoding.GetBytes("Demo for Async I/O")

    Using SourceStream As FileStream = File.Open(filename, FileMode.OpenOrCreate)
        SourceStream.Seek(0, SeekOrigin.End)
        Await SourceStream.WriteAsync(result, 0, result.Length)
    End Using

End Sub
```

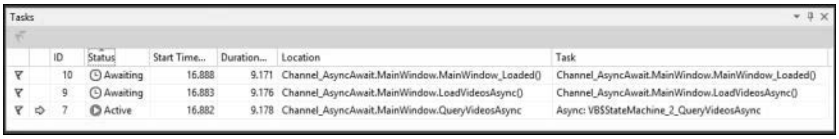
In this particular example, the code is written exactly like in the synchronous counterpart, except for the `Async` modifier and the `Await` statement that invokes the `WriteAsync` method (instead of running `Write`). By using the `Async` pattern, writing applications that work with streams and remain responsive has become dramatically simpler.

IMPLEMENTING CUSTOM AWAITERS

By using the Task-based asynchrony, you can work with instances of the `Task` class and use `Await` when waiting for their results. Using `Task.Run` and combinators will usually avoid the need to create custom types that can be used along with `Await`. However, in some situations you might want to define your own awaitable types. Reasons for making this can be various for example, performing some work over a control in the user interface from within a background thread. To accomplish this, you define a `Structure` or `Class` that implements the `INotifyCompletion` interface to expose a method called `OnCompleted`; then you need a method called `GetAwaiter`. The MSDN documentation does not provide enough information about building custom types that can be used with Await, but fortunately a blog post series by Lucian Wischik from Microsoft shows how to create one. You can find it at http://blogs.msdn.com/b/lucian/archive/2012/11/28/how-to-await-a-storyboard-and-other-things.aspx. There is also another blog post from the Stephen Toub, in which he discusses how to implement a custom awaitable type to perform asynchronous operations with sockets; you can find it at http://blogs.msdn.com/b/pfxteam/archive/2011/12/15/10248293.aspx.

Debugging Tasks

Visual Studio 2015 allows you to collect information about asynchronous tasks with the Tasks window. This is not a new tool; it has been available since Visual Studio 2012, but only for the Task Parallel Library. It now provides support for the `Async`/`Await` pattern in Visual Studio 2013 and requires Windows 8 or higher. To understand how it works, place a breakpoint on the `QueryVideosAsync` method in the sample WPF application and press **F5**. When the breakpoint is hit, select **Debug**, **Windows**, **Tasks** and press **F11** to step into the code. As you step into asynchronous method calls, the Tasks window shows information on each task, as shown in Figure 42.8.



Tasks

		ID	Status	Start Time...	Duration...	Location	Task
		10	Awaiting	16.888	9.171	Channel_AsyncAwait.MainWindow.MainWindow_Loaded()	Channel_AsyncAwait.MainWindow.MainWindow_Loaded()
		9	Awaiting	16.883	9.176	Channel_AsyncAwait.MainWindow.LoadVideosAsync()	Channel_AsyncAwait.MainWindow.LoadVideosAsync()
	➪	7	Active	16.882	9.178	Channel_AsyncAwait.MainWindow.QueryVideosAsync	Async: VB$StateMachine_2_QueryVideosAsync

FIGURE 42.8 The Tasks window.

As you can see, the Tasks window shows what task is active and what other tasks are awaiting the completion of the active task. It shows the duration, the method that started the asynchronous task (in the Location column), and the actual running task, such as a state machine (in the Task column). If you hover over the values in the ID column, you will also get additional information about the task execution order. This very useful tool helps you understand the execution flow of your asynchronous code.

Summary

Building applications that remain responsive whatever task they are executing is something that developers must take care of, especially from the user interface perspective. This chapter explained how to use asynchrony to build responsive applications by first discussing old-style programming models such as the Event-based Asynchronous Pattern and the Asynchronous Programming Model. Both provide techniques to write asynchronous code that runs on separate threads. However, both have some limitations, such as code complexity, issues in returning information to caller threads or functions, and managing errors effectively. Visual Basic 2015 offers the asynchronous pattern based on the `Async` and `Await` keywords, which enable you to keep the UI thread free and write code that is similar to the synchronous approach and is much easier to read and maintain. You invoke an asynchronous task, and then its result is returned some time later, but the control is immediately returned to the caller. When the result is available, an implicit callback enables you to consume the result of the asynchronous operation effectively. The `Async` pattern relies on the concept of task and on the `Task` class, which means that asynchrony easily includes support for cancellation, progress, anonymous delegates, and concurrent operations. The .NET Framework 4.6 itself exposes built-in classes that use the new `Async` pattern for making asynchronous I/O operations much easier with particular regard to streams and network requests. The `Async` pattern and the `Async`/`Await` keywords can be used across multiple platforms and presentation technologies.

Index

Symbols & Numerics

<c> tag, 1253-1254

<param> tag, 1255

' (comment symbol), 72

1-click deployment, 940-941

A

ABC (Address, Binding, Contract), 1014-1015
- autogenerated contracts, 1018-1019
- IService1.vb file, 1017-1018

abstract classes, inheritance, 383

abstraction, 646

accessing
- base class members, 383-387
  - MyBase keyword, 383-385
  - MyClass keyword, 385-387
- ClickOnce, programmatic access, 1301-1302
- directories, 515-518
- Microsoft Azure Management Portal, 954-955
- properties, 275
- recently used templates, 23-24

actions, 207-208

adaptive user interfaces, 1007

AddHandler keyword, 431-432

adding references to COM libraries, 94-95

Address property (WCF), 1014-1015

ADO.NET, 599-603
- connecting to databases, 601-603
  - deleting data, 602-603
  - insert operations, 601-602

query operations, 603
updating data, 602
connection modes, 601
data providers, 600-601
DataSets, 603-608
creating, 604-608
serialization, 1091-1092
ADO.NET Entity Framework. *See* EF (Entity Framework)
advanced compiling options, 55-58
conditional compilation, 57
constants, 57-58
extension methods, custom extension methods, 581-583
generating serialization assemblies, 58
optimizations, 56-57
advanced language features
anonymous types, 585-586
array literals, 575-577
extension methods, 577-584
exporting, 583-584
generic variance, 596-598
contra variance, 597-598
covariance, 596-597
lambda expressions, 587-595
debugging, 594-595
lexical closures, 593-594
multiline lambdas, 591
Sub lambdas, 591-593
type inference, 590
local type inference, 571-575
relaxed delegates, 586-587
advantages of generics, 413-414
aggregation operators, 628-629
aliasing, 77
analyzers, building, 1315-1336
animations, 840-848
ColorAnimation, 844-845
creating, 847-848
DoubleAnimation, 841-843
events, 845-847
annotating documents, 862-864
anonymous iterators, 469-470
anonymous types, 585-586
APIs
for ASP.NET, 910
Compiler APIs, 1308
Fluent APIs, 724-726
Scripting APIs, 1309
in UWP, 975-976
Windows APIs
P/Invokes, 1237-1238
P/Invokes, encapsulating, 1239-1240
APM (Asynchronous Programming Model), 1142-1143
Application class
members, 773
Startup and Exit events, handling, 774
application domains, 1179-1181
creating, 1179-1181
sandboxing, 1186-1188
application events, 569-570
Application state, 920
Application tab settings (My Project window), 43-48
Application Type field (My Project window), 44
Application.myapp file, 83-85
applications
debugging, 58-65, 199-200
breakpoints, 60-62, 204-208
debug attributes, 226-230
Edit and Continue feature, 64-65
exceptions, 231-232
"Just My Code" debugging, 202-203
light bulb icon, 249-254
in managed code, 216-230

Microsoft Azure Debugging Environment, 968
Mixed Mode debugging, 202
PerfTips, 208-209
preparing an example, 199-200
Run to Cursor command, 201
runtime errors, 62-64
Set Next Statement command, 201
Show Next Statement command, 202
Step Into command, 201
Step Over command, 201
visualizers, 215-216
warnings, 247-249
deploying
with ClickOnce, 1289-1294
with InstallShield LE, 1272-1283
to Microsoft Azure, 964-967
with Windows Installer, 1270-1271
execution environment, 3
localizing, 901-902, 906-908
MAT, 902
settings, 90-92
updating with ClickOnce, 1296-1297
WPF, creating, 749-751
Canvas panel, 767-768
DockPanel panel, 768
Grid panel, 762-764
Logical Tree, 757-758
StackPanel panel, 765-766
ViewBox panel, 769
Visual Tree, 758
WrapPanel panel, 766-767
XAML, 751-757
applying
animations
ColorAnimation, 844-845
DoubleAnimation, 841-843
attributes, 1217-1220
arguments, 1218
assembly-level, 1219
CLS, 307
custom attributes, 1221-1223
inheritance, 370-375
styles, 828-833
inheritance, 831
triggers, 831-833
apps, creating universal Windows apps, 977-998
app bar, 987-994
app manifest, 998-1001
defining a data template, 986-987
implementing a data model, 980-981
Internet connection, 981-986
navigation, 987-994
playing media, 997-998
project templates, 978-980
raising Toast notifications, 996-997
sharing content, 995-996
architecture
of collections, 442
of LINQ, 613-614
of .NET Core 5, 11-12
of .NET Framework 4.6, 6
of WPF, 747-749
DirectX libraries, 747
PresentationFramework layer, 748
arguments, 73, 280-285
for attributes, 1218
CObj conversion function, 1223-1224
optional arguments, 283-284
optional nullable arguments, 284-285
ParamArray arguments, 282-283
arithmetic operators, 172-174
ArrayList collection, 442-445

arrays, 165-167, 311
  jagged arrays, 169, 576-577
  literals, 575-577
  multidimensional arrays, 168-169
  ParamArray arguments, 282-283
  passing as arguments, 282
  ReDim keyword, 168
  System.Array class, 169-172
ASP.NET applications
  controls, 917-919
    HTML controls, 918-919
    server controls, 917-918
  creating for the cloud, 961-964
  creating with Visual Studio
    data controls, adding, 929-932
    data model, adding, 927
    filtering capabilities, adding, 932-933
    forcing login for security, 937-939
    master pages, 924-927
    model binding, 935-937
    navigation controls, adding, 933
    new web form, adding, 927-929
    strongly typed data controls, 934-935
  developing, 910-914
    event handling, 919-920
    page requests, 911
    platforms, 910
    web forms, 914-916
  project templates, 912-914
  scalability, 911-912
  state management, 920-923
    Application state, 920
    Cache state, 921-922
    Context state, 922
    cookies, 922
    Session state, 922
    ViewState state, 923
assemblies, 1177-1179
  application domains, 1179-1181
    creating, 1179-1181
    sandboxing, 1186-1188
  Async pattern, 1158-1160
  binding, 1178
  CAS, 1182
  deploying, XCopy deployment, 1262
  Event-based Asynchrony, Async pattern, 1155-1157
  executing, 1179-1181
  GAC, 1263-1267
  information stored in, 1178
  installing and uninstalling, 1264, 1267
  loading, 1194
  location of, 1178
  marking as CLS compliant, 307-308
  metadata, 1179, 1192-1194, 1217
    inspecting, 1195-1196
    retrieving, 1194-1196
  MsCorlib.dll, 10
  in .NET Compiler Platform, 1309
  in .NET Framework 4.6, 8-9
  PIAs, 95
    Deploy Without PIAs feature, 95-97
  private assemblies, 1178
  probing, 1178
  references, 92-97
  reflection, 1191-1192
    Caller Information, 1212-1215
    late binding, 1212
    System.Reflection.Emit namespace, 1206-1212
  resources, 86-90
  serialization assemblies, generating, 58
  shared assemblies, 1178
  signing with strong names, 1179, 1265-1267
  types, reflecting, 1196-1204

Assembly Information button (My Project window), 44-46
Assembly Name field (My Project window), 43
AssemblyInfo.vb file, 85-86
assembly-level attributes, 1219
assigning structures to variables, 354
assignment operators, 174-175
associations, 646
Async pattern, 1143-1151, 1155-1160
  documentation, 1160-1161
  exception handling, 1161
  implementing cancellation, 1165-1168
  reporting progress, 1168-1170
  when to use, 1147
  where to use, 1146
asynchronous I/O file operations, 1171-1175
asynchronous lambda expressions, 1170-1171
asynchronous programming, 1139-1140
  APM, 1142-1143
  Async pattern, 1143-1151, 1155-1157
    asynchronous lambda expressions, 1170-1171
    documentation, 1160-1161
    reporting progress, 1168-1170
    when to use, 1147
    where to use, 1146
  EAP, 1141-1142
  Event-Based Asynchrony, 1140
    callbacks, 1151-1155
  Task-Based Asynchrony, 1161-1165
    combinators, 1163-1165
    debugging tasks, 1175
    switching threads, 1162-1163
attributes, 78, 1217
  applying, 1217-1220
  arguments, 1218
    CObj conversion function, 1223-1224
  assembly-level, 1219
  conditional APTCA, 1186-1188
  custom attributes, 1220-1225
    applying, 1221-1223
    writing, 1220-1221
  debug attributes, 226-230
  DllImport attribute, performing P/Invokes, 1237-1238
  inheritance, 1224-1225
  .NET Framework 4.6 security attributes, 1185
  parameters, applicable data types, 1221
  reflecting, 1225-1226
  StructLayout, converting types to unmanaged, 1241-1243
  VBFixedString, converting types to unmanaged, 1243
audio files, playing, 545
auto-completing code typing, 30-32
auto-implemented read-only properties, 73, 271-272
Autos window, 215
avoiding namespace ambiguities, 340-341
Await keyword, 1145, 1158-1160
  custom awaiters, 1175
Azure platform
  cloud storage, 952-953
  services offered by Microsoft Azure, 952

B

background compiler, 52
Background GC, 324
BAML (Binary Application Markup Language), 906
Barrier class, 1115-1116
Base Class Library, 5

base classes, 369
  members, accessing, 383-387
    MyBase keyword, 383-385
    MyClass keyword, 385-387
BCL (Base Class Library), 9-10
  GAC, 1263-1267
  namespaces, 328
BigInteger value type, 112-113
binary files, reading and writing, 525-526
binary serialization
  deep copy, 1076
  exception handling, 1075
binding, 1178
  built-in bindings (WCF), 1028-1029
  late binding, 1212
Binding markup extension, 870-871
Binding property (WCF), 1014-1015
bit flags, 366
BitArray collection, 449-450
BitmapCacheBrush, 826-828
BitVector32 collection, 450-451
bitwise operators, 177-179
Blend, 833
Blob storage, 953
BlockingCollection(of T) collection, 1127-1128
Border control, 781-782
boxing, 123-124
breaking method execution, 289-290
breakpoints, 60-62, 204-208
  actions, 207-208
  conditions, 205-207
Browser Applications (WPF), 774-777
browsing online code samples, 25-26
brushes, 812-828
  BitmapCacheBrush, 826-828
  CaretBrush, 822-823
  DrawingBrush, 825
  ImageBrush, 818-821
  LinearGradientBrush, 815-816
  RadialGradientBrush, 817
  SelectionBrush, 822-823
  SolidColorBrush, 813-814
  VisualBrush, 823-825
bubbling strategy, 761
building
  analyzers, 1315-1336
  code refactoring, 1336-1345
  complex queries with anonymous types, 681-682
  custom collections, 462-463
  custom exceptions, 390-392
  custom value types, 113
built-in bindings (WCF), 1028-1029
built-in extension methods, 579-580
built-in trace listeners, 219
built-in types, inspecting, 67-68
Button control, 782
ByRef keyword, 281
ByVal keyword, 280

C

Cache state, 921-922
cached events, 915-916
Calendar control, 782-783
Call Hierarchy window (Visual Studio 2015), 69-70
Call Stack window, 210-212
callbacks, 1151-1155
Caller Information, 1191, 1212-1215
canceling tasks, 1113-1114
cancellation, implementing in Async, 1165-1168
Canvas panel, 767-768
CaretBrush, 822-823

CAS (Code Access Security), 1182
catching exceptions, 240-241, 246-247
  from wrapped objects, 1233
CheckBox control, 783
CIL (Common Intermediate Language), 8-9
Class Designer, 473-484
  designing objects, 475-479
  diagrams
    creating, 482-483
    exporting, 483-484
  enabling, 474-475
  implementing derived classes, 479-482
Class View window (Visual Studio), 484
classes, 72, 267, 310. *See also* collections
  abstract classes, inheritance, 383
  Application class, 772-774
    members, 773
    Startup and Exit events, handling, 774
  attributes, 78
  Barrier class, 1115-1116
  base classes, 369
    members, accessing, 383-387
  comparing with modules, 349-350
  CultureInfo class, 902
  custom attributes, 1220-1225
    writing, 1220-1221
  DataContext class, instantiating, 658
  DbContext class, 697-698
  Debug class, 216-219
    methods, 216
    output, 217-219
  declaring, 267-268
  derived classes, 369
  fields, 269-270
  inheritance, 74-75
    attribute inheritance, 1224-1225
    constraints, 420-421
    custom exceptions, building, 390-392
    interfaces, 400-402
    MustInherit keyword, 382-383
    MustOverride keyword, 382-383
    NotInheritable keyword, 381-382
    overriding, 375
    polymorphism, 375-376
    shadowing, 388-389
  instantiating, 76
  iterator classes, implementing, 470-472
  libraries, 268
  LINQ to SQL, 647-658
    .dbml files, 652
    declaring, 658
    inheritance, 651
    System.Data.Linq.DataContext class, 650
  members, invoking, 76
  MessageBox, 340-341
  Monitor class, 1103
  nested classes, 268-269
  Parallel class, 1106
  ParallelEnumerable class, 1133-1134
  ParallelLoopState class, 1122
  ParallelOptions class, 1107
  partial classes, 290-293
  portable classes, creating, 491-494
  shared classes, 302
  System.Array class, 169-172
  System.Console class, 29
  System.Data.Linq.DataContext class, 650
  System.Diagnostics.Process class, 1096
  System.Enum, 362-365
  System.GC, 323-324
  System.IO.Directory class, 515-518
  System.IO.DirectoryInfo class, 518
  System.IO.DriveInfo class, 519

System.IO.File class, 520-521
System.IO.FileInfo class, 522-523
System.IO.Path class, 514-515
System.IO.Stream class, 524
System.Object, 373-375
Task class, 1109-1110
TaskFactory class, 1107
TaskScheduler class, 1107
Trace class, 219-225
variables, local variables, 270
classic publishing, 941-943
ClickOnce, 1287
application files, configuring, 1295
applications
deploying, 1289-1294
handling, 1288
updating, 1296-1297
configuring, 1294-1299
options, configuring, 1297-1299
prerequisites, 1296
programmatically accessing, 1301-1302
Registration-Free COM, 1302-1304
security, configuring, 1299-1300
when to use, 1288-1289
cloning reference types, 126
closing processes, 1097
cloud computing
application templates, 961-962
Microsoft Azure, 951
ASP.NET applications, creating, 961-964
Cloud Storage Studio, 972
Management Portal, 954-955
registering for, 953
roles, 967-970
SDK, downloading, 954
services offered, 952
SQL databases, creating, 955-961
storage, 952-953
WCF services, hosting, 1031-1032
WebJobs, 970
Cloud Storage Studio, 972
CLR (Common Language Runtime), 6-9
POCO, 1091
writing unmanaged code, 7-8
CLS (Common Language Specification), 306-311
abstract classes, 383
applying, 307
arrays, 311
classes, 310
enumerations, 366-367
interfaces, 402
marking assemblies as compliant, 307-308
methods, 310
namespaces, 342
naming conventions, 308-309
properties, 310
structures, rules for, 360
CObj conversion function, 1223-1224
code. *See also* live code analysis
breakpoints, 60-62
CAS, 1182
comments, 72
generating, 1210-1211
implicit line continuation, 78-80
invoking dynamically, 1204-1206
managed code, debugging, 216-230
organizing with partial classes, 290-293
redundant code, organizing, 263-266
refactoring, 254-266
Encapsulate Field quick action, 261-262
Extract Interface quick action, 254-256
Extract Method refactoring, 256-257
Inline Rename, 257-259

Introduce Constant refactoring, 262-263
Introduce Local Variable refactoring, 260-261
Introduce Temporary Variable refactoring, 259-260
sample code
browsing online code samples, 25-26
downloading, 2
unmanaged code, 1237
code editor, 28-35
IntelliSense technology, 30-32
live static code analysis, 250-254
Navigate To feature, 34-35
Peek Definition, 32-33
scroll bar, 33-34
touch improvements, 32
XML literals, 735-737
embedded expressions, 740-743
iterators, 743
LINQ queries, 738-740
local type inference, 736-737
null-propagating operator, 740
zooming in/out, 30
Code First approach for EF, 716-726
coding your model, 717-719
data annotations, 722-724
executing data operations, 719-722
Fluent APIs, 724-726
Code First Migration tool, 1044
Code Metrics, 1349-1351
Code Snippet Editor, 1364
code-continuation arrows, 2
Codelens, 35
CodeRush Xpress, 1364
coercion, 287
Collect method, 323
collection initializers, 453-454
collections, 441
architecture, 442
concurrent collections, 463, 1124-1128
BlockingCollection(of T), 1127-1128
ConcurrentBag(of T), 1124-1125
ConcurrentDictionary(TKey, TValue), 1126-1127
ConcurrentQueue(of T), 1125
ConcurrentStack(of T), 1126
custom collections, building, 462-463
generic collections, 451-461
Dictionary(of TKey, TValue) collection, 455-456
LinkedList(of T) collection, 458-460
List(of T) collection, 451-453
ObservableCollection(of T) collection, 456-458
Queue(of T) collection, 460
ReadOnlyCollection(of T) collection, 454-455
ReadOnlyObservableCollection(of T) collection, 458
serializing, 1087
SortedDictionary(of TKey, TValue) collection, 456
Stack(of T) collection, 460
immutable collections, 463-464
iterators, 464-472
anonymous iterators, 469-470
benefits of, 465-466
exiting from, 468
iterator classes, implementing, 470-472
scope of, 465
simple iterators, 467-468
Try..Catch..Finally blocks, 469
nongeneric collections, 442-451
ArrayList collection, 442-445
BitArray collection, 449-450
BitVector32 collection, 450-451

HashTable collection, 446-447
HybridDictionary collection, 448
ListDictionary collection, 447
NameValueCollection collection, 449
OrderedDictionary collection, 447-448
Queue collection, 445-446
SortedList collection, 448
Stack collection, 446
StringCollection collection, 448
StringDictionary collection, 448-449
null-conditional operator, 461-462
ColorAnimation, 844-845
COM (Component Object Model) architecture, 1229
calling objects from WPF, 1234-1237
catching exceptions, 1233
importing components into Visual Studio, 1230-1232
libraries
Dll hell, 1261-1262
references, adding, 94-95
releasing objects, 1233-1234
using in .NET code, 1233
combinators, 1163-1165
combining
delegates, 430-431
multiple constraints, 420-421
ComboBox control, 784-785
Command window, 210
comments, 72
XML comments, 1245-1246
<c> tag, 1253-1254
<param> tag, 1255
compiled help files, generating, 1258
complex documentation, 1251
documenting permissions requirements, 1257
enabling, 1247
generics, 1258
implementing, 1248-1258
links to other references, specifying, 1257-1258
lists, creating, 1256-1257
MSDN recommended tags, 1250-1251
referring to external documentation file, 1256
scope of, 1250
Common Type System, 99-103
boxing, 123-124
reference types, 113-116
unboxing, 124-125
value types, 103-113
analyzing content of, 110-111
applying, 106-108
assigning, 108-110
BigInteger, 112-113
building custom value types, 113
naming conventions, 104
comparing
classes and modules, 349-350
delegates and function pointers, 427
.NET Framework 4.6 and 4.5, 7
reference and value types, 116-120
strings, 136-138
comparison operators, 182-184
compiled help files, generating, 1258
Compiler APIs, 1308
compiling projects, 13-15, 48
advanced compiling options
conditional compilation, 57
constants, 57-58
generating serialization assemblies, 58
advanced options, 55-58
optimizations, 56-57
background compiler, 52

configurations
creating, 50-52
warning configurations, 53-55
Debug configuration, 49
JIT compiler, 9
light bulb icon, 249-254
.NET Compiler Platform, 13-14
.NET Native compiler, 14-15
options, 52-53
target CPU, specifying, 53
complex documentation, 1251
complex queries, building with anonymous types, 681-682
components
of .NET 2015, 3
of .NET Core 5, 11
compressing data
with streams, 527-532
Zip archives, 532-534
concatenating strings, 149-150
concatenation operators, 182, 641
Conceptual Scheme Definition Language, 693-694
concurrent collections, 463, 1124-1128
BlockingCollection(of T), 1127-1128
ConcurrentBag(of T), 1124-1125
ConcurrentDictionary(TKey, TValue), 1126-1127
ConcurrentQueue(of T), 1125
ConcurrentStack(of T), 1126
ConcurrentBag(of T) collection, 1124-1125
ConcurrentDictionary(TKey, TValue) collection, 1126-1127
ConcurrentQueue(of T) collection, 1125
ConcurrentStack(of T) collection, 1126
conditional APTCA, 1186-1188
conditional code blocks, 193-196
conditional compilation, 57
conditioning inheritance, 380-383
conditions, 205-207
configurations
creating, 50-52
Debug configuration, 49
Release configuration, 49-50
warning configurations, 53-55
configuring
ClickOnce, 1294-1299
application files, 1295
options, 1297-1299
prerequisites, 1296
security, 1299-1300
WCF services, 1033
connecting to databases, 601-603
connection modes for ADO.NET, 601
constants, 196
compilation constants, 57-58
grouping with enumerations, 361-367
constraints, 419-421
inheritance constraints, 420-421
method constraints, 419
multiple constraints, combining, 420-421
type constraints, 419-420
constructors, 119-120, 295-301
inheritance, 387
invoking, 296
nested invocations, 299-300
object initializers, 300-301
overloading, 298-300
private constructors, 300
shared constructors, 306
XmlSerializer, 1081
consuming
functions, 1068-1070
generics, 416-417
OData services, 1060-1068
creating a client application, 1061-1065
CRUD operations, 1067-1068
querying data, 1065-1067

WCF services, 1024-1029
invoking service members, 1026-1029
proxy class, 1025-1026
service references, adding, 1024-1025
ContentControl element, 780-781
contra variance, 597-598
Contract property (WCF), 1014-1015
control templates, 833-836
controls, 779-780
ASP.NET, 917-919
server controls, 917-918
Border control, 781-782
Button control, 782
Calendar control, 782-783
CheckBox control, 783
ComboBox control, 784-785
ContentControl element, 780-781
DataGrid, 872-876
DataGrid control, 784-785
DatePicker control, 786
DocumentViewer control, 787
Expander control, 788
Frame control, 789
GroupBox control, 790
HTML controls, 918-919
Image control, 790
Label control, 790
ListBox control, 791
ListView control, 792-793
MediaElement control, 794
Menu control, 794-796
panels, 762-769
Canvas panel, 767-768
DockPanel panel, 768
Grid panel, 762-764
StackPanel panel, 765-766
ViewBox panel, 769
WrapPanel panel, 766-767
PasswordBox control, 796-797
ProgressBar control, 797-798
RadioButton control, 799
RichTextBox, 864-866
RichTextBox control, 800
ScrollBar control, 800
ScrollViewer control, 800-801
Separator control, 801
Slider control, 801-802
StatusBar control, 802
strongly typed data controls, 934-935
TabControl control, 803-804
TextBlock control, 804-805
TextBox control, 805
ToolBar control, 805-806
TreeView control, 806-807
VirtualizingStackPanel control, 767
WebBrowser control, 808
WindowsFormsHost control, 808-809
conversion functions, 131-132
conversion operators, 129-135, 630-632
CType, 133-135
DirectCast operator, 133-135
narrowing conversions, 130-131
widening conversions, 130
converting
between reference and value types
boxing, 123-124
deep copy, 127-128
GetType keyword, 128-129
implicit conversion, 121-123
shallow copy, 126-127
unboxing, 124-125
strings to dates, 155-156

types to unmanaged, 1240-1243
StructLayout attribute, 1241-1243
VBFixedString attribute, 1243
cookies, 922
copying strings, 143
CopyToDataTable method, 682-683
CoreCLR, 11
covariance, 596-597
CPU
parallel computing, 1105
debugging tools, 1122-1123
Parallel class, 1106
ParallelOptions class, 1107
TaskFactory class, 1107
TaskScheduler class, 1107
target CPU, specifying, 53
creating
animations, 847-848
application domains, 1179-1181
bindings, 872
configurations, 50-52
custom events, 437-439
custom exceptions, 390-392
DataSets, 604-608
dates, 153-155
diagrams in Class Designer, 482-483
EDMs, 688-702
generics, 414-415
master-details forms, 886-891
.msi packages, 1274-1283
adding installation files, 1277-1278
configuring Registry on target machine, 1280
creating shortcuts, 1279-1280
environment variables, 1283
installation requirements, 1275-1277
setting properties, 1283
specifying application information, 1274
specifying dialogs, 1282
OData endpoints, 1036-1063
adding a data model, 1038-1044
adding controllers, 1045-1046
CRUD operations, 1047-1052
enabling Code First Migration, 1044
handling relationships between entities, 1052-1054
implementing queries, 1046-1047
publishing the endpoints, 1054-1055
querying data via HTTP requests, 1056-1060
registering the OData service, 1044-1045
portable classes, 491-494
portable libraries, 494-503
projects, 21-22, 27-29
refactoring, 1336-1345
shared projects, 504-512
SQL databases for Microsoft Azure, 955-961
structures, 351
tabular data forms, 877-885
threads, 1098-1099
types, 485-491
universal Windows apps, 977-998
app bar, 987-994
app manifest, 998-1001
defining a data template, 986-987
implementing a data model, 980-981
Internet connection, 981-986
navigation, 987-994
playing media, 997-998
project templates, 978-980
raising Toast notifications, 996-997
sharing content, 995-996

WPF applications, 745-746, 749-751
Canvas panel, 767-768
DockPanel panel, 768
Grid panel, 762-764
Logical Tree, 757-758
StackPanel panel, 765-766
ViewBox panel, 769
Visual Tree, 758
WrapPanel panel, 766-767
XAML, 751-757
XML documents, 732
CType operator, 133-135
overloading, 359-360
CultureInfo class, 902
cultures, 539-540
custom attributes, 1220-1225
applying, 1221-1223
arguments, CObj conversion function, 1223-1224
inheritance, 1224-1225
parameters, data types, 1221
writing, 1220-1221
custom awaiters, 1175
custom collections, building, 462-463
custom events, creating, 437-439
custom exceptions, building, 390-392
custom extension methods, 581-583
custom logic, implementing for WCF, 1019-1023
custom objects, serialization, 1078-1080
custom providers for LINQ, 613
custom serialization, 1083-1086
events, 1085-1086
custom types, exposing, 274-275
custom validations, 673-675

D

data annotations, 722-724
data binding, 869-876
Binding markup extension, 870-871
bindings, creating, 872
DataGrid control, 872-876
drag-and-drop data binding, 876-900
master-details forms, creating, 886-891
IValueConverter interface, implementing, 897-900
modes, 871-872
ObservableCollection(of T) collection, 872-876
string formatters, implementing, 895-897
views, 891-895
data providers, 600-601
databases. *See also* LINQ; SQL databases
connecting to, 601-603
insert operations, 601-602
LINQ, queries, 658-663
querying, 603
SQL databases for Microsoft Azure
creating, 955-961
migrating to, 958-959
updating data, 602
DataContext class, instantiating, 658
DataGrid control, 784-785, 872-876
DataSets, 603-608
creating, 604-608
LINQ to DataSets, 679-682
CopyToDataTable method, 682-683
Field(of T) method, 684
SetField(of T) method, 684
DatePicker control, 786
dates, 153
adding values to, 158-159
creating, 153-155
formatting, 156-158

DateTimeOffset structure, 159-160
DbContext class, 697-698
  instantiating, 703-704
.dbml files, 652
Debug class, 216-219
  methods, 216
  output, 217-219
Debug configuration, 49
DebuggerBrowsable attribute, 226-227
DebuggerDisplay attribute, 227-228
DebuggerStepperBoundary attribute, 226
DebuggerTypeProxy attribute, 228-230
debugging applications, 58-65, 199-200
  attributes, 226-230
  breakpoints, 60-62, 204-208
    actions, 207-208
    conditions, 205-207
  Edit and Continue feature, 64-65
  exceptions, 231-232
    catching, 240-241
    hierarchy, 236-237
    ignoring, 240-241
    System.Exception, 233
    Try..Catch..Finally blocks, 233-236
    When keyword, 245-246
  "Just My Code" debugging, 202-203
  light bulb icon, 249-254
  in managed code, 216-230
    Debug class, 216-219
    Trace class, 219-225
  Microsoft Azure Debugging Environment, 968
  Mixed Mode debugging, 202
  PerfTips, 208-209
  preparing an example, 199-200
  Run to Cursor command, 201
  runtime errors, 62-64
  Set Next Statement command, 201
  Show Next Statement command, 202
  Step Into command, 201
  Step Over command, 201
  universal Windows apps, 1002-1009
  visualizers, 215-216
  warnings, 247-249
declarative mode (XAML), 756-757
Declare keyword, performing P/Invokes, 1237-1238
declaring
  classes, 267-268
    LINQ to SQL, 658
  delegates, 428-430
  interface variables, 397
  objects, 432-433
  optional parameter overloads, 288
deep copy, 127-128, 1076
default imports, 338-339
default properties, 275-276
deferred execution in LINQ to Objects, 623-625
defining, interfaces, 393-394
delegates, 427
  comparing with function pointers, 427
  declaring, 428-430
  generics support, 428
  multicast delegates, 430-431
  relaxed delegates, 586-587
  syntax, 428-429
deleting
  directories, 517
  entities, 705-706
  entities with LINQ to SQL, 667-668
Deploy Without PIAs feature, 95-97
deploying
  applications
    with ClickOnce, 1289-1294
    with InstallShield LE, 1272-1283
    to Microsoft Azure, 964-967
    with Windows Installer, 1270-1272
  assemblies, XCopy deployment, 1262

web applications, 927-949
1-click deployment, 940-941
classic publishing, 941-943
MSDeploy, 943-946
XBAP applications, 777
derived classes, 369, 479-482
deserialization, 1073
binary deserialization, 1074
XAML content, 1088
XML deserialization, 1082
designer tools, 751
Class Designer, 473-484
designing objects, 475-479
diagrams, creating, 482-483
enabling, 474-475
implementing derived classes, 479-482
EDM designer tool windows, 698-702
Desktop mode (Windows 10), 1008-1009
destroying object instances, 313-314
destructors
Dispose method, 317-323
Finalize method, 315-316
implementing, 319-322
developing ASP.NET web applications, 910-914
controls, 917-919
HTML controls, 918-919
server controls, 917-918
event handling, 919-920
page requests, 911
platforms, 910
project templates, 912-914
scalability, 911-912
state management, 920-923
Application state, 920
Cache state, 921-922
Context state, 922
cookies, 922
Session state, 922
ViewState state, 923
web forms, 914-916
Diagnostic Tools, 1351-1354
diagrams, exporting, 483-484
dialogs, 809-810
Dictionary(of TKey, TValue) collection, 455-456
direct routing strategy, 761
DirectCast operator, 133-135
directives
Imports directives, 76-77
default imports, 338-339
namespace alias, 336-338
organizing, 263-266, 339
Option Infer directive, 573-574
Region directives, 77-78
directories
accessing, 515-518
deleting, 517
exception handling, 519
manipulating, System.IO.Path class, 514-515
DirectX libraries, 747
Discover What's New section (Visual Studio 2015), 20
Dispose method, 317-323
Dll hell, 1261-1262
DllImport attribute, performing P/Invokes, 1237-1238
docking, tool windows, 36
DockPanel panel, 768
documentation. *See also* documents
for Async pattern, 1160-1161
compiled help files, generating, 1258
Microsoft Azure Developer Center, 953
MSDN Library, 65-66
Object Browser window, 66-67
online help, 65-66

for Windows API, 1244
XML comments
lists, creating, 1256-1257
referring to external documentation file, 1256
documents
annotating, 862-864
highlighting, 862-864
manipulating, 857-866
XPS documents, viewing, 866-867
DocumentViewer control, 787
DoubleAnimation, 841-843
downloading
Cloud Storage Studio, 972
InstallShield LE, 1271-1272
Microsoft Azure SDK, 954
source code for this book, 2
drag-and-drop data binding, 876-900
IValueConverter interface, implementing, 897-900
master-details forms, creating, 886-891
string formatters, implementing, 895-897
tabular data forms, creating, 877-885
views, 891-895
DrawingBrush, 825
duplicate or overlapping catch blocks warning, 55
dynamic code, executing, 1204-1206

E

EAP (Event-Based Asynchronous Pattern), 1141-1142
Edit and Continue feature, 64-65
editing
default values in thread pool, 1101
strings, 145-148
EDMs (Entity Data Models), 688, 1023, 1036
creating, 688-702
DbContext class, 697-698
designer tool windows, 698-702
entities
adding, 704-705
deleting, 705-706
updating, 706-707
foreign key columns support, 691
object context, 697
optimistic concurrency, handling, 707-708
querying
with Entity SQL, 712
with LINQ to Entities, 710-712
schemas
Conceptual Scheme Definition Language, 693-694
Mapping Definition Language, 696
Store Schema Definition Language, 694-696
validating data, 708-710
EF (Entity Framework), 687-688
Code First approach, 716-726
coding your model, 717-719
data annotations, 722-724
executing data operations, 719-722
Fluent APIs, 724-726
EF 7, 689
mapping stored procedures, 712-716
Model First approach, 716
elements operators, 641-642
Ellipse shape, 787-788
embedded expressions, 740-743
empty strings, checking for, 138
enabling
Class Designer, 474-475
XML comments, 1247
Encapsulate Field quick action, 261-262
encapsulating P/Invokes, 1239-1240

endpoints
  OData, creating, 1036-1063
    adding a data model, 1038-1044
    adding controllers, 1045-1046
    CRUD operations, 1047-1052
    enabling Code First Migration, 1044
    handling relationships between entities, 1052-1054
    implementing queries, 1046-1047
    publishing the endpoints, 1054-1055
    querying data via HTTP requests, 1056-1060
    registering the OData service, 1044-1045
  WCF, 1014
entities, 646
  ADO.NET Entity Framework. *See* EF (Entity Framework)
  DbContext class, instantiating, 703-704
  deleting, 667-668
  insert operations, 663-666
  updating, 666-667
  using SQL syntax, 676
Entity SQL, querying EDMs, 712
enumerations
  as bit flags, 366
  grouping constants, 361-367
  as return values from methods, 365-366
environment variables, specifying for .msi packages, 1283
equality operators, 640
Error List window, 39-40
errors
  exceptions, 231-232
    for binary serialization, handling, 1075
    catching, 240-241, 246-247
    catching from wrapped objects, 1233
    custom exceptions, building, 390-392
    for directories and pathnames, handling, 519
    for files, handling, 523
    hierarchy, 236-237
    ignoring, 240-241
    from P/Invokes, handling, 1243-1244
    System. Exception, 233
    task exceptions, 1111-1113
    throwing, 242-245
    Try..Catch..Finally blocks, 233-236
    for WCF, handling, 1030-1031
    When keyword, 245-246
  fixing, 250-254
  warnings, 247-249
Event-based Asynchrony, 1140-1142
  Async pattern, 1155-1160
    documentation, 1160-1161
  callbacks, 1151-1155
events, 427, 431-433
  animation events, 845-847
  application events, 569-570
  for ASP.NET applications, 919-920
  cached events, 915-916
  custom events, creating, 437-439
  for custom serialization, 1085-1086
  GC, subscribing to, 324-325
  My.Settings, 555-556
  nonserialized events, 1080
  passing information, 435-437
  postback events, 915-916
  raising, 434-437
  registering for, 431-432
  routed events, 760-761
  WithEvents keyword, 432-433
  WPF events, 759-761
examples of LINQ, 611-612
exceptions, 231-232
  for Async pattern, 1161
  for binary serialization, handling, 1075

catching, 240-241, 246-247
from wrapped objects, 1233
custom exceptions, building, 390-392
for directories and pathnames, handling, 519
Exit Try statements, 241
for files, handling, 523
hierarchy, 236-237
ignoring, 240-241
from P/Invokes, handling, 1243-1244
PLINQ exceptions, handling, 1136-1137
System.Exception, 233
properties, 237-240
task exceptions, 1111-1113
throwing, 242-245
Try..Catch..Finally blocks, 233-236
nesting, 241
for WCF, handling, 1030-1031
When keyword, 245-246
executing
assemblies, 1179-1181
dynamic code, 1204-1206
execution environment, 3
Exit Try statements, 241
exiting
from iterators, 468
from methods, 289-290
Expander control, 788
exporting
diagrams from Class Designer, 483-484
extension methods, 583-584
exposing custom types, 274-275
extending
My namespace, 562-566
My.Application, 564-566
My.Resources, 566
My.Settings, 566
extension methods, 344, 577-584
customizing, 581-583
exporting, 583-584
external processes, launching, 1096
Extract Interface quick action, 254-256
Extract Method refactoring, 256-257

F

Featured Videos section (Visual Studio 2015), 20
Fiddler, 1365
Field(of T) method, 684
fields
of classes, 269-270
read-only fields, 270
scope, 270
shared fields, 302
files
asynchronous I/O file operations, 1171-1175
code files
Application.myapp file, 83-85
AssemblyInfo.vb file, 85-86
Resources.resx file, 86-90
Settings.settings file, 90-92
.dbml files, 652
exception handling, 523
help files
compiled help files, generating, 1258
complex documentation, 1251
IService1.vb file, 1017-1018
manipulating
System.IO.File class, 520-521
System.IO.FileInfo class, 522-523
media files, playing, 853-856
permissions, 523, 1183-1184

prerequisites, 1296
text files, reading and writing, 524-525
Web.Config file, 1017
FileSystem property, 543-544
Finalize method, 315-316
finalizers, 120
finding projects, 29
fixing errors, 250-254
Flexera, 1271
flow documents, 857-862
Fluent APIs, 724-726
folders
accessing, 515-518
deleting, 517
formatting
dates, 156-158
strings, 138-143
forms
master-details forms, creating, 886-891
tabular data forms, creating, 877-885
web forms, 914-916
cached and postback events, 915-916
page lifetime, 915
Frame control, 789
frameworks
.NET Core 5, 3, 10-12. *See also* .NET Core 5
architecture, 11-12
components, 11
CoreCLR, 11
open source layers, 12
.NET Framework 4.6, 5-10. *See also* .NET Framework 4.6
architecture, 6
assemblies, 8-9
BCL, 9-10
CLR, 7-9
comparing with .NET Framework 4.5, 7
ILDasm, 9
function/operator without return value warning, 54
functions
consuming, 1068-1070
conversion functions, 131-132
installing and uninstalling, 1264, 1267
signing with strong names, 1265-1267
FxCop live code analysis, 1345-1349

G

GAC (global assembly cache), 93, 1261-1267. *See also* assemblies
garbage collection, 314-315, 323-325
Background GC, 324
Finalize method, 315-316
forcing, 315
generations, 324-325
subscribing to events, 324-325
System.GC class, 323-324
Generate from Usage feature (Visual Studio), 485-491
generating complex objects, 489-491
generating shared members, 488
generating
code, 1210-1211
compiled help files, 1258
serialization assemblies, 58
shared members, 488
generation operators, 632-633
generations, 324-325
generic collections, 451-461
Dictionary(of TKey, TValue) collection, 455-456
LinkedList(of T) collection, 458-460
List(of T) collection, 451-453
ObservableCollection(of T) collection, 456-458

ObservableCollection(of T) collection, 872-876
Queue(of T) collection, 460
ReadOnlyCollection(of T) collection, 454-455
ReadOnlyObservableCollection(of T) collection, 458
serializing, 1087
SortedDictionary(of TKey, TValue) collection, 456
Stack(of T) collection, 460
generic variance, 596-598
contra variance, 597-598
covariance, 596-597
generics, 287-288, 413-414. *See also* constraints
advantages of, 413-414
consuming, 416-417
creating, 414-415
delegate support for, 428
generic methods, implementing, 418-419
nullable types, 422-425
GetValueOrDefault method, 423-424
syntax, 422
overloading type parameters, 421-422
in WCF, 1022-1023
XML comments, 1258
GetName method, 364
GetType keyword, 128-129
GetValueOrDefault method, 423-424
GetValues method, 363
Global keyword, 343
global namespaces, 342-345
goals of this book, 1
gradient brushes
LinearGradientBrush, 815-816
RadialGradientBrush, 817
graphical representation of inheritance, 370
Grid panel, 762-764
GroupBox control, 790
grouping constants with enumerations, 361-367
grouping operators, 634-637
GUIDs, 164-165

H

handling errors
exceptions
hierarchy, 236-237
ignoring, 240-241
System.Exception, 233
throwing, 242-245
Try..Catch..Finally blocks, 233-236
warnings, 247-249
HashTable collection, 446-447
help page (Visual Studio 2015), 65-66
highlighting documents, 862-864
historical debugger, 1354-1359
hosting WCF services, 1031-1032
HTML controls, 918-919
HybridDictionary collection, 448

I

IComparable interface, 405-407
Icon field (My Project window), 44
icons, light bulb, 249-254
IConvertible interface, 408-410
identifiers, 80
for namespaces, 342
IDisposable interface, 317-319
IEnumerable interface, 403-405
IFormattable interface, 410-412

ignoring
  exceptions, 240-241
  warning messages, 54
IIS (Internet Information Services), hosting WCF services, 1031-1032
ILDasm (IL Disassembler), 9
Image control, 790
ImageBrush, 818-821
images, viewing, 851-853
immutable collections, 463-464
imperative mode (XAML), 756-757
implementing
  destructors, 319-322
  dynamic code, 1204-1206
  events, 434-437
  generic methods, 418-419
  interfaces, 394-399
    behaviors, 398
    IComparable, 405-407
    IConvertible, 408-410
    IEnumerable, 403-405
    IFormattable, 410-412
  iterator classes, 470-472
  permissions, 1183-1184
  WCF
    autogenerated contracts, 1018-1019
    custom logic, 1019-1023
  WCF services, 1015-1023
    project templates, 1015-1016
  XML comments, 1248-1258
implicit conversion, 121-123
implicit conversion warning, 54
implicit line continuation, 78-80
  LINQ, 612
implicit type warning, 54
importing COM components into Visual Studio, 1230-1232
Imports directives, 76-77
  default imports, 338-339
  namespace alias, 336-338
  organizing, 263-266, 339
information messages, 39
inheritance, 74-75
  from abstract classes, 383
  applying, 370-375
  attribute inheritance, 1224-1225
  constraints, 420-421
  constructors, 387
  custom exceptions, building, 390-392
  graphical representation of, 370
  interfaces, 400-402
  "is-a" relationships, 369
  LINQ to SQL classes, 651
  MustInherit keyword, 382-383
  MustOverride keyword, 382-383
  NotInheritable keyword, 381-382
  overriding, 375
    NotOverridable keyword, 379-380
    shared members, 389-390
  polymorphism, 375-376
  shadowing, 388-389
  structures, 355
  style inheritance, 831
Inherits keyword, 370
Inline Rename, 257-259
insert operations, 601-602
  using LINQ to SQL, 663-666
inspecting
  assembly information, 1195-1196
  built-in types, 67-68
  strings, 143-145
installed templates, searching for, 25
installing. *See also* ClickOnce; InstallShield LE; Windows Installer
  assemblies, 1264
  Windows 10 SDK, 976

InstallShield LE, 1269-1270
- downloading, 1271-1272
- .msi packages, creating
  - adding installation files, 1277-1278
  - configuring Registry on target machine, 1280
  - creating shortcuts, 1279-1280
  - environment variables, 1283
  - installation requirements, 1275-1277
  - setting properties, 1283
  - specifying application information, 1274
  - specifying dialogs, 1282
- Project Assistant, 1273-1274
- SetupExample project, 1271-1272
- Windows Installer distributable packages, deploying, 1285

instance variable accesses shared members warning, 55

instantiating
- classes, 76
- DataContext class, 658
- DbContext class, 703-704
- objects, 295-301
- structures, 352

IntelliSense technology, 30-32, 207
- overriding, 378-379
- XML comments, 1246-1247, 1250, 1255-1256

IntelliTrace, 1354-1359

interfaces, 119, 402-412
- CLS-compliant, 402
- defining, 393-394
- IComparable, 405-407
- IConvertible, 408-410
- IEnumerable, 403-405
- IFormattable, 410-412
- implementing, 394-399
  - behaviors, 398
- inheritance, 400-402
- IValueConverter interface, implementing, 897-900
- naming conventions, 402
- partial interfaces, 399
- passing as method arguments, 398-399
- polymorphism, 399-400
- scope of, 394
- variables, declaring, 397

interpolating strings, 141-143

Introduce Constant refactoring, 262-263

Introduce Local Variable refactoring, 260-261

Introduce Temporary Variable refactoring, 259-260

invoking. *See also* P/Invokes
- code, 1204-1206
- constructors, 296
  - nested invocations, 299-300
- members, 76
- methods, 279-280
- structures, 352
- WCF service members, 1026-1029

"is-a" relationships, 369

IsDefined method, 364

IService1.vb file, 1017-1018

isolation, 1179

iterators, 188-191, 441, 464-472
- anonymous iterators, 469-470
- benefits of, 465-466
- exiting from, 468
- iterator classes, implementing, 470-472
- scope of, 465
- simple iterators, 467-468
- Try..Catch..Finally blocks, 469
- XML literals, 743

IValueConverter interface, implementing, 897-900

J

jagged arrays, 169, 576-577
JIT (just-in-time) compiler, 9
JSON (JavaScript Object Notation), serialization, 1091
"Just My Code" debugging, 202-203
JustDecompile, 1365

K

keyboard, managing, 545
keywords
  AddHandler, 431-432
  Await keyword, 1145, 1158-1160
  ByRef, 281
  ByVal, 280
  Declare, performing P/Invokes, 1237-1238
  Default, 276
  GetType, 128-129
  Global, 343
  Inherits, 370
  Let, 630
  Me, 270
  MustInherit, 382-383
  MustOverride, 382-383
  MyBase, 383-385, 387
  MyClass, 385-387
  NotInheritable, 381-382
  NotOverridable, 379-380
  Overridable, 377-380
  Partial, 294
  RaiseEvent, 434
  ReDim, 168
  REM keyword, 72
  reserved keywords, 80-82
  Throw, 242-245
  unreserved, 82
  When, 245-246
  WithEvents, 432-433
killing unresponsive processes, 1097
kinds, 1313

L

Label control, 790
lambda expressions, 587-595
  debugging, 594-595
  lexical closures, 593-594
  multiline lambdas, 591
  Sub lambdas, 591-593
  type inference, 590
language features in Visual Basic 2015, 571
  anonymous types, 585-586
  array literals, 575-577
  extension methods, 577-584
    custom extension methods, 581-583
    exporting, 583-584
  generic variance
    contra variance, 597-598
    covariance, 596-597
  lambda expressions, 587-595
    debugging, 594-595
    lexical closures, 593-594
    multiline lambdas, 591
    Sub lambdas, 591-593
    type inference, 590
  local type inference, 571-575
  relaxed delegates, 586-587
languages, translating, 903-905
late binding, 1212
late binding warning, 54
launching external processes, 1096

layered architecture, WPF, 747-749
  DirectX libraries, 747
  PresentationFramework layer, 748
learning resources for Visual Studio 2012, 1363-1364
lexical closures, 593-594
libraries, 268
  assemblies, 1177-1179
    binding, 1178
    executing, 1179-1181
    information stored in, 1178
    installing and uninstalling, 1264
    loading, 1194
    location of, 1178
    metadata, 1179, 1192-1194
    private assemblies, 1178
    probing, 1178
    reflection, 1191-1192
    shared assemblies, 1178
    signing with strong names, 1179, 1265-1267
  COM libraries, adding references, 94-95
  portable libraries, creating, 494-503
  Task Parallel Library, 1105-1107
    tasks, 1108-1116
  TPL, 463
lifetime of objects, managing
  destructors, 319-322
  Dispose method, 317-323
  Finalize method, 315-316
  garbage collection, 314-315, 323-325
    generations, 324-325
  memory allocation, 313-314
  object resurrection, 322-323
  releasing objects, 314-315
  Using..End Using statement, 319
light bulb icon, 249-254
limitations
  of LINQ, 610
  of Model First approach, 716
  of XML serialization, 1081
LinearGradientBrush, 815-816
LinkedList(of T) collection, 458-460
LINQ, 599-611
  architecture, 613-614
  examples of, 611-612
  implicit line continuation, 612
  language support, 612
  limitations of, 610
  LINQ to DataSets, 679-682
    CopyToDataTable method, 682-683
    Field(of T) method, 684
    SetField(of T) method, 684
  LINQ to Objects, 615-616
    aggregation operators, 628-629
    concatenation operators, 641
    conversion operators, 630-632
    deferred execution, 623-625
    elements operators, 641-642
    equality operators, 640
    generation operators, 632-633
    grouping operators, 634-637
    ordering operators, 633
    partitioning operators, 643-644
    projection operators, 626-627
    querying in-memory objects, 616-623
    restriction operators, 627-628
    set operators, 633-634
    standard query operators, 625-626
    union operators, 637-639
  LINQ to XML, 730-735
    System.Xml.Linq namespace, 730
    XML documents, creating, 732
    XML documents, parsing, 732-733
    XML documents, querying, 734-735

Parallel LINQ, 1128-1137
controlling queries, 1134-1136
exception handling, 1136-1137
ParallelEnumerable class, 1133-1134
providers, 613
queries, measuring performance of, 1129-1131
query expressions, 611
syntax, 610-611
LINQ to Entities
querying EDMs, 710-712
LINQ to SQL, 645-658
abstraction, 646
associations, 646
classes, 647-658
.dbml files, 652
declaring, 658
inheritance, 651
System.Data.Linq.DataContext class, 650
custom validations, 673-675
entities, 646
deleting, 667-668
inserting, 663-666
SQL syntax, 676
updating, 666-667
logging, 671-672
mapping stored procedures, 668-671
optimistic concurrency, 675-676
Parallel LINQ
binary operators, 1133
intensive work, simulating, 1129
ordering sequences, 1132-1133
queries, 658-663
LINQPad, 1365
ListBox control, 791
ListDictionary collection, 447
listeners, trace listeners, 219-225
built-in trace listeners, 219
setting in configuration files, 224-225
List(of T) collection, 451-453
lists, creating for documentation, 1256-1257
ListView control, 792-793
live code analysis, 1306-1349
analyzers, building, 1315-1336
FxCop, 1345-1349
live static code analysis, 250-254
Live Visual Tree, 777
loading
assemblies, 1194
XML documents, 732-733
Local LB, 599
local type inference, 571-575, 736-737
scope of, 574-575
local variables, 270
localization, 901-902
MAT, 902
Windows Forms, 903-906
WPF, 906-908
Locals window, 209-210
location of assemblies, 1178
logging
Caller Information, 1212-1215
LINQ to SQL, 671-672
logical operators, 175-176
Logical Tree, 757-758
loops, 191-193

M

managed code
application domains, 1179-1181
creating, 1179-1181

debugging, 216-230
Debug class, 216-219
Trace class, 219-225
Management Portal (Microsoft Azure), 954-955
managing
documents, 857-866
annotation service, 862-864
object lifetime
destructors, 319-322
Dispose method, 317-323
Finalize method, 315-316
garbage collection, 314-315, 323-325
memory allocation, 313-314
object resurrection, 322-323
releasing objects, 314-315
Using..End Using statement, 319
windows in WPF, 770-772
manipulating
classes, System.IO.Path class, 514-515
documents, 857-866
files
System.IO.File class, 520-521
System.IO.FileInfo class, 522-523
strings, 527
manual installation, packaging applications for, 947-949
map-mode scroll bar, 33-34
Mapping Definition Language, 696
mapping stored procedures, 668-671
in EF, 712-716
marking assemblies as CLS compliant, 307-308
master pages, 924-927
master-details forms, creating, 886-891
MAT (Multilingual App Toolkit), 902
Me keyword, 270
measuring performance
of LINQ queries, 1129-1131
of PLINQ queries, 1131-1132
media
images, viewing, 851-853
playing, 853-856
MediaElement control, 794
members
of Application class, 773
of base classes, accessing, 383-387
MyBase keyword, 383-385
MyClass keyword, 385-387
invoking, 76
of My namespace, 537-538
My.Application, 538-542
of My.User, 560
shared members, 301-306
overriding, 389-390
memory allocation
object lifetime, managing, 313-314
structures, 355
memory streams, 526-527
Menu control, 794-796
MessageBox class, 340-341
messages
in Error List window, 39-40
WCF, 1014
metadata, 8, 1179, 1192-1194, 1217
inspecting, 1195-1196
reflection, 1191-1192
retrieving, 1194-1196
methods, 73, 278-290
arguments, 73, 280-285
optional arguments, 283-284
optional nullable arguments, 284-285
ParamArray arguments, 282-283
Call Hierarchy window (Visual Studio 2015), 69-70

Collect, 323
constraints, 419
CopyToDataTable, 682-683
for Debug class, 216
Dispose, 317-323
enumerations as return values, 365-366
exiting from, 289-290
extension methods, 344, 577-584
customizing, 581-583
exporting, 583-584
Field(of T), 684
Finalize, 315-316
generic methods, implementing, 418-419
generics, 287-288
GetName, 364
GetValueOrDefault, 423-424
GetValues, 363
invoking, 279-280
IsDefined, 364
overloading, 285-289
coercion, 287
generics, 287-288
optional parameter overloads, 288
parameters, 280
Parse, 364-365
partial methods, 293-295
custom validations, 673-675
ReRegisterForFinalize, 323
scope of, 278
SetField(of T) method, 684
shared methods, 304-306
SuppressFinalize, 323
System.Convert methods, 132-133
TestResurrection, 322-323
ToString, 364-365
WaitForPendingFinalizers, 323

Microsoft Azure, 951
applications
deploying, 964-967
templates, 961-962
ASP.NET applications, creating for the cloud, 961-964
cloud storage, 952-953
Cloud Storage Studio, 972
Management Portal, 954-955
registering for, 953
roles, 967-970
multiple roles, adding to projects, 970
options, configuring, 968-969
SDK, downloading, 954
services offered, 952
SQL databases
creating, 955-961
migrating to, 958-959
WCF services, hosting, 1031-1032
WebJobs, 970
Microsoft Azure Debugging Environment, 968
Microsoft Azure Developer Center, 953
migrating
to Azure SQL databases, 958-959
from CAS to .NET Framework 4.6, 1189
Code First Migration tool, 1044
Visual Basic 6 to 2015, 233
MilCore, 747
Mixed Mode debugging, 202
model binding, 935-937
Model First approach, 716
modules, 73-74, 347-349
comparing with classes, 349-350
partial modules, 348-349
scope of, 349
Monitor class, 1103
MSBuild, 5
MsCorlib.dll, 10

MSDeploy, 943-946
  manual installation, packaging applications for, 947-949
  packages, 943-944
  Web Deploy, 945-946
MSDN Library, 65-66
.msi packages, 1270
  creating, 1274-1283
    adding installation files, 1277-1278
    configuring Registry on target machine, 1280
    creating shortcuts, 1279-1280
    environment variables, 1283
    installation requirements, 1275-1277
    setting properties, 1283
    specifying application information, 1274
    specifying dialogs, 1282
multicast delegates, 430-431
multidimensional arrays, 168-169, 576-577
multiline lambdas, 591
multiline strings, 148-149
Multilingual App Toolkit, 1365
multiple constraints, combining, 420-421
multi-targeting, 22-23
multithreading, 1098-1099, 1139-1140
  passing parameters, 1099
  threads, creating, 1098-1099
MustInherit keyword, 382-383
MustOverride keyword, 382-383
MVC (Model-View-Controller), 910
  scaffolding, 923
My namespace, 537-538
  customizations, 566-570
  extending, 562-566
  My.Application, 538-542
    cultures, 539-540
    deployment information, 540-541
    retrieving command-line arguments, 542
    retrieving environment information, 541
    writing entries to Windows' application log, 541-542
  My.Computer, 542-550
    accessing the network, 547-548
    clipboard, 544-545
    FileSystem property, 543-544
    managing the keyboard, 545
    playing audio files, 545
    Registry, 546-547
    retrieving computer information, 548-550
  My.Resources, 556-560
    retrieving resources by name in code, 560
    WPF resources, 556-557
  My.Settings, 550-556
    application-level only settings, 552-553
    events, 555-556
  My.User, 560-562
  My.WebServices, 562
My Project window, 42-48, 83
  Application tab settings, 43-48
  resources, 86-90
My.Application, 538-542
  cultures, 539-540
  deployment information, 540-541
  extending, 564-566
  retrieving command-line arguments, 542
  retrieving environment information, 541
  writing entries to Windows' application log, 541-542
MyBase keyword, 383-385, 387
MyClass keyword, 385-387
My.Computer, 542-550
  accessing the network, 547-548
  clipboard, 544-545
  extending, 564-566
  FileSystem property, 543-544

managing the keyboard, 545
playing audio files, 545
Registry, 546-547
retrieving computer information, 548-550
My.Resources, 556-560
extending, 566
retrieving resources by name in code, 560
WPF resources, 556-557
My.Settings
application-level only settings, 552-553
events, 555-556
extending, 566
My.User, 560-562
My.WebServices, 562

N

NameOf operator, 150-153
namespaces, 10, 75, 327-330
aliasing, 77
avoiding ambiguities, 340-341
CLS rules, 342
extending, 344-345
global namespaces, 342-345
identifiers, 342
imported namespaces, 752-753
Imports directives, 336-338
default Imports, 338-339
My Namespace, 537-538
My namespace
customizations, 566-570
extending, 562-566
My.Application, 538-542
My.Computer, 542-550
My.Settings, 542-550
nested namespaces, 76, 332-335
in .NET Compiler Platform, 1309
purpose of, 331-332
root namespace, 335-336
scope of visibility, 335
Smart Name Resolution, 341-342
System.Reflection.Emit, 1206-1212
System.Xml namespace, 729
System.Xml.Linq namespace, 730
naming conventions
for CLS, 308-309
for identifiers, 80
for interfaces, 402
for value types, 104
narrowing conversions, 130-131
Navigate To feature (code editor), 34-35
navigation bar (Visual Studio 2015), 28
nested classes, 268-269
nested invocations, 299-300
nested namespaces, 76, 332-335
nested Try..Catch..Finally blocks, 241
.NET, 3
.NET 2015, 3-4
.NET Compiler Platform, 13-14
analyzers, building, 1315-1336
assemblies, 1309
Code Metrics, 1349-1351
Compiler APIs, 1308
live code analysis, 1306-1349
analyzers, building, 1315-1336
FxCop, 1345-1349
namespaces, 1309
refactoring, building, 1336-1345
Scripting API, 1309
semantics, 1313-1315
syntax, 1309-1313
workspaces, 1315

.NET Core 5, 1, 3, 10-12
- architecture, 11-12
- components, 11
- CoreCLR, 11
- .NET Native, 14-15
- open source layers, 12
- for Windows 10, 1009

.NET Foundation, 12
.NET Framework 4.6, 5-10
- ADO.NET, 600-603
  - connecting to databases, 601-603
  - connection modes, 601
  - data providers, 600-601
  - DataSets, 603-608
- ADO.NET Entity Framework. *See* EF (Entity Framework)
- architecture, 6
- assemblies, 8-9
- asynchronous I/O file operations, 1171-1175
- Base Class Library, 5
- BCL, 9-10
- CLR, 7-9
  - writing unmanaged code, 7-8
- COM objects, using in code, 1233
- comparing with .NET Framework 4.5, 7
- ILDasm, 9
- LINQ
  - architecture, 613-614
  - examples of, 611-612
  - implicit line continuation, 612
  - language support, 612
  - limitations of, 610
  - providers, 613
  - query expression, 611
- localization, 902
  - MAT, 902
  - Windows Forms, 903-906
- migration from CAS-based code, 1189
- MSBuild, 5
- objects, exposing in COM, 1235-1237
- security, 1182-1189
  - attributes, 1185
  - migration from CAS-based code, 1189
  - permissions, 1183-1184
  - sandboxing, 1186-1188
  - Transparency Level 2, 1184-1186
  - transparency model, 1182-1183
  - updates to, 1182-1183
- thread pool, 1099-1101
  - editing default values, 1101
  - querying, 1100

.NET Native, 14-15
networking with streams, 534-535
new features in Visual Studio 2015, 17-18
New Project window (Visual Studio 2015), 22
News section (Visual Studio 2015), 19
nongeneric collections, 442-451
- ArrayList collection, 442-445
- BitArray collection, 449-450
- BitVector32 collection, 450-451
- HashTable collection, 446-447
- HybridDictionary collection, 448
- ListDictionary collection, 447
- NameValueCollection collection, 449
- OrderedDictionary collection, 447-448
- Queue collection, 445-446
- SortedList collection, 448
- Stack collection, 446
- StringCollection collection, 448
- StringDictionary collection, 448-449

nonserialized events, 1080
NotInheritable keyword, 381-382
NotOverridable keyword, 379-380

nullable types, 422-425
  GetValueOrDefault method, 423-424
  null-propagating operator, 424-425
  syntax, 422
null-conditional operator, 185-188, 461-462
  syntax, 461-462
null-propagating operator, 424-425
  and XML literals, 740

O

Oasis Consortium, 1036
Object Browser window, 66-67, 442
objects
  complex objects, generating, 489-491
  declaring with WithEvents keyword, 432-433
  initializers, 300-301
  instantiating, 295-301
  lifetime, managing
    destructors, 319-322
    Dispose method, 317-323
    Finalize method, 315-316
    garbage collection, 314-315, 323-325
    memory allocation, 313-314
    object resurrection, 322-323
    Using..End Using statement, 319
  LINQ to Objects, 615-616
    aggregation operators, 628-629
    concatenation operators, 641
    conversion operators, 630-632
    deferred execution, 623-625
    elements operators, 641-642
    equality operators, 640
    generation operators, 632-633
    grouping operators, 634-637
    ordering operators, 633
    partitioning operators, 643-644
    projection operators, 626-627
    querying in-memory objects, 616-623
    restriction operators, 627-628
    set operators, 633-634
    union operators, 637-639
  .NET, exposing in COM, 1235-1237
  out-of-scope, 315
  releasing, 314-315
  serialization, 1074-1080
    binary serialization, 1074-1076
    custom objects, 1078-1080
    nonserialized events, 1080
    SOAP serialization, 1077-1078
  in System.Xml.Linq namespace, 730
ObservableCollection(of T) collection, 456-458, 872-876
OData (Open Data Protocol), 1035-1036
  endpoints, creating, 1036-1063
    adding a data model, 1038-1044
    adding controllers, 1045-1046
    CRUD operations, 1047-1052
    enabling Code First Migration, 1044
    handling relationships between entities, 1052-1054
    implementing queries, 1046-1047
    publishing the endpoints, 1054-1055
    querying data via HTTP requests, 1056-1060
    registering the OData service, 1044-1045
  functions, consuming, 1068-1070
  services, consuming, 1060-1068
    creating a client application, 1061-1065
    CRUD operations, 1067-1068
    querying data, 1065-1067
One ASP.NET, 912
online help, 65-66
online templates, 23-24

OOP (object-oriented programming), 267
  polymorphism, 375-376
open source layers, .NET Core 5, 12
operators
  arithmetic operators, 172-174
  assignment operators, 174-175
  binary operators, 1133
  bitwise operators, 177-179
  comparison operators, 182-184
  concatenation operators, 182
  conversion operators, 129-135
    CType, 133-135
    DirectCast operator, 133-135
    narrowing conversions, 130-131
    TryCast operator, 133-135
    widening conversions, 130
  CType, overloading, 359-360
  logical operators, 175-176
  null-conditional operator, 185-188, 461-462
  null-propagating operator, 424-425
  overloading, 356-360
  shift operators, 179-180
  short-circuiting operators, 176-177
  standard query operators, 625-626
    aggregation operators, 628-629
    concatenation operators, 641
    conversion operators, 630-632
    elements operators, 641-642
    equality operators, 640
    generation operators, 632-633
    grouping operators, 634-637
    ordering operators, 633
    partitioning operators, 643-644
    projection operators, 626-627
    quantifiers, 640-641
    restriction operators, 627-628
    set operators, 633-634
    union operators, 637-639
  ternary If operator, 180-182
optimistic concurrency
  in EDMs, 707-708
  resolving, 675-676
optimizations, 56-57
optional arguments, 283-284
optional nullable arguments, 284-285
OrderedDictionary collection, 447-448
ordering operators, 633
organizing
  Imports directives, 263-266, 339
  structures, 356
  types within namespaces, 327-330
    CLS rules, 342
    global namespaces, 342-345
    nested namespaces, 332-335
    root namespace, 335-336
    scope of visibility, 335
    Smart Name Resolution, 341-342
out-of-scope objects, 315
Output window, 41-42
overloading
  constructors, 298-300
  derived classes, 380
  methods
    coercion, 287
    declaring optional parameter overloads, 288
    generics, 287-288
    properties, 288-289
  operators, 356-360
  type parameters, 421-422
overloading methods, 285-289
overriding, 375, 377-380
  MustOverride keyword, 382-383
  NotOverridable keyword, 379-380
  shared members, 389-390

P

packages, 943-944
    app packages, creating and testing, 1009-1010
    manual installation, packaging applications for, 947-949
page lifetime for web forms, 915
page requests, 911
panels, 762-769
    Canvas panel, 767-768
    DockPanel panel, 768
    Grid panel, 762-764
    StackPanel panel, 765-766
    ViewBox panel, 769
    WrapPanel panel, 766-767
Parallel class, 1106
parallel computing
    concurrent collections, 1124-1128
        BlockingCollection(of T), 1127-1128
        ConcurrentBag(of T), 1124-1125
        ConcurrentDictionary(TKey, TValue), 1126-1127
        ConcurrentQueue(of T), 1125
        ConcurrentStack(of T), 1126
    Parallel class, 1106
    parallel loops, 1116-1122
        Parallel..For loop, 1117-1119
        Parallel..ForEach loop, 1119-1121
        partitioning, 1121-1122
    ParallelOptions class, 1107
    Task Parallel Library, 1106-1107
    TaskFactory class, 1107
    tasks
        canceling, 1113-1114
        debugging tools, 1122-1123
        exception handling, 1111-1113
        returning values, 1110-1111
        running with Parallel.Invoke, 1108-1109
        Task class, 1109-1110
    TaskScheduler class, 1107
Parallel LINQ, 1128-1137
    binary operators, 1133
    exception handling, 1136-1137
    intensive work, simulating, 1129
    ordering sequences, 1132-1133
    ParallelEnumerable class, 1133-1134
    queries
        controlling, 1134-1136
        measuring performance of, 1131-1132
ParallelEnumerable class, 1133-1134
Parallel..For loop, 1117-1119
Parallel..ForEach loop, 1119-1121
ParallelLoopState class, 1122
ParallelOptions class, 1107
ParamArray arguments, 282-283
parameters, 280
    applicable data types for attributes, 1221
Parse method, 364-365
parsing XML documents, 732-733
partial classes, 290-293
Partial keyword, 294
partial methods, 293-295
    custom validations, 673-675
partial modules, 348-349
partitioning operators, 643-644
passing event information, 435-437
PasswordBox control, 796-797
pathnames
    exception handling, 519
    manipulating, 514-515
Peek Definition, 32-33
performance of LINQ queries, measuring, 1129-1131
performing P/Invokes, 1237-1238
PerfTips, 208-209

permissions, 523
  in .NET Framework 4.6, 1183-1184
  requirements, documenting, 1257
PIAs (primary interoperability assemblies), 95
  Deploy Without PIAs feature, 95-97
PInvoke.net website, 1244
P/Invokes, 1237
  encapsulating, 1239-1240
  exception handling, 1243-1244
  types, converting to unmanaged, 1240-1243
    StructLayout attribute, 1241-1243
    VBFixedString attribute, 1243
platforms for ASP.NET, 910
playing audio files, 545
playing media, 853-856
POCO (Plain Old CLR Objects), 1091
polymorphism, 375-376
  interfaces, 399-400
  overriding, 376
portable classes, creating, 491-494
portable libraries, creating, 494-503
postback events, 915-916
preparing
  debugging example, 199-200
  reflection example, 1193-1194
PresentationFramework layer (WPF), 748
private assemblies, 1178
private constructors, 300
private scope, 276
probing, 1178
processes
  closing, 1097
  isolation, 1179
  killing, 1097
  launching, 1096
  querying, 1097
programmatically accessing ClickOnce, 1301-1302
programming languages in Visual Studio 2015, 12-13
ProgressBar control, 797-798
Project Assistant (InstallShield), 1273-1274
Project Roslyn, 14
projects. *See also* My Project window
  code files, 82-83
    Application.myapp file, 83-85
    AssemblyInfo.vb file, 85-86
    Resources.resx file, 86-90
  compiling, 48
    advanced options, 55-58
    conditional compilation, 57
    creating configurations, 50-52
    Debug configuration, 49
    generating serialization assemblies, 58
    options, 52-53
    Release configuration, 49-50
    target CPU, specifying, 53
    warning configurations, 53-55
  compiling projects
    background compiler, 52
    optimizations, 56-57
  creating, 21-22, 27-29
  finding, 29
  multi-targeting, 22-23
  role options, configuring, 968-969
  round-tripping, 20
  SetupExample, 1271-1272
  shared projects, 503-512
  solutions, 20
  templates, 21
    accessing recently used templates, 23-24
    cloud-hosted application templates, 961-962
    online templates, 23-24
    searching for installed templates, 25
    WPF template, 749

properties, 72-73, 271-276, 310
- accessing, 275
- auto-implemented read-only properties, 271-272
- default properties, 275-276
- overloading, 288-289
- read-only properties, 272-274
- setting for .msi packages, 1283
- shared properties, 303-304
- for System.Exception, 237-240
- of WPF windows, 770
- write-only properties, 274

Properties window, 40-41
proxy class, 1025-1026
public scope, 276
publish process (ClickOnce), 1289-1294

Q

quantifiers, 640-641
query expressions, 611
querying. *See also* LINQ
- databases, 603
- DataSets, 679-682
- EDMs
  - with Entity SQL, 712
  - with LINQ to Entities, 710-712
- in-memory objects, 616-623
- with LINQ to SQL, 658-663
- .NET thread pool, 1100
- processes, 1097
- standard query operators, 625-626
- XML documents, 734-735

Queue collection, 445-446
Queue(of T) collection, 460
quick actions
- Encapsulate Field, 261-262
- Extract Interface, 254-256

Quick Launch tool, 67-69
Quick Watch window, 213-214

R

RadialGradientBrush, 817
RadioButton control, 799
RaiseEvent keyword, 434
raising events, 434-437
reading
- binary files, 525-526
- text files, 524-525

read-only auto-implemented properties, 73
read-only fields, 270
read-only properties, 272-274
ReadOnlyCollection(of T) collection, 454-455
ReadOnlyObservableCollection(of T) collection, 458
read/write locks, 1103-1104
recently used templates, accessing, 23-24
rectangle shape, drawing, 799-800
recursive cloning, 128
recursive operator or property access warning, 55
ReDim keyword, 168
redundant code, organizing, 263-266
refactoring, 254-266
- building, 1336-1345
- Encapsulate Field quick action, 261-262
- Extract Interface quick action, 254-256
- Extract Method refactoring, 256-257
- Inline Rename, 257-259
- Introduce Constant refactoring, 262-263
- Introduce Local Variable refactoring, 260-261
- Introduce Temporary Variable refactoring, 259-260

reference types, 100-103, 113-116
  cloning, 126
  comparing with value types, 116-120
  converting to value type
    boxing, 123-124
    deep copy, 127-128
    GetType keyword, 128-129
    implicit conversion, 121-123
    shallow copy, 126-127
    unboxing, 124-125
references, 92-97
  adding to COM libraries, 94-95
  ByRef keyword, 281
reflection, 1191-1192, 1225-1226
  assemblies, loading, 1194
  Caller Information, 1212-1215
  late binding, 1212
  metadata, inspecting, 1195-1196
  sample assembly, preparing, 1193-1194
  System.Reflection.Emit namespace, 1206-1212
  types, 1196-1204
Region directives, 77-78
registering
  for events, 431-432
  for GC events, 324-325
  for Microsoft Azure, 953
  OData service, 1044-1045
  with Windows Store, 976
Registration-Free COM, 1302-1304
regular expressions, 294
relaxed delegates, 586-587
Release configuration, 49-50
releasing objects, 314-315
  COM objects, 1233-1234
  Using..End Using statement, 319
REM keyword, 72
RemoveHandler keyword, 431-432
ReRegisterForFinalize method, 323
reserved keywords, 80-82
resolving optimistics concurrency, 675-676
Resources.resx file, 86-90
restriction operators, 627-628
resurrecting objects, 322-323
rethrowing exceptions, 243
RichTextBox control, 800, 864-866
roles, 967-970
  multiple roles, adding to projects, 970
  options, configuring, 968-969
root namespace, 335-336
Root Namespace field (My Project window), 43
RotateTransform, 838
round-tripping, 20
routed events, 760-761
routing strategies, 761
Run to Cursor command, 201
runtime errors, 62-64

S

sample code
  browsing online code samples, 25-26
  downloading, 2
sandboxing, 1186-1188
scaffolding, 923
scalability of web applications, 911-912
ScaleTransform, 838
scope
  of fields, 270
  of interfaces, 394
  of iterators, 465
  of local type inference, 574-575
  of methods, 278

of modules, 349
of namespaces, 335
of types, 276-278
of XML comments, 1250
Scripting API, 1309
scroll bar (code editor), 33-34
ScrollBar control, 800
ScrollViewer control, 800-801
SDKs
Windows 10 SDK, 976
Windows SDK, 15
searching for installed templates, 25
security
CAS, 1182
ClickOnce, configuring, 1299-1300
login, forcing on web applications, 937-939
in .NET Framework 4.6, 1182-1189
attributes, 1185
conditional APTCA, 1186-1188
migration from CAS-based code, 1189
permissions, 1183-1184
sandboxing, 1186-1188
Transparency Level 2, 1184-1186
transparency model, 1182-1183
updates to, 1182-1183
permissions, 523
requirements, documenting, 1257
SelectionBrush, 822-823, 922
semantics, .NET Compiler Platform, 1313-1315
Separator control, 801
serialization, 1073
in ADO.NET Entity Framework, 1091-1092
custom serialization, 1083-1086
events, 1085-1086
JSON serialization, 1091
objects, 1074-1080
binary serialization, 1074-1076
custom objects, 1078-1080
nonserialized events, 1080
SOAP serialization, 1077-1078
WCF serialization, 1088-1091
with XAML, 1086-1088
deserialization, 1088
generic collections, 1087
XML serialization, 1081-1083
customizing, 1082-1083
limitations of, 1081
serialization assemblies, generating, 58
server controls, 917-918
Server Explorer tool (Visual Studio), 970-971
services
Microsoft Azure offerings, 952
OData, 1035-1036
consuming, 1060-1068
registering, 1044-1045
OData (Open Data Protocol)
endpoints, creating, 1036-1063
WCF, 1014
ABC, 1014-1015
configuring, 1033
consuming, 1024-1029
custom logic, implementing, 1019-1023
EDM, 1023
endpoints, 1014
exception handling, 1030-1031
generics, 1022-1023
implementing, 1015-1023
Set Next Statement command, 201
set operators, 633-634
settings, 90-92
Application tab settings (My Project window), 43-48
SetupExample project, 1271-1272
shadowing, 380, 388-389
shallow copy, 126-127
shared assemblies, 1178

shared classes, 302
  modules, 347-349
    partial modules, 348-349
    scope of, 349
shared constructors, 306
shared fields, 302
shared members
  generating, 488
  overriding, 389-390
shared methods, 304-306
shared projects, 503-512
shared properties, 303-304
shift operators, 179-180
short-circuiting operators, 176-177
shortcuts, creating for .msi files, 1279-1280
Show Next Statement command, 202
signing assemblies with strong names, 1179, 1265-1267
Silverlight, 6
simple iterators, 467-468
single types, reflecting, 1203-1204
SkewTransform, 838-839
Slider control, 801-802
Smart Name Resolution, 341-342
SOAP serialization, 1077-1078
SolidColorBrush, 813-814
Solution Explorer window, 36-38
solutions, 20
SortedDictionary(of TKey, TValue) collection, 456
SortedList collection, 448
spans, 1313
spell check for RichTextBox control, 865-866
SQL databases
  creating for Microsoft Azure, 955-961
  LINQ to SQL, 645-646
    abstraction, 646
    associations, 646
    classes, 647-658
    custom validations, 673-675
    deleting entities, 667-668
    entities, 646
    inserting entities, 663-666
    logging, 671-672
    mapping stored procedures, 668-671
    optimistic concurrency, 675-676
    SQL syntax for entities, 676
    updating entities, 666-667
  migrating to Microsoft Azure, 958-959
Stack collection, 446
Stack(of T) collection, 460
StackPanel panel, 765-766
standard query operators, 625-626
  aggregation operators, 628-629
  concatenation operators, 641
  conversion operators, 630-632
  elements operators, 641-642
  equality operators, 640
  generation operators, 632-633
  grouping operators, 634-637
  ordering operators, 633
  partitioning operators, 643-644
  projection operators, 626-627
  quantifiers, 640-641
  restriction operators, 627-628
  set operators, 633-634
  union operators, 637-639
Start method, 1096
Start Page (Visual Studio 2015), 18-19
  Discover What's New section, 20
  Featured Videos section, 20
  News section, 19
  What's New on Microsoft Platforms section, 20
Startup Object field (My Project window), 44

state management, 920-923
  Application state, 920
  Cache state, 921-922
  Context state, 922
  cookies, 922
  Session state, 922
  ViewState state, 923
status bar colors in Visual Studio 2015, 18-19
StatusBar control, 802
Step Into command, 201
Step Over command, 201
storage, cloud storage, 952-953
Store Schema Definition Language, 694-696
stored procedures, mapping, 668-671
  in EF, 712-716
streams, 524-535
  binary streams, reading and writing, 525-526
  compressing data, Zip archives, 532-534
  data compression, 527-532
  manipulating strings, 527
  memory streams, 526-527
  networking with, 534-535
  text streams, reading and writing, 524-525
string formatters, implementing, 895-897
StringCollection collection, 448
StringDictionary collection, 448-449
strings, 135-136
  comparing, 136-138
  concatenating, 149-150
  converting to dates, 155-156
  copying, 143
  editing, 145-148
  formatting, 138-143
  inspecting, 143-145
  interpolating, 141-143
  manipulating, 527
  multiline strings, 148-149
  null strings, checking for, 138
strong-named assemblies, 1265-1267
StructLayout attribute, converting types to unmanaged, 1241-1243
structure of ClickOnce deployments, 1294
structures, 74, 351-356
  assigning to variables, 354
  CLS rules, 360
  creating, 351
  DateTimeOffset, 159-160
  inheritance limitations, 355
  instantiating, 352
  invoking, 352
  members, scope qualifiers, 354-355
  memory allocation, 355
  organizing, 356
  passing to methods, 354
styles, 828-833
  inheritance, 831
  triggers, 831-833
Sub lambdas, 591-593
submitting apps to Windows Store, 1010-1011
subscribing to GC events, 324-325
SuppressFinalize method, 323
symbols, comment symbol ('), 72
synchronous programming, 1140
SyncLock..End..SyncLock statement, 1101-1102
syntax
  for delegates, 428-429
  for LINQ, 610-611
  .NET Compiler Platform, 1309-1313
  for nullable types, 422
  null-conditional operator, 461-462
  SQL syntax for entities, 676
System.Array class, 169-172
System.Console class, 29

System.Convert methods, 132-133
System.Data.Linq.DataContext class, 650
System.Diagnostics.Process class, 1096
System.Enum class, 362-365
System.Exception, 233
  properties, 237-240
System.GC class, 323-324
System.IO.Directory class, 515-518
System.IO.DirectoryInfo class, 518
System.IO.DriveInfo class, 519
System.IO.File class, 520-521
System.IO.FileInfo class, 522-523
System.IO.Path class, 514-515
System.IO.Stream class, 524
System.Object class, 373-375
System.Reflection.Emit namespace, 1206-1212
System.Runtime.CompilerServices.RuntimeWrappedException, catching exceptions from wrapped objects, 1233
System.Windows.Style class, 828
System.Xml namespace, 729
System.Xml.Linq namespace, 730

T

Tablet mode (Windows 10), 1007
tabular data forms, creating, 877-885
tags (XML)
  <c> tag, 1253-1254
  <param> tag, 1255
  recommended tags for XML comments, 1250-1251
target CPU, specifying, 53
Target Framework combo box (My Project window), 48
Task class, 1109-1110
Task Parallel Library, 1105-1107
  Parallel class, 1106
  parallel loops, 1116-1122
    Parallel..For loop, 1117-1119
    Parallel..ForEach loop, 1119-1121
    partitioning, 1121-1122
  ParallelOptions class, 1107
  TaskFactory class, 1107
  tasks, 1108-1116
    canceling, 1113-1114
    exception handling, 1111-1113
    returning values, 1110-1111
    running with Parallel.Invoke, 1108-1109
    Task class, 1109-1110
  TaskScheduler class, 1107
Task-Based Asynchrony, 1161-1165
  combinators, 1163-1165
  custom awaiters, 1175
  debugging tasks, 1175
  switching threads, 1162-1163
TaskFactory class, 1107
tasks, 1108-1116
  Barrier class, 1115-1116
  canceling, 1113-1114
  debugging tools, 1122-1123
  exception handling, 1111-1113
  returning values, 1110-1111
  running with Parallel.Invoke, 1108-1109
TaskScheduler class, 1107
templates
  accessing recently used templates, 23-24
  for cloud-hosted applications, 961-962
  control templates, 833-836
  online templates, 23-24
  project templates, 21
  searching for installed templates, 25
  for universal Windows apps, 978-980
  WCF project templates, 1015-1016
  WPF project template, 749

ternary If operator, 180-182
testing app packages, 1009-1010
TestResurrection method, 322-323
text files, reading and writing, 524-525
TextBlock control, 804-805
TextBox control, 805
themes for Visual Studio 2015, 19
thread pool
  editing default values, 1101
  querying, 1100
threading, 1095
  APM, 1142-1143
  creating threads, 1098-1099
  killing unresponsive processes, 1097
  multithreading, 1098-1099, 1139-1140
  .NET thread pool, 1099-1101
  passing parameters, 1099
  processes
    closing, 1097
    launching, 1096
    querying, 1097
  synchronization, 1101-1104
    Monitor class, 1103
    read/write locks, 1103-1104
    SyncLock..End..SyncLock statement, 1101-1102
  Task-Based Asynchrony, 1162-1163
Threads window, 214
Throw keyword, 242-245
time, representing, 160-161
TimeZone type, 162-164
TimeZoneInfo type, 164
Toast notifications, implementing in universal Windows apps, 996-997
tool windows
  docking, 36
  Error List window, 39-40
  Output window, 41-42
  Properties window, 40-41
  Solution Explorer window, 36-38
tool windows (Visual Studio 2015), 35-42
ToolBar control, 805-806
ToString method, 364-365
touch improvements to code editor, 32
TPL (Task Parallel Library), 463
Trace class, 219-225
trace listeners, 219-225
  built-in trace listeners, 219
  setting in configuration files, 224-225
transformations, 836-840
  multiple transformations, applying, 840
  RotateTransform, 838
  ScaleTransform, 838
  SkewTransform, 838-839
  TranslateTransform, 839-837
TranslateTransform, 839-837
translating languages, 903-905
Transparency Level 2, 1184-1186
transparency model, 1182-1183
TreeView control, 806-807
triggers, applying styles with, 831-833
TryCast operator, 133-135
Try..Catch..Finally blocks, 233-236
  exiting from, 241
  iterators, 469
  nesting, 241
tunnel strategy, 761
type inference, 590
types, 9
  anonymous types, 585-586
  applicable data types for attributes, 1221
  built-in types, inspecting, 67-68
  constraints, 419-420
  converting to unmanaged, 1240-1243
    StructLayout attribute, 1241-1243
    VBFixedString attribute, 1243

creating, 485-491
custom types, exposing, 274-275
enumerations, 361-367
generics, 287-288, 413-414
  consuming, 416-417
  creating, 414-415
  generic methods, implementing, 418-419
  overloading type parameters, 421-422
late binding, 1212
namespaces, 75, 327-330
  aliasing, 77
  avoiding ambiguities, 340-341
  CLS rules, 342
  extending, 344-345
  global namespaces, 342-345
  My namespace, 537-538
  nested namespaces, 332-335
  purpose of, 331-332
  root namespace, 335-336
  scope of visibility, 335
  Smart Name Resolution, 341-342
nullable types, 422-425
  GetValueOrDefault method, 423-424
  null-propagating operator, 424-425
  syntax, 422
Peek Definition, 32-33
properties, 72-73
reference types, 100-103, 113-116
  cloning, 126
reflecting, 1196-1204
scope, 276-278
TimeZone, 162-164
value types, 100-113
  analyzing content of, 110-111
  applying, 106-108
  assigning, 108-110
  BigInteger, 112-113
    building custom value types, 113
    naming conventions, 104

U

UAC (User Account Control), application settings, 46-48
unboxing, 124-125
uninstalling assemblies, 1264
union operators, 637-639
Universal Windows Platform, creating universal Windows apps, 977-998
  app bar, 987-994
  app manifest, 998-1001
  defining a data template, 986-987
  implementing a data model, 980-981
  Internet connection, 981-986
  navigation, 987-994
  playing media, 997-998
  project templates, 978-980
  raising Toast notifications, 996-997
  sharing content, 995-996
unmanaged code, 1237
  writing, 7-8
unreserved keywords, 82
unresponsive processes, killing, 1097
unused local variable warning, 54
updating
  applications with ClickOnce, 1296-1297
  entities, 666-667, 706-707
use of variable prior to assignment warning, 54
Using..End Using statement, 319
UWP (Universal Windows Platform), 12, 974-976
  APIs, 975-976

V

validation rules, LINQ to SQL, 673-675
value types, 100-113
- analyzing content of, 110-111
- applying, 106-108
- assigning, 108-110
- BigInteger, 112-113
- building custom value types, 113
- comparing with reference types, 116-120
- converting to reference type
  - boxing, 123-124
  - deep copy, 127-128
  - GetType keyword, 128-129
  - implicit conversion, 121-123
  - shallow copy, 126-127
  - unboxing, 124-125
- enumerations, 361-367
- naming conventions, 104

variables
- interface variables, declaring, 397
- local variables, 270
- structures, assigning, 354

variance
- contra variance, 597-598
- covariance, 596-597

VBFixedString attribute, converting types to unmanaged, 1243
View Windows Settings button (My Project window), 46-48
ViewBox panel, 769
viewing
- images, 851-853
- XPS documents, 866-867

views, 891-895
ViewState state, 923
VirtualizingStackPanel control, 767
Visual Basic 14, 13
Visual Basic 2012, learning resources, 1363-1364
Visual Basic 6, migrating to Visual Basic 2015, 233
Visual Studio 2015 IDE, 17
- Autos window, 215
- Blend, 833
- Call Hierarchy window, 69-70
- Call Stack window, 210-212
- Class Designer, 473-484
  - designing objects, 475-479
  - diagrams, creating, 482-483
  - diagrams, exporting, 483-484
  - enabling, 474-475
  - implementing derived classes, 479-482
- Class View window, 484
- code editor, 29-35
  - IntelliSense technology, 30-32
  - Navigate To feature, 34-35
  - Peek Definition, 32-33
  - scroll bar, 33-34
  - touch improvements, 32
  - zooming in/out, 30
- Code Snippet Editor, 1364
- CodeRush Xpress, 1364
- Command window, 210
- designer tools, 751
- Diagnostic Tools, 1351-1354
- Generate from Usage feature, 485-491
  - generating shared members, 488
- Generate from Usage feature (Visual Studio), generating complex objects, 489-491
- help page, 65-66
- importing COM objects, 1230-1232
- InstallShield LE, 1269-1270
- integrated programming languages, 12-13
- IntelliTrace, 1354-1359
- Locals window, 209-210

Microsoft Azure SDK, downloading, 954
My Project window, 42-48, 83
  Application tab, 43-48
navigation bar, 28
new features, 17-18
New Project window, 22
projects, 20-21
  Application.myapp file, 83-85
  AssemblyInfo.vb file, 85-86
  compiling, 48
  creating, 21-22, 27-29
  finding, 29
  multi-targeting, 22-23
  Resources.resx file, 86-90
  role options, configuring, 968-969
  templates, 21
Quick Launch tool, 68-69
scaffolding, 923
Server Explorer tool, 970-971
Start Page, 18-19
  Discover What's New section, 20
  Featured Videos section, 20
  News section, 19
  What's New on Microsoft Platforms section, 20
status bar colors, 18-19
themes, 19
Threads window, 214
tool windows, 35-42
  docking, 36
  Error List window, 39-40
  Output window, 41-42
  Properties window, 40-41
  Solution Explorer window, 36-38
user developer tools
  Fiddler, 1365
  LINQPad, 1365
Watch windows, 212-214
  web applications, creating
    data controls, adding, 929-932
    data model, adding, 927
    filtering capabilities, adding, 932-933
    forcing login for security, 937-939
    master pages, 924-927
    model binding, 935-937
    navigation controls, adding, 933
    new web form, adding, 927-929
    strongly typed data controls, 934-935
Visual Studio Gallery, 1366
Visual Studio Power Tools, 1365
Visual Tree, 758
VisualBrush, 823-825
visualizers, 215-216

W

WaitForPendingFinalizers method, 323
warning configurations, 53-55
warning messages, 39
  ignoring, 54
warnings, 247-249
Watch windows, 212-214
WCF (Windows Communication Foundation), 1013-1014
  ABC, 1014-1015
    autogenerated contracts, 1018-1019
    IService1.vb file, 1017-1018
  built-in bindings, 1028-1029
  custom logic, implementing, 1019-1023
  EDM, 1023
  endpoints, 1014
  exceptions, handling, 1030-1031
  generics, 1022-1023
  JSON serialization, 1091
  serialization, 1088-1091

services
configuring, 1033
consuming, 1024-1029
hosting, 1031-1032
implementing, 1015-1023
services, implementing, 1015-1016
web applications, ASP.NET, 910-914
creating with Visual Studio
data controls, adding, 929-932
data model, adding, 927
filtering capabilities, adding, 932-933
forcing login for security, 937-939
master pages, 924-927
model binding, 935-937
navigation controls, adding, 933
new web form, adding, 927-929
strongly typed data controls, 934-935
deploying, 927-949
1-click deployment, 940-941
classic publishing, 941-943
MSDeploy, 943-946
developing
controls, 917-919
event handling, 919-920
web forms, 914-916
page requests, 911
platforms, 910
project templates, 912-914
scalability, 911-912
state management, 920-923
Application state, 920
Cache state, 921-922
Context state, 922
cookies, 922
Session state, 922
ViewState state, 923
Web Deploy, 945-946
web forms, 914-916
cached and postback events, 915-916
page lifetime, 915
web roles, 968
WebBrowser control, 808
Web.Config file, 1017
WebJobs, 970
websites
Flexera, 1271
.NET Foundation, 12
Oasis Consortium, 1036
PInvoke.net, 1244
Visual Studio Gallery, 1366
What's New on Microsoft Platforms section (Visual Studio 2015), 20
When keyword, 245-246
widening conversions, 130
windows, managing in WPF, 770-772
Windows 8.X, 1229
Windows 10
adaptive user interfaces, 1007
Desktop mode, 1008-1009
.NET Core, 1009
SDK, installing, 976
Tablet mode, 1007
UWP, 975-976
APIs, 975-976
Windows API
documentation, 1244
P/Invokes
encapsulating, 1239-1240
exception handling, 1243-1244
performing, 1237-1238
Windows Forms, localization, 903-906
Windows Installer, 1270
Windows Registry, 546-547
Windows SDK, 15

Windows Store
  apps, 1270
  registering with, 976
  submitting apps to, 1010-1011
WindowsFormsHost control, 808-809
WinRT (Windows Runtime), 1229
with..end with statements, 196-197
WithEvents keyword, 432-433
worker roles, 968
workspaces, 1315
WPF (Windows Presentation Foundation), 746
  animations, 840-848
    ColorAnimation, 844-845
    creating, 847-848
    DoubleAnimation, 841-843
    events, 845-847
  Application class, 772-774
    members, 773
    Startup and Exit events, handling, 774
  applications, creating, 745-746, 749-751
    Canvas panel, 767-768
    DockPanel panel, 768
    Grid panel, 762-764
    Logical Tree, 757-758
    StackPanel panel, 765-766
    ViewBox panel, 769
    Visual Tree, 758
    WrapPanel panel, 766-767
    XAML, 751-757
  Browser Applications, 774-777
  brushes, 812-828
    BitmapCacheBrush, 826-828
    CaretBrush, 822-823
    DrawingBrush, 825
    ImageBrush, 818-821
    LinearGradientBrush, 815-816
    RadialGradientBrush, 817
    SelectionBrush, 822-823
    SolidColorBrush, 813-814
    VisualBrush, 823-825
  COM objects, calling, 1234-1237
  control templates, 833-836
  controls, 779-780
    Border control, 781-782
    Button control, 782
    Calendar control, 782-783
    CheckBox control, 783
    ComboBox control, 784-785
    ContentControl element, 780-781
    DataGrid control, 784-785
    DatePicker control, 786
    DocumentViewer control, 787
    Expander control, 788
    Frame control, 789
    GroupBox control, 790
    Image control, 790
    Label control, 790
    ListBox control, 791
    ListView control, 792-793
    MediaElement control, 794
    Menu control, 794-796
    PasswordBox control, 796-797
    ProgressBar control, 797-798
    RadioButton control, 799
    RichTextBox control, 800
    ScrollBar control, 800
    ScrollViewer control, 800-801
    Separator control, 801
    Slider control, 801-802
    StatusBar control, 802
    TabControl control, 803-804
    TextBlock control, 804-805
    TextBox control, 805
    ToolBar control, 805-806

TreeView control, 806-807
WebBrowser control, 808
WindowsFormsHost control, 808-809
data binding, 869-876
bindings, creating, 872
DataGrid control, 872-876
drag-and-drop data binding, 876-900
IValueConverter interface, implementing, 897-900
modes, 871-872
ObservableCollection(of T) collection, 872-876
string formatters, implementing, 895-897
views, 891-895
dialogs, 809-810
documents, manipulating, 857-866
Ellipse shape, 787-788
event handling, 759-761
images, viewing, 851-853
improvements to, 747
layered architecture, 747-749
DirectX libraries, 747
PresentationFramework layer (WPF), 748
Live Visual Tree, 777
localization, 906-908
media, playing, 853-856
RichTextBox control, 864-866
styles, 828-833
inheritance, 831
triggers, 831-833
transformations, 836-840
multiple transformations, applying, 840
RotateTransform, 838
ScaleTransform, 838
SkewTransform, 838-839
TranslateTransform, 839-837
window management, 770-772

WrapPanel panel, 766-767
wrappers, exception handling, 1233
write-only properties, 274
writing
binary files, 525-526
custom attributes, 1220-1221
entries to Windows' application log, 541-542
text files, 524-525
unmanaged code, 7-8
WSDL (Web-Service Definition Language), 1021

X

XAML, 751-757
controls, adding to user interface, 753-757
declarative mode, 756-757
imperative mode, 756-757
imported namespaces, 752-753
serialization
deserialization, 1088
generic collections, 1087
Windows Store apps, 752
XBAP (XAML Browser Application), 774
XCopy deployment of assemblies, 1262
XML
app manifest, 998-1001
LINQ to XML, 730-735
System.Xml.Linq namespace, 730
XML documents, creating, 732
XML documents, parsing, 732-733
XML documents, querying, 734-735. *See also* XML comments; XML literals
serialization, 1081-1083
customizing, 1082-1083
deserialization, 1082
limitations of, 1081

Settings.settings file, 90-92
WSDL, 1021
XML comments, 1246
<c> tag, 1253-1254
<param> tag, 1255
compiled help files, generating, 1258
complex documentation, 1251
documenting permissions requirements, 1257
enabling, 1247
generics, 1258
implementing, 1248-1258
links to other references, specifying, 1257-1258
lists, creating, 1256-1257
MSDN recommended tags, 1250-1251
referring to external documentation file, 1256
scope of, 1250
XML literals, 735-736, 737
embedded expressions, 740-743
iterators, 743
LINQ queries, 738-740
local type inference, 736-737
null-propagating operator, 740
XmlSerializer constructor, 1081
XPS documents, viewing, 866-867

Y-Z

Zip archives, 532-534
zooming code editor in/out, 30

UNLEASHED

Unleashed takes you beyond the basics, providing an exhaustive, technically sophisticated reference for professionals who need to exploit a technology to its fullest potential. It's the best resource for practical advice from the experts, and the most in-depth coverage of the latest technologies.

informit.com/unleashed

OTHER UNLEASHED TITLES

C# 5.0 Unleashed
ISBN-13: 9780672336904

ASP.NET Dynamic Data Unleashed
ISBN-13: 9780672335655

Microsoft System Center 2012 Unleashed
ISBN-13: 9780672336126

System Center 2012 Configuration Manager (SCCM) Unleashed
ISBN-13: 9780672334375

System Center 2012 R2 Configuration Manager Unleashed: Supplement to System Center 2012 Configuration Manager (SCCM) Unleashed
ISBN-13: 9780672337154

Windows Server 2012 Unleashed
ISBN-13: 9780672336225

Microsoft Exchange Server 2013 Unleashed
ISBN-13: 9780672336119

Microsoft Visual Studio 2015 Unleashed
ISBN-13: 9780672337369

System Center 2012 Operations Manager Unleashed
ISBN-13: 9780672335914

Microsoft Dynamics CRM 2013 Unleashed
ISBN-13: 9780672337031

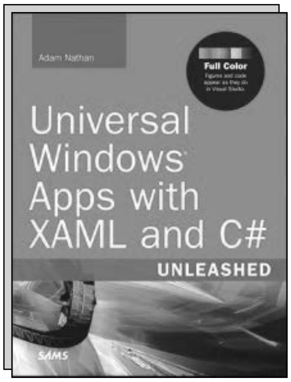
Microsoft Lync Server 2013 Unleashed
ISBN-13: 9780672336157

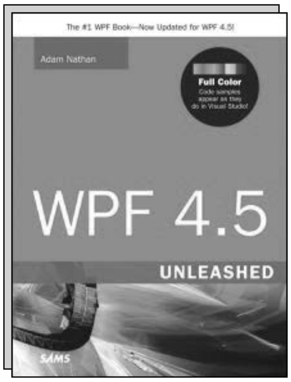
Visual Basic 2012 Unleashed
ISBN-13: 9780672336317

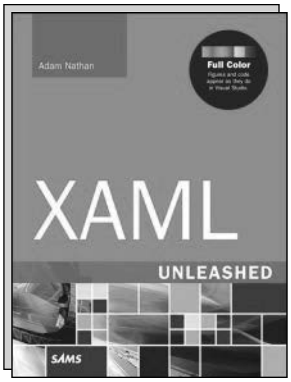
Microsoft SQL Server 2014 Unleashed
ISBN-13: 9780672337291

SAMS

informit.com/sams


Universal Windows Apps with XAML and C# Unleashed
ISBN-13: 9780672337260


WPF 4.5 Unleashed
ISBN-13: 9780672336973


XAML Unleashed
ISBN-13: 9780672337222

SAMS

THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams
SAFARI BOOKS ONLINE

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS Safari Books Online

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit **informit.com/newsletters**.
- Access FREE podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library at **safari.informit.com**.
- Get tips from expert blogs at **informit.com/blogs**.

Visit **informit.com/learn** to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **informit.com/socialconnect**.

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS Safari Books Online