C# 5.0
UNLEASHED
Contents at a Glance

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1 Introducing the .NET Platform</td>
<td>5</td>
</tr>
<tr>
<td>2 Introducing the C# Programming Language</td>
<td>55</td>
</tr>
<tr>
<td>3 Getting Started with .NET Development Using C#</td>
<td>103</td>
</tr>
<tr>
<td>4 Language Essentials</td>
<td>175</td>
</tr>
<tr>
<td>5 Expressions and Operators</td>
<td>251</td>
</tr>
<tr>
<td>6 A Primer on Types and Objects</td>
<td>301</td>
</tr>
<tr>
<td>7 Simple Control Flow</td>
<td>351</td>
</tr>
<tr>
<td>8 Basics of Exceptions and Resource Management</td>
<td>407</td>
</tr>
<tr>
<td>9 Introducing Types</td>
<td>463</td>
</tr>
<tr>
<td>10 Methods</td>
<td>501</td>
</tr>
<tr>
<td>11 Fields, Properties, and Indexers</td>
<td>547</td>
</tr>
<tr>
<td>12 Constructors and Finalizers</td>
<td>585</td>
</tr>
<tr>
<td>13 Operator Overloading and Conversions</td>
<td>609</td>
</tr>
<tr>
<td>14 Object-Oriented Programming</td>
<td>649</td>
</tr>
<tr>
<td>15 Generic Types and Methods</td>
<td>701</td>
</tr>
<tr>
<td>16 Collection Types</td>
<td>755</td>
</tr>
<tr>
<td>17 Delegates</td>
<td>789</td>
</tr>
<tr>
<td>18 Events</td>
<td>843</td>
</tr>
<tr>
<td>19 Language Integrated Query Essentials</td>
<td>913</td>
</tr>
<tr>
<td>20 Language Integrated Query Internals</td>
<td>977</td>
</tr>
<tr>
<td>21 Reflection</td>
<td>1057</td>
</tr>
<tr>
<td>22 Dynamic Programming</td>
<td>1119</td>
</tr>
<tr>
<td>23 Exceptions</td>
<td>1175</td>
</tr>
<tr>
<td>24 Namespaces</td>
<td>1221</td>
</tr>
<tr>
<td>25 Assemblies and Application Domains</td>
<td>1241</td>
</tr>
<tr>
<td>26 Base Class Library Essentials</td>
<td>1301</td>
</tr>
<tr>
<td>27 Diagnostics and Instrumentation</td>
<td>1373</td>
</tr>
<tr>
<td>28 Working with I/O</td>
<td>1399</td>
</tr>
<tr>
<td>29 Threading and Synchronization</td>
<td>1443</td>
</tr>
<tr>
<td>30 Task Parallelism and Data Parallelism</td>
<td>1513</td>
</tr>
<tr>
<td>31 Asynchronous Programming</td>
<td>1551</td>
</tr>
<tr>
<td>32 Introduction to Windows Runtime</td>
<td>1643</td>
</tr>
<tr>
<td>Index</td>
<td>1671</td>
</tr>
</tbody>
</table>
Table of Contents

Introduction 1
Who Should Read This Book? .......................................................... 2
What You Need to Know Before You Read This Book .................. 2
How This Book Is Organized ......................................................... 3

1 Introducing the .NET Platform 5
A Historical Perspective ............................................................... 5
A 10,000-Feet View of the .NET Platform ....................................... 9
The Common Language Infrastructure ......................................... 12
The Multilanguage Aspect of .NET ............................................... 15
Introducing .NET Assemblies ....................................................... 16
The Common Type System Explained .......................................... 17
Executing Managed Code ............................................................. 24
Diving into the Common Language Runtime ............................... 32
The Base Class Library .................................................................. 51
Summary ..................................................................................... 54

2 Introducing the C# Programming Language 55
The Evolution of C# ....................................................................... 55
A Sneak Peek at the Future ........................................................... 95
Summary .................................................................................... 102

3 Getting Started with .NET Development Using C# 103
Installing the .NET Framework .................................................... 103
Your First Application: Take One ................................................ 113
Visual Studio 2012 ....................................................................... 119
Your First Application: Take Two ................................................ 127
Summary .................................................................................... 173

4 Language Essentials 175
The Entry Point ........................................................................... 175
Keywords ..................................................................................... 181
A Primer on Types ........................................................................ 184
Built-In Types ............................................................................. 190
Local Variables ........................................................................... 212
Intermezzo on Comments .......................................................... 223
Arrays ......................................................................................... 230
5 Expressions and Operators  
What Are Expressions? ................................. 251  
The Evaluation Stack .................................... 255  
Arithmetic Operators .................................... 258  
String Concatenation .................................. 269  
Shift Operators .......................................... 274  
Relational Operators .................................... 275  
Logical Operators ....................................... 277  
Conditional Operators ................................. 281  
An Operator's Result Type ............................ 284  
Null-Coalescing Operator ............................ 285  
Assignment ............................................... 287  
Summary ................................................... 299  

6 A Primer on Types and Objects  
Implicit Versus Explicit Conversions .............. 301  
The typeof Operator: A Sneak Peek at Reflection 319  
Default Value Expression ........................... 322  
Creating Objects with the new Operator .......... 324  
Member Access ......................................... 336  
Invocation Expressions ............................... 340  
Element Access ........................................ 348  
Summary ................................................... 349  

7 Simple Control Flow  
What Are Statements, Anyway? .................... 351  
Expression Statements .............................. 353  
The Empty Statement ................................ 355  
Statement Blocks ...................................... 356  
Declarations ............................................. 357  
Selection Statements ................................. 358  
Iteration Statements ................................. 375  
A Peek at Iterators .................................... 391  
Loops in the Age of Concurrency ................. 398  
The goto Statement .................................... 400  
The return Statement ............................... 404  
Summary ................................................... 406
# Table of Contents

## 8 Basics of Exceptions and Resource Management 407
- Exception Handling .............................................. 407
- Deterministic Resource Cleanup ................................. 438
- Locking on Objects ................................................. 448
- Summary ............................................................. 462

## 9 Introducing Types 463
- Types Revisited ....................................................... 463
- Classes Versus Structs .............................................. 466
- Type Members ....................................................... 486
- Summary ............................................................. 499

## 10 Methods 501
- Defining Methods ................................................... 501
- Specifying the Return Type ....................................... 502
- Declaring Parameters ............................................. 504
- Overloading .......................................................... 519
- Extension Methods ............................................... 524
- Partial Methods ..................................................... 534
- Extern Methods ..................................................... 538
- Refactoring .......................................................... 540
- Code Analysis ....................................................... 545
- Summary ............................................................. 546

## 11 Fields, Properties, and Indexers 547
- Fields ......................................................................... 547
- An Intermezzo About Enums ...................................... 563
- Properties ............................................................. 574
- Indexers ................................................................. 580
- Summary ............................................................. 583

## 12 Constructors and Finalizers 585
- Constructors .......................................................... 585
- Static Constructors .................................................. 592
- Destructors (Poorly Named Finalizers) ......................... 595
- Summary ............................................................. 608

## 13 Operator Overloading and Conversions 609
- Operators ............................................................. 609
- Conversions ......................................................... 633
- Summary ............................................................. 647
## Contents

14 **Object-Oriented Programming** 649
   - The Cornerstones of Object Orientation .................................................. 649
   - Inheritance for Classes .......................................................................... 663
   - Protected Accessibility ........................................................................... 674
   - Polymorphism and Virtual Members ...................................................... 676
   - Abstract Classes .................................................................................... 688
   - Interface Types ..................................................................................... 690
   - Summary ............................................................................................. 699

15 **Generic Types and Methods** 701
   - Life Without Generics ........................................................................... 701
   - Getting Started with Generics ................................................................ 704
   - Declaring Generic Types ........................................................................ 707
   - Using Generic Types ............................................................................. 712
   - Performance Intermezzo ........................................................................ 714
   - Operations on Type Parameters .............................................................. 718
   - Generic Constraints .............................................................................. 720
   - Generic Methods .................................................................................. 736
   - Co- and Contravariance ......................................................................... 743
   - Summary ............................................................................................. 754

16 **Collection Types** 755
   - Nongeneric Collection Types ................................................................. 755
   - Generic Collection Types ....................................................................... 765
   - Thread-Safe Collection Types ................................................................. 778
   - Other Collection Types .......................................................................... 786
   - Summary ............................................................................................. 787

17 **Delegates** 789
   - Functional Programming ....................................................................... 789
   - What Are Delegates? ............................................................................. 794
   - Delegate Types ...................................................................................... 794
   - Delegate Instances ................................................................................ 798
   - Invoking Delegates ............................................................................... 811
   - Putting It Together: An Extensible Calculator .......................................... 815
   - Case Study: Delegates Used in LINQ to Objects .................................... 819
   - Asynchronous Invocation ...................................................................... 823
   - Combining Delegates ............................................................................ 835
   - Summary ............................................................................................. 842
18 Events
The Two Sides of Delegates .................................................. 844
A Reactive Application ......................................................... 845
How Events Work ............................................................... 853
Raising Events, the Correct Way ........................................... 855
add and remove Accessors ................................................... 857
Detach Your Event Handlers ................................................ 861
Recommended Event Patterns .............................................. 871
Case Study: INotifyPropertyChanged Interfaces and UI Programming ................................................. 880
Countdown, the GUI Way .................................................... 890
Event Interoperability with WinRT ......................................... 896
Introduction to Reactive Programming ................................ 898
Summary ............................................................................. 911

19 Language Integrated Query Essentials
Life Without LINQ .............................................................. 914
LINQ by Example ............................................................... 921
Query Expression Syntax ................................................... 931
Summary ............................................................................. 975

20 Language Integrated Query Internals
How LINQ to Objects Works ................................................. 977
Standard Query Operators .................................................. 1000
The Query Pattern ............................................................. 1033
Parallel LINQ ....................................................................... 1036
Expression Trees ............................................................... 1045
Summary ............................................................................. 1055

21 Reflection
Typing Revisited, Static and Otherwise ................................ 1058
Reflection ............................................................................ 1063
Lightweight Code Generation .............................................. 1091
Expression Trees ............................................................... 1101
Summary ............................................................................. 1117

22 Dynamic Programming
The dynamic Keyword in C# 4.0 .......................................... 1119
DLR Internals ................................................................. 1137
Office and COM Interop ..................................................... 1159
Summary ............................................................................. 1174
## Contents

**23 Exceptions**
- Life Without Exceptions .................................................. 1175
- Introducing Exceptions .................................................... 1178
- Exception Handling ......................................................... 1180
- Throwing Exceptions ....................................................... 1196
- Defining Your Own Exception Types ................................. 1198
- (In)famous Exception Types .............................................. 1201
- Summary ........................................................................ 1220

**24 Namespaces**
- Organizing Types in Namespaces ........................................ 1221
- Declaring Namespaces ..................................................... 1227
- Importing Namespaces .................................................... 1231
- Summary ........................................................................ 1240

**25 Assemblies and Application Domains**
- Assemblies ....................................................................... 1241
- Application Domains ....................................................... 1286
- Summary ........................................................................ 1298

**26 Base Class Library Essentials**
- The BCL: What, Where, and How? ...................................... 1303
- The Holy System Root Namespace ...................................... 1311
- Facilities to Work with Text .............................................. 1356
- Summary ........................................................................ 1372

**27 Diagnostics and Instrumentation**
- Ensuring Code Quality .................................................... 1374
- Instrumentation ................................................................ 1388
- Controlling Processes ...................................................... 1396
- Summary ........................................................................ 1398

**28 Working with I/O**
- Files and Directories ........................................................ 1399
- Monitoring File System Activity ........................................ 1407
- Readers and Writers ....................................................... 1409
- Streams: The Bread and Butter of I/O ............................... 1415
- A Primer to (Named) Pipes .............................................. 1434
- Memory-Mapped Files in a Nutshell ................................. 1437
- Overview of Other I/O Capabilities ................................. 1440
- Summary ........................................................................ 1440
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Threading and Synchronization</td>
<td>1443</td>
</tr>
<tr>
<td></td>
<td>Using Threads</td>
<td>1444</td>
</tr>
<tr>
<td></td>
<td>Thread Pools</td>
<td>1474</td>
</tr>
<tr>
<td></td>
<td>Synchronization Primitives</td>
<td>1482</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>1511</td>
</tr>
<tr>
<td>30</td>
<td>Task Parallelism and Data Parallelism</td>
<td>1513</td>
</tr>
<tr>
<td></td>
<td>Pros and Cons of Threads</td>
<td>1514</td>
</tr>
<tr>
<td></td>
<td>The Task Parallel Library</td>
<td>1515</td>
</tr>
<tr>
<td></td>
<td>Task Parallelism</td>
<td>1520</td>
</tr>
<tr>
<td></td>
<td>Data Parallelism</td>
<td>1542</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>1550</td>
</tr>
<tr>
<td>31</td>
<td>Asynchronous Programming</td>
<td>1551</td>
</tr>
<tr>
<td></td>
<td>Why Asynchronous Programming Matters</td>
<td>1551</td>
</tr>
<tr>
<td></td>
<td>Old Asynchronous Programming Patterns</td>
<td>1564</td>
</tr>
<tr>
<td></td>
<td>Asynchronous Methods and <code>await</code> Expressions</td>
<td>1584</td>
</tr>
<tr>
<td></td>
<td>Behind the Scenes</td>
<td>1610</td>
</tr>
<tr>
<td></td>
<td>Advanced Topics</td>
<td>1634</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>1641</td>
</tr>
<tr>
<td>32</td>
<td>Introduction to Windows Runtime</td>
<td>1643</td>
</tr>
<tr>
<td></td>
<td>What Is Windows Runtime?</td>
<td>1643</td>
</tr>
<tr>
<td></td>
<td>Creating a Windows Runtime Component</td>
<td>1658</td>
</tr>
<tr>
<td></td>
<td>Overview of the Windows Runtime APIs</td>
<td>1667</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>1669</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>1671</td>
</tr>
</tbody>
</table>
About the Author

Bart J.F. De Smet is a software development engineer on Microsoft’s Cloud Programmability Team, an avid blogger, and a popular speaker at various international conferences. In his current role, he’s actively involved in the design and implementation of Reactive Extensions for .NET (Rx) and on an extended “LINQ to Anything” mission. You can read about Bart’s technical adventures on his blog at http://blogs.bartdesmet.net/bart.

His main interests include programming languages, virtual machines and runtimes, functional programming, and all sorts of theoretical foundations. In his spare time, Bart likes to go out for a hike in the wonderful nature around Seattle, read technical books, and catch up on his game of snooker.

Before joining the company in October 2007, Bart was active in the .NET community as a Microsoft Most Valuable Professional (MVP) for C#, while completing his Bachelor of Informatics, Master of Informatics, and Master of Computer Science Engineering studies at Ghent University, Belgium.
Acknowledgments

Writing this book was a huge undertaking that would have proven impossible without the support of many people. I’d like to apologize upfront for forgetting any of you. (I’ll buy you a Belgian beer if I did.)

First and foremost, I cannot thank my family enough for the support they’ve given me over the years to pursue my dreams. Their support for my 6-year university studies in Ghent and tolerance for my regular absence to participate in the technical community have all been essential ingredients. If this weren’t enough, my move across the Pacific Ocean to go and work at the Microsoft headquarters has put us through the ultimate test. Words fall short to describe how incredibly lucky I am to have their ongoing support. Thanks once more!

I wouldn’t have ended up in the world of computer science if not for some of my teachers. For my first exposure to computers, I have to go back to 1993, checking sums during the mathematics lessons at elementary school. Thanks to “Meester Wilfried” for his MS-DOS and GWBASIC powered calculator that shaped my future. In high school, several people kept me on this track, as well. Math teachers Paul, Geert, and Ronny had to endure endless conversations about programming languages. In a weird twist of history, I never got educated in informatics in high school, but nonetheless I spent countless hours in the computer rooms of my school. Without the support of Hans De Four, I wouldn’t have gotten where I am today. Sorry for all the network downtime caused by my continuous experiments with ProfPass, domain controllers, and whatnot.

Looking back over 10 years in history, I’m eternally grateful to the people at the local Microsoft subsidiary in Belgium (back then called Benelux) for adopting me in the early .NET community and giving me the chance to work on various projects. In particular, I want to thank my very first contact at Microsoft, Gunther Beersaerts, for all the advice he gave me over the years. Gunther’s been a true source of inspiration, encouraging me to take the speaker stand at various conferences.

During a few summers in the early 2000s, many Microsoft Belgium people provided a nice place for me to grow and learn while working on various exciting projects. Thanks to Chris Volckerick for taking me on board to build the (now defunct) http://www.dotnet.be website, using what was called ASP+ back then. Later, Gerd De Bruycker took me under his wing to develop the first MSDN home page for Microsoft Belux. Your passion for the developer community has always stuck with me (not just that wild community VIP party in Knokke).

A bigger project called SchoolServer came around in the summer of 2004 and lasted for the years after. Christian Ramioul’s faith in my technical skills needed to land this project was unbelievable. And getting to know the IT professional audience that had to work with the solution wouldn’t have been possible without the wonderful collaboration I had with Ritchie Houtmeyers (remember the countless hours spent in our server room office?) and
Ricardo Noulez. Big thanks go to Bart Vande Ghinste for giving me a crash course on COM+.

Over the years, I’ve had the honor to interact with a tremendous number of community members at various conferences. Mentioning all of them would be a Herculean task, so I won’t even attempt. I want to call out a few, though. First of all, thanks to the Belgian developer evangelism team for their relentless support over the years: Gerd De Brucyker and Tom Mertens, you’ve done a great job. Today’s community is in great hands with Katrien De Graeve, Hans Verbeeck, and Arlindo Alves. Hans De Smael, you continue to be my ongoing source of debugging and bit-twiddling inspiration. Finally, and sadly enough, this list wouldn’t be complete without taking a moment to remember the late David Boschmans and Patrick Tisseghem, who passed away suddenly: We miss you!

Finally, we enter my Redmond-based Microsoft Corporation career that started in October 2007, thanks to Scott Guthrie’s mail through my blog asking me to interview with the company. Ultimately, I ended up working on Windows Presentation Foundation’s AppModel team, where I felt welcome from day one. In particular, I want to thank my first office mate, Chango Valtchev, for the countless hours he spent to bring me up to speed in the codebase, sharing tons of debugging insights, and epic hikes. Of my first couple of managers, Grzegorz Zygmunt and Adam Smith have been great in helping me shape my early career and provided room for my speaking engagements abroad.

Once I finally started writing this book in 2009, a lot of my colleagues were put through the test. My office mates Mike Cook and Eric Harding had to withstand the most boring stories on various language constructs, generated IL code, functional programming adventures, and ways to (ab)use the C# programming language. Benjamin Westbrook, whom I’ve worked with for several months, underwent a similar faith during lunchtime. I have to thank Ben for sharing the things he enjoys most when reading technical books: I hope you find some of your stylistic ideas here and there throughout the book. Patrick Finnigan deserves a special mention here, too. Not only for being a great colleague taking over some of the work I’ve been doing on the team, but even more so as a great book reviewer with tons of feedback both technically and stylistically. Thanks a lot!

Thanking all the other WPF colleagues I’ve worked with and who gave me various technical insights would take up way too much space. Instead, here’s a roll-up of folks I’m very grateful to have worked with. Adam, Alik, Andre, Dwayne, Joe, Eric, Matt, Saied, Zia: Thanks a ton.

In the middle of the book-writing adventure, I transitioned to the Cloud Programmability Team. Thanks to Erik Meijer for taking me on board in the oasis he’s created for innovative and creative ideas, allowing me to work on one of my key passions: LINQ. My colleagues Danny Van Velzen, Jeffrey Van Gogh, Mark Shields, and Wes Dyer have been fantastic to bring me up to speed. Endless technical discussions have been a tremendous source of inspiration that contributed directly to this book’s contents. This is also the right spot to thank my professor Raymond Boute. It turns out Erik and I caught the passion for functional programming from the same professor, set a few decades apart in another twist of history.
I can’t thank the Sams team enough, in particular Neil Rowe for his incredible patience with me. Even though I always knew writing this book was going to be a huge task, lots of unexpected twists made the schedule more challenging every time. Combine this with an ever-growing page count and changing table of contents: I’m very grateful I could write the book I think is right for the C# programmer’s audience with virtually no constraints. Also thanks to the technical team for leading the way through new authoring and publication software and assisting with my numerous technical requests. A special word of thanks goes to the technical reviewer, Boyd Nolan, and various other team members who participated in various reviews. Writing a book is not only about teaching your readers, it’s also a lot about learning things yourself (including some of the English language, thanks Keith).

This book would not exist if not for the wonderful C# language and its designers. So, I want to thank Anders Hejlsberg and the entire language design team for giving us the most favorite .NET language out there. This big thank you also applies to the Common Language Runtime (CLR) team for bringing a managed runtime to a wide variety of platforms. Internal resources have been very helpful in providing valuable insights, in particular on our C# Discussion List.

Last but not least, I want to thank the waiters and waitresses in various downtown Bellevue restaurants for tolerating my regular book-writing presence, hiding behind a laptop screen and asking for endless soda refills.
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 www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.
Introduction

Does the world need yet another book about C#? Very rightfully, you might be asking this question right now (and so did some of my colleagues when I first told them about this book project). In short, what sets this book apart from many others is its in-depth coverage of how things work. Personally, I firmly believe in education that stimulates the student’s curiosity. That’s one of the things this book aims to do.

*The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity.*

—Albert Einstein

Understanding how a language brings its features to life is always a good thing, which will help you on many fronts. For one thing, coming up with a proper software design requires a good insight in the à la carte menu of language features, so that you can pick the ones best suited for the job at hand and that will not leave you with a bitter after-taste. Also, solid knowledge about the language and its caveats will prove invaluable while debugging your (or someone else’s) code. Occasional historical perspectives interwoven throughout this book help you to understand why the language looks the way it does today.

A tremendous number of .NET libraries have been born over the years, each addressing specific needs for particular applications. Doing justice to any of those by trying to reduce their coverage to a few tens of pages seems overly optimistic. Moreover, different developers have different needs: Maybe you’re in charge of user interface (UI) design or web development, but you may well be specialized in service-oriented architectures or designing a data-access layer. Each of those domains has very specific concepts that deserve whole books dedicated to them.

For all those reasons, this book shies away from even attempting such shallow in-breadth coverage of the .NET Framework. Instead, we aim at the common ground where all developers meet: the way they express their thoughts through programming languages. In this book, you’ll get essential insights in the foundations of the platform and one of the most commonly used languages, C#. Armed with this knowledge, you should be able to discover and understand many technologies that build on the platform.

As a concrete example, today’s libraries are built using object-oriented programming, so we spend our time explaining the capabilities of this feature. Similarly, recent application programming interfaces (APIs) (such as Language Integrated Query [LINQ]) have started to leverage the expressiveness of programming constructs (such as lambda expression) borrowed from the functional world, so we take a look at how those work.
Finally, at the intersection of different developer audiences, there are quite a few libraries that no one can live without. Examples include primitive types, collections, parallel programming capabilities, performing I/O operations, and so on. Discussing those libraries has several benefits. Not only does the reader get a good idea about the essential toolset the .NET Framework has to offer, but it also allows us to illustrate various language features using them. A good example is the discussion of generic types and LINQ through the lens of collections.

I sincerely hope you enjoy reading this book as much as I enjoyed writing it. I've learned a lot during the process, and I'm confident you will, too.

*Homines dum docent discunt.*
(Latin phrase translated “Men learn while they teach.”)
—Seneca, *Epistolae, VII, 7*

Happy coding!
Bart J.F. De Smet
Bellevue, Washington

**Who Should Read This Book?**

This book is for anyone who wants to learn the C# programming language in depth, understanding how language features truly work. While giving you those insights, you’ll learn where and how to use the features to design various kinds of software. Essential tools and techniques to carry out tasks such as debugging are covered, too.

If you've already had prior exposure to C#—maybe a previous version of the language—this book will help you grasp the new language features that have been added over the years. If you're new to the C# language and/or the .NET platform as a whole, this book will guide you through the language in a step-by-step manner.

In summary, this book not only teaches the language's capabilities, but it also looks behind the scenes to build a solid foundation that will aid you in studying other parts of the .NET platform. Because programming languages are at the heart of a developer's expressiveness, such a foundation is essential no matter what your day-to-day programming activities are.

**What You Need to Know Before You Read This Book**

No prior knowledge of the C# programming language is assumed, although it helps to have a basic idea of typical language constructs. Any kind of modern programming background can help here. For example, readers with a background in C, C++, or Java will feel at home with the C# syntax immediately. Those coming from C++ or Java will have no issue appreciating the power of object-oriented programming. For Visual Basic developers, the different syntax might be a hurdle to overcome, but lots of concepts will sound familiar.
Likely the most important thing to have is technical curiosity and the willingness to learn and truly understand a (new) programming language and the platform on which it’s built.

**How This Book Is Organized**

Two writing principles have been taken as a guide in this book:

- Avoid backward references, causing mental jumps for the readers. In other words, this book tells you the story of how various language features are built on top of each other, starting from primitive constructs such as expressions and statements. Sometimes making a little jump is unavoidable due to the historical evolution the language has undergone. In such a case, I present you with the basics of the feature in question and refer to a later chapter for in-depth coverage.

- Samples of technologies are interspersed with the coverage of language features that underpin them. For example, a discussion of generics naturally leads to an overview of various collection types in the .NET Framework. Similarly, a good explanation of Language Integrated Query (LINQ) cannot take place without coverage of constructs such as extension methods and lambda expressions.

From a 10,000-feet point of view, this book consists of three core pieces:

- The first four chapters introduce the .NET platform, the tooling ecosystem, and the C# programming language. In this part, a good historical perspective is provided that will help you understand why those technologies were created and how they evolved.

- Chapters 5 through 25 cover the C# programming language itself, with immediate application of language features where applicable. A further breakdown looks as follows:

  - We start by looking at basic constructs, such as expressions, operators, and statements. These language elements should be familiar to anyone who’s been programming before, allowing things such as arithmetic, control flow, and so on. Finally, we introduce the notion of exception handling.

  - Next, our attention is aimed at larger units of code. To set the scene, we start by introducing the notion of types, and then cover members such as methods and properties.

  - This naturally leads to a discussion of object-oriented programming in Chapter 14, covering the notions of objects, classes, interfaces, virtual methods, and so on.

  - After explaining generic types and methods, we move on to orthogonal language features, such as delegates, events, and Language Integrated Query (introduced in C# 3.0). Constructs borrowed from functional programming are covered here, too.
Next, we revise our notions of typing and introduce runtime services, such as reflection, that allow a more dynamically typed code characteristic. This brings us to an in-depth discussion of C# 4.0’s dynamic feature, including the Dynamic Language Runtime infrastructure that underpins it.

To put the icing on the cake, we take a closer look at the largest units of code the programming language deals with: namespaces and assemblies. The latter of the two touches quite a few runtime concepts, such as the global assembly cache, native images, and application domains, all of which are explained here, too.

Finally, the last chapters give an overview of the .NET Framework libraries about which every good developer on the platform should know. Here we cover essential types in the BCL, how to perform various kinds of I/O, diagnostic capabilities, and the increasingly important domain of multithreaded and asynchronous programming.

All in all, this book takes a language-centric approach to teaching how to program rich and possibly complex applications on the .NET Framework. Activities such as API design, coming up with proper application architectures, and even debugging issues with existing code all benefit from a deep understanding of the language and runtime. That’s precisely the sweet spot this book aims for.
CHAPTER 3

Getting Started with .NET Development Using C#

Time to set yourself up for a successful journey through the world of .NET development using C#. An obvious first thing to tackle is to install the .NET Framework and the necessary tools so that we can start running and writing code. One tool we pay a fair amount of attention to is Visual Studio 2012, but we cover other useful tools, too.

To get our hands dirty, we write a simple C# application and highlight some of the important language elements, go through the process of building it using various tools, and look at how we can debug code.

Installing the .NET Framework

The first logical step to get started with writing code targeting the .NET Framework is to install it on your development machine. At this point, let’s skip the in-depth discussion on how to deploy the .NET Framework to the machines where your application code is to run ultimately, be it a client computer, a web server, or even the cloud.

The .NET Framework Version Landscape

Over the years, various versions of the .NET Framework have been released. In this book, we cover the latest (at the time of this writing) release of the .NET Framework, version 4.5, which goes hand in hand with the Visual Studio 2012 release.
Does that mean you can’t use the knowledge you gain from .NET 4.5 programming to target older releases of the framework? Certainly not! Although the .NET Framework has grown release after release, the core principles have remained the same, and a good part of the evolution is to be found at the level of additional application programming interfaces (APIs) that are made available through the class libraries. A similar observation of evolution obviously holds on the level of the managed languages: Features are added every time around that often require additional framework support (for example, Language Integrated Query [LINQ] in C# 3.0, `dynamic` in version 4). As you can imagine, keeping track of all those versions and their key differentiating features isn’t always easy. To make things more clear, take a look at Table 3.1.

**TABLE 3.1 .NET Platform Version History**

<table>
<thead>
<tr>
<th>Version</th>
<th>Codename</th>
<th>Visual Studio</th>
<th>C#</th>
<th>VB</th>
<th>Flagship Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Lightning</td>
<td>2002 (7.0)</td>
<td>1.0</td>
<td>7.0</td>
<td>Managed code</td>
</tr>
<tr>
<td>1.1</td>
<td>Everett</td>
<td>2003 (7.1)</td>
<td>1.0</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>Whidbey</td>
<td>2005 (8.0)</td>
<td>2.0</td>
<td>8.0</td>
<td>Generics</td>
</tr>
<tr>
<td>3.0</td>
<td>WinFX</td>
<td>2005 (8.0)</td>
<td>2.0</td>
<td>8.0</td>
<td>WPF, WCF, WF</td>
</tr>
<tr>
<td>3.5</td>
<td>Orcas</td>
<td>2008 (9.0)</td>
<td>3.0</td>
<td>9.0</td>
<td>LINQ</td>
</tr>
<tr>
<td>4.0</td>
<td>Dev10</td>
<td>2010 (10.0)</td>
<td>4.0</td>
<td>10.0</td>
<td>Dynamic</td>
</tr>
<tr>
<td>4.5</td>
<td>Dev11</td>
<td>2012 (11.0)</td>
<td>5.0</td>
<td>11.0</td>
<td>Asynchronous programming</td>
</tr>
</tbody>
</table>

Notice that new releases of the .NET Framework typically go hand in hand with updates to the Visual Studio tooling support. A notable exception to this rule was the .NET 3.0 release, where Visual Studio 2005 additions were made to support the newly added features (for example, by providing designer support for Windows Presentation Foundation [WPF]). Notice, however, how the managed languages evolve at a slightly slower pace. It’s perfectly imaginable that a future release of the .NET Framework will still be using C# 5.0 and VB.NET 11.0. History will tell.

**NOTE: WHAT ABOUT OPERATING SYSTEM INTEGRATION?**

Being a logical extension to the Win32 API for Windows programming, it very much makes sense to have the framework components readily available together with various versions of the Windows operating system. However, Windows and the .NET Framework have been evolving at a different pace, so the innovation on the level of the .NET Framework hasn’t always been immediately available with the operating system out there at that point in time.

One first little piece of integration with the operating system happened with Windows XP, where the image loaded was made aware of the existence of managed code, to be able to load managed executables with fewer workarounds than would be required otherwise. In the Windows Server 2003 era, the 1.1 release of the .NET Framework was brought to the operating system so that the ASP.NET web stack was available out of the box for use in web server installations.
The bigger integration story happened around Vista, driven by the WinFX vision of enhancing core system capabilities like windowing (with WPF) and communication (with WCF). For a long time during the development of Vista—known as Longhorn at the time—WinFX formed a core pillar of the next-generation operating system, and development proceeded hand in hand with the operating system. Only later was WinFX decoupled from Vista and ported back to other platforms, resulting in what became known as .NET Framework 3.0. This said, the .NET Framework 3.0 components still shipped with Windows Vista as an optional Windows component that can be turned on or off. The same holds for Windows Server 2008.

With the release of Windows 7, this tradition continued by making the .NET Framework 3.5 components available out of the box. More recently, with Windows 8, the .NET Framework 4.5 shipped out of the box with the product. But there’s more. Thanks to the tight integration with Windows Runtime, we can now truly speak of a better together story where the power of Windows is readily accessible for .NET developers.

What Table 3.1 doesn’t show is the versioning of the Common Language Runtime (CLR). There’s a very important point to be made about this: The CLR evolves at a much slower pace than the libraries and languages built on top of it. Slow most often has a pejorative feel to it, but for the CLR this is a good thing: The less churn made to the core of runtime, the more guarantees can be made about compatibility of existing code across different versions of the .NET Framework. Figure 3.1 illustrates this nicely based on the .NET Framework 3.x history.

From this figure, you can see how both .NET Framework 3.0 and .NET Framework 3.5 are built to run on top of the existing CLR 2.0 runtime bits. This means that for all the goodness that ships with those versions of the .NET Framework, no changes were required to the core execution engine, a good sign of having a solid runtime that’s ready to take on a big job.

NOTE: RED BITS VERSUS GREEN BITS

A concept you might sometimes hear from Microsoft people in the context of framework versioning is that of red bits and green bits. The categorization of framework assemblies in red bits and green bits was introduced in the .NET 3.x timeframe to distinguish between new assemblies (green bits) and modifications to existing ones (red bits). Although .NET 3.x mostly added new library functionality to the existing .NET 2.0 layer, some updates were required to assemblies that had already shipped. With the distinction between red bits and green bits, development teams kept track of those modifications also to minimize the changes required to red bits to reduce the risk of breaking existing applications.

What all this means in practice is that .NET 3.0 is a superset of .NET 2.0, but with some updates to the .NET 2.0 binaries, in the form of a service pack. Those service packs are also made available by themselves because they contain very valuable fixes and optimizations, and they are designed to be fully backward compatible so as not to break existing code. Windows Update automatically deploys these service packs to machines that already have the framework installed.
NOTE: LOST IN TRANSLATION

Even more remarkable than the capability to add gigantic libraries like WPF and Windows Communication Foundation (WCF) on an already existing runtime without requiring modifications to it is the fact that very powerful language extensions have been made in .NET 3.5 with the introduction of LINQ. However, none of those new language additions required changes to the runtime or intermediate language (IL). Therefore, C# 3.0 and VB 9.0 programs can run on the .NET 2.0 CLR. Even more, it’s theoretically possible to cross-compile C# 3.0 programs into C# 2.0 code with an equivalent meaning. A paper proving this claim was written by a group of language designers at Microsoft and is titled “Lost in Translation.”

One caveat, though: Don’t take this to mean that C# 3.0 programs can be ported blindly to .NET 2.0 because implementations of various LINQ providers ship in various .NET 3.5 assemblies.

Another advantage that comes from keeping the runtime the same across a set of framework releases is the capability to reuse existing tooling infrastructure (for example, for debugging). With the release of Visual Studio 2008, this capability became visible to .NET developers under the form of multitargeting support. What this feature enables is to use
Installing the .NET Framework

Visual Studio 2008 to target .NET Framework 2.0, 3.0, and 3.5 using the same comfortable tooling environment. And with .NET 4.0 and .NET 4.5—as you’ll see later in this chapter when we explore Visual Studio 2012—multitargeting has been extended to support all releases from .NET 2.0 to 4.5.

What about .NET Framework 1.x? Development targeting those platforms will always be tied to the use of the releases of Visual Studio that shipped with it (that is, Visual Studio .NET versions 2002 and 2003). Too many fundamental changes to runtime infrastructure were made between versions 1.x and 2.0 of the CLR, making multitargeting support for .NET 1.x unfeasible. Luckily nowadays, the use of .NET 1.x has largely been phased out. If you still have .NET 1.x applications around, now is the time to port them to a more recent version of the platform (preferably .NET 4.0, of course).

But why should someone care to target an older release of the .NET Framework? Most commonly, the answer is to be found in deployment constraints within companies, web hosting providers, and so on. Having tooling support to facilitate this multitargeting is pure goodness and also means you can benefit from core enhancements to the Visual Studio tools while targeting older releases of the .NET Framework.

.NET Framework 4.5

The particular version of the .NET Framework we target in this book is .NET 4.5, using Visual Studio 2012 and C# 5.0. Other than the .NET 3.x releases, .NET 4.x has a new version of the CLR underneath it, and obviously—in the grand tradition—it comes with a bunch of new class libraries that will make life easier for developers.

Two key features about .NET 4.x are important to point out here:

▶ **Side-by-side support:** This means that .NET 4.x can be installed next to existing versions of the .NET Framework. What’s so special about this compared to .NET 3.x? The key difference is updates to existing class library assemblies are no longer carried out in-place, but new versions are put next to the existing ones.

▶ **Backward compatibility:** This provides tremendous value to developers, allowing reuse of existing code and components. In practice, it means that existing code that was compiled against .NET 2.0 or 3.x in the past can now be targeted at .NET 4.x without requiring source-level code changes.

Figure 3.2 illustrates a machine with all the versions of the .NET Framework installed next to one another.

**NOTE: WHAT’S UP WITH THOSE VERSION NUMBERS?**

The full version numbers of the CLR and .NET Framework installations and binaries can be somewhat distracting at first sight. Where do they come from?

In the .NET Framework 1.x timeframe, the build number (the third component of the version number) was simply created incrementally. Version 1.0 released at build 3705, and version 1.1 ended up at build 4322.
With .NET 2.0, it made sense to give more meaning to the build number, and the pattern ymmd was chosen: one digit for the year (2005), two digits for the month (July), and two for the day (27).

This approach worked very nicely until the theoretical 35th day of the 55th month of the year 2006: The metadata encoding for build numbers cannot exceed 65535, so we’re out of luck using this pattern in its full glory. The result was a compromise. The month and year encodings are kept the same, but the year is now considered relative to the start of the release. For the .NET 4.0 release, the start of the release was in 2007, so from Figure 3.2, one can infer that .NET 4.0 was built on March 19, 2012. Because the .NET 4.5 release is an in-place update, the folder name has stayed the same, even though the revision number of the files in the folder has gone up.

Besides having various versions of the .NET Framework, .NET 4.0 pioneered the availability of different “flavors.” Around the .NET 3.5 timeframe it became apparent that the size of the .NET Framework had grown too large to enable fast friction-free installs, which are especially important for client application deployments. Significant factors for such deployments are download times and installation times.

To streamline typical deployments of the framework, a split of the .NET Framework class libraries was made, factoring out so-called Client Profile assemblies. The Client Profile bubble contained the Base Class Library (BCL) and libraries required to write client applications using WPF and WCF. The remaining part (referred to as Extended Profile) was layered on top of the Client Profile subset and contained features like ASP.NET that client applications typically don’t need. As a result, the deployment and installation footprint of the Client Profile is kept small, while it’s still possible to upgrade such an installation to the full framework. Figure 3.3 shows the layered cake architecture of those profiles in the .NET 4.0 timeframe.
Together with this split, Visual Studio 2010 extended its notion of multitargeting to the various “profiles” of the .NET Framework. By doing so, developers didn’t have to memorize which libraries are available in each profile. When the Client Profile subset is selected, Visual Studio prevents assemblies from the Full framework from being referenced.

During the .NET 4.5 timeframe, a big investment was made to reduce download and installation sizes and to decrease the installation time. As a result, the split of a Client Profile and Extended Profile was no longer necessary, resulting in the discontinuation of the Client Profile package.

.NET 4.5 introduces a new notion of profiles though, through the new Portable Library support. When writing portable class libraries, you can select the target platforms the library can run on. For example, you could build a portable library targeting .NET Framework 4.5, .NET for Windows Store applications, and Windows Phone 8. The resulting project will have access only to the .NET Framework functionality available in the intersection for those platforms, hence ensuring the resulting assembly works on all of the selected platforms. In the past, developers had to maintain separate builds of their libraries for each target platform, causing major grief. Portable Library aims to take away this pain.

To make the intersection of APIs across different target platforms as stable as possible, the bottommost layer of the framework was split off in a “.NET Core” set of assemblies. This foundation is what future releases of the framework are based on, and is used by .NET 4.5 on the desktop, .NET for Windows Store applications, and Windows Phone 8. When you are targeting older platforms, such as Silverlight or .NET 4.0, the set of APIs available in the intersection will differ because of prior differences between different flavors of the .NET Framework. With the .NET Core and Portable Library refactoring in place, the fragmentation of the framework libraries should be reduced substantially.
Running the Installer

Playtime! To write code on the .NET Framework 4.5, let’s start by installing the Full .NET Framework package. That’s really all you need to get started with managed code development.

Where to get it? Browse to http://msdn.com/netframework and click the link to the .NET Framework 4.5 download. The installer itself should be straightforward to run: Accept the license agreement, get a cup of coffee, and you’re ready to go. On Windows 8, you can use the Turn Windows Features On or Off Control Panel applet to enable or disable the .NET Framework 4.5 feature. Figure 3.4 shows the default feature selection on a clean Windows 8 installation.

What Got Installed?

When the installation is complete, it’s good to take a quick look at what was installed to familiarize yourself with where to find stuff.

The Runtime Shim

The runtime shim is really something you shouldn’t care much about, but it’s a convenient way to find out the latest version of the installed CLR on the machine. The purpose of the shim is to load the correct version of the CLR to execute a given application, a particularly important task if multiple versions of the runtime are installed.

You can find the shim under %windir%\system32, with the name mscoree.dll. By looking at the file properties (shown in Figure 3.5), you’ll find out about the latest common language runtime version on the machine.
Although the file description states “Microsoft .NET Runtime Execution Engine,” this is not the CLR itself, so where does the runtime itself live?

**The .NET 4.0 CLR**

Having a runtime shim is one thing; having the runtime itself is invaluable. All runtime installations live side by side in the %windir%\Microsoft.NET\Framework folder. On 64-bit systems, there’s a parallel Framework64 folder structure. Having two “bitnesses” of the CLR and accompanying libraries is required to allow applications to run either as 32-bit (Windows On Windows, or WOW) or 64-bit.

Starting with .NET 4.0, the CLR itself is called clr.dll (previously, mscorwks.dll), as shown in Figure 3.6. The same CLR is used for .NET 4.5, so we’ll refer to it using the 4.0 version number.

**The Global Assembly Cache**

The Global Assembly Cache (GAC) is where class library assemblies are loaded for use in .NET applications. You can view the GAC under %windir%\assembly, but a
command-line directory listing reveals the structure of the GAC in more detail. We discuss the role of the GAC and the impact on your own applications exhaustively in Chapter 25, “Assemblies and Application Domains.”

Figure 3.7 shows the structure of the .NET 4.0 GAC containing the 4.0 version of the System.dll assembly, one of the most commonly used assemblies in the world of managed code development.

![Figure 3.7 Inside the GAC.](image)

**NOTE: GAC SPLITTING**

Notice the v4.0 prefix in the name of the folder containing the .NET 4.0 version of System.dll? This is an artifact of the “GAC splitting” carried out in .NET 4.0. This simple naming trick hides assemblies targeting different versions of the runtime so that a specific version of the CLR doesn’t try to load an assembly that’s incompatible with it. In the preceding example, CLR 2.0 will recognize only the first folder as a valid entry in the GAC, whereas CLR 4.0 recognizes only the second one. This truly shows the side-by-side nature of the different runtimes.

**Tools**

Besides the runtime and class library, a set of tools get installed to the framework-specific %windir%\Microsoft.NET\Framework folder. Although you’ll only use a fraction of those on a regular basis—also because most of those are indirectly used through the Visual Studio 2012 graphical user interface (GUI)—it’s always good to know which tools you have within reach. My favorite tool is, without doubt, the C# compiler, csc.exe. Figure 3.8 shows some of the tools that ship with the .NET Framework installation.

You can find other tools here, too, including other compilers, the IL assembler, MSBuild, the NGEN native image generator tool, and so on.

We explore quite a few of the tools that come with the .NET Framework throughout this book, so make sure to add this folder to your favorites.
Your First Application: Take One

With the .NET Framework installation in place, we’re ready to develop our first .NET application. But wait a minute... where are the development tools to make our lives easy? That’s right, for just this once, we’ll lead a life without development tools and go the hardcore route of Notepad-type editors and command-line compilation to illustrate that .NET development is not tightly bound to the use of specific tools like Visual Studio 2012. Later in this chapter, we get our feet back on the ground and explore the Visual Studio 2012 tooling support, which will become your habitat as a .NET developer moving forward.

NOTE: THE POWER OF NOTEPAD AND THE COMMAND LINE

Personally, I’m a huge fan of coding with the bare minimum tools required. Any text editor, the good old command-line interpreter, and the C# compiler suffice to get the job done. True, colleagues think I endure a lot of unnecessary pain because of this approach, but I’m a true believer.

But why? For a couple of reasons, really. For one, it helps me memorize commonly used APIs; for the more specialized ones, I keep MSDN online open. But more important, the uncooperative editor forces me into a coding mode, where thinking precedes typing a single character.

For any decent-sized project, this approach becomes much less attractive. The ability to navigate code efficiently and use autocomplete features, source control support, an integrated debugging experience, and so on—all these make the use of a professional editor like Visual Studio 2012 invaluable.

However, I recommend everyone go back to the old-fashioned world of Notepad and the command line once in a while. One day, you might find yourself on an editor-free machine solving some hot issue, and the ability to fall back to some primitive development mode will come in handy, for sure. Anyway, that’s my five cents.

So as not to complicate matters, let’s stick with a simple command-line console application for now. Most likely, the majority of the applications you’ll write will either be GUI
applications or web applications, but console applications form a great ground for experimentation and prototyping.

Our workflow for building this first application is as follows:

▶ Writing the code using Notepad
▶ Using the C# command-line compiler to compile it
▶ Running the resulting program

Writing the Code

Clichés need to be honored from time to time, so what's better than starting with a good old Hello World program? Okay, let's make it a little more complicated by making a generalized Hello program, asking for the user's name to show a personalized greeting message.

Open up Notepad, enter the following code, and save it to a file called Hello.cs:

```csharp
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter your name: ");
        string name = Console.ReadLine();
        Console.WriteLine("Hello " + name);
    }
}
```

Make sure to respect the case of letters: C# is a case-sensitive language. In particular, if you come from a Java or C/C++ background, be sure to spell `Main` with a capital `M`. Without delving too deeply into the specifics of the language just yet, let's go over the code quickly.

On the first line, we have a using directive, used to import the `System` namespace. This allows us to refer to the `Console` type further on in the code without having to type its full name `System.Console`.

Next, we're declaring a class named `Program`. The name doesn't really matter, but it's common practice to name the class containing the entry point of the application `Program`. Notice the use of curly braces to mark the start and end of the class declaration.

Inside the `Program` class, we declare a static method called `Main`. This special method is recognized by the common language runtime as the entry point of the managed code program and is where execution of the program will start. Notice the method declaration is indented relative to the containing class declaration. Although C# is not a whitespace-sensitive language, it's good to be consistent about indentation.
Finally, inside the `Main` method we’ve written a couple of statements. The first one makes a method call to the `Write` method on the `Console` type, printing the string "Enter your name:" to the screen. In the second line, we read the user’s name from the console input and assign it to a local variable called `name`. This variable is used in the last statement, where we concatenate it to the string "Hello " using the `+` operator to print it to the console by using the `WriteLine` method on the `Console` type.

**Compiling It**

To run the code, we must compile it because C# is a compiled language (at least in today’s world without an interactive read-eval-print-loop [REPL] loop C# tool). The act of compiling the code results in an assembly ready to be executed on the .NET runtime.

Open a command prompt window and change the directory to the place where you saved the Hello.cs file. As an aside, the use of `.cs` as the extension is not a requirement for the C# compiler; it’s just a best practice to store C# code files as such.

Because the search path doesn’t contain the .NET Framework installation folder, we have to enter the fully qualified path to the C# compiler, `csc.exe`. Recall that it lives under the framework version folder in `%windir%\Microsoft.NET\Framework`. Just run the `csc.exe` command, passing in Hello.cs as the argument, as illustrated in Figure 3.9.

![Running the C# compiler](Figure 3.9)

If the user has installed Visual Studio 2012, a more convenient way to invoke the compiler is from the Visual Studio 2012 command prompt. This specialized command prompt has search paths configured properly such that tools like `csc.exe` will be found.

**NOTE: MSBUILD**

As you’ll see later on, very rarely will you invoke the command-line compilers directly. Instead, MSBuild project files are used to drive build processes.
Running It
The result of the compilation is an executable called hello.exe, meaning that we can run it immediately as a Windows application (see Figure 3.10). This differs from platforms like Java where a separate application is required to run the compiled code.

That wasn’t too hard, was it? To satisfy our technical curiosity, let’s take a look at the produced assembly.

Inspecting Our Assembly with ILSpy
Knowing how things work will make you a better developer. One great thing about the use of an IL format in the .NET world is the capability to inspect compiled assemblies at any point in time without requiring the original source code.

Three commonly used tools to inspect assemblies include the .NET Framework IL disassembler tool (ildasm.exe), .NET Reflector from Red Gate, and ILSpy. For the time being, we’ll use ILSpy, which you can download for free from http://www.ilspy.net.

When you run the tool for the first time, it loads the ILSpy assembly as the assembly to inspect, including all of its dependencies, as shown in Figure 3.11.
Because we’re not interested in the decompilation of ILSpy, we’ll load our own hello.exe using File, Open. This adds “hello” to the list of loaded assemblies, after which we can start to drill down into the assembly’s structure, as shown in Figure 3.12.

Looking at this structure gives us a good opportunity to explain a few concepts briefly. As we drill down in the tree view, we start from the “hello” assembly we just compiled. Assemblies are just CLR concepts by themselves and don’t have direct affinity to file-based storage. Indeed, it’s possible to load assemblies from databases or in-memory data streams, too. Hence, the assembly’s name does not contain a file extension.

When expanding the assembly’s entry, we encounter a node with a {} logo. This indicates a namespace and is a result of ILSpy’s decompilation intelligence, as the CLR does not know about namespaces by itself. Namespaces are a way to organize the structure of APIs by grouping types in a hierarchical tree of namespaces (for example, System.Windows.Forms). To the CLR, types are always referred to—for example, in IL code—by their fully qualified name (like System.Windows.Forms.Label). In our little hello.exe program we didn’t bother to declare the Program class in a separate namespace, so ILSpy shows a “.” to indicate the global namespace.

Finally, we arrive at our Program type with the Main method inside it. Let’s take a look at the Main method now, simply by selecting it from the tree view. Figure 3.13 shows the result.

The pane on the right shows the decompiled code back in C#. It’s important to realize this didn’t use the hello.cs source code file at all. The hello.exe assembly doesn’t have any link back to the source files from which it was compiled. “All” ILSpy does is reconstruct the C# code from the IL code inside the assembly. You can clearly see that’s the case because the name of the name variable was changed into str. ILSpy’s interpretation of our Main method is semantically correct, though; we could have written the code like this.

Notice the drop-down box in the toolbar at the top. Over there, we can switch to other views on the disassembled code (for example, plain IL). Let’s take a look at that, too, as shown in Figure 3.14.
CHAPTER 3   Getting Started with .NET Development Using C#

FIGURE 3.14   IL disassembler for the Main method.

What you’re looking at now is the code as the runtime sees it to execute it. Notice a few things here:

▶ Metadata is stored with the compiled method to indicate its characteristics: .method tells it’s a method, private controls visibility, cil reveals the use of IL code in the method code, and so on.

▶ The execution model of the CLR is based on an evaluation stack, as revealed by the .maxstack directive and naming of certain IL instructions (pop and push, not shown in our little example).

▶ Method calls obviously refer to the methods being called, but observe how there’s no trace left of the C# using-directive namespace import and how all names of
methods are fully qualified (for example, `[mscorlib]System.Console::WriteLine(string)).

Our local variable “name” has lost its name because the execution engine needs to know only about the existence (and type) of local variables, not their names. (The fact that it shows up as `str` is due to ILSpy’s attempt to be smart about making up variable names.)

You might have noticed a few strange things in the IL instructions for the `Main` method: Why are those `nop` (which stands for `no operation`) instructions required? The answer lies in the way we compiled our application, with optimizations turned off. This default mode causes the compiler to preserve the structure of the input code as much as possible to make debugging easier. In this particular case, the curly braces surrounding the `Main` method code body were emitted as `nop` instructions, which allows a breakpoint to be set on that line.

**TIP**

Explore the `csc.exe` command-line options (`/?`) to find a way to turn on optimization and recompile the application. Take a look at the disassembler again (you can press F5 in ILSpy to reload the assemblies from disk) and observe the `nop` instructions are gone.

---

**Visual Studio 2012**

Now that we’ve seen the hardcore way of building applications using plain old text editors and the C# command-line compiler, it’s time to get more realistic by having a look at professional tooling support provided by the Visual Studio 2012 products. Figure 3.15 shows the Visual Studio 2012 logo, reflecting the infinite possibilities of the technology.

**FIGURE 3.15  The Visual Studio 2012 logo.**

Since the very beginning of software development on the Microsoft platform, Visual Studio has been an invaluable tool to simplify everyday development tasks significantly. One core reason for this is its integrated development environment (IDE) concept, which is really an expansive term with an overloaded meaning today. Although it originally stood for the combination of source code editing and debugging support, today’s IDE has capabilities that stretch a whole range of features such as the following:
Source code editing with built-in language support for various languages such as C#, Visual Basic, F# and C++, including things such as syntax coloring, IntelliSense auto-completion, and so on.

Refactoring support is one of the powerful tools that makes manipulating code easier and allows for the restructuring of existing code with just a few clicks in a (mostly) risk-free manner.

Exploring code is what developers do most of their time in the editors. Navigating between source files is just the tip of the iceberg, with the editor providing means to navigate to specific types and members.

Visualization of project structures bridges the gap between architecture, design, and implementation of software. In the spirit of Unified Markup Language (UML), class designers and architecture explorers are available right inside the tool.

Designers come into play when code is to be generated for common tasks that benefit from a visual development approach. Typical examples include GUI design, web page layout, object/relational mappings, workflow diagrams, and so on.

Debugging facilities are the bread and butter for every developer to tame the complexities of analyzing code behavior and (hopefully not too many) bugs by stepping through code and analyzing the state of execution.

Project management keeps track of the various items that are part of a software development project, including source files, designer-generated files, project-level settings, and so on.

Integrated build support is tightly integrated with project management features and allows immediate invocation of the build process to produce executable code and feed build errors and warnings back to the IDE.

Source control and work item tracking are enterprise-level features for managing large-scale collaborative software development projects, providing means to check in/out code, open and close bugs, and so on.

Extensibility might not be the most visible feature of the IDE but it provides a huge opportunity for third parties to provide extensions for nearly every aspect of the tooling support.

Editions

I feel like a marketing guy saying so, but to “differentiate between the needs for various software development groups,” Visual Studio 2012 is available in different editions. For a full overview, we refer to the MSDN website, but here’s a short summary nonetheless:

Visual Studio Express Editions are free downloads targeted at providing rich tooling support to build great apps for the Web, the new Windows 8 platform, Windows Phone, as well as the classic desktop. Each edition comes with language support for C#, Visual Basic, and C++.
Visual Studio Professional Edition is aimed at the professional developer and at small teams. Compared to the Express Editions, it bundles the project types for all application types in one suite, also including support for development of Windows Services, cloud applications, and so on. In addition, rich tooling for testing is included.

Visual Studio Premium Edition extends the Professional Edition by adding tools for agile development teams, using Team Foundation Server (TFS). In addition, tools are included for project management, UI testing, code coverage analysis, lab infrastructure management, and much more.

Visual Studio Ultimate Edition is the largest edition available, and extends on the Premium Edition functionality by adding tooling for historical debugging (IntelliTrace), web performance and load testing, a richer unit test framework (Fakes), tools for architecture diagrams, and so forth.

**NOTE: TLA OF THE DAY: SKU**

The different editions of Visual Studio—and similarly for other products—are often referred to as SKUs by Microsoft representatives. SKU is a TLA, a three-letter acronym that refers to *shelve-kept unit*. It comes from the days software was mostly distributed in cardboard boxes that were kept on shelves in the software store around the corner. Today, though, lots of developers get their tools through downloads, MSDN subscriptions, or enterprise agreements.

In this book, we mainly focus on language and framework-level aspects of programming on the .NET platform, which are separate from the tooling support. However, when covering tooling support, we assume the reader has access to at least the Professional Edition of Visual Studio 2012. This said, many of the features covered (such as debugging support to name an essential one) are available in the Express Edition, too. From time to time, we’ll have a peek at Team System-level features, as well, but in a rather limited fashion.

Oh, and by the way, Visual Studio is available in different (natural) languages beyond just English. However, this book refers to the English vocabulary used in menus, dialogs, and so on.

**NOTE: VISUAL STUDIO SHELL**

In fact, Visual Studio is a highly extensible shell for all sorts of development and management tools. An example of a tool that’s built on the Visual Studio environment is the SQL Server Management Studio. To allow the use of this infrastructure for use by third-party tools, there’s the so-called Visual Studio Shell. One can go even further and embed Visual Studio capabilities in a separate application by using the Visual Studio for Applications (VSTA) platform.
Expression

Applications with GUIs, either for Windows or the Web, are typically not just built by development teams. An important peer to the developer involved in GUI development is a professional designer working on the look and feel for the application’s user experience.

Platform-level technologies like Windows Extensible Application Markup Language (XAML), Windows Presentation Foundation (WPF), Silverlight, and ASP.NET are built with this fact in mind, allowing for rich styling capabilities and a separation between developer code and UI definitions (for example, in terms of XAML). This very powerful concept enables developers and designers to work in parallel with one another.

Although this book focuses on the developer aspect of .NET application development, it’s important to know about the Expression family of tools that your designer peers can use. You can find more information about those tools at http://www.microsoft.com/expression.

Installing Visual Studio 2012

Installation of Visual Studio 2012 should be straightforward. If you are using at least the Professional Edition of the product, check boxes will appear to install managed code/native code development support (see Figure 3.16). Make sure to check the Managed Code option or switch the Options page to the more verbose mode where you can turn individual features on or off.

Depending on the number of features you select (I typically do a full installation to avoid DVD or other install media hunts afterward), installation might take a while. If you don’t already have those installed, various prerequisites, such as the .NET Framework, will get installed as well, potentially requiring a reboot or two. But it’s more than worth the wait.
Once Visual Studio setup has completed, install the product documentation, also known as the Help Library. Although the Visual Studio help system can hook up to the online version of MSDN seamlessly, it’s convenient to have the documentation installed locally if you can afford the disk space. To do so, go to the Start Menu and find the Manage Help Settings entry under the Visual Studio 2012, Visual Studio Tools folder. Figure 3.17 shows the user interface (UI) of this tool, where one can install content from the installation disk or by downloading it.

![Figure 3.17](image)

**FIGURE 3.17** Visual Studio 2012 Documentation configuration.

**A Quick Tour Through Visual Studio 2012**

With Visual Studio 2012 installed, let’s take a quick tour through the IDE you’ll be spending a lot of your time as a developer in.

**What Was Installed**

Depending on the edition you have installed, a number of tools have been installed in parallel with the Visual Studio 2012 editor itself. Figure 3.18 shows the Windows 8 Start screen entry for Visual Studio 2012 for an Ultimate Edition installation on a 64-bit machine. (Obviously, users of previous releases of the operating system will find similar entries in the classic Start menu.) A few notable entries here are as follows:

- Developer Command Prompt for VS2012 provides a command prompt window with several environment variables set, including additions to the search path to locate various tools such as the command-line compilers.
- Remote Debugger is one of my favorite tools when debugging services or other types of applications that run on a remote machine. It enables you to enable debugging applications over the network right from inside Visual Studio 2012.
Go ahead and use the Visual Studio 2012 command prompt to recompile our first application we created earlier in this chapter. You should find that csc.exe is on the search path, so you can simply invoke it without specifying the full path.

Another tool that was installed is ildasm.exe, the IL disassembler. Go ahead and use it to inspect the hello.exe assembly, looking for the Main method’s IL code. Because we’ll be using this tool from time to time, it’s good to know from where you can launch it.

Splash Screen and Start Page

Figure 3.19 shows the Visual Studio 2012 splash screen. Prior to Visual Studio 2010, the splash screen showed the different extensibility packages that were installed. Now this information is available from the Help, About menu.

NOTE: CHOOSE YOUR MOTHER TONGUE

If this is the first time you’ve started Visual Studio 2012, a dialog appears from which you select a settings template to configure the IDE for a specific programming language. You can either stick with a general settings template or indicate your preferred language (your programming mother tongue, so to speak). If you’re presented with this option, feel free to go ahead and select the C# template.

All this means is some settings will be optimized for C# projects (for example, the default language selection in the New Project dialog), but other languages are still available to you at any point in time. Hence the word preference.

The first thing you’ll see in Visual Studio is the Start page shown in Figure 3.20. It provides links to various tasks (for example, to reload recently used projects). An RSS feed shows news items from the MSDN website.

FIGURE 3.20  Visual Studio 2012 Start page.
Core UI Elements

The menu and toolbar contain a wealth of features. (We cover only the essential ones.) Make sure to explore the menu a bit for things that catch your eye. Because we haven’t loaded a project yet, various toolbars are not visible yet.

Various collapsible panels are docked on the borders. There are several of those, but only a few are visible at this point: Toolbox, Solution Explorer, and Error List are the ones we’ll interact with regularly. More panels can be enabled through the View menu, but most panels have a contextual nature and will appear spontaneously when invoking certain actions (for example, while debugging). Figure 3.21 shows how panels can be docked at various spots in the editor. The little pushpin button on the title bar of a panel can be used to prevent it from collapsing. As you get more familiar with the editor, you’ll start rearranging things quite a bit to adapt to your needs.

![Customizing the look and feel by docking panels.](image)

**NOTE: “INNOVATION THROUGH INTEGRATION” WITH WPF**

If you’ve used earlier releases of Visual Studio, you no doubt have noticed the different look and feel of the IDE in the 2010 and 2012 versions. Starting with Visual Studio 2010, large portions of the UI have been redesigned to use WPF technology.

This has several advantages in both the short and long term, and today we’re seeing just the tip of the iceberg of capabilities this unlocks. For example, by having the code editor in WPF, whole new sets of visualizations become possible. To name just one example, imagine what it’d be like to have code comments with rich diagrams in it to illustrate some data flow or architectural design. Also, the rebranding in Visual Studio 2012 was made possible in a relatively short release cycle thanks to the flexibility of XAML and WPF.

It’s worth pointing out explicitly that Visual Studio is a hybrid managed and native (mostly for historical reasons) code application. An increasing number of components are written using managed code, and new extensibility APIs are added using the new Managed Extensibility Framework (MEF). Another great reason to use managed code!
Your First Application: Take Two

To continue our tour through Visual Studio 2012, let’s make things a bit more concrete and redo our little Hello C# application inside the IDE.

New Project Dialog

The starting point to create a new application is the New Project dialog, which can be found through File, New, Project or invoked by Ctrl+Shift+N. A link is available from the Projects tab on the Start Page, too. A whole load of different project types are available, also depending on the edition used and the installation options selected. Actually, the number of available project templates has grown so much over the years that the dialog was redesigned in Visual Studio 2010 to include features such as search.

Because I’ve selected Visual C# as my preferred language at the first start of Visual Studio 2012, the C# templates are shown immediately. (For other languages, scroll down to the Other Languages entry on the left.) Subcategories are used to organize the various templates. Under the Windows category, we find the following commonly used project types:

- Console Application is used to create command-line application executables. This is what we’ll use for our Hello application.
- Class Library provides a way to build assemblies with a .dll extension that can be used from various other applications (for example, to provide APIs).
- Portable Class Library is new in Visual Studio 2012 and is used to create class libraries that can run on multiple .NET Framework flavors (such as Silverlight, Windows Phone, .NET 4.5, and so on).
- Windows Forms Application creates a project for a GUI-driven application based on the Windows Forms technology, targeting the classic Windows desktop.
- WPF Application is another template for GUI applications but based on the new and more powerful WPF framework, also targeting the classic Windows desktop.

Visual Studio 2012 adds the Windows Store category with templates used to build applications targeting the Windows 8 platform:

- Different XAML templates are available as starting points for the GUI design of a Windows Store application (for example, using a grid or a split view).
- Class Library (Windows Store apps) gives you a way to build class library assemblies that you can reuse across different Windows Store app projects.
- Windows Runtime Component allows for the creation of WinMD components using C#. Such components can be used for Windows Store apps built-in managed code, JavaScript, and C++.
- When you are writing web applications, the Web category is a good starting point, providing different templates for ASP.NET-based applications.
We cover other types of templates, too, but for now those are the most important ones to be aware of. Figure 3.22 shows the New Project dialog, where you pick the project type of your choice.

Notice the NET Framework 4.5 drop-down at the top of the dialog. This is where the multitargeting support of Visual Studio comes in. In this list, you can select to target older versions of the framework, all the way back to 2.0. Give it a try and select the 2.0 version of the framework to see how the dialog filters out project types that are not supported on that version of the framework.

For now, keep .NET Framework 4.5 selected, mark the Console Application template, and specify Hello as the name for the project. Notice the Create Directory for Solution check box. Stay tuned. We’ll get to the concept of projects and solutions in a while. Just leave it as is for now. Figure 3.23 shows the result of creating the new project.

Once the project has been created, it is loaded, and the first (and in this case, only) relevant file of the project shows up. In our little console application, this is the Program.cs file containing the managed code entry point.

Notice how an additional toolbar (known as the Text Editor toolbar), extra toolbar items (mainly for debugging), and menus have been made visible based on the context we’re in now.
With the new project created and loaded, make the Solution Explorer (usually docked on the right side) visible, as shown in Figure 3.24. Slightly simplified, Solution Explorer is a mini file explorer that shows all the files that are part of the project. In this case, that’s just Program.cs. Besides the files in the project, other nodes are shown as well:

- Properties provides access to the project-level settings (see later) and reveals a code file called AssemblyInfo.cs that contains assembly-level attributes, something we discuss in Chapter 25.

- References is a collection of assemblies the application depends on. Notice that by default quite a few references to commonly used class libraries are added to the project, also depending on the project type.

**NOTE:** WORRIED ABOUT UNUSED REFERENCES?

People sometimes freak out when they see a lot of unused references. Our simple Hello application will actually use only the System assembly (which contains things such as the basic data types and the Console type), so there are definitely grounds for such a worry. However, rest assured that there’s no performance impact in having unused assembly references because the CLR loads referenced assemblies only when they’re actually used. As time goes on, you’ll become more familiar with the role of the various assemblies that have been included by default.
So, what’s the relation between a solution and a project? Fairly simple: Solutions are containers for one or more projects. In our little example, we have just a single Console Application project within its own solution. The goal of solutions is to be able to express relationships between dependent projects. For example, a Class Library project might be referred to by a Console Application in the same solution. Having them in the same solution makes it possible to build the whole set of projects all at once.

**NOTE: SOURCE CONTROL**

For those of you who’ll be using Visual Studio 2012 in combination with TFS, Solution Explorer is also one of the gateways to source control, enabling you to perform check-in/check-out operations on items in the solution, just to name one thing.

---

**Project Properties**

Although we don’t need to reconfigure project properties at this point, let’s take a quick look at the project configuration system. Double-click the Properties node for our Hello project in Solution Explorer (or right-click and select Properties from the context menu). Figure 3.25 shows the Build tab in the project settings.

As a concrete example of some settings, I’ve selected the Build tab on the left, but feel free to explore the other tabs at this point. The reason I’m highlighting the Build configuration at this point is to stress the relationship between projects and build support, as will be detailed later on.
Time to take a look at the center of our development activities: writing code. Switch back to Program.cs and take a look at the skeleton code that has been provided:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

There are a few differences with the code we started from when writing our little console application manually.
First of all, more namespaces with commonly used types have been imported by means of using directives. Second, a namespace declaration is generated to contain the `Program` class. We talk about namespaces in more detail in the next chapters, so don’t worry about this for now. Finally, the `Main` entry point has a different signature: Instead of not taking in any arguments, it now does take in a string array that will be populated with command-line arguments passed to the resulting executable. Because we don’t really want to use command-line arguments, this doesn’t matter much to us. We discuss the possible signatures for the managed code entry point in Chapter 4, “Language Essentials,” in the section “The Entry Point.”

Let’s write the code for our application now. Recall the three lines we wrote earlier:

```csharp
static void Main()
{
    Console.Write("Enter your name: ");
    string name = Console.ReadLine();
    Console.WriteLine("Hello " + name);
}
```

As you enter this code in the editor, you’ll observe a couple of things. One little feature is auto-indentation, which positions the cursor inside the `Main` method indented a bit more to the right than the opening curly brace of the method body. This enforces good indentation practices (the behavior of which you can control through the Tools, Options dialog). More visible is the presence of IntelliSense. As soon as you type the member lookup dot operator after the `Console` type name, a list of available members appears that filters out as you type. Figure 3.26 shows IntelliSense in action.

![IntelliSense while typing code.](image)

After you’ve selected the `Write` method from the list (note you can press Enter or the spacebar as soon as the desired member is selected in the list to complete it further) and
you type the left parenthesis to supply the arguments to the method call, IntelliSense pops up again showing all the available overloads of the method. You learn about overloading in Chapter 10, “Methods,” so just type the “Enter your name:” string.

IntelliSense will help you with the next two lines of code in a similar way as it did for the first. As you type, notice different tokens get colorized differently. Built-in language keywords are marked with blue, type names (like Console) have a color that I don’t know the name of but that looks kind of lighter bluish, and string literals are colored with a red-brown color. Actually, you can change all those colors through the Tools, Options dialog.

**NOTE: WORRIED ABOUT UNUSED NAMESPACE IMPORTS?**

Just as with unused references, people sometimes freak out when they see a lot of unused namespace imports. Again, this is not something to worry about but for a different reason this time. Namespaces are a compile-time aid only, telling the compiler where to look for types that are used throughout the code. Even though the preceding code has imported the System.Text namespace, you won’t find a trace of it in the compiled code because we’re not using any of the types defined in that namespace.

Agreed, unused namespace imports can be disturbing when reading code, so Visual Studio comes with an option to weed out unused namespace imports by right-clicking the code and selecting Organize Usings, Remove Unused Usings.

If you try this on our code, you’ll see that only the System namespace import remains, and that’s because we’re using the Console type that resides in that namespace. Figure 3.27 shows this handy feature.

![FIGURE 3.27 Reducing the clutter of excessive imported namespaces.](image)

Another great feature about the code editor is its background compilation support. As you type, a special C# compiler is running constantly in the background to spot code defects early. Suppose we have a typo when referring to the name variable; it will show up almost immediately, marked by red squiggles, as shown in Figure 3.28.

If you’re wondering what the yellow border on the left side means, it simply indicates the lines of code you’ve changed since the file was opened and last saved. If you press Ctrl+S to save the file now, you’ll see the lines marked green. This feature helps you find code you’ve touched in a file during a coding session by providing a visual cue, which is quite handy if you’re dealing with large code files.
Build Support

As software complexity grows, so does the build process: Besides the use of large numbers of source files, extra tools are used to generate code during a build, references to dependencies need to be taken care of, resulting assemblies must be signed, and so on. You probably don’t need further convincing that having integrated build support right inside the IDE is a great thing.

In Visual Studio, build is integrated tightly with the project system because that’s ultimately the place where source files live, references are added, and properties are set. To invoke a build process, either use the Build menu (see Figure 3.29) or right-click the solution or a specific project node in Solution Explorer. A shortcut to invoke a build for the entire solution is F6.

TIP

Although launching a build after every few lines of code you write might be tempting, I recommend against such a practice. For large projects, this is not feasible because build times might be quite long (though C# code compilation is relatively fast); but more important, this style has the dangerous potential of making developers think less about the code they write.

Personally, I try to challenge myself to write code that compiles immediately without errors or even warnings. The background compilation support for C# in the code editor helps greatly to achieve this goal, catching silly typos early, leaving the more fundamental code flaws something to worry about.
Behind the scenes, this build process figures out which files need to compile, which additional tasks need to be run, and so on. Ultimately, calls are made to various tools such as the C# compiler. This is not a one-way process: Warnings and errors produced by the underlying tools are bubbled up through the build system into the IDE, allowing for a truly interactive development experience. Figure 3.30 shows the Error List pane in Visual Studio 2012.

Starting with Visual Studio 2005, the build system is based on a .NET Framework technology known as MSBuild. One of the rationales for this integration is to decouple the concept of project files from exclusive use in Visual Studio. To accomplish this, the project file (for C#, that is a file with a .csproj extension) serves two goals: It’s natively recognized by MSBuild to drive build processes for the project, and Visual Studio uses it to keep track of the project configuration and all the files contained in it.

To illustrate the project system, right-click the project node in Solution Explorer and choose Unload Project. Next, select Edit Hello.csproj from the same context menu (see Figure 3.31).

In Figure 3.32, I’ve collapsed a few XML nodes in the XML editor that is built into Visual Studio. As you can see, the IDE is aware of many file formats. Also notice the additional menus and toolbar buttons that have been enabled as we’ve opened an XML file.
From this, we can see that MSBuild projects are XML files that describe the structure of the project being built: what the source files are, required dependencies, and so forth. Visual Studio uses MSBuild files to store a project’s structure and to drive its build. Notable entries in this file include the following:

- The Project tag specifies the tool version (in this case, version 4.0 of the .NET Framework tools, including MSBuild itself), among other build settings.
- PropertyGroups are used to define name-value pairs with various project-level configuration settings.
- ItemGroups contain a variety of items, such as references to other assemblies and the files included in the project.
Using an `Import` element, a target file is specified that contains the description of how to build certain types of files (for example, using the C# compiler).

You’ll rarely touch up project files directly using the XML editor. However, for advanced scenarios, it’s good to know it’s there.

Now that you know how to inspect the MSBuild project file, go ahead and choose Reload Project from the project’s node context menu in Solution Explorer. Assuming a successful build (correct the silly typo illustrated before), where can the resulting binaries be found? Have a look at the project’s folder, where you’ll find a subfolder called bin. Underneath this one, different build flavors have their own subfolder. Figure 3.33 shows the Debug build output.

For now, we’ve just built one particular build flavor: Debug. Two build flavors, more officially known as solution configurations, are available by default. In Debug mode, symbol files with additional debugging information are built. In Release mode, that’s not the case, and optimizations are turned on, too. This is just the default configuration, though: You can tweak settings and even create custom configurations altogether. Figure 3.34 shows the drop-down list where the active project build flavor can be selected.
NOTE: THE ROLE OF PDB FILES IN MANAGED CODE

In the introductory chapters on the CLR and managed code, we stressed the important role metadata plays, accommodating various capabilities such as IntelliSense, rich type information, reflection facilities, and so on. Given all this rich information, you might wonder how much more information is required to support full-fledged debugging support. The mere fact that managed code assemblies still have PDB files (Program Database files) reveals there’s a need for additional “debugging symbols.” One such use is to map compiled code back to lines in the sources. Another one is to keep track of names of local variable names, something the CLR doesn’t provide metadata storage for.

One of the biggest advantages of the MSBuild technology is that a build can be done without the use of Visual Studio or other tools. In fact, MSBuild ships with the .NET Framework itself. Therefore, you can take any Visual Studio project (since version 2005, to be precise) and run MSBuild directly on it. That’s right: Visual Studio doesn’t even need to be installed. Not only does this allow you to share your projects with others who might not have the IDE installed, but it also makes automated build processes possible (for example, by TFS). Because you can install TFS on client systems nowadays, automated (that is, nightly) build of personal projects becomes available for individual professional developers, too.

In fact, MSBuild is nothing more than a generic build task execution platform that has built-in notions of dependency tracking and timestamp checking to see what parts of an existing build are out of date (to facilitate incremental, and hence faster, builds). The fact it can invoke tools such as the C# compiler is because the right configuration files, so-called target files, are present that declare how to run the compiler. Being written in managed code, MSBuild can also be extended easily. See the MSDN documentation on the subject for more information.

To see a command-line build in action, open a Developer Command Prompt for VS2012 from the Start menu, change the directory to the location of the Hello.csproj file, and invoke msbuild.exe (see Figure 3.35). The fact there’s only one recognized project file extension causes MSBuild to invoke the build of that particular project file.

Because we already invoked a build through Visual Studio for the project before, all targets are up-to-date, and the incremental build support will avoid rebuilding the project altogether.

TIP

Want to see a more substantial build in action? First clean the project’s build output by invoking msbuild /target:clean. Next, you can simply rebuild by issuing the msbuild command again.

To convince yourself the C# compiler got invoked behind the scenes, turn on verbose logging by running msbuild /clp:verbosity=detailed. This causes a spew of output to be emitted to the console, in which you’ll find an invocation of csc.exe with a bunch of parameters.
Debugging Support

One of the first features that found a home under the big umbrella of the IDE concept was integrated debugging support on top of the editor. This is obviously no different in Visual Studio 2012, with fabulous debugging support facilities that you’ll live and breathe on a day-to-day basis as a professional developer on the .NET Framework.

The most commonly used debugging technique is to run the code with breakpoints set at various places of interest, as shown in Figure 3.36. Doing so right inside a source code file is easy by putting the cursor on the line of interest and pressing F9. Alternative approaches include clicking in the gray margin on the left or using any of the toolbar or menu item options to set breakpoints.

To start a debugging session, press F5 or click the button with the VCR Play icon. (Luckily, Visual Studio is easier to program than such an antique and overly complicated device.) Code will run until a breakpoint is encountered, at which point you’ll break in the debugger, as illustrated in Figure 3.37.
Notice a couple of the debugging facilities that have become available as we entered the debugging mode:

- The Call Stack pane shows where we are in the execution of the application code. In this simple example, there’s only one stack frame for the Main method, but in typical debugging sessions, call stacks get much deeper. By double-clicking entries in the call stack list, you can switch back and forth between different stack frames to inspect the state of the program.

- The Locals pane shows all the local variables that are in scope, together with their values. More complex object types will result in more advanced visualizations and the ability to drill down into the internal structure of the object kept in a local variable. Also, when hovering over a local variable in the editor, its current value is shown to make inspection of program state much easier.
The Debug toolbar has become visible, providing options to continue or stop execution and step through code in various ways: one line at a time, stepping into or over methods calls, and so on.

More advanced uses of the debugger are sprinkled throughout this book, but nevertheless let’s highlight a few from a 10,000-foot view:

- The Immediate window enables you to evaluate expressions, little snippets of code. This way, you can inspect more complex program state that might not be immediately apparent from looking at individual variables. For example, you could execute a method to find out about state in another part of the system.
- The Breakpoints window simply displays all breakpoints currently set and provides options for breakpoint management: the ability to remove breakpoints or enable/disable them.
- The Memory window and Registers window are more advanced means of looking at the precise state of the machine by inspecting memory or processor registers. In the world of managed code, you won’t use those very often.
- The Disassembly window can be used to show the processor instructions executed by the program. Again, in the world of managed code this is less relevant (recall the role of the Just-in-Time [JIT] compiler), but all in all the Visual Studio debugger is usable for both managed and native code debugging.
- The Threads window shows all the threads executing in a multithreaded application. Since .NET Framework 4, new concurrency libraries have been added to System.Threading and new Parallel Stacks and Parallel Tasks windows have been added to assist in debugging those, too.

Debugging is not necessarily initiated by running the application straight from inside the editor. Instead, you can attach to an already running process, even on a remote machine, using the Remote Debugger.

Visual Studio 2010 introduced the IntelliTrace feature, which enables a time-travel mechanism to inspect the program’s state at an earlier point in the execution (for example, to find out about some state corruption that happened long before a breakpoint was hit).

### NOTE: ALTERNATIVE DEBUGGERS

The Visual Studio IDE is not the only debugger capable of dealing with managed code, although it’s likely the most convenient one due to its rich graphical nature, which allows direct visual inspection of various pieces of state and such.

Command-line savvy developers on the Windows platform will no doubt have heard about CDB and its graphical counterpart, WinDbg. Both are available from the Microsoft website as separate downloads, known as the Debugger Tools for Windows.

Although the original target audience for CDB and WinDbg consists of Win32 native code developers and driver writers, an extension for managed code ships right with the .NET Framework. This debugger extension is known as SOS, which stands for Son of Strike,
with Strike being an old code name for the CLR. You can find it under the framework installation folder in a file called sos.dll. We take a look at the use of SOS sporadically—for example, in Chapter 18, “Events,” to debug a memory leak in the sidebar called “Using SOS to Trace Leaks.”

Besides SOS, there’s also a purely managed code debugger called MDbg, which stands for Managed Debugger. This one, too, comes as a command-line debugger. Originally meant as an example to illustrate the use of the CLR debugger APIs, I find it a useful tool from time to time when I don’t have Visual Studio installed.

Given the typical mix of technologies and tools applications are written with nowadays, it’s all-important to be able to flawlessly step through various types of code during the same debugging session. In the world of managed code, one natural interpretation of this is the ability to step through pieces of code written in different managed languages, such as C# and Visual Basic. Visual Studio goes even further by providing the capability to step through other pieces of code: T-SQL database stored procedures, workflow code in Windows Workflow Foundation (WF), JavaScript code in the browser, and so on. Core pillars enabling this are the capability to debug different processes simultaneously (for example, a web service in some web server process, the SQL Server database process, the web browser process running JavaScript) and the potential for setting up remote debugging sessions.

Object Browser

With the .NET Framework class libraries ever growing and other libraries being used in managed code applications, the ability to browse through available libraries becomes quite important. You’ve already seen IntelliSense as a way to show available types and their available members, but for more global searches, different visualizations are desirable. Visual Studio’s built-in Object Browser is one such tool (see Figure 3.38).

![Object Browser visualizing the System.Core assembly.](Figure 3.38)
This tool feels a lot like ILSpy, with the ability to add assemblies for inspection, browse namespaces, types, and members, and a way to search across all of those. It doesn’t have decompilation support, though.

**NOTE: .NET FRAMEWORK SOURCE CODE**

Want to see the .NET Framework source code itself? This has been a longstanding request from the community to help boost the understanding of framework functionality and, in answer to this request, Microsoft has started to make parts of the source code for the .NET Framework available through a shared source program starting from .NET 3.5. Even more so, Visual Studio has been enhanced to be able to step through .NET Framework source code available from the Microsoft servers.


**Code Insight**

An all-important set of features that form an integral part of IDE functionality today is what we can refer to collectively as “code insight” features. No matter how attractive the act of writing code may look—because that’s what we, developers, are so excited about, aren’t we?—the reality is we spend much more time reading existing code in an attempt to understand it, debug it, or extend it. Therefore, the ability to look at the code from different angles is an invaluable asset to modern IDEs.

**NOTE**

For the examples that follow, we shy away from our simple Hello application because its simplicity does not allow us to illustrate more complex software projects. Instead, we use one of the sample applications that you can download through the Help, Samples menu in Visual Studio 2012. In this dialog, search for “LINQ Sample Queries” to locate the C# sample project that illustrates the use of LINQ.

To start with, three closely related features are directly integrated with the code editor through the context menu, shown in Figure 3.39. These enable navigating through source code in a very exploratory fashion.

Go To Definition simply navigates to the place where the highlighted “item” is defined. This could be a method, field, local variable, and so on. We talk about the meaning of those terms in the next few chapters.

Find All References is similar in nature but performs the opposite operation: Instead of finding the definition site for the selection, it looks for all use sites of it. For example, when considering changing the implementation of some method, you better find out who’s using it and what the impact of any change might be.
View Call Hierarchy was added in Visual Studio 2010 and somewhat extends upon the previous two in that it presents the user with a hierarchical representation of outgoing and incoming calls for a selected member. Figure 3.40 shows navigation through some call hierarchy.

So far, we’ve been looking at code with a fairly local view: hopping between definitions, tracing references, and drilling into a hierarchy of calls. Often, you want to get a more global view of the code to understand the bigger picture. Let’s zoom out gradually and explore more code exploration features that make this task possible.

Another addition in Visual Studio 2010 was the support for sequence diagrams, which can be generated using Generate Sequence Diagram from the context menu in the code.
editor. People familiar with UML notation will immediately recognize the visualization of sequence diagrams. They enable you to get an ordered idea of calls being made between different components in the system, visualizing the sequencing of such an exchange.

Notice that the sequence diagrams in Visual Studio are not passive visualizations. Instead, you can interact with them to navigate to the corresponding code if you want to drill down into an aspect of the implementation. This is different from classic UML tools where the diagrams are not tightly integrated with an IDE. Figure 3.41 shows a sequence diagram of calls between components.

FIGURE 3.41 A simple sequence diagram.

To look at a software project from a more macroscopic scale, you can use the Class Diagram feature in Visual Studio, available since version 2008. To generate such a diagram, right-click the project node in Solution Explorer and select View Class Diagram. The Class Diagram feature provides a graphical veneer on top of the project’s code, representing the defined types and their members, as well as the relationships between those types (such as object-oriented inheritance relationships, as discussed in Chapter 14, “Object-Oriented Programming”).

Once more, this diagram visualization is interactive, which differentiates it from classical approaches to diagramming of software systems. In particular, the visualization of the various types and their members is kept in sync with the underlying source code so that documentation never diverges from the actual implementation. But there’s more. Besides visualization of existing code, you can use the Class Diagram feature to extend existing code or even to define whole new types and their members. Using Class Diagrams you can do fast prototyping of rich object models using a graphical designer. Types generated by the designer will have stub implementations of methods and such, waiting for code to be
supplied by the developer at a later stage. Figure 3.42 shows the look and feel of the Class Diagram feature.

![Image of a class diagram](https://via.placeholder.com/150)

**FIGURE 3.42** A class diagram for a simple type hierarchy.

Other ways of visualizing the types in a project exist. We’ve already seen the Object Browser as a way to inspect arbitrary assemblies and search for types and their members. In addition to this, there’s the Class View window that restricts the view to the projects in the current solution. A key difference is this tool’s noninteractive nature: It’s a one-way visualization of types.

Finally, to approach a solution from a high-level view, there’s the Architecture Explorer (illustrated in Figure 3.43). This one can show the various projects in a solution and the project items they contain, and you can drill down deeper into the structure of those items (for example, types and members). By now, it should come as no surprise this view on the world is kept in sync with the underlying implementation, and the designer can be used to navigate to the various items depicted. What makes this tool unique is its rich analysis capabilities, such as the ability to detect and highlight circular references, unused references, and so on.
NOTE: IT’S AN ML WORLD: DGML

Designer tools are typically layered on top of some markup language (ML); for example, web designers visualize HTML, and in WPF and WF, they use XAML. This is no different for the Architecture Explorer’s designer, which is based on a new format called DGML, for Directed Graph Markup Language. In essence, it describes a graph structure based on nodes and links and hence can be used for a variety of tools that require such graph representations/visualizations.

Integrated Help

During the installation of Visual Studio 2012, I suggested that you install the full MSDN documentation locally using the Manage Help Settings utility. Although this is not a requirement, it’s convenient to have a wealth of documentation about the tools, framework libraries, and languages at your side at all times.

Although you can launch the MSDN library directly from the Start menu by clicking the Microsoft Visual Studio 2012 Documentation entry, more regularly you’ll invoke it through the Help menu in Visual Studio or by means of the context-sensitive integrated help functionality. Places where help is readily available from the context (by pressing F1)
include the Error List (to get information on compiler errors and warnings) and the code
editor itself (for lookup of API documentation). Notice that starting with Visual Studio
2012, documentation is provided through the browser rather than a standalone applica-
tion. This mirrors the online MSDN help very closely.

**NOTE: COMMUNITY CONTENT**

Online MSDN documentation at msdn.microsoft.com has a more recent feature addition,
allowing users to contribute content in some kind of wiki style. For example, if the use of
a certain API is not as straightforward as you might expect, chances are good that some
other user has figured it out and shared it with the world over there.

**Designers**

Since the introduction of Visual Basic 1.0 (as early as 1991), Rapid Application
Development (RAD) has been a core theme of the Microsoft tools for developers. Rich
designers for UI development are huge time savers over a coding approach to accomplish
the same task. This was true in the world of pure Win32 programming and still is today,
with new UI frameworks benefiting from designer support. But as you will see, designers
are also used for a variety of other tasks outside the realm of UI programming.

**Windows Forms**

In .NET 1.0, Windows Forms (WinForms) was introduced as an abstraction layer over the
Win32 APIs for windowing and the common controls available in the operating system.
By nicely wrapping those old dragons in the `System.Windows.Forms` class library, the
creation of UIs became much easier. And this is not just because of the object-oriented
veneer provided by it, but also because of the introduction of new controls (such as the
often-used `DataGrid` control) and additional concepts, such as data binding to bridge
between data and representation.

Figure 3.44 shows the Windows Forms designer in the midst of designing a UI for a simple
greetings program. On the left, the Toolbox window shows all the available controls we
can drag and drop onto the designer surface. When we select a control, the Properties
window on the right shows all the properties that can be set to control the control’s
appearance and behavior.

To hook up code to respond to various user actions, you can create event handlers
through that same Properties window by clicking the “lightning” icon on the toolbar.
Sample events include Click for a button, TextChanged for a text box, and so on. And
the most common event for each control can be wired up by simply double-clicking the
control. For example, double-clicking the selected button produces an event handler for
a click on Say Hello. Now we find ourselves in the world of C# code again, as shown in
Figure 3.45.
The straightforward workflow introduced by Windows Forms turned it into a gigantic success right from the introduction of the .NET Framework. Although we now have the Windows Presentation Foundation (WPF) as a new and more modern approach to UI development, there are still lots of Windows Forms applications out there. (So it's in your interest to know a bit about it.)

**NOTE: CODE GENERATION AND THE C# LANGUAGE**

You might be wondering how the UI definition for the previous WinForms application is stored. Is there a special on-disk format to describe graphical interfaces, or what? In the world of classic Visual Basic, this was the case with .frm and .frx files. With WinForms, though, the answer is no: The UI definition is simply stored as generated C# (or VB) code,
using the System.Windows.Forms types, just as you’d do yourself if you were to define a UI without the use of the designer. Actually, the designer is a live instance of your UI but with certain interactions rewired to the designer’s functionality (for example, when clicking a button, it gets selected).

So where does this code live? In the screenshot with the event handler method; notice the call to InitializeComponent in the constructor of the Form class. When you right-click the call and Go to Definition, you’ll see another code file opens with the extension .designer.cs:

```csharp
#region Windows Form Designer generated code

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.label1 = new System.Windows.Forms.Label();
    this.button1 = new System.Windows.Forms.Button();
    this.textBox1 = new System.Windows.Forms.TextBox();

Here you’ll find code that sets all the properties on the controls, adds them to the form, wires up event handlers, and more.

Notice that the XML document comment on top of the InitializeComponent method saying not to modify this code as it gets generated by the graphical designer and changes will get overridden (at best) or might confuse the designer resulting in weird behavior. Why is this important to point out? Well, the first release of the designer in .NET 1.0 had to use the first version of the C# language. Nothing wrong with that, of course, except for the fact that the generated code had to be emitted to the same file as the one containing the event handlers’ code. Although technically challenging to ensure the user’s code is not tampered with when updating the generated code, there was a bigger flaw. Developers, curious as they are, felt tempted to tweak the generated code from time to time, despite the warning comment on top of it, sometimes with disastrous results. As a way to mitigate this (partly), code was emitted inside a #region preprocessor directive to collapse it in the code editor, hiding it from the developer by default.

A better way to deal with this situation was highly desirable, and the solution came online in the .NET Framework 2.0 with the introduction of C# 2.0’s partial classes. In essence, a partial class allows the definition of a class to be spread across multiple files. Windows Forms was one of the first tools to take advantage of this by emitting generated code to a separate file (with a .designer.cs extension) while keeping user-written code elsewhere. In this regard, notice the partial keyword on the class definition shown in Figure 3.45. As an implication, the designer can always rewrite the entire generated file, and the generated code file can be hidden from the user more efficiently. Actually, just for that reason, by default Solution Explorer doesn’t show this file.

With this, we finish our discussion of Windows Forms for now and redirect our attention to its modern successor: WPF.
Windows Presentation Foundation

With the release of the .NET Framework 3.0 (formerly known as WinFX), a new UI platform was introduced: Windows Presentation Foundation. WPF solves a number of problems:

- Mixed use of various UI technologies, such as media, rich text, controls, vector graphics, and so on, was too hard to combine in the past, requiring mixed use of GDI+, DirectX, and more.
- Resolution independence is important to make applications that scale well on different form factors.
- Decoupled styling from the UI definition allows you to change the look and feel of an application on-the-fly without having to rewrite the core UI definition.
- A streamlined designer-developer interaction is key to delivering compelling user experiences because most developers are not very UI savvy and want to focus on the code rather than the layout.
- Rich graphics and effects allow for all sorts of UI enrichments, making applications more intuitive to use.

One key ingredient to achieve these goals—in particular the collaboration between designers and developers—is the use of XAML. In essence, XAML is a way to use XML for creating object instances (for example, to represent a UI definition). The use of such a markup language allows true decoupling of the look and feel of an application from the user’s code. As you can probably guess by now, Visual Studio has an integrated designer (code named Cider) for WPF (see Figure 3.46).

![The integrated WPF designer.](image-url)
As in the Windows Forms designer, three core panes are visible: the Toolbox window containing controls, the Properties window with configuration options for controls and the ability to hook up event handlers, and the designer sandwiched in between.

One key difference is in the functionality exposed by the designer. First of all, observe the zoom slider on the left, reflecting WPF’s resolution-independence capabilities. A more substantial difference lies in the separation between the designer surface and the XAML view at the bottom. With XAML, no typical code generation is involved at design time. Instead, XAML truly describes the UI definition in all its glory.

Based on this architecture, it’s possible to design different tools (such as Expression Blend) that allow refinement of the UI without having to share out C# code. The integrated designer therefore provides only the essential UI definition capabilities, decoupling more-involved design tasks from Visual Studio by delegating those to the more-specialized Expression Blend tool for use by professional graphical designers.

Again, double-clicking the button control generates the template code for writing an event handler to respond to the user clicking it. Although the signature of the event handler method differs slightly, the idea is the same. Figure 3.47 shows the generated empty event handler for a WPF event.

Notice, though, that there’s still a call to InitializeComponent in the Window1 class’s constructor. But didn’t I just say there’s no code generation involved in WPF? That’s almost true, and the code generated here does not contain the UI definition by itself. Instead, it contains the plumbing required to load the XAML file at runtime, to build up the UI. At the same time, it contains fields for all the controls that were added to the UI for you to be able to address them in code. This generated code lives in a partial class definition stored in a file with a .g.i.cs extension, as illustrated in Figure 3.48. To see this generated code file, toggle the Show All Files option in Solution Explorer.

Notice how the XAML file (which gets compiled into the application’s assembly in a binary format called Binary Application Markup Language [BAML]) is loaded through the generated code. From that point on, the XAML is used to instantiate the UI definition, ready for it to be displayed by WPF’s rendering engine.
As an aside, you can actually create WPF applications without using XAML at all by creating instances of the window and control types yourself. In other words, there’s nothing secretive about XAML; it’s just a huge convenience not to have to go through the burden of defining objects by hand.

**NOTE: LIGHTING UP THE WEB WITH SILVERLIGHT**

There’s no reason why the advantages of WPF with regard to designer support, rich graphics layout capabilities, and so on should not be extended to the Web. That’s precisely what Silverlight is about. Originally dubbed WPF/E, for WPF Everywhere, Silverlight is a cross-platform (Windows, Mac, Linux) and cross-browser (Internet Explorer, Firefox, Safari) subset of the CLR and .NET Framework class libraries (including WPF) that you can use to create rich Web experiences. In the field of UI design, it shares a lot of the WPF concepts, including the use of XAML to establish a designer-developer collaboration foundation. Given all of this, it’s very straightforward for WPF developers to leverage Silverlight and vice versa.

Since Visual Studio 2010, Silverlight project support has been added to the IDE, requiring only additional installation of the Silverlight SDK.

A little tidbit for geeks: The main Silverlight in-process browser DLL is called agcore, as a subtle hint to the chemical symbol for silver. I’ll leave it to your imagination to figure out what was first: agcore or the public Silverlight name.

**Windows Workflow Foundation**

A more specialized technology, outside the realm of UI programming, is the Windows Workflow Foundation (abbreviated WF, not WWF, to distinguish from a well-known organization for the conservation of the environment). Workflow-based programming enables the definition and execution of business processes, such as order management, using graphical tools. The nice thing about workflows is they have various runtime services to support transaction management, long-running operations (that can stretch multiple hours, day, weeks or even years), and so on.

The reason I’m mentioning WF right after WPF is the technology they have in common: XAML. In fact, XAML is a generic language to express object definitions using an
XML-based format, which is totally decoupled from UI specifics. Because workflow has a similar declarative nature, it just made sense to reuse the XAML technology in WF, as well (formerly dubbed XOML, for Extensible Orchestration Markup Language).

Figure 3.49 shows the designer of WF used to define a sequential workflow.

The golden triad (Toolbox, Properties, and designer) is back again. This time in the Toolbox you don’t see controls but so-called activities with different tasks, such as control flow, transaction management, sending and receiving data, invoking external components (such as PowerShell), and so on. Again, the Properties window is used to configure the selected item. In this simple example, we receive data from an operation called AskUserName, bind it to the variable called name, and feed it in to a WriteLine activity called SayHello. The red bullet next to SayHello is a breakpoint set on the activity for interactive debugging, illustrating the truly integrated nature of the workflow designer with the rest of the Visual Studio tooling support.

For such a simple application it’s obviously overkill to use workflow, but you get the idea. A typical example of a workflow-driven application is order management, where orders might need (potentially long-delay) confirmation steps, interactions with credit card payment services, sending out notifications to the shipping facilities, and so on. Workflow provides the necessary services to maintain this stateful long-running operation, carrying out suspend and resume actions with state (de)hydration when required.

**NOTE: WPF STRIKES AGAIN**

Not only is Visual Studio presented using WPF technology, the new workflow designer is too. This clearly illustrates the richness that WPF can provide. Actually, the workflow designer can be rehosted in custom applications, too.
Also introduced right from the inception of the .NET Framework is ASP.NET, the server-side web technology successor to classic Active Server Pages (ASP). Core differences between the old and the new worlds in web programming with ASP-based technologies include the following:

- Support for rich .NET languages, leveraging foundations of object-oriented programming, eliminating the use of server-side script as with VBScript in classic ASP.
- First-class notion of controls that wrap the HTML and script aspects of client-side execution.
- Related to control support is the use of an event-driven approach to control interactions with the user, hiding the complexities of HTTP postbacks or AJAX script to make callbacks to the server.
- Various aspects, such as login facilities, user profiles, website navigation, and so on, have been given built-in library support to eliminate the need for users to reinvent the wheel for well-understood tasks. An example is the membership provider taking care of safe password storage, providing login and password reset controls, and so on.
- Easy deployment due to the .NET's xcopy vision. For instance, when requiring a class library to be deployed to the server, there's no need to perform server-side registrations in the world of .NET.
- A rich declarative configuration system makes deployment of web applications easier, having settings stored in a file that's deployed with the rest of the application over any upload mechanism of choice.

From the Visual Studio point of view, ASP.NET has rich project support with a built-in designer and deployment facilities. Figure 3.50 shows ASP.NET's page designer.

By now, designers should start to look very familiar. This time around, the markup is stored in HTML, containing various ASP.NET controls with an <asp: prefix. The runat attribute set to server reveals the server-side processing involved, turning those controls into browser-compatible markup:

```html
<asp:Button ID="Button1" runat="server" Text="Say Hello" />
```

Again, the Toolbox contains a wealth of usable controls available for web development, and the Properties window joins the party to assist in configuring the controls with respect to appearance, behavior, data binding, and more. Starting with Visual Studio 2012, the web page designer only shows the HTML and ASP.NET markup. No visual designer is included anymore, in favor of the separate Expression Web tool.

Hooking up event handlers is done from the markup view, by adding an attribute to the control, pointing at the handler method that can be generated on-the-fly. Figure 3.51 shows the result of adding a Click handler to a Button control. What goes on behind the scenes is much more involved. Although you still write managed code, ASP.NET wires up
event handlers through postback mechanisms at runtime. With the introduction of AJAX, various postback operations can be made asynchronous as well. By doing so, no whole page refreshes have to be triggered by postback operations, improving the user experience a lot.

To simplify testing ASP.NET applications, a lightweight version of Internet Information Services (IIS), called IIS Express, comes with Visual Studio 2012. Figure 3.52 shows the notification area icon for IIS Express used in a debugging session (by a press of F5, for example).
Debugging ASP.NET applications is as simple as debugging any regular kind of application, despite the more complex interactions that happen under the covers. In the latest releases of Visual Studio, support has been added for richer JavaScript debugging as well, making the debugging experience for web applications truly end to end.

Different application models to write web applications exist. This quick tour showed you the oldest approach using web forms. More recent additions to the ASP.NET stack include several versions of the MVC framework. Refer to books on ASP.NET for in-depth information.

Visual Studio Tools for Office

Office programming has always been an area of interest to lots of developers. With the widespread use of Office tools, tight integration with those applications provides an ideal interface to the world for business applications. Originally shipped as a separate product, Visual Studio Tools for Office (VSTO) is now integrated with Visual Studio and has support to create add-ins for the Office 2007 and later versions of Word, Excel, Outlook, PowerPoint, Visio, and InfoPath. Support for SharePoint development has been added, as well, significantly simplifying tasks like deployment, too.

One of the designer-related innovations in Visual Studio 2012 is built-in support to create Office ribbon extensions, as shown in Figure 3.53.

NOTE: C# 4.0 DYNAMIC IN THE WORLD OF VSTO

Visual Studio 2010 and .NET Framework 4.0 are great releases for developers who target Office. With the underlying Office APIs written in COM, use from inside C# has always been quite painful due to the lack of optional and named parameters, the required use of “by ref” passing for all sorts of parameters, and the loose typing of the Office APIs. Because of all this, C# code targeting the Office APIs has always looked quite cumbersome.

C# 4.0 eliminates all those problems, making the code look as it was intended to in the world of the Office COM-based APIs. In addition, one of the core features that makes this possible—dynamic typing—proves useful in lots of other domains, too.

Furthermore, there’s the concept of No PIA (primary interop assembly), significantly improving the deployment story for managed Office add-ins. PIAs contain wrappers for the Office APIs but can be quite large (in the order of several megabytes). Previously, those
needed to be deployed together with the application and were loaded into memory as a whole at runtime. With the No PIA feature, the used portions of the PIAs can be linked in to the application’s assembly, eliminating the deployment burden and reducing the memory footprint.

![Server Explorer in Visual Studio 2012](image)

**FIGURE 3.53** Ribbon designer support in Visual Studio 2012.

**Server Explorer**

Modern software is rarely ever disconnected from other systems. Database-driven applications are found everywhere, and so are an increasing number of service-oriented applications. Server Explorer is one of the means to connect to a server, explore aspects of it, and build software components that are used to interact with the system in question. Figure 3.54 shows one view of Server Explorer, when dealing with database connections. Adding a Component file to the project, you get an empty design surface ready for drag and drop of different types of server objects.

Server Explorer has built-in support for a variety of commonly used server-side technologies, including the following:

- A variety of database technologies, with support for SQL Server, Access, Oracle, OLEDB, and ODBC. Connecting to a database visualizes things such as tables and stored procedures.
- Event logs are useful from a management perspective both for inspection and the emission of diagnostic information during execution of the program. .NET has rich support to deal with logging infrastructure.
- Management Classes and Events are two faces for the Windows Management Instrumentation (WMI) technology, allowing for thorough querying and modification of the system’s configuration.
Message queues enable reliable, possibly offline, communication between machines using the Microsoft Message Queuing (MSMQ) technology. To send and receive data to and from a queue, a mapping object can be made.

Performance counters are another cornerstone of application manageability, providing the capability to emit diagnostic performance information to counters in the system (for example, the number of requests served per second by a service).

The Services node provides a gateway to management of Windows Services, such as querying of installed services, their states, and configuration and to control them. In fact, C# can even be used to write managed code OS services.

For example, in Figure 3.55, a component designer was used to create a management component containing management objects for a Windows server, a performance counter, and an event log. No code had to be written manually thanks to the drag-and-drop support from the Server Explorer onto the designer surface. You can use the Properties window to tweak settings for the generated objects.
CHAPTER 3  Getting Started with .NET Development Using C#

FIGURE 3.55  Component designer surface with management objects.

NOTE: WHAT’S A COMPONENT?

The term component is probably one of the most overloaded words in the world of software design. In the context of Visual Studio’s Component project item, it refers to a subtype of the Component base class found in the System.ServiceModel namespace. What precisely makes up such a component is discussed in Chapter 27, “Diagnostics and Instrumentation,” where components are used quite often. In essence, components make it possible to share code, access it remotely, manage memory correctly, and so on. And on top of that, the notion of designer support is closely tied to the component model, too.

Server Explorer is not only involved in the creation of management-focused components. In various other contexts, Server Explorer can be used to drive the design of a piece of software. One such common use is in the creation of database mappings, something so common we dedicate the whole next section to it.

Database Mappers

Almost no application today can live without some kind of data store. An obvious choice is the use of relational databases, ranging from simple Access files to full-fledged client/server database systems such as SQL Server or Oracle. While having library support for communicating with the database is a key facility present in the .NET Framework through the System.Data namespaces, there’s more to it.

One of the biggest challenges of database technologies is what’s known as impedance mismatch between code and data. Where databases consist of tables that potentially participate in relationships between one another, .NET is based on object-oriented programming; therefore, a need exists to establish a two-way mapping between relational data and objects. In this context, two-way means it should be possible to construct objects out
of database records, while having the ability to feed changes back from the objects to the database.

To facilitate this, various mapping mechanisms have been created over the years, each with its own characteristics, making them applicable in different contexts. At first, this might seem a bit messy, but let’s take a look at them in chronological order. We won’t go into detail on them: Whole books have been written explaining all of them in much detail. For now, let’s just deal with databases in .NET programming.

**DataSet**

.NET Framework 1.0 started coloring the database mapping landscape by providing a means for offline data access. This was envisioned by the concept of occasionally connected clients. The core idea is as follows.

First, parts of a database are queried and mapped onto rich .NET objects, reflecting the structure of the database records with familiar managed types. Next, those objects can be used for visualization in UIs through mechanisms like data binding in ASP.NET and Windows Forms. In addition, objects can be directly updated in-memory, either directly through code or through data-binding mechanisms. An example of a popular control used in data binding is a DataGrid, which presents the data in a tabular form, just like Excel and Access do.

Visualizing and updating in-memory objects that originate from a database is just one piece of the puzzle. What about tracking the changes made by the user and feeding those back to the database? That’s precisely one of the roles of the offline mapping established through a DataSet, in collaboration with so-called data adapters that know how to feed changes back when requested (for example, by emitting `UPDATE` statements in SQL).

A DataSet can be used in two ways. The most interesting one is to create a strongly typed mapping where database schema information is used to map types and create full-fidelity .NET objects. For example, a record in a Products table gets turned into a Product object with properties corresponding to the columns, each with a corresponding .NET type.

To create a strongly typed DataSet, Visual Studio provides a designer that can interact with Server Explorer. This makes it incredibly easy to generate a mapping just by carrying out a few drag-and-drop operations. Figure 3.56 shows the result of creating such a mapping.

**NOTE: THE FUTURE OF DATASET**

Some people believe that the use of DataSet has become redundant since LINQ’s introduction in .NET 3.5 and its new mapping mechanisms. Nothing is further from the truth. As a matter of fact, there’s even a LINQ to DataSet provider in the .NET Framework class libraries.

DataSet is still a convenient way to represent tabular data, regardless of the type of underlying data store. The reason this works is because DataSet was intentionally designed to be decoupled from a particular database provider and to serve as a generic data container mechanism.
One of the key advantages of DataSet is its direct support for XML-based serialization. In fact, the extension of a strongly typed DataSet is .xsd, revealing this relationship. When generating mappings from database schemas, you’re actually creating an XML schema capturing type definitions and their mutual relationship. The command-line tool xsd.exe that ships with the .NET Framework developer tools can be used to generate C# or VB code from such a schema, just like the integrated designer does.

FIGURE 3.56   DataSet designer.

LINQ to SQL

After the relatively calm .NET 2.0 and 3.0 releases on the field of database mapping technologies, Language Integrated Query (LINQ) was introduced in .NET 3.5. As discussed in Chapter 2, “Introducing the C# Programming Language” (and detailed in Chapter 18, “Events,” and Chapter 19, “Language Integrated Query Essentials”), LINQ provides rich syntax extensions to both C# and VB, to simplify data querying regardless of its shape or origin. Besides LINQ providers used to query in-memory object graphs or XML data, a provider targeting SQL Server database queries shipped with .NET Framework 3.5.

In a similar way to the DataSet designer, LINQ to SQL comes with tooling support to map a database schema onto an object model definition. Figure 3.57 shows the result of such a mapping using the Northwind sample database. One core difference with DataSet lies in the SQL-specific mapping support, as opposed to a more generic approach. This means the LINQ to SQL provider has intimate knowledge of SQL’s capabilities required to generate SQL statements for querying and create/update/delete (CRUD) operations at runtime.
Similar to the DataSet designer, Server Explorer can be used to drag and drop tables (among other database items) onto the designer surface, triggering the generation of a mapping. Notice how relationships between tables are detected, as well, and turned into intuitive mappings in the object model.

Once this mapping is established, it’s possible to query the database using LINQ syntax against the database context object. This context object is responsible for connection maintenance and change tracking so that changes can be fed back to the database.

It’s interesting to understand how the designer generates code for the mapping object model. Most designers use some kind of markup language to represent the thing being designed. ASP.NET takes an HTML-centered approach, WPF uses XAML, and DataSet is based on XSD. For LINQ to SQL, an XML file is used containing a database mapping definition, hence the extension .dbml.

To turn this markup file into code, a so-called single file generator is hooked up in Visual Studio, producing a .cs or .vb file, depending on the project language. Figure 3.58 shows the code generation tool configured for .dbml files used by LINQ to SQL. The generated code lives in the file with .designer.cs extension. Other file formats, such as .diagram and .layout, are purely used for the look and feel of the mapping when displayed in the designer. Those do not affect the meaning of the mapping in any way.
Not surprisingly, the emitted code leverages the partial class feature from C# 2.0 once more. This allows for additional code to be added to the generated types in a separate file. But there’s more: A C# 3.0 feature is lurking around the corner, too. Notice the Extensibility Method Definitions collapsed region in Figure 3.59?

FIGURE 3.58 How the DBML file turns into C# code.

FIGURE 3.59 Generated LINQ to SQL mapping code.
You’ll see such a region in the various generated types, containing *partial method* definitions. In the data context type in Figure 3.59, one such partial method is `OnCreated`:

```csharp
{
    #region Extensibility Method Definitions
    partial void OnCreated();
    #endregion

    public NorthwindDataContext(string connection)
    : base(connection, mappingSource)
    {
        OnCreated();
    }
}
```

The idea of partial methods is to provide a means of extending the functionality of the autogenerated code efficiently. In this particular example, the code generator has emitted a call to an undefined `OnCreated` method. By doing so, an extensibility point has been created for developers to leverage. If it’s desirable to take some action when the data context is created, an implementation for `OnCreated` can be provided in the sister file for the partial class definition. This separates the generated code from the code written by the developer, which allows for risk-free regeneration of the generated code at all times.

**ADO.NET Entity Framework**

Finally, we’ve arrived at the latest of database mapping technologies available in the .NET Framework: the Entity Framework. Introduced in .NET 3.5 SP1, the Entity Framework provides more flexibility than its predecessors. It does this by providing a few key concepts, effectively decoupling a conceptual model from the mapping onto the database storage. This makes it possible to have different pieces of an application evolve independent of each other, even when the database schema changes. The Entity Framework also benefits from rich integration with the WCF services stack, especially OData-based WCF Data Services.

Figure 3.60 presents an architectural overview.

On the right is the execution architecture, a topic we’ll save for later. The most important takeaway from it is the ability to use LINQ syntax to query a data source exposed through the Entity Framework. In return for such a query, familiar .NET objects come back. That’s what mapping is all about.

Under the covers, the data source has an Entity Client Data Provider that understands three things:

- The conceptual model captures the intent of the developer and how the data is exposed to the rest of the code. Here entities and relationships are defined that get mapped into an object model.
The storage model is tied to database specifics and defines the underlying storage for the data, as well as aspects of the configuration. Things such as table definitions, indexes, and so on belong here.

Mappings play the role of glue in this picture, connecting entities and relationships from the conceptual model with their database-level storage as specified in the storage model.

To define both models and the mapping between the two, Visual Studio 2012 has built-in designers and wizards for the ADO.NET Entity Framework, as shown in Figure 3.61.

**NOTE: WHAT’S IN A NAME? ADO.NET**

ADO.NET was introduced in .NET Framework 1.0 as the successor to the popular ADO technology available for COM developers, including the Visual Basic classic community. ADO stands for ActiveX Data Objects and was by itself a successor to other database access technologies such as RDO and DAO. Luckily, all of that belongs to the past, and in fact the only relevant thing ADO.NET shares with its predecessor is its name. All concepts in ADO.NET fit seamlessly in the bigger picture of managed code and an object-oriented programming style.
Unit Testing

A proven technique to catch bugs and regressions early is to use unit tests that exercise various parts of the system by feeding in different combinations of input and checking the expected output. Various unit testing frameworks for .NET have been created over the years (NUnit being one of the most popular ones), and for the past few releases Visual Studio has built-in support for unit testing.

To set the scene, consider a very simple `Calculator` class definition, as shown here:

```csharp
public static class Calculator
{
    public static int Add(int a, int b)
    {
        return a + b;
    }

    public static int Subtract(int a, int b)
    {
        return a - b;
    }
}
```
To verify the behavior of our Calculator class, we want to call the calculator’s various methods with different inputs, exercising regular operation as well as boundary conditions. This is a trivial example, but you get the idea.

Unit tests in Visual Studio are kept in a separate type of project that’s hooked up to a test execution harness, reporting results back to the user. This underlying test execution infrastructure can also be used outside Visual Studio (for example, to run tests centrally on some source control server). While different types of test projects exist, unit tests are by far the most common, allowing for automated testing of a bunch of application types. Manual tests describe a set of manual steps to be carried out to verify the behavior of a software component. Other types of test projects include website testing, performance testing, and so on.

To create a unit test project, right-click the solution in Solution Explorer and choose Add, New Project to add a test project (see Figure 3.62).

![Figure 3.62 Creating a new unit test project.](image)
Next, right-click the newly created project node in Solution Explorer, and choose Add Reference. In the Reference Manager dialog, add a reference to the project containing the Calculator (see Figure 3.63).

![Reference Manager - CalculatorTests](image)

**FIGURE 3.63** Add a cross-project reference.

The unit test project contains an empty test class with an empty test method, as shown here:

```csharp
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

Our task is now to replace the code in the template with test methods that check the behavior of our Calculator. A much too simplistic example is shown here:

```csharp
[TestMethod]
public void AddTest()
{
    int a = 28;
    int b = 14;
    int expected = 42;
    int actual;
    actual = Calculator.Add(a, b);
    Assert.AreEqual(expected, actual);
}
```
To assert the expected behavior, we use helper methods on the `Assert` class. For example, the `Assert.AreEqual` test checks for equality of the supplied arguments.

**NOTE: TEST GENERATION WITH PEX**

From the preceding example, it’s clear that Visual Studio 2012 does not possess magical powers to understand your code and to thus generate a series of unit tests by itself. This does not mean such a thing is impossible to achieve, though.

By analyzing code carefully, specialized tools can infer lots of valid test cases that hit interesting conditions. In the preceding example, we haven’t written a test that deals with overflow situations when the two arguments to the `Add` method are too big for their sum to be represented as a 32-bit integer. Tools could infer such cases by looking at the types being used.

Another appealing property of automated test generation is the capability to ensure high numbers of code coverage. Assume you have some code with a bunch of conditional branches, leading to an explosion in the possible execution paths. Flow analysis tools can generate different sets of input values so that various code paths in the unit being tested are hit.

If all of this sounds like a wonderful dream, wake up now. With Pex, Microsoft Research has created such a toolkit that plugs in to Visual Studio. Pex stands for Program Exploration, reflecting its automated test case generation powers based on reasoning about the program. If you care about test coverage (you should!), Pex is definitely something to check out. Visit http://research.microsoft.com/Pex for more information.

The nice thing about using Pex with .NET 4.0 is its synergy with managed code contracts, something we’ll talk about later. An example of a contract is constraining the range of an input value, a so-called precondition. Contracts not only serve documentation purposes but are also used to enforce correctness by means of theorem provers or runtime checks. But combining the information captured in contracts with Pex is even more exciting. Pex can use this wealth of information to come up with more test cases that check violations of contracts and such.

Does all of this mean you should no longer write unit tests yourself? No. Although Pex can take over the burden of generating various types of tests, there’s still lots of value in writing more complex test cases that exercise various concrete scenarios your software component needs to deal with. In other words, Pex enables you to focus more on the more involved test cases while relieving you from the creation of slightly more boring (but nevertheless important) test cases.

Once unit tests are written, they’re ready to be compiled and executed in the test harness. This is something you’ll start to do regularly to catch regressions in code when making changes. Figure 3.64 shows a sample test run result, triggered through the Test, Run, All Tests menu item.

Turns out I introduced some error in the `Subtract` method code, as caught by the unit test. Or the test could be wrong. Regardless, a failed test case screams for immediate attention to track down the problem. Notice you can also debug through tests cases, just like regular program code.
Tightly integrated with unit testing is the ability to analyze code coverage. It’s always a worthy goal to keep code coverage numbers high (90% as a bare minimum is a good goal, preferably more) so that you can be confident about the thoroughness of your test cases. Visual Studio actually has built-in code highlighting to contrast the pieces of code that were hit during testing from those that weren’t.

**Team Development**

To finish off our in-depth exploration of Visual Studio 2012 tooling support, we take a brief look at support for developing software in a team context. Today’s enterprise applications are rarely ever written by a single developer or even by a handful of developers. For example, the .NET Framework itself has hundreds of developers and testers working on it on a day-to-day basis.

**Team System and Team Foundation Server**

To deal with the complexities of such an organization, Visual Studio Team System (VSTS) provides development teams with a rich set of tools. Besides work item and bug tracking, project status reporting, and centralized document libraries, source control is likely the most visible aspect of team development.

The entry point for the use of Team Foundation Server (TFS) is the Team Explorer window integrated in Visual Studio 2012 (see Figure 3.65).

Here is a quick overview of the different parts of the Team Explorer:

- The drop-down at the top represents the TFS server we’re connected to. One of the nice things about TFS is its use of HTTP(S) web services (so there is no hassle with port configurations). Each server can host different team projects.

- Work Items is the collective name for bug descriptions and tasks assigned to members of the team. Queries can be defined to search on different fields in the database. Via the Work Items view, bugs can be opened, resolved, and so on.
CHAPTER 3  Getting Started with .NET Development Using C#

FIGURE 3.65   Team Explorer in Visual Studio 2012.

- **Documents** displays all sorts of documentation—Word documents, Visio diagrams, plain old text files, and such—that accompany the project. Those are also available through a SharePoint web interface.

- **Reports** leverages the SQL Server Reporting Services technology to display information about various aspects of the project to monitor its state. Examples include bug counts, code statistics, and so on.

- **Builds** allows developers to set up build definitions that can be used for product builds, either locally or remotely. It’s a good practice for team development to have a healthy product build at all times. Automated build facilities allow configuration of daily builds and such.

- **Source Control** is where source code is managed through various operations to streamline the process of multiple developers working on the code simultaneously. This is further integrated with Solution Explorer.

**Source Control**

Source control stores source code centrally on a server and provides services to manage simultaneous updates by developers. When a code file requires modification, it’s checked out to allow for local editing. After making (and testing) the changes, the opposite operation of checking in is used to send updates to the source database. If a conflicting edit is detected, tools assist in resolving that conflict by merging changes.

Figure 3.66 shows the presence of source control in Visual Studio 2012, including rich context menus in Solution Explorer and the Source Control Explorer window.

Other capabilities of source control include rich source code versioning (enabling going back in time), shelving edits for code review by peer developers, correlation of check-ins to resolved bugs, and the creation of branches in the source tree to give different feature crews their own playgrounds.
Summary

In this chapter, we installed the .NET Framework 4.5 and went through the motions of building our first trivial but illustrative C# console application. While doing so, we focused on the development process of writing and compiling code, and then we took a look at how to inspect it using ILSpy. Because it’s unrealistic today to build software without decent tooling support, we explored various aspects of the Visual Studio 2012 family. We covered integrated source exploration, build and debugging support, and took a peek at the various project types and associated tools available.

In the next chapter, we leave the realm of extensive tooling for a while and learn about the core fundamentals of the C# language.
Index

.NET and COM: The Complete Interoperability Guide, 602
.NET Framework, 1-4, 11-12, 27 54, 103, 173
.NET 4.5, 107-109
  asynchronous programming, 1429-1433
applications
  deployment, 10
  types, 11
assemblies, 16-17
BCL (Base Class Library), 11, 51,
  1301-1303, 1372
  assemblies, 1304-1308
  default project references, 1303-1304
  encoding text, 1371
  formatting text, 1357-1362
history, 53-54
namespaces, 51-52, 1304-1306
Object Browser, 1305-1306
  parsing text to objects, 1362-1363
string methods, 1366-1369
StringBuilder class, 1369-1371
  support, 11
  System namespace, 1311-1320,
  1344-1356
classes, 19-20
CLI (Common Language Infrastructure),
  12-14
CLR (Common Language Runtime), 32-33
  application domains, 37-39
  assembly loading, 35-36
  automatic memory management, 43-46
  bootstrapping runtime, 33-35
  entry points, 33
  exception handling, 46-48
  JIT compilation, 39-41
  NGEN (native image generation), 41-43
  shims, 34
CLS (Common Language Specification),
  23-24
COM interop, 1163-1172
  component-driven development, 10
  continuations, hidden, 1533
creating applications, 113-119
  delegates, 21-22
  enums, 19-20
  events, 850-852, 908
  exception handling, 11
  executing managed code, 24-31
GAC (Global Assembly Cache), 111-112
  generics, 23
  installing, 103-112
  interfaces, 20-21
  members, 22-23
  metadata, 1058-1059
  modules, 16
  multiple language, 10
    support, 10, 15
  usability, 1058-1062
OOP (object-oriented programming), 10
  primitive types, 19
  refactoring, 1280-1281
runtime shim, 110-111
  source code, 143
  structs, 19-20
thread pools, 1475-1482
  type
    hierarchy, 203-204
    safety, 18-19
    system, 17-24
unified runtime infrastructure, 11
  versions, 103-107
web services, 12

A

aborted threads, 1456
absolute time, 1327-1329
abstract classes, 688-690
abstracting concurrency, 1481
access, type members, 486-489
accessing
fields, 548-551
members, 1106-1107
accessors, add and remove, 857-861
ACID transactions, locks, 461
add accessors, 857-861
Add method, 792
addressing, 64-bit, limitations, 1506
ADO.NET Entity Framework, 165-166
Aggregate query operator, 1019-1022
AggregateException type, 1218-1220, 1526-1527
unwrapping, 1624-1627
aggregation query operators, 1019-1026
AJAX (Asynchronous JavaScript and XML), 898-900
dictionary suggest, 908-911
aliases
built-in types, 212
extern, 1235-1238
importing namespaces, 1234-1235
All query operator, 1026-1027
analysis, code, 1374-1376
anonymous closures, 63
anonymous function expressions, 801-802
anonymous iterators, 993
anonymous methods, 63, 345-347
anonymous types, 66
LINQ (Language Integrated Query), 941-944
Any query operator, 1026-1027
APIs (application programming interfaces), 6
expression trees, 1103-1114
fuent, extension methods, 68
public, renaming parameters, 812
WinRT (Windows Runtime), 1667-1668
Windows Store application, 1644-1646
APM (Asynchronous Programming Model), 899, 1421-1433, 1564-1569
versus EAP (Event-based Asynchronous Pattern), 1569-1570
methods, 1423-1428
threading state, 1424-1427
appcontainer.exe target, 1245
application domains, 1241, 1286, 1298-1299
CLR (Common Language Runtime), 37-39
creating, 1287-1288
cross-domain communication, 1288-1297
managed add-in framework, 1296-1298
application extensibility, 1069-1080
built-in operations, 1071-1072
defining interface, 1070-1071
extensions
deploying, 1076-1077
loading, 1074-1076
writing, 1076-1077
MEF (Managed Extensibility Framework), 1069, 1077-1080
user interface, 1072-1074
ApplicationException class, 1201
applications
I/O (input/output), 1399-1400, 1440-1441
localizable, 1360
monitoring
event logs, 1388-1391
performance counters, 1391-1395
reactive, 845-852
delegates, 846-849
trivial console, running, 179-180
ArgumentException class, 1193, 1213
ArgumentNullException class, 1193, 1213-1214
ArgumentOutOfRangeException class, 1214-1215
arguments, 504
arithmetic expressions, toy compiler, 1093-1100
arithmetic operators, 258-259
character arithmetic, 262-263
decimal arithmetic, 261-262
floating-point arithmetic, 260-261
integer arithmetic, 259
nullable value types, 269
overflow checking, 263-269
unary plus and minus, 263
array types, broken covariance, 745-747
arrays, 230-239, 249
initializers, 234-236
internal representation, 231
jagged, 236-237
multidimensional, 238-239
parameters, 510-511
AJAX (Asynchronous JavaScript and XML)

language support, 91-95
latency, 1552
patterns, 89-91, 1564
  APM (asynchronous programming model), 1564-1569
  EAP (Event-based Asynchronous Pattern), 1569-1571
  exception behavior, 1576-1578
  method naming, 1573-1575
  overloading, 1573-1575
  progress reporting, 1575-1576
  synchronization behavior, 1578-1579
  TAP (Task-based Asynchronous Pattern), 1571-1573, 1579-1584
saving evaluation state, stack spilling, 1630-1634
scalability, 1561-1564
simplifying, .NET 4.5, 1429-1433
WinRT (Windows Runtime), 1656
Asynchronous Programming Model (APM), 899, 1421-1433, 1564-1569
  versus EAP (Event-based Asynchronous Pattern), 1569-1570
methods, 1423-1428
threading state, 1424-1427
asynchronous read and write I/O operations, 1420-1433
atomicity, 1483-1486
attributes
custom
defining, 1086-1087
discovering, 1089-1091
reflection, 1085-1091
storage, 1088-1089
InternalsVisibleTo, 1276-1277
ThreadStatic, 1464-1467
auto-implemented properties, 73-75, 578, 882-883
automatic memory management, CLR (Common Language Runtime), 43-46
Average query operator, 1023-1024
await expressions, 616
asynchronous, 1584-1585, 1588-1591
  manual callback plumbing, 1597-1603
  synchronization behavior, 1603-1607
cascading completion, 1593
continuations, 1536
awaitable types, building, 1634-1640

B
background threads, 1458-1460
BackgroundWorker component, 1507-1510
backward compatibility, 59, 183
Bar method, 479
barriers, synchronization, 1506
base calls, 687
base class constraints, generic types, 727-728
base class members, hiding, 672-674
BCL (Base Class Library), 11, 51, 1301-1303, 1372
  assemblies, 1304-1306
 mscorlib, 1306-1308
  System, 1306-1308
  System.Core, 1308-1311
  default project references, 1303-1304
  encoding text, 1371
  formatting text, 1357-1362
  history, 53-54
  namespaces, 1304-1306
  organization, 51-52
Object Browser, 1305-1306
parsing text to objects, 1362-1363
string methods, 1366-1369
StringBuilder class, 1369-1371
System namespace, 1311
  arrays, 1315-1318
  BitInteger type, 1320-1322
  complex numbers, 1322-1324
  GC (garbage collector), 1344-1351
  GUID values, 1335-1337
  interacting with environment, 1339-1344
  lazy initiation, 1353-1354
  native interop, 1351-1353
  nullability, 1337-1338
  primitive value types, 1311-1315
System.Math class, 1318-1320
tuple types, 1354-1356
Uri type, 1338-1339
BeforeFieldInit type attribute, 1205
BeginInvoke method, 1427
Big O notation, 756
binary expressions, 1105
BinaryWriter class, 1418-1420
binders, DLR (Dynamic Language Runtime), 1137-1143
BindingFlags enum, 1084-1085
bindings, query expressions, 972-974
BitArray type, 763
blocked threads, interrupting, 1457-1458
blocking, 1556
inheritance, 671
blocks
exception handling, 352
statements, 351, 356
Boole, George, 200
Boolean logical operators, 279-281
Boolean types, 200-201
bootstrapping runtime, CLR (Common Language Runtime), 33-35
boxed value types, 622
boxing
conversions, 637-638
types, 478-483
break statement, 378-379
broken covariance, array types, 745-747
bugs, common source, 241-243
build support, Visual Studio 2012 projects, 134-138
building WinRT (Windows Runtime) components, 1662-1665
built-in conversions, 634
boxing and unboxing, 637-638
enumeration, 634-635
nullable, 635
numeric, 634
reference, 635-637
built-in types, 190-212
aliases, 212
Boolean, 200-201
decimal, 199-200
floating-point, 194-198
integral, 190-194
object, 203-205
string, 201-202

C

C# programming language, 15
enriching core features, 58-63
evolution, 55
managed code development, 56-58
name origin, 57
C++ programming language, 6, 15
CaaS (compiler as a service), 97-99
caching, 1422
calendar systems, 1331-1332
call sites, DLR (Dynamic Language Runtime), 1137-1143
call stacks, overflowing, 413
caller info attributes, optional parameters, 516-519
calling
methods
generic, 737
optional parameters, 513-516
through delegates, 343
calls
base, 687
tail, 1212
virtual, 683-687
cancellation, tasks, 1536-1538
Cardone, Felice, 347
CAS (Code Access Security), 258
CLR (Common Language Runtime), 48
case labels, 366
cast expressions, 302-307, 309
syntax, 303
Cast projection query operator, 1008
catching exceptions, 426-427
CDS (Coordination Data Structures), 102
character arithmetic, operators, 262-263

How can we make this index more useful? Email us at indexes@samspublishing.com
character literals, 194
checked arithmetic, 265-266, 352
Church, Alonzo, 347, 811
Class Library, Visual Studio 2012, 127
classes, 465
  abstract, 688-690
  ApplicationException, 1201
  ArgumentNullException, 1193, 1213-1214
  ArgumentOutOfRangeException, 1214-1215
  ArgumentNullException, 1193, 1213
  ArgumentOutOfRangeException, 1214-1215
  ArgumentException, 1193, 1213
  ArrayTypeMismatchException, 1204
  Assembly, 1282-1283
  BCL (Base Class Library), 11, 51,
    1301-1303
    assemblies, 1304-1311
    default project references, 1303-1304
    formatting text, 1357-1362
    history, 53-54
    namespace organization, 51-52
    namespaces, 1304-1306
    System namespace, 1311-1356
  BinaryWriter, 1418-1420
  CLI (Common Language Infrastructure), 19-20
  CountDownEvent, 1501-1502
  Debugger.IsAttached, 1385
derived, designing events, 878-880
  DirectoryInfo, 1407
  DirectoryNotFoundException, 1193
  DriveInfo, 1400
  DynamicMetaObject, 1154-1156
  DynamicObject, 1149-1153
  EventLog, 1388-1391
  Expression, 1103-1104
  File, 1409-1415
  FileInfo, 1406-1407
  FileNotFoundException, 418, 1193
  IndexOutOfRangeException, 1203
  inheritance, 663-666
    single, 667-668
  InsufficientMemoryException, 1208
  InvalidCastException, 1203
  InvalidOperationException class, 1215
  Lazy, 1353-1354
  ManualResetEvent, 1500-1502

MemoryStream, 1417
MFCs (Microsoft Foundation Classes), 6
NotImplementedException, 1215-1216
NotSupportedException, 1193, 1216-1217
OOP (object-oriented programming), 662
PipeStream, 1434
Process, 1374, 1396
static, 527, 595
StreamWriter, 1417-1418
Stream, 1415-1434
StreamReader, 1414-1415, 1417-1418
StreamWriter, 1414-1415
StringBuilder, 1369-1371
versus structs, 466-486
System.Convert, 644
System.Exception, 1198
System.Math, 1318-1320
System.Object, 306, 702, 703
  banning, 204-205
  Equals method, 622-630
  ReferenceEquals method, 628-630
TaskCreationOptions, 1522
TaskFactory, 1521-1522
TaskScheduler, 1540-1542
TextWriter, 1413-1414
Thread, 1448-1453
ThreadLocal, 1467-1470, 1482
TimeZone, 1332-1333
TypeConvert, 645-646
WaitHandle, 1502-1503

clauses
  from, source selection, 933-938
  group, 953-960
  into, 966-971
  join, 960-965
  let, 972-974
  select, projection, 938-944

cleanup logic, 408
CLFS (Common Log File System), 1440
CLI (Common Language Infrastructure), 15
classes, 19-20
enumerations, 19-20
structures, 19-20
dynamic, 1062-1063
evaluation stack, 255-258
initializers, 66-67
invocation, 340-348
lambda, 69-71, 347-348, 807-809, 1107-1110
logical operators, 277-281
new operator, 324-336
null-coalescing operators, 285-287
operator overloading, 609-633
operator result type, 284-285
operators, 252-254
relational operators, 275-277
shift operators, 274-275
string concatenation, 269-274
subexpressions, 254
trees, 71-73, 810-811
unary, 1105
function pointers, 345
FxCop, 362
generation, locks, 457-458
homoiconicity, 73
IL, 27-28
IL-generated, dumping, 1271-1273
input validation, 410-413
inspecting, ILSpy, 116-119
JIT-generated, dumping, 561, 1271-1273
Just My Code feature, disabling, 422
LCG (Lightweight Code Generation), 1091
Hello World program, 1091-1093
toy compiler for arithmetic expressions, 1093-1100
locks, 352
loops, 398-400
managed, 1446-1448
executing, 24-31
managed development, 56-58
measuring performance, 1386-1388
optimization, 256, 282
quotations, 72-73
reflection, 1057, 1063, 1117
application extensibility, 1069-1080
custom attributes, 1085-1091
events, 1083-1084
fields, 1084-1085
indexers, 1082-1083
late-bound property access, 1083
methods, 1080-1081
properties, 1082-1083
System.Type type, 1064-1066
types, 1066-1068
typing, 1058-1063
running, 116
runtime disasters, 413-416
Sandcastle project code, 230
snippets
  common tasks, 490
  writing, 1200
stack traces, logging, 1385-1386
statement trees, 1110-1114
statements, 351-353
  blocks, 351, 356
  declaration, 351
  declarations, 357-358
  empty, 355-356
  exception handling, 352
  expression, 351, 353-355
  goto, 400-403
  iteration, 352, 375-397
  jump, 352
  resource management, 352
  return, 404-406
  selection, 352, 358-375
StyleCop, 362
synchronization, 1506-1511
primitives, 1482-1510
syntax versus semantics, 619
threads, 1444-1446
  background, 1458-1460
  creating, 1448-1450
  debugging techniques, 1471-1474
  exceptions, 1463-1464
  foreground, 1458-1460
  frozen, 1471-1474
  IDs, 1461
  life cycle, 1453-1458
  managed, 1458-1463
    naming, 1460
per-thread state, 1481-1482
pools, 1474-1482
starting, 1450-1453
stopping, 1454-1456
Thread class, 1448-1453
threading apartments, 1461-1463
thread-local storage, 1470-1471
thread-specific state, 1464-1471
types, 184-185
unsafe, 1318
Visual Studio 2012 projects, 143-148
whitespace sensitivity, 360-361
writing, 114-115
Code Access Security (CAS), 48-49, 258
Code Analysis, 545-546
code editor, Visual Studio 2012, 131-133
code metrics, projects, calculating, 541
CodeDOM, 1110
collection initializers, 334-336, 706
syntax, 772
collection types, 701-703, 755, 787
generic, 765-778
nongeneric, 755-764
hash tables, 757-760
queues, 761
stacks, 762-763
specialized, 786-787
thread-safe, 778-786
collections, GC (garbage collector), 1348-1349
COM (Component Object Model), 9
component-driven development, 7
error handling, 1177-1178
interop, 1159-1161
.NET 3.5, 1163-1168
.NET 4.0, 1169-1172
dispatch services, 1161
embedding PIs, 1172-1174
improving, 82-85
marshaling services, 1161
obsolescence, 50
combining delegates, 835-842
command line, 113
commands
PrintException, 1190
StopOnException, 1190

subt, 1401
Threads, 1191
comments, 223-230
delimited, 226-227
documentation, 227-230
single-line, 223-224
Common Language Runtime (CLR). See CLR (Common Language Runtime)
Common Language Specification (CLS), 14, 23-24
Common Log File System (CLFS), 1440
Common Type System (CTS), 14, 56
compile method, 1096-1100
compile time type, versus runtime type, 206
compiler-generated expression trees, 1101-1103
compilers
CaaS (compiler as a service), 97-99
extension methods, marking and finding, 531-534
optimization, 256
restrictions, 666
compiling code, 115
complex numbers, System namespace, 1322-1324
compliance, 176
components, WinRT (Windows Runtime)
activation, 1653-1655
building, 1662-1665
creating, 1658-1667
debugging, 1667
using, 1665-1667
writing, 1658-1662
compound assignments, 290-292
overloading operators, 617
compound keys, 956-960
computed keys, 956
Concat query operator, 1028-1029
concatenation, string, operators, 269-274
concrete types, constructors, 326
concurrency, 97, 99-102
abstracting, 1481
conditional debugger output, 1381
conditional operators, 281-284, 617-621
console application, running, 179-180
Console Application, Visual Studio 2012, 127

How can we make this index more useful? Email us at indexes@samspublishing.com
 constants, 557-559
eums, 565
local variables, 216-218
constituent types, 577-578
constraints, generic types, 720-721
base class, 727-728
constructor, 728-735
default constructor, 728-735
interface-based, 721-727
constructors, 22, 326-329, 585, 608
cancrete types, 326
default, 587-589
generic types, 728-735
inheritance, 664
initializers, 591-592
instance, 585-592
static, 592-595
structs, 589-591
Contains query operator, 1027
context switches, 1445, 1514
contextual keywords, 60, 182-183, 1586-1587
continuations
await expressions, 1536
exceptions, 1527
hidden, NET Framework, 1531-1536
specifying options, 1534-1536
tasks, 1533-1534
continue statement, 378-379
contracts, 243
code, 1375-1376, 1379-1381
as interfaces, 691-695, 1295-1297
covariance, 85-88
generic, 743-754, 798
safety guarantees, 748-749
control flow
asynchronous methods, 1595-1597
asynchronous programming, 1627-1630
controlling processes, 1396-1398
conversions, 609, 633, 647
built-in, 634
boxing and unboxing, 637-638
enumeration, 634-635
nullable, 635
numeric, 634
reference, 635-637
date and time values, 1329-1331
explicit, 301-319
IConvertible interface, 644-645
implicit, 301-319
System.Convert class, 644
TypeConvert class, 645-646
user-defined, 638-644
cooperative scheduling, 454-455
yielding, 1457
Coordination Data Structures (CDS), 102
Count query operator, 1022
CountDownEvent class, 1501-1502
CountdownEvent synchronization, 1479
covariance, 85-88
broken, array types, 745-747
generic, 798
generic types, 743-754
safety guarantees, 748-749
CreateInstance method, 728
cross-domain communication, application
domains, 1288-1297
C-style function pointers, 794, 800
CTS (Common Type System), 14, 56
curly braces, 422
custom attributes
defining, 1086-1087
discovering, 1089-1091
reflection, 1085-1091
storage, 1088-1089
custom ordering, query expressions, 952

D

data
binding, WPF (Windows Presentation
Foundation), 884
code, 184-185
versus code, 343
in-memory, 914-915
LINQ (Language Integrated Query), 921-923
parallelism, 1542-1550
structures, 724
database mappers, 160-161
ADO.NET Entity Framework, 165-166
DataSet, 161-162
LINQ to SQL, 162-165
databases
horizontal partitioning, 946-947
relational, 915-919
LINQ (Language Integrated Query), 923-929
vertical partitioning, 944
DataSet, 161-162
date and time values, 1327-1335
conversions, 1329-1331
DateTime values, 1327-1335
DateTimeOffset value, 1335
debug support, Visual Studio 2012 projects, 139-142
Debugger.IsAttached class, 1385
debuggers, controlling, 1383-1385
debugging
code
controlling debugger, 1383-1385
diagnostic output, 1381-1383
MDAs (Managed Debugging Assistants), 1189
WinRT (Windows Runtime) components, 1667
debugging code, IntelliTrace, 1191-1192
decimal arithmetic, operators, 261-262
decimal types, 199-200
declaration statements, 351
declarations
asynchronous methods, 1585-1588
fields, 548
generic types, 707-712
local variables, 212-213
method parameters, 504-519
namespaces, 1227-1230
pairwise, relational and equality operators, 621-622
properties, 575-578
statements, 357-358
versus
assignments, 288-290
imperative, 1516-1519
virtual members, 680-681
declarative languages, 95
declarative programming, 97
decomposing, types, 465
Decrement method, 1504-1505
decrement operators
expression statements, 355
overloading, 616-617
prefix and postfix, 297-299
default constructors, 587-589
constraints, generic types, 728-735
default value expressions, 322-324
default values, type parameter operations, 718-720
defaults, members, 658
defining
custom attributes, 1086-1087
exception types, 1198-1201
extension methods, 526-528
finalizers, 597
flags enums, 571-572
indexers, 580-582
interfaces, 691-692
method overloads, 519-520
methods, 501-502
operators, 610-611
rules, 24
definite assignments, 292-296
delegate invocation, expressions, 341-348
delegates, 21-22, 789, 794, 842, 844-845, 911
anonymous function expressions, 801-802
asynchronous, invocation, 1427
calling through, 343
closures, 802-807
combining, 835-842
EventHandler, 871-878, 901
expression trees, 810-811
extensible calculator, 815-819
versus function pointers, 345
instances, 798-800
creating, 343-347
invocation, 1107-1110
invoking, 811-815, 844
asynchronous invocation, 823-835
lambda expressions, 807-809
LINQ (Language Integrated Query), 819-823
1682  delegates

MulticastDelegate, 796
plain use, 849-850
types, 794-798
generic, 814-815
delimited comments, 226-227
dependencies, language, 1307-1308
deployment
assemblies, 1249-1252
Xcopy, 1264-1265
derived classes, designing events, 878-880
designers
ASP.NET, 155-157
VSTO (Visual Studio Tools for Office), 157-158
Windows Forms, 148-150
WPF (Windows Presentation Foundation), 151-153
designing events, derived classes, 878-880
destructors, 585, 595-608, 862
defining, 597
garbage collection, 601-607
implementing, 600
running, 597-600
detaching, event handlers, 861-870
Deterministic Resource Clean Up, 438-448
DGML (Directed Graph Markup Language), 147
diagnostic, debugging code, 1381-1383
dictionary suggest, 900-901
AJAX, 908-911
direct invocation, expressions, 340-341
Directed Graph Markup Language (DGML), 147
directives, preprocessing, 224-226
directories, 1402-1404
paths, 1405-1406
DirectoryInfo class, 1407
directoryNotFoundException class, 1193
discovering custom attributes, 1089-1091
dispatch services, COM interop, 1161
Dispose method, 603-607
Distinct restriction operator, 1003
DivideByZeroException, 1201
DLR (Dynamic Language Runtime), 80-82, 1137, 1158-1159
dynamic dispatch, 1143-1149
dynamic operations, 1157-1158
DynamicMetaObject, 1154-1156
DynamicObject, 1149-1153
DNA (Distributed interNet Applications Architecture), 8
documentation comments, 227-230
DOM (Document Object Model) APIs, 919, 1644
domains, application, 1286, 1298-1299
creating, 1286
cross-domain communication, 1288-1297
managed add-in framework, 1296-1298
domain-specific languages (DSLs), 97
do.while statement, 379-380
DriveInfo class, 1400
drives, listing, 1400-1402
DSLs (domain-specific languages), 97
duck typing, 336
dumping IL and JIT-generated code, 1271-1273
dynamic binders, 1137-1143
dynamic call sites, 1137-1143
dynamic dispatch, DLR (Dynamic Language Runtime), 1143-1149
dynamic expressions, 1062-1063
dynamic keyword, 79-80, 1119-1121
deferred overload resolution, 1124-1126
dynamic type, 1121-1122
dynamic typing, 1122-1124
IronPython, 1128-1137
using, 1128-1137
Dynamic Language Runtime (DLR), 80-82
dynamic languages, 75-88
versus static, 77-79
dynamic parameters, 1138
dynamic programming, 1119, 1174
COM interop, 1159-1161
.NET 3.5, 1163-1168
.NET 4.0, 1169-1172
dispatch services, 1161
embedding PIA,s, 1172-1174
marshaling services, 1161
DLR (Dynamic Language Runtime), 1137
dynamic binders, 1137-1143
dynamic call sites, 1137-1143
dynamic dispatch, 1143-1149
dynamic operations, 1157-1159
DynamicMetaObject, 1154-1156
DynamicObject, 1149-1153
dynamic keyword, 1119-1121
deferred overload resolution, 1124-1126
dynamic type, 1121-1122
dynamic typing, 1122-1124
System.Dynamic type, 1126-1128
using, 1128-1137
dynamic type, 1121-1122
dynamic typing, 205-206, 312
dynamic keyword, 1122-1124
member access, 338-339
versus static, 207
DynamicMetaObject class, 1154-1156
DynamicObject class, 1149-1153

e
e
EAP (Event-based Asynchronous Pattern), 899, 1569-1571
versus APM (Asynchronous Programming Model), 1569-1570
editions, Visual Studio 2012, 120-121
Einstein, Albert, 1
ElementAt restriction operator, 1006
ElementAtOrDefault restriction operator, 1006
elements, access, 348-349
embedded resources, assemblies, 1278-1279
Empty source generator, 1001
empty statements, 355-356
capsulation, 653-654
encoding text, 1371
Entity Framework, 165-166
entry points, 175-181
CLR (Common Language Runtime), 33
signatures, 177-179
enumerations
conversions, 634-635
logical operators, 277-278
e
enums, 563-564
CLI (Common Language Infrastructure), 19-20
constants, 565
enumerating, 567-568
flags, describing, 570-573
members, assigning values to, 565-566
string representations, 566-567
switch statement, 573-574
System.Enum type, 566-569
underlying types, 564
values
converting integral values to, 568-569
converting strings to, 569
environment error conditions, 416-420
epsilon-delta definitions, 260
equality checks, types, 275-276
equality operators
overloading, 621
pairwise declaration, 621-622
Equals method
GetHashCode consistency, 625-628
overloading operators, 622-632
overriding, 623-625
required properties, 625
errors
environment error conditions, 416-420
handling
COM (Component Object Model), 1177-1178
propagation, 408-409
Win32, 601-602
tasks, dealing with, 1524-1531
Evaluate method, 620
evaluation stack, 255-258
evaluators, 371
event handlers, 851
detaching, 861-870
EventHandler<T>, 875-878, 901
naming, 872
Visual Basic, declarative approach, 889-890
event logs, monitoring software, 1388-1391
EventArgs, 871-875
Event-based Asynchronous Pattern (EAP), 899

How can we make this index more useful? Email us at indexes@samspublishing.com
EventHandler delegate, 871-878, 901
EventLog class, 1388-1391
events, 22, 843, 853-855, 911
.NET, 850-852
eexisting, 908
add accessors, 857-861
delegates, 844-845
designing, derived classes, 878-880
EAP (Event-based Asynchronous Pattern), 899
event handlers
detatching, 861-870
EventHandler, 871-878, 901
naming, 872
GUIs (graphical user interfaces), 890-896
handling, 844
multithreading interaction, 856
patterns, 871-880
raising, 855-857
reactive applications, 845-852
delegates, 846-850
reactive programming, 898-905
reflection, 1083-1084
remove accessors, 857-861
signaling, 1498-1503
UI frameworks, 885-890
UnobservedTaskException, 1529-1530
WinRT (Windows Runtime), interoperability, 896-898
Except query operator, 1028
exception handling
SEH (structured exception handling), 1175
statements, 352
exception text, printing, 918
ExceptionDispatchInfo type, 1624
exceptions, propagation, 429-431
exceptions, 243, 407, 462, 1175, 1178-1180, 1220
behavior, asynchronous programming, 1576-1578
catching, 426-427
causes, 410-420
checked, 1179
continuations, 1527
first-chance, enabling, 424
handling, 11, 407-409, 421-431, 1180-1183, 1193-1196
CLR (Common Language Runtime), 46-48
filters, 431
finally clause, 432-437
first-chance exceptions, 1187-1190
handler order, 1181
IntelliTrace, 1191-1192
ty statements, 1183-1186
as objects, 409-410
propagation, 429-431, 1619-1624
rethrowing, 427-429
SEH (structured exception handling), 1175
ThreadAbortException, 458
threads, 1463-1464
throwing, 420-421, 1196-1198
types, 1201
AggregateException, 1218-1220
ApplicationException, 1201
ArgumentOutOfRangeException, 1193, 1216-1217
ArgumentNullException, 1193, 1213-1214
ArgumentOutOfRangeException, 1214-1215
ArrayTypeMismatchException, 1204
defining, 1198-1201
DirectoryNotFoundException, 1193
DivideByZeroException, 1201
ExecutionEngineException, 1212-1213
FileNotFoundException, 1193
FormatException, 1217-1218
InsufficientMemoryException, 1208
InvalidCastException, 1203
InvalidOperationException, 1215
NotImplementedException, 1215-1216
NotSupportedException, 1193, 1216-1217
NullReferenceException, 1202-1203
ObjectDisposedException, 1206-1208
OutOfMemoryException, 1208-1209
OverflowException, 1201-1202
PathTooLongException, 1193
StackOverflowException, 1209-1212
TypeInitializationException, 1204-1205
How can we make this index more useful? Email us at indexes@samspublishing.com
expressions

orderby keyword, 946-952
patterns, 952
select clause, 938-944
syntax, 931-974
where clause, 944-947
regular, 1363-1366
versus statements, 1102-1103
string concatenation, 269-274
subexpressions, 254
trees, 71-73, 810-811
unary, 1105
ExpressionVisitor type, 1114-1117
extensibility, 1069-1080
built-in operations, 1071-1072
defining interface, 1070-1071
extensions
deploying, 1076-1077
loading, 1074-1076
writing, 1076-1077
MEF (Managed Extensibility Framework), 1069, 1077-1080
user interface, 1072-1074
Extensible Application Markup Language (XAML), 122
extensible calculator, 815-819
EXTensible Stylesheet Language (XSLT), 230
extension methods, 68-69, 524-534, 1214
compilers, marking and finding, 531-534
defining, 526-528
versus expandos, 534
importing namespaces, 1238-1240
LINQ to Objects, 980-984
overload resolution, 528-529
extern aliases, 1235-1238
extern methods, 538-539

F

F# programming language, 15
factory methods, 187
fibers, 1445

fields, 22, 547, 583-584
accessing, 548-551
constants, 557-559
declaring, 548
default values, automatic assignment, 552
initializing, 551-555
naming conventions, 548
object initializers, 334
read-only, 555-557
reflection, 1084-1085
volatile, 559-563
File class, 1409-1415
file system, monitoring activity, 1407-1409
FileInfo class, 1406-1407
FileNotFoundException class, 418, 1193
files
directories, 1402-1404
memory-mapped, 1437-1440
PE/COFF, 1420
filters, 1197
exception handling, 431
where clause, 944-947
finalization, 598-600, 1349-1350
finalizers, 22, 585, 595-607, 608
defining, 597
GC (garbage collector), 601-607, 1349-1350
implementing, 600
running, 597-600
finally clause, exception handling, 432-437
finding
assemblies, 1267-1271
operators, 611-612
First restriction operator, 1005
first-chance exceptions, 1187-1190
enabling, 424
first-class functions, 793-794
FirstOrDefault restriction operator, 1005
flags
checking for, 572-573
descrribing, 570-573
floating-point arithmetic, operators, 260-261
Gyro generic types, 1687

G

GAC (Global Assembly Cache), 36, 1258-1262
.NET Framework, 111-112
inspecting, 1258-1260
installing assemblies in, 1260-1262
GC (garbage collector), 438-440, 1344-1351
collections, 1348-1349
finalization, 1349-1350
IDisposable, 601-607
memory pressure, 1349
weak references, 1350-1351
generic co- and contravariance, 798
generic collection types, 765-778
generic methods, 502, 523-524, 736-743
calling, 737
constraints, 720-721
base class, 727-728
default constructor, 728-735
interface-based, 721-727
contravariance, 743-754
safety guarantees, 748-749
covariance, 743-754
safety guarantees, 748-749
declaring, 707-712
delegates, 814-815
Gyro, 707
performance, 714-718
polymorphism, 707
static type checking, 705
universal quantification, 707
GetAccessControl method, 1404
GetHashCode method, 625-628
GetResult method, 1636
GetType method, 540
Global Assembly Cache (GAC). See GAC (Global Assembly Cache)
goto statement, 366, 400-403
green bits, 105
group clause, query expressions, 953-960
GroupBy query operator, 1013-1015, 1025-1026
grouping query operators, 1013-1016
GroupJoin query operator, 1015-1016
groups, methods, 520-522, 799
GUIs (graphical user interfaces), events, 890-896
Gyro generic types, 707

How can we make this index more useful? Email us at indexes@samspublishing.com
handling

events, 844, 871-875
exceptions, 11, 407-409, 421-431, 1180-1183, 1193-1196
CLR (Common Language Runtime), 46-48
finally clause, 431
first-chance exceptions, 1187-1190
handler order, 1181
IntelliTrace, 1191-1192
try statements, 1183-1186
hash codes, 757-760
hash tables, nongeneric collection types, 757-760
Haskell programming language, 931-932
headers, 1224
methods, 502
heap allocated closures, 804
heaps, 1353
versus stacks, 469-478, 590
Hejlsberg, Anders, 57
Hello World program
compiling, 115
inspecting, 116-119
running, 116
writing, 114-115
helpers, interlocked, 1504-1506
hidden continuations, .NET Framework, 1531-1536
hiding, base class members, 672-674
Hindley, J. Roger, 347
homoiconicity
code, 73
expression trees, 1048-1050
horizontal partitioning, databases, 946-947
hosting, IronPython, 1130-1132
Hungarian notation, 892
Hypotenuse method, 344

I

ICloneable method, 764
ICloneable, 764
ICriticalNotifyCompletion interface, 1638
identifiers, naming, 358
identity key selector, 947-948
identity projection, select clause, 938-940
IDisposable interface, 601-607
resource cleanup, 444-448
IEnumerable interface, 388-389, 978-980
IEqualityComparer interface, 627
if statement, 358-363
IFormattable interface, 1357
Iif function, 284
ILanguage, 1654-1655
IL (Intermediate Language)
code, 27-28
generated, dumping, 1271-1273
round tripping, 26
ILDASM tool, 713, 1651
ILSpy, inspecting assemblies, 116-119
images, native, generation, 1271-1275
“immediate if” function, 284
immutability, 100
imperative languages versus declarative, 1516-1519
implementing
finalizers, 600
indexers, 582-583
interfaces, 695-699
implicit conversions, versus explicit, 301-319
implicit implementation, interfaces, 696-697
implicitly typed declarations, local variables, 218-223
implicitly typed local variables, 771
importing namespaces, 1231-1240
aliases, 1234-1235
extension methods, 1238-1240
name clashes, 1230
Increment method, 1504-1505
increment operators
expression statements, 355
overloading, 616-617
prefix and postfix, 297-299
indexers, 22, 580, 583-584
  defining, 580-582
  implementing, 582-583
  reflection, 1082-1083
indexing manual arrays, 297
IndexOutOfRangeException class, 1203
indirect invocation, expressions, 341-348
infoof operator, 1083
ingheritance, 654-659
  blocking, 671
  classes, 663-666
  single, 667-668
  constructors, 664
  interfaces, multiple, 669-670
  members, 665
initial capacity, constructor overloads, 761
initialization, zero-initialization, 591
initializers
  arrays, 234-236
  collection, 334-336, 706
  syntax, 772
  constructors, 591-592
  expressions, 66-67
  object, 329-334, 941-944
  properties versus fields, 334
initializing fields, 551-555
in-memory data, 914-915
  LINQ (Language Integrated Query), 921-923
InnerScope method, 214
INotifyProperty interfaces, 880-890
input validation, code, 410-413
inspecting, GAC (Global Assembly Cache), 1258-1260
inspecting assemblies, ILSpy, 116-119
installing
  .NET Framework, 103-112
  Visual Studio 2012, 122
instance constructors, 585-592
instance members, versus static members, 490-495
instances, 249
  delegates, 798-800
  creating, 343-347
  object, creating, 1289-1290
  types, 186-187
instantiating objects, 1106
instantiating types, 1068
instrumentation, System.Diagnostics
  namespace, 1388-1396
InsufficientMemoryException class, 1208
integer arithmetic, operators, 259
integral bitwise logical operators, 277
integral types, 190-194
integral values, enum values, converting to, 568-569
IntelliSense
  member access, 337-339
  query expressions, 922
IntelliTrace, debugging, 1191-1192
interface-based constraints, generic types, 721-727
interfaces, 20-21
  APIs (application programming interfaces), 6
  as contracts, 691-695, 1295-1297
  defining, 691-692, 1070-1071
  design recommendations, 693-695
  GUIs (graphical user interfaces), events, 890-896
  IConvertible, 644-645
  ICriticalNotifyCompletion, 1638
  IEnumerable, 388-389, 978-980
  IEqualityComparer, 627
  IFormattable, 1357
  IObserver, 905-908
  IObserver, 905-908
  IQueryProvider, 1052-1055
  IRule, 1375
  OOP (object-oriented programming), 662
  single-method, 693-694
types, 690-699
  versioning, 693
  zero-method, 693-694
interference, types, 738-741
interlocked helpers, 1504-1506
internal representation, arrays, 231
internal visibility, 1275-1276
InternalsVisibleTo attribute, 1276-1277
interning strings, 632
interoperability, WinRT (Windows Runtime) events, 896-898

interoperability facilities, CLR (Common Language Runtime), 49-50

interrupting blocked threads, 1457-1458

Intersect query operator, 1028

into clause, query expressions, 966-971

IntPtr value, System namespace, 1351-1353

InvalidOperationException class, 1215

invalidation

asynchronous delegates, 1427

expressions, 340-348

delegate, 341-348

method, 340-341

parallel, 1538-1539

invoking

deleates, 811-815

asynchronous invocation, 823-835

members, 1106-1107

I/O (input/output), 1399-1400, 1440-1441

asynchronous read and write operations, 1420-1433

caching, 1422

Common Log File System (CLFS), 1440

directories, 1402-1404

paths, 1405-1406

file system, monitoring activity, 1407-1409

FileInfo class, 1406-1407

isolated storage, 1440

listing drives, 1400-1402

memory-mapped files, 1437-1440

Open Packaging Convention (OPC), 1440

pipes, 1434-1436

readers, 1409-1410, 1411-1415

serial port communication, 1440

streams, 1415, 1433-1434

StreamReader class, 1417-1418

StreamWriter class, 1417-1418

writers, 1410-1415

IObservable interface, 905-908

IObserver interface, 905-908

IQueryable interface, 1052-1055

IronPython

dynamic keyword, 1128-1137

hosting, 1130-1132

IRule interface, 1375

is keyword, type checks, 637

is operator, 307-312

isolated storage, I/O (input/output), 1440

iteration statements, 352, 375

do...while, 379-380

for, 380-382

foreach, 382-390

while, 375-379

iterators, 62-63, 391-397

anonymous, 993

generating code for, 394

lazy evaluation, 990-999

LINQ to Objects, 984-990

J

J++ programming language, 8

jagged arrays, 236-237

JIT compilation, 39-41

JIT-generated code, dumping, 1271-1273

join clause, query expressions, 960-965

Join method, 1369

Join query operator, 1016-1018

joining query operators, 1016-1018

jump statements, 352

Just My Code feature, disabling, 422

K

Kennedy, Andrew, 60, 707

dkeys

compound, 956-960

computed, 956

identity, 947-948

simple, 956
LINQ (Language Integrated Query) 1691

keywords, 181-183
  ascending, 947
dynamic, 79-80, 1119-1121
dynamic type, 1121-1124
System.Dynamic type, 1126-1128
using, 1128-1137
lock, 1486-1489
orderby, 946-952
override, 678
primitive types, 1315
reuse, 183
reuse of, 735-736
syntax highlighting, Visual Studio, 183
type checks, is and as, 637
var, 218-219
Kleene closure operators, 61, 246

lambda expressions, 22, 69-71, 347-348
delegates, 807-809
expression trees, 1107-1110
Language Integrated Query (LINQ). See LINQ (Language Integrated Query)
language projections, WinRT (Windows Runtime), 1655-1658
language support, asynchronous programming, 91-95
languages
  dependencies, 1307-1308
  mini-languages, 1359
  mixing, 30-31
Last restriction operator, 1005
LastOrDefault restriction operator, 1005
late-bound invocation of methods, 1080-1081
late-bound property access, reflection, 1083
latency, asynchronous programming, 1552
Lazy class, 1353-1354
lazy evaluation, iterators, 990-999
lazy initiation, 1353-1354
LCG (Lightweight Code Generation), 1091
  Hello World program, 1091-1093
toy compiler for arithmetic expressions, 1093-1100
leaf nodes, expression trees, 1104-1105
leaks, 865
  tracing, SOS (Son of Strike), 865
let clause, 972-974
libraries
  BCL (Base Class Library), 11, 51, 1301-1303, 1372
  assemblies, 1304-1308
default project references, 1303-1304
  encoding text, 1371
  history, 53-54
  namespaces, 51-52, 1304-1306
  Object Browser, 1305-1306
  parsing text to objects, 1362-1363
  string methods, 1366-1369
  StringBuilder class, 1369-1371
  support, 11
  System namespace, 1311-1320, 1344-1356
  runtime, 1307-1308
  TPL (Task Parallel Library), 1444, 1515-1520
  library target, assemblies, 1245
  life cycle, threads, 1453-1458
  lifted operators, 276-277, 612-615
Lightning, 8-9
LINQ (Language Integrated Query), 57, 63-65, 913, 920-921, 975, 977, 1055
anonymous types, 941-944
benefits, 914-920
catalyst, 335
delegates, 819-823
expression trees, 1045
  homoiconicity, 1048-1050
  query expressions, 1045-1048, 1050-1055
  group clause, 953-960
  join clause, 960-965

How can we make this index more useful? Email us at indexes@samspublishing.com
LINQ to Objects, 977
  extension methods, 980-984
  IEnumerable interface, 978-980
  IEnumerator interface, 978-980
  iterators, 984-990
  lazy evaluation, 990-999
  query operator methods, 983
in-memory data, 921-923
methods, 766
MinLINQ, 1031
origins, 920
PLINQ (Parallel Language Integrated Query),
  102, 1036, 1041-1043
  AsOrdered operator, 1042-1043
  ForAll method, 1043-1045
  optimization, 1036-1040
  tweaking parallel querying behavior, 1043
query expressions
  from clause, 933-938
  into clause, 966-971
  let clause, 972-974
  orderby keyword, 946-952
  syntax, 931-934
  where clause, 944-947
query operators, 1000
  aggregation, 1019-1026
  grouping, 1013-1016
  joining, 1016-1018
  local, 1031-1033
  ordering, 1012-1013
  predicates, 1026-1027
  projection, 1007-1012
  remote, 1031-1033
  restriction, 1002-1007
  sequence persistence, 1029-1031
  sequencing, 1027-1029
  set theoretical, 1027-1029
  source generators, 1000-1002
query pattern, 1033
  methods, 1033-1034
  overloading query expression syntax, 1034-1036
relational databases, 923-929
XML (eXtensible Markup Language),
  929-931
LINQ to Objects, 977
  extension methods, 980-984
  IEnumerable interface, 978-980
  IEnumerator interface, 978-980
  iterators, 984-990
  lazy evaluation, 990-999
  query operator methods, 983
listing drives, 1400-1402
literals
  decimal, 199-200
  integral, 192-193
  real, 198
  string, 201-202
loading assemblies, 1283-1286
  at runtime, 1264-1271
  local query operators, 1031-1033
local variables, 212-223
  assignments, 215-216
  constants, 216-218
  declarations, 212-213
    implicitly typed, 218-223
  scope, 213-215
  type inference, 65-66
localizable applications, 1360
locating
  assemblies, 1267-1271
  operators, 611-612
  lock keyword, 1486-1489
  lock statement, 453-457
  locking on objects, 448-462
    lock statement, 453-457
  locks, 459-462
    ACID transactions, 461
    code generation, 457-458
    exposed, 1497-1498
logging
  stack traces, 1385-1386
  Windows Store applications, 1268
logic, cleanup, 408
logic programming, 96
logical operators, 277
  Boolean, 279-281
  enumerations, 277-278
  integral bitwise, 277
  non-short-circuiting, 620
  short-circuiting, 617-621
methods 1693

logs, event, 1388-1391
LongCount query operator, 1022
loops, 398-400
  ForEach, 1548-1550
  REPLs (read-eval-print-loops), 1130, 1364

M
magic strings, 1361
Main method, 25, 117, 176, 359, 501
  asynchronous, 1607-1609
  signature, 176-177
managed add-in framework, 1296-1298
managed code, 1446-1448
  executing, 24-31
Managed Debugging Assistants (MDAs), 1189
Managed Extensibility Framework (MEF), 1069,
  1077-1080, 1298
managed threads, 1458-1463
manifests, assembly, 26-27
manual array indexing, 297
manual callback plumbing, asynchronous
  methods and await expressions, 1597-1603
ManualResetEvent class, 1500-1502
MarshalByRefObject calculator, 1293-1294
marshaling services, COM interop, 1161
Max query operator, 1025
MDAs (Managed Debugging Assistants), 1189
measuring code performance, 1386-1388
MEF (Managed Extensibility Framework), 1069,
  1077-1080, 1298
member access, 336-339
  dynamic typing, 338-339
  encapsulation, 653-654
IntelliSense, 337-339
members, 22-23
  accessing, 1106-1107
  base class, hiding, 672-674
defaults, 658
elems, assigning values to, 565-566
inheritance, 665
invoking, 1106-1107
type, 486
  limiting access, 486-489
  static versus instance, 490-495
  visibility, 488-489
virtual
declaring, 680-681
overriding, 678-680
polymorphism, 676-687
memory, automatic management, CLR
  (Common Language Runtime), 43-46
memory pressure, GC (garbage collector), 1349
memory streams, 1416-1417
memory-mapped files, 1437-1440
MemoryStream class, 1417
messaging, named pipes, 1434
metadata, 28-30, 1058-1059
  Windows Metadata format, 1650-1652
meta-programming, 73
method calls, expression statements, 353-354
method invocation, expressions, 340-341
methods, 22, 408, 501, 546
  Add, 792
  anonymous, 63, 345-347
APM (Asynchronous Programming Model),
  1423-1428
arguments, 504
AsParallel, 1037-1040
asynchronous, 1584-1585
  control flow, 1595-1597
  declaring, 1585-1588
  execution, 1591-1595
Main, 1607-1609
manual callback plumbing, 1597-1603
refactoring, 1593
returning from, 1614-1619
state machine, 1610-1614
Bar, 479
BeginInvoke, 1427
calling, optional parameters, 513-516
compile, 1096-1100
CreateInstance, 728
Decrement, 1504-1505
defining, 501-502
Dispose, 603-607

How can we make this index more useful? Email us at indexes@samspublishing.com
Equals
GetHashCode consistency, 625-628
overloading operators, 622-632
overriding, 623-625
required properties, 625
Evaluate, 620
extension, 68-69, 524-534, 1214
defining, 526-528
versus expandos, 534
importing namespaces, 1238-1240
LINQ to Objects, 980-984
marking and finding, 531-534
overload resolution, 528-529
extern, 538-539
factory, 187
finalizer, 22
ForAll, 1043-1045
generic, 502, 523-524, 736-743
calling, 737
GetAccessControl, 1404
GetHashCode, 625-628
GetResult, 1636
GetType, 540
groups, 520-522, 799
headers, 502
Hypotenuse, 344
ICloneable, 764
Increment, 1504-1505
InnerScope, 214
Join, 1369
late-bound invocation of, 1080-1081
LINQ query pattern, 1033-1034
Main, 25, 117, 176, 359, 501, 1607-1609
signature, 176-177
naming, 1573-1575
op_Explicit, 641
overloading, 519-524
defining overloads, 519-520
resolution, 522-524
Pad, 1368-1369
Parallel.For, 1548-1550
Parallel.Invoke, 1539
parameters, 504
arrays, 510-511
declaring, 504-519
named, 511-512, 513-516
optional, 511-519
output, 508-510
reference, 507-508
value, 505-507
partial, 498, 534-538
PrintUsage, 491
Process.Start, 1396-1398
query operator, 983
QueueUserWorkItem, 1476-1478
ReadAllLines, 1410
refactoring, 540-545
ReferenceEquals, 628-630
reflection, 1080-1081
return type, specifying, 502-504
Run, 1522
SchedulePayment, 494
SetAccessControl, 1404
signatures, 176-177
Sort, 1377
string, 1366-1369
String.Concat, 1368
ToString, 676, 679, 686, 1357, 1369, 1385
Trim, 1368-1369
Workflow, 1639
MFCs (Microsoft Foundation Classes), 6
Min query operator, 1025
mini-languages, 1359
MinLINQ, 1031
mixing languages, 30-31
modifiers, CLR (Common Language Runtime), 675-676
modules, 16
assemblies, 1242-1244
monitoring
applications
event logs, 1388-1391
performance counters, 1391-1395
Moore, Gordon, 99
mscorlib assembly, 1306-1308
MulticastDelegate objects, 796, 836
multidimensional arrays, 238-239
multilanguage usability, .NET platform, 1058-1062
multiparadigm programming language, 95-97
multiple inheritance, interfaces, 669-670
multiple tasks, 1539-1540
multithreading, 855
interaction, events, 856
offloading computation, 1560
mumble types, 743
mutability, 468
mutable value types, 484-486
mutexes, 1489-1492
semaphores, 1492-1495

N
name clashes, namespaces, 1230
importation, 1230
named parameters, 332, 513-516, 765
methods, 511-512
syntax, 83
named pipes, 1434-1436
namespaces, 1221, 1240
aliases, 1234-1235
extern, 1235-1238
assemblies, 1224-1227
versus assemblies, 1304-1306
declaring, 1227-1230
folder structures, 210
importing, 1231-1240
aliases, 1234-1235
extension methods, 1238-1240
name clashes, 1230
name clashes, 1230
importation, 1230
naming conventions, 1229
organization, BCL (Base Class Library), 51-52
System, 1229, 1311
arrays, 1315-1318
BitInteger type, 1320-1322
complex numbers, 1322-1324
date and time values, 1327-1335
GC (garbage collector), 1344-1351
generating random numbers, 1324-1327
GUID values, 1335-1337
interacting with environment, 1339-1344
lazy initialization, 1353-1354
nullability, 1337-1338
primitive value types, 1311-1315
System.Math class, 1318-1320
tuple types, 1354-1356
Uri type, 1338-1339
System.Collections, 51
controlling processes, 1396-1398
ensuring code quality, 1374-1388
instrumentation, code access security, 1388-1396
System.Data, 51
System.Diagnostics, 1373, 1381-1383, 1398
System.Globalization, 51
System.IO, 51
System.Linq, 52
System.Net, 52
System.Numerics, 643, 1322-1324
System.Reflection, 52
System.Security, 52
System.ServiceModel, 52
System.Text, 52, 1369
System.Web, 52
System.Xml, 52
types, organizing in, 1221-1227
visibility, 1229-1230
Windows, 1229
naming
assemblies, 1249-1252
strong, 1252-1257
event handlers, 872
identifiers, 358
methods, 1573-1575
threads, 1460
naming conventions
fields, 548
namespaces, 1229
types, 209
n-ary ordering, query expressions, 949-951
Nathan, Adam, 602, 1353
native image generation, 41-43, 1271-1275
native interop, 1351-1353
network cards, time, 1554
new operator, creating objects, 324-336
New Project dialog (Visual Studio 2012), 127-128
NGEN (native image generation), 41-43, 1271-1275
non-case-sensitive suffixes, 193
nongeneric collection types, 755-764
queues, 761
stacks, 762-763
non-short-circuiting logical operators, 620
nontrivial ordering, query expressions, 948-949
nontrivial projections, 940-941
Notepad, 113
Hello World program, writing, 114-115
NotImplementedException class, 1215-1216
NotSupportedException class, 1193, 1216-1217
null, 275-276
null reference, 239-243
nullability, 248-249
operators, 612-615
nullable Boolean logic, 280-281
nullable conversions, 635
nullable types, 61
nullable value types, 243-249
arithmetic operators, 269
null-coalescing operators, 285-287
NullReferenceException, 1202-1203
numeric conversions, 634

O
Object Browser, 1305-1306
object initializers, 329-334
properties versus fields, 334
object instances, creating, 1289-1290
object model, expression trees, 1094-1096
object types, 203-205
ObjectDisposedException, 1206-1208
object-oriented programming (OOP). See OOP (object-oriented programming)
objects, 63-75, 249, 301, 349
creating, new operator, 324-336
delegates, 794, 842
anonymous function expressions, 801-802
closures, 802-807
combining, 835-842
expression trees, 810-811
instances, 798-800
invoking, 811-815
lambda expressions, 807-809
LINQ (Language Integrated Query), 819-823
types, 794-798
disposal, 440-441
IInspectable, 1654-1655
initializers, 941-944
instantiating, 1106
locking on, 448-462
lock statement, 453-457
parsing text to, 1362-1363
reflection info, types, 720
SafeHandle, 607
space leaks, 806-807
Timer, rooting, 1580
TreeNode, 668
types, 186-187
offloading computation, multithreading, 1560
OffType restriction operator, 1006-1007
OOP (object-oriented programming), 95, 649-653, 699, 790-791
.NET platform, 10
classes, 662
abstract, 688-690
inheritance, 663-674
encapsulation, 653-654
inheritance, 654-659
classes, 663-674
interface types, 690-699
interfaces, 662
polymorphism, 659-661
virtual members, 676-687
protected accessibility, 674-676
opExplicit method, 641
OPC (Open Packaging Convention), 1440
Open Packaging Convention (OPC), 1440
operations, type parameters, 718-720
operators, 299, 609-610
arithmetic, 258-259
  character arithmetic, 262-263
decimal arithmetic, 261-262
floating-point arithmetic, 260-261
integer arithmetic, 259
nullable value types, 269
overflow checking, 263-269
unary plus and minus, 263
arity, 252
as, 312-317
AsOrdered, 1042-1043
associativity, 253-254
conditional, 281-284, 617-621
conversion, 633
  built-in, 634-638
conversions
  IConvertible interface, 644-645
  System.Convert, 644
  TypeConvert class, 645-646
user-defined, 638-644
defining, 610-611
equality
  overloading, 621
  pairwise declaration, 621-622
finding, 611-612
infoof, 1083
is, 307-312
Kleene, 61
Kleene closure, 246
lifted, 612-615
logical, 277
  Boolean, 279-281
  enumerations, 277-278
  integral bitwise, 277
  non-short-circuiting, 620
  short-circuiting, 617-621
new, creating objects, 324-336
nullability, 612-615
null-coalescing, 285-287
overloading, 609-610, 647
  compound assignments, 617
decrement operators, 616-617
drawbacks, 610
equality operators, 621
Equals method, 622-632
increment operators, 616-617
support for, 615-622
postfix increment and decrement, 297-299
expression statements, 355
overloading, 616-617
precedence, 252-253
prefix increment and decrement, 297-299
expression statements, 355
overloading, 616-617
query, 819, 1000
  aggregation, 1019-1026
  grouping, 1013-1016
  joining, 1016-1018
  local, 1031-1033
  ordering, 1012-1013
  predicates, 1026-1027
  projection, 1007-1012
  remote, 1031-1033
  restriction, 1002-1007
  sequence persistence, 1029-1031
  source generators, 1000-1002
relational, 275-277
  lifted, 276-277
  pairwise declaration, 621-622
result type, 284-285
shift, 274-275
string concatenation, 269-274
translation, 633
typeof, 319-322
optimization
code, 256, 282
PLINQ (Parallel Language Integrated Query), 1036-1040
optional parameters
caller info attributes, 516-519
methods, 511-519
  declaring, 512-513
orderby keyword, query expressions, 946-952
ordering
  custom, query expressions, 952
  n-ary, query expressions, 949-951
  nontrivial, query expressions, 948-949
ordering

secondary, 949-951
syntactical, 741
ordering query operators, 1012-1013
organizing types in namespaces, 1221-1227
OutOfMemoryException, 1208-1209
output parameters, methods, 508-510
overflow checking, arithmetic operators, 263-269
OverflowException, 1201-1202
overflowing call stacks, 413
overloading
asynchronous patterns, 1573-1575
methods, 519-524
defining overloads, 519-520
resolution, 522-524, 528-529
operators, 609-610, 647
compound assignments, 617
decrement, 616-617
drawbacks, 610
equality, 621
Equals method, 622-632
increment, 616-617
support for, 615-622
query expression syntax, 1034-1036
overloads, simple, 1544-1547
override keyword, 678
overriding
Equals method, 623-625
virtual members, 678-680

P

Pad method, 1368-1369
page designer, ASP.NET, 155-157
pairwise declaration, relational and equality operators, 621-622
parallel For loops, 1543-1548
parallel programming, 100-101, 1513, 1550
continuations, 1531-1536
data parallelism, 1542-1550
declarative languages, 1516-1519
ForEach loops, 1548-1550
imperative languages, 1516-1519
parallel invocation, 1538-1539
task parallelism, 1519-1520
creating tasks, 1520-1523
tasks
cancellation, 1536-1538
dealing with errors, 1524-1531
multiple, 1539-1540
retrieving results, 1523-1524
threads, pros and cons, 1514-1515
TPL (Task Parallel Library), 1515-1520
architecture, 1515-1516
Parallel.For method, 1548-1550
Parallel.Invoke method, 1538-1539
ParallelQuery type, 1041-1042
parameterized threads, 1452
parameters
dynamic, 1138
methods, 504
arrays, 510-511
declaring, 504-519
named, 511-512, 513-516
optional, 511-519
output, 508-510
reference, 507-508
value, 505-507
named, 765
syntax, 83
renaming, public APIs, 812
types, operations, 718-720
parsing
strings, 1312-1314
text to objects, 1362-1363
partial methods, 498, 534-538
partial types, 496-498
partitioners, parallelism, 1549-1550
paths, directories, 1405-1406
PathTooLongException, 1193
patterns
asynchronous programming, 89-91, 1564
APM (asynchronous programming model), 1564-1569
EAP (Event-based Asynchronous Pattern), 1569-1571
exception behavior, 1576-1578
method naming, 1573-1575
overloading, 1573-1575
progress reporting, 1575-1576
synchronization behavior, 1578-1579
TAP (Task-based Asynchronous Pattern), 1571-1573, 1579-1584
events, 871-880
queries, 960
query expressions, query expressions, 952
PE/COFF files, 1420
performance, code, measuring, 1386-1388
performance counters, monitoring applications, 1391-1395
per-thread state, 1481-1482
PIAs (primary interop assemblies), 50
  embedding, 1172-1174
P/Invoke, 49
pipes, named, 1434-1436
PipeStream class, 1434
plain use, delegates, 849-850
PLINQ (Parallel Language Integrated Query), 65, 102, 1036, 1041-1043
  AsOrdered operator, 1042-1043
  ForAll method, 1043-1045
  optimization, 1036-1040
  tweaking parallel querying behavior, 1043
pointers, C-style function pointers, 794, 800
polymorphism, 659-661
generic types, 707
virtual members, 676-687
pools, threads, 1474-1482
Portable Class Library, Visual Studio 2012, 127
postfix, 257
postfix increment and decrement operators, 297-299
  expressions, 355
  overloading, 616-617
PowerShell (Windows), 178
precedence, operators, 252-253
predicate query operators, 1026-1027
preemptive scheduling, 454-455
prefix increment and decrement operators, 297-299
  expression statements, 355
  overloading, 616-617
Premium Edition, Visual Studio, 121
preprocessing directives, 224-226
primitive types, 19
primitive value types
  keywords, 1315
  System namespace, 1311-1315
type names, 1315
primitives
  synchronization, 1482-1510
  atomicity, 1483-1486
  interlocked helpers, 1504-1506
  lock keyword, 1486-1489
  locks, 1495-1498
  monitors, 1486-1489
  mutexes, 1489-1492
  semaphores, 1492-1495
  signaling events, 1498-1503
PrintException command, 1190
printing exception text, 918
PrintUsage method, 491
Process class, 1374, 1396
processes
  controlling, 1396-1398
  starting, 1396-1398
Process.Start method, 1396-1398
Professional Edition, Visual Studio, 121
programming
  asynchronous, 88-95, 101, 1551-1552, 1561, 1641
  APM (Asynchronous Programming Model), 899
  arbitrary control flow, 1627-1630
  await expressions, 1584-1585
  building awaitable types, 1634-1640
  language support, 91-95
  latency, 1552
  methods, 1584-1597
  patterns, 89-91, 1564-1584
  saving evaluation state, 1630-1634
  scalability, 1561-1564
  WinRT (Windows Runtime), 1656
dynamic, 1119, 1174
COM interop, 1159-1174
deferred overload resolution, 1124-1126
DLR (Dynamic Language Runtime), 1137
dynamic keyword, 1119-1122, 1128-1137
dynamic languages, 75-88
extensible calculator, 815-819
functional, 789-794
meta-73
OOP (object-oriented programming), 649-653, 699, 790-791
abstract classes, 688-690
classes, 662
encapsulation, 653-654
inheritance, 654-659, 663-674
interface types, 690-699
interfaces, 662
polymorphism, 659-661, 676-687
protected accessibility, 674-676
virtual members, 676-687
parallel, 100-101, 1513, 1550
data parallelism, 1542-1550
ForEach loops, 1548-1550
invocation, 1538-1539
multiple tasks, 1539-1540
task cancellation, 1536-1538
task parallelism, 1519-1542
threads, 1514-1515
TPL (Task Parallel Library), 1515-1520
reactive, 898-905
side-effect-free, 101
Win32, 6
project folders, usage-first development, 496
projection query operators, 1007-1012
projects
assemblies, 1262-1263
Code Analysis, 545-546
code metrics, calculating, 541
maintainability, 541
Visual Studio 2012, 127
code, 143-148
database mappers, 160-165
designers, 148-158
properties, 130
team development, 171-172
unit testing, 167-171
propagation, exceptions, 429-431, 1619-1624
properties, 22, 185, 575, 579, 583-584
assemblies, 1245-1249
auto-implemented, 73-75, 578, 882-883
declaring, 575-578
Equals method, 625
object initializers, 334
reflection, 1082-1083
late-bound access, 1083
trivial, 73
protected accessibility, OOP (object-oriented programming), 674-676
public APIs, renaming parameters, 812
Python
IronPython, dynamic keyword, 1128-1137
types, 1132-1136

Q

queries
optimization, 1029
patterns, 960
query expression, patterns, 952
query expressions
into clause, 966-971
expression trees, 1050-1055
from clause, source selection, 933-938
group clause, 953-960
IntelliSense, 922
join clause, 960-965
let clause, 972-974
orderby keyword, 946-952
ordering
custom, 952
n-ary, 949-951
nontrivial, 948-949
secondary, 949-951
overloading syntax, 1034-1036
select clause, 938-944
syntax, 931-934
translation, LINQ (Language Integrated Query), 1045-1048
where clause, 944-947
query operator methods, LINQ to Objects, 983
query operators, 819, 1000
aggregation, 1019-1026
grouping, 1013-1016
joining, 1016-1018
local, 1031-1033
ordering, 1012-1013
predicates, 1026-1027
projection, 1007-1012
remote, 1031-1033
restriction, 1002-1007
sequence persistence, 1029-1031
sequencing, 1027-1029
set theoretical, 1027-1029
source generators, 1000-1002
query pattern (LINQ), 1033
methods, 1033-1034
overloading query expression syntax, 1034-1036
querying behavior, PLINQ (Parallel Language Integrated Query), tweaking, 1043
QueueUserWorkItem method, 1476-1478
quotations, 72-73

R
RAD (Rapid Application Development), 148
raising events, 855-857
random numbers, generating, System namespace, 1324-1327
Range source generator, 1001
Rapid Application Development (RAD), 148
reactive applications, 845-852
delegates, 846-849
plain use, 849-850
events, .NET, 850-852
Reactive Extensions (RX), 843
reactive programming, 898-905
read operations (asynchronous), 1420-1433
ReadAllLines method, 417, 1410
readers, files, 1409-1410
read-eval-print loops (REPLs), 98, 1364
read-only fields, 555-557
real literals, 198
red bits, 105
redundant assignments, 295
reentrant calls, guarding against, 1604
refactoring
.NET Framework, 1280-1281
methods, 540-545
reference assemblies, 1311
reference parameters, methods, 507-508
reference types, 188-190
restrictions, 735-736
versus value types, 466-470
referenced assemblies, loading, 1266-1267
ReferenceEquals method, 628-630
references
BCL (Base Class Library), 1303-1304
weak, 719
referencing assemblies, 1262-1264
reflection, 479, 1057, 1063, 1117
APIs (application programming interfaces), 179
application extensibility, 1069-1080
assemblies, 1282-1286
custom attributes, 1085-1091
events, 1083-1084
expression trees, 1101
API (application programming interface), 1103-1114
compiler-generated, 1101-1103
fields, 1084-1085
indexers, 1082-1083
LCG (Lightweight Code Generation), 1091
Hello World program, 1091-1093
toy compiler for arithmetic expressions, 1093-1100
methods, 1080-1081
properties, 1082-1083
late-bound access, 1083
as relational database, 1082-1083
System.Type, 1064-1066
typing, 1058-1063, 1066-1068
reflection info object, types, 720
regular expressions, 1363-1366

How can we make this index more useful? Email us at indexes@samspublishing.com
relational databases, 915-919
  horizontal partitioning, 946-947
  LINQ (Language Integrated Query), 923-929
  reflection, 1082-1083
  vertical partitioning, 944
relational operators, 275-277
  lifted, 276-277
  pairwise declaration, 621-622
relative time, 1333-1335
remote query operators, 1031-1033
remoting, flavors, 1290-1293
remove accessors, 857-861
renaming parameters, public APIs, 812
Repeat source generator, 1001
REPLs (read-eval-print-loops), 77, 98, 1130, 1364
resource cleanup, 438
  garbage collection, 438-440
  IDisposable, 444-446
  object disposal, 440-441
  using statement, 441-444
resource management, 352
restriction query operators, 1002-1007
restrictions
  compilers, 666
  reference types, 735-736
  value types, 735-736
result type, operators, 284-285
results
  tasks, retrieving, 1523-1524
  rethrowing, exceptions, 427-429
  return statement, 404-406
  return type
    methods, specifying, 502-504
    return values, 1179
  reuse, keywords, 183
rethrow Polish notation, 257
Reverse query operator, 1029
rooting, Timer objects, 1580
rules, defining, 24
Run method, starting tasks, 1522
running
  code, 116
  finalizers, 597-600
  trivial console application, 179-180
runtime
  assembly strong-name verification, 1257-1258
  bootstrapping, CLR (Common Language Runtime), 33-35
  loading assemblies at, 1264-1271
  runtime binder, 1120
  runtime disasters, code, 413-416
  runtime libraries, 1307-1308
  runtime shim, .NET Framework, 110-111
  runtime type versus compile type, 206
  RX (Reactive Extensions), 843, 901, 911
S
SafeHandle objects, 607
safety, types, 1345
safety guarantees, co- and contravariance, 748-749
Sandcastle project code, 230
saving evaluation state, 1630-1634
scalability, asynchronous programming, 1561-1564
SchedulePayment method, 494
scheduling abstracting, 1481
scope, local variables, 213-215
secondary ordering, query expressions, 949-951
security, CAS (Code Access Security), 258
SEH (structured exception handling), 1175
select clause, projection, 938-944
Select projection query operator, 1007
selection statements, 352, 358
  if, 358-363
  switch, 363-375
SelectMany projection query operator, 1008-1010
semantics, versus syntax, 619
semaphores, 1489, 1492-1495
sequence persistence query operators, 1029-1031
SequenceEqual query operator, 1027
sequencing query operators, 1027-1029
serial port communication, 1440
Server Explorer, 158-160
set theoretical query operators, 1027-1029
SetAccessControl method, 1404
shift operators, 274-275
shims, CLR (Common Language Runtime), 34
short-circuiting logical operators, 617-621
side-effect-free programming, 101
signaling events, 1498-1503
signatures
  entry points, 177-179
  methods, 176-177
signing keys, strong-name, 1253-1255
Silverlight, 122, 153
simple keys, 956
simple overloads, 1544-1547
single inheritance, classes, 667-668
Single restriction operator, 1006
single-dimensional arrays, 231-233
single-line comments, 223-224
single-method interfaces, 693-694
SingleOrDefault restriction operator, 1006
singletons, 494
SIPs (software isolated processes), 38
64-bit addressing, limitations, 1506
Skip restriction operator, 1003
SkipWhile restriction operator, 1003
Smye, Don, 60
snippets
  common tasks, 490
  writing, 1200
software isolated processes (SIPs), 38
software transactional memory (STM), 101
Solution Explorer (Visual Studio 2012), 129-130
solutions, assemblies, 1262-1263
Sort method, 1377
SOS (Son of Strike), tracing leaks, 865
source control, Visual Studio 2012, 172
source generators, query operators, 1000-1002
space leaks, 806-807
specialized collection types, 786-787
SpinLock struct, 1498
splash screen, Visual Studio 2012, 124-125
stack spilling, evaluation state, saving, 1630-1634
stack traces, logging, 1385-1386
stack-allocated closures, 804
StackOverflowException, 1209-1212
stacks
  evaluation, 255-258
  versus heaps, 469-478, 590
  nongeneric collection types, 762-763
starting code, 1450-1453
starting processes, 1396-1398
statement trees, 1110-1114
statements, 351-353, 406
blocks, 351, 356
checked contexts, 352
declarations, 351, 357-358
empty, 355-356
exception handling, 352
expression, 351, 353
assignments, 354-355
decrement operators, 355
increment operators, 355
method calls, 353-354
  versus, 1102-1103
foreach, 757
  hidden cast, 841-842
goto, 366, 400-403
iteration, 352, 375
do...while, 379-380
for, 380-382
foreach, 382-390
while, 375-379
iterators, 391-397
jump, 352
lock, 453-457
resource management, 352
return, 404-406
selection, 352, 358
  if, 358-363
  switch, 363-375
switch, 317-319
  enums, 573-574
try, exception handling, 1183-1186
try-catch-finally, 1180
unchecked contexts, 352
using, resource cleanup, 441-444

How can we make this index more useful? Email us at indexes@samspublishing.com
static classes, 527, 595
static constructors, 592-595
static languages versus dynamic, 77-79
static members versus instance members, 490-495
static type checking, 705
static typing, 80, 301
versus dynamic, 207
STM (software transactional memory), 101
StopOnException command, 1190
stopping threads, 1454-1456
Stopwatch, measuring code performance, 1386-1388
storage, custom attributes, 1088-1089
Stream class, 1415-1434
StreamReader class, 1414-1415, 1417-1418
streams, 1415, 1433-1434
asynchronous read and write I/O operations, 1420-1433
memory, 1416-1417
StreamReader class, 1417-1418
StreamWriter class, 1417-1418
StreamWriter class, 1414-1415, 1417-1418
string methods, 1366-1369
string representations, enums, 566-567
string types, 201-202
StringBuilder class, 1369-1371
String.Concat method, 1368
strings
character sequences, 1366
checking, 1366-1367
enum values, converting to, 569
format, 1357-1362
interning, 632
magic, 1361
parsing, 1312-1314
verbatim, 1366
strong naming assemblies, 1252-1257
strong-name signing keys, 1253-1255
strong-name verification, assemblies, 1257-1258
structs, 465, 589-591
versus classes, 466-486
CLI (Common Language Infrastructure), 19-20
SpinLock, 1498
StyleCop, 362
subexpressions, 254
subst command, 1401
suffixes, non-case-sensitive, 193
Sum query operator, 1023-1024
switch statement, 317-319, 363-375
ums, 573-574
switches, context, 1514
symbols, defining for conditional compilation, 225
Syme, Don, 707
synchronization, 1443-1444, 1506-1510, 1511
asynchronous await expressions, 1603-1607
atomicity, 1483-1486
barriers, 1506
behavior, asynchronous programming, 1578-1579
CountdownEvent, 1479
primitives, 1482-1510
atomicity, 1483-1486
interlocked helpers, 1504-1506
lock keyword, 1486-1489
locks, 1495-1498
monitors, 1486-1489
mutexes, 1489-1492
semaphores, 1492-1495
signaling events, 1498-1503
SynchronizationContext, 1506
synchronous processing, versus asynchronous, 1556
syntactical ordering, 741
syntax
cast, 303
collection initializers, 772
named parameters, 83
overloading query expression, 1034-1036
versus semantics, 619
syntax highlighting, keywords, Visual Studio, 183
System assembly, 1306-1308
System namespace, 51, 1229, 1311
arrays, 1315-1318
BitInteger type, 1320-1322
complex numbers, 1322-1324
date and time values, 1327-1335
GC (garbage collector), 1344-1351
generating random numbers, 1324-1327
GUID values, 1335-1337
interacting with environment, 1339-1344
lazy initiation, 1353-1354
nullability, 1337-1338
primitive value types, 1311-1315
System.Math class, 1318-1320
tuple types, 1354-1356
Uri type, 1338-1339
System.Collections namespace, 51
System.Convert class, 644
System.Core assembly, 1308-1311
System.Data namespace, 51
System.Delegate type, 895
System.Diagnostics namespace, 51, 1373, 1381-1383, 1398
controlling processes, 1396-1398
ensuring code quality, 1374-1388
instrumentation, 1388-1396
System.Drawing types, 363-365
System.Dynamic type, 1126-1128
System.Enum type, 566-569
System.Exception base class, 1198
System.Globalization namespace, 51
System.IO namespace, 51
System.Linq namespace, 52
System.Net namespace, 52
System.Numerics namespace, 643, 1322-1324
System.Object class, 306, 702
banning, 204-205
Equals method
GetHashCode consistency, 625-628
overloading operators, 622-632
overriding, 623-625
required properties, 625
performance, 703
ReferenceEquals method, 628-630
System.Reflection namespace, 52
systems, type, 17-24
System.Security namespace, 52
System.ServiceModel namespace, 52
System.Text namespace, 52, 1369
System.Type type, 319, 1064-1066
System.Web namespace, 52
System.Windows namespace, 52
System.Xml namespace, 52

T

tail calls, 1212
Take restriction operator, 1003-1005
TakeWhile restriction operator, 1003-1005
TAP (Task-based Asynchronous Pattern), 1522, 1571-1573, 1583-1584, 1641
methods, implementing, 1579-1583
Task constructor, creating tasks, 1520-1521
Task Manager, non-responsive applications, 1560
Task Parallel Library (TPL). See TPL (Task Parallel Library)
task parallelism, 1519-1520
creating, 1520-1523
Task-based Asynchronous Pattern (TAP). See TAP (Task-based Asynchronous Pattern)
TaskCreationOptions class, 1522
TaskFactory class, creating tasks, 1520-1522
tasks
cancellation, 1536-1538
continuations, 1533-1534
creating, 1520-1523
dealing with errors, 1524-1531
multiple, 1539-1540
creating, 1520-1523
parallelism, 1519-1520
retrieving results, 1523-1524
starting, 1520-1523
methods, implementing, 1579-1583
TAP (Task-based Asynchronous Pattern), 1522, 1571-1573, 1583-1584, 1641
Task Manager, non-responsive applications, 1560
TaskScheduler class, 1540-1542
team development, Visual Studio 2012 projects, 171-172
Team Foundation Server (TFS), 171-172
test-driven development (TDD), 1373
text
  encoding, 1371
  formatting, 1357-1362
    format strings, 1357-1362
    IFormattable interface, 1357
  parsing to objects, 1362-1363
TextWriter class, 1413-1414
TFS (Team Foundation Server), 171-172
this reference, 456
Thread class, 1448-1453
ThreadAbortException, 458
treading apartments, 1461-1463
threading state, APM (Asynchronous Programming Model), 1423-1428, 1424-1427
thread-local data, 1547-1548
thread-local storage, 1470-1471
ThreadLocal class, 1467-1470, 1482
threads, 1443-1444, 1446, 1514-1515
  background, 1458-1460
  blocked, interrupting, 1457-1458
  creating, 1448-1450
  cross-threading violations, 1443
d debugging techniques, 1471-1474
extceptions, 1463-1464
foreground, 1458-1460
frozen, 1471-1474
ideal number, 1514-1515
IDs, 1461
life cycle, 1453-1458
managed, 1458-1463
naming, 1460
parameterized, 1452
per-thread state, 1481-1482
pools, 1474-1482
starting, 1450-1453
stopping, 1454-1456
Thread class, 1448-1453
thread pools, .NET 4.0, 1540-1542
threading apartments, 1461-1463
thread-local storage, 1470-1471
thread-specific state, 1464-1471
Threads command, 1191
thread-safe collection types, 778-786
thread-specific state, 1464-1471
ThreadStatic attribute, 1464-1467
throwing exceptions, 420-421, 1196-1198
  rethrowing, 427-429
time
calendar systems, 1331-1332
date and time values, 1327-1335
network cards, 1554
relative, 1333-1335
zones, 1332-1333
Timer objects, rooting, 1580
threads, accuracy, 849
TimeSpan type, 329-330
TimeSpan value, 1333-1335
TimeZone class, 1332-1333
ToArray query operator, 1030
ToDictionary query operator, 1030-1031
ToList query operator, 1030
ToString method, 676, 679, 686, 1369, 1385
  formatting text, 1357
TPL (Task Parallel Library), 101, 1444, 1515-1520
  architecture, 1515-1516
  tracing leaks, SOS (Son of Strike), 865
type inference, 76-77
types 1707

type names, primitive value types, 1315
type systems
  CLR (Common Language Runtime), 301
  static typing, 301
TypeConvert class, 645-646
TypelnitializationException, 1204-1205
type of operator, 319-322
types, 184-190, 249, 301, 319, 349, 463-465, 499, 1066-1068
  AggregateException, 1526-1527
  unwrapping, 1624-1627
anonymous, 66
  LINQ (Language Integrated Query), 941-944
array, broken covariance, 745-747
awaitable, building, 1634-1640
background, 247
BitArray, 763
BitInteger, 1320-1322
boxing, 478-483
built-in, 190-212
  aliases, 212
  Boolean, 200-201
  decimal, 199-200
  floating-point, 194-198
  integral, 190-194
  object, 203-205
  string, 201-202
checks, is and as keywords, 637
code, 184-185
collection, 701-703, 755, 787
generic, 765-777
  nongeneric, 755-764, 757-760
  specialized, 786-787
  thread-safe, 778-786
compile time, 206
cr e concrete, constructors, 326
constituent, 577-578
decomposing, 465
degl e getes, 794-798
duck typing, 336
dynamic, 1121-1122
dynamic typing, 205-206, 312
  member access, 338-339
enums, underlying, 564
equality checks, 275-276
exception, 1201
  AggregateException, 1218-1220
  ArgumentException, 1201
  ArgumentException, 1193, 1213
  ArgumentNullException, 1193, 1213-1214
  ArgumentOutOfRangeException, 1214-1215
  ArrayTypeMismatchException, 1204
defining, 1198-1201
  DirectoryNotFoundException, 1193
  DivideByZeroException, 1201
  ExecutionEngineException, 1212-1213
  FileNotOpenedException, 1193
  FormatException, 1217-1218
  IndexOutOfRangeException, 1203
  InsufficientMemoryException, 1208
  InvalidCastException, 1203
  InvalidOperationExce ption, 1215
  NotImplementedException, 1215-1216
  NotSupportedExce ption, 1193
  NotSupportedException, 1202-1203
  ObjectDisposedException, 1206-1208
  OperationException, 1208-1209
  OverflowException, 1201-1202
  PathTooLongException, 1193
  StackOverflowException, 1209-1212
  TypelnitializationException, 1204-1205
  UnauthorizedAccessException, 1193
ExceptionDispatchInfo, 1624
exceptions, NotSupportedExce ption, 1216-1217
ExpressionVisitor, 1114-1117
canstraints, 720-721
  contravariance, 743-754
covariance, 743-754
  declaring, 707-712
delegates, 814-815
  Gyro, 707
  polymorphism, 707
  static type checking, 705
  universal quantification, 707

How can we make this index more useful? Email us at indexes@samspublishing.com
heaps versus stacks, 469-478
instances, 186-187, 319
instantiating, 1068
interface, 690-699
interference, 738-741
local variable inference, 65-66
members, 22-23, 486
limiting access, 486-489
static versus instance, 490-495
visibility, 488-489
mumble, 743
mutable value, 484-486
namespaces, organizing in, 1221-1227
naming conventions, 209
nullable, 61
nullable value, 243-249
objects, 186-187
ParallelQuery, 1041-1042
parameters, operations, 718-720
partial, 496-498
performance, 714-718
primitive, 19
Python, 1132-1136
reference, 188-190
restrictions, 735-736
versus value, 466-470
reflection info object, 720
runtime, 206
safety, 18-19, 1345
structs versus classes, 466-486
System.Delegate, 895
System.Drawing, 363-365
System.Dynamic, 1126-1128
System.Enum, 566-569
systems, 17-24
System.Type, 319, 1064-1066
TimeSpan, 329-330
tuple, 1354-1356
Uri, 1338-1339
value, 188
boxed, 622
restrictions, 735-736
variables, 187-188
visibility, 486-488
Windows.ApplicationModel, 1667
Windows.Data, 1667
Windows.Devices, 1668
Windows.Foundation, 1668
Windows.Graphics, 1668
Windows.Media, 1668
Windows.Networking, 1668
Windows.Security, 1668
Windows.Storage, 1668
Windows.System, 1668
Windows.UI, 1668

U

Uls (user interfaces)
frameworks, events, 885-890
programming, INotifyPropertyChanged, 880-890
Visual Studio 2012, 125-126
Ultimate Edition, Visual Studio, 121
unary expressions, 1105
unary plus and minus, 263
UnauthorizedAccessException, 1193
unboxing conversions, 637-638
unchecked arithmetic, 265-266, 352
unhandled exceptions
.NET 4.0, 1527-1529
.NET 4.5, 1530-1531
Unified Code Object Model, 98
Union query operator, 1028
unit testing, Visual Studio 2012 projects, 167-171
universal quantification, generic types, 707
UnobservedTaskException event, 1529-1530
unsafe code, 1318
unwrapping AggregateException type, 1624-1627
Uri, 1338-1339
usage-first development, project folders, 496
user mode scheduling, 1445
user-defined conversions, 609, 633, 638-644
using directive, System.Text namespace, 1369-1370
using statement, resource cleanup, 441-444
volatile fields

value parameters, methods, 505-507
value types, 188
boxed, 622
versus reference types, 466-470
restrictions, 735-736
values
date and time, 1327-1335
DateTimeOffset, 1335
enums
converting integral values to, 568-569
converting strings to, 569
fields, automatic assignment, 552
GUID, 1335-1337
IntPtr, 1351-1353
return, 1179
TimeSpan, 1333-1335
var keyword, 218-219
variables, 249
assignments, 287-288-290
definite, 292-296
redundant, 295
captured, 802-807
declarations, 288-290
foreach loop, scoping, 805-806
local, 212-223
assignments, 215-216
constants, 216-218
declarations, 212-213
implicitly typed declarations, 218-223
scope, 213-215
reference types, 188-190
types, 187-188
value, 188
variance
coop- and contravariance, 743-754
contravariance, 85-88
covariance, 85-88
when to use, 753-754
verbatim strings, 1366
versioning
assemblies, 1249-1252
interfaces, 693

versions, .NET Framework, 103-107
vertical partitioning, relational databases, 944
VES (Virtual Execution System), 13
virtual dispatch, 683-687
Virtual Execution System (VES), 13
virtual members
declaring, 680-681
overriding, 678-680
polymorphism, 676-687
visibility
assemblies, 1274-1277
internal, 1275-1276
namespaces, 1229-1230
types, 486-488
members, 488-489
Visual Basic, 7
aggregates, 1026
Visual Basic .NET, 15
Visual Studio 2012, 119-120, 123-126, 173
code editor, 131-133
deditions, 120-121
expression, 122
installing, 122
keywords, syntax highlighting, 183
New Project dialog, 127-128
project types, 127
projects
build support, 134-138
code, 143-148
database mappers, 160-165
debug support, 139-142
designers, 148-158
properties, 130
team development, 171-172
unit testing, 167-171
Solution Explorer, 129-130
source control, 172
splash screen, 124-125
templates, 127
UI elements, 124-126
VSTO (Visual Studio Tools for Office), 157-158
Visual Studio Team System (VSTS), 171-172
volatile fields, 559-563
von Neumann machine model, 343
VSTO (Visual Studio Tools for Office), 157-158
VSTS (Visual Studio Team System), 171-172

W
WaitHandle class, 1502-1503
WCF (Windows Communication Foundation), 53, 105, 694
weak references, 719
   GC (garbage collector), 1350-1351
web services, .NET platform, 12
WF (Workflow Foundation), 53
where clause, filtering, 944-947
Where restriction operator, 1002-1003
while statement, 375-379
whitespace sensitivity, 360-361
Win32 programming, 6
Windows Communication Foundation (WCF), 53, 694
Windows Distributed interNet Applications Architecture (DNA), 8
Windows Forms Application, Visual Studio 2012, 127
   designer, 148-150
Windows Management Instrumentation (WMI), 1396
Windows Metadata format, 1650-1652
Windows namespace, 1229
Windows OS, garbage collection, 603-604
Windows PowerShell, 178, 1396
Windows Presentation Foundation (WPF), See WPF (Windows Presentation Foundation)
Windows Runtime (WinRT), 16, 843
   Windows Shell, execute behavior, 1396-1398
Windows Store application, WinRT (Windows Runtime), 1644-1646
Windows Workflow Foundation (WWF), 153-154
Windows.ApplicationModel type, 1667
Windows.Data type, 1667
Windows.Devices type, 1668
Windows.Foundation type, 1668
Windows.Graphics type, 1668
Windows.Media type, 1668
Windows.Networking type, 1668
Windows.Security type, 1668
Windows.Storage type, 1668
Windows.System type, 1668
Windows.UI type, 1668
winexe target, assemblies, 1245
winmdobj target, assemblies, 1245
WinRT (Windows Runtime), 16, 843, 1643-1644, 1669
   APIs (application programming interfaces), 1667-1668
   Windows Store application, 1644-1646
   asynchronous programming, 1656
   components
      activation, 1653-1655
      building, 1662-1665
      creating, 1658-1667
      debugging, 1667
      using, 1665-1667
      writing, 1658-1662
   events, interoperability, 896-898
   IInspectable object, 1654-1655
   language projections, 1655-1658
   Windows Metadata format, 1650-1652
   WMI (Windows Management Instrumentation), 1396
   Workflow Foundation (WF), 53
   Workflow method, 1639
   WPF (Windows Presentation Foundation), 53, 122, 175
      data binding, 884
      designer, 151-153
   WPF Application, Visual Studio 2012, 127
   write operations (asynchronous), 1420-1433
   writers, files, 1410-1415
   writing
      code, 114-115
      snippets, 1200
   WinRT (Windows Runtime) components, 1658-1662
   WWF (Windows Workflow Foundation), 153-154
X

XAML (Extensible Application Markup Language), 122, 497-498
Xcopy, deployment, 1264-1265
XML (eXtensible Markup Language), 919-920
    LINQ (Language Integrated Query), 929-931
XSD (XML Schema Definition), LINQ (Language Integrated Query), 929-931
XSLT (EXtensible Stylesheet Language), 230

Z

zero-initialization, 591
zero-method interfaces, 693-694
1, 1010-1012

How can we make this index more useful? Email us at indexes@samspublishing.com