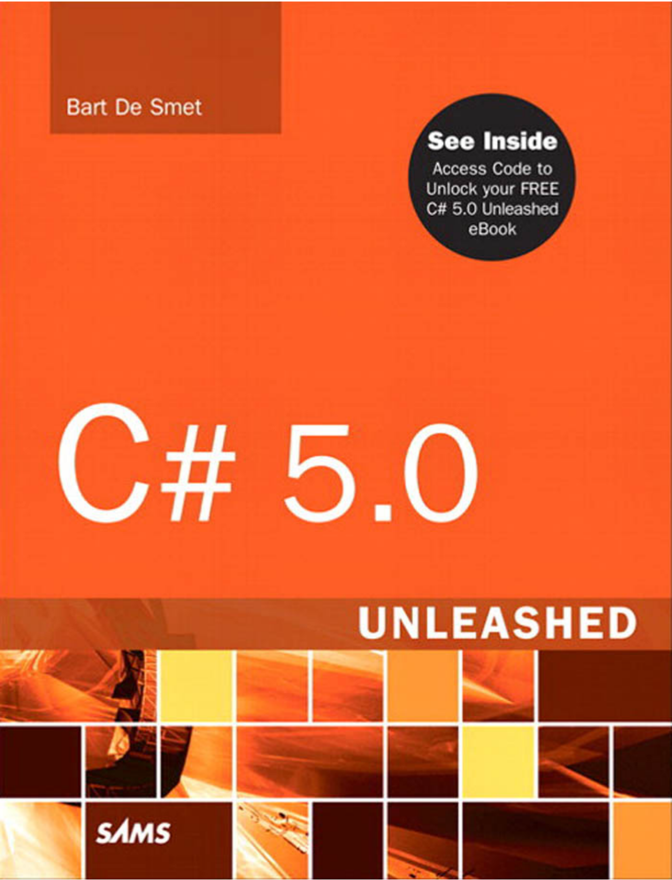


Bart De Smet

See Inside
Access Code to
Unlock your FREE
C# 5.0 Unleashed
eBook

C# 5.0

UNLEASHED

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Bart De Smet

C# 5.0

UNLEASHED

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

C# 5.0 Unleashed

Copyright © 2013 by Pearson Education, Inc.

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

ISBN-13: 978-0-672-33690-4

ISBN-10: 0-672-33690-1

Library of Congress Cataloging-in-Publication Data is on file.

Printed in the United States of America

First Printing: April 2013

Trademarks

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

Warning and Disclaimer

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

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Editor-in-Chief
Greg Wiegand

Acquisitions Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Keith Cline

Indexer
Brad Herriman

Proofreader
Debbie Williams

Technical Editor
Christopher Wilcox

Editorial Assistant
Cindy Teeters

Cover Designer
Anne Jones

Compositor
Nonie Ratcliff

Contents at a Glance

Table of Contents

5 Expressions and Operators 251

6 A Primer on Types and Objects 301

7 Simple Control Flow 351

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 Smaele, 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.

This page intentionally left blank

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 aftertaste. 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#

IN THIS CHAPTER

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

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

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.

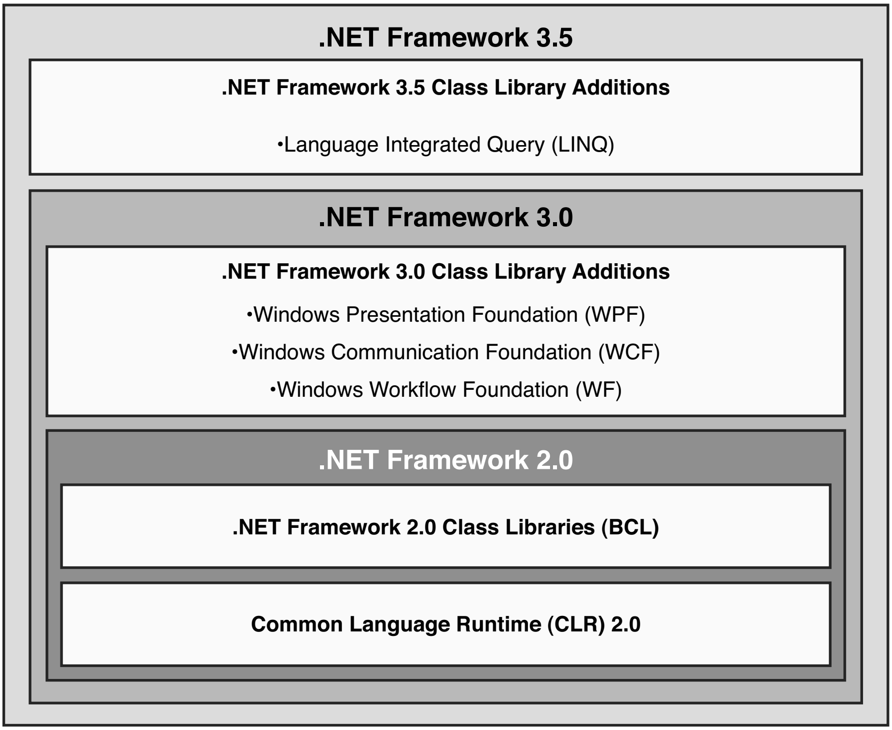


FIGURE 3.1 .NET Framework 3.x is built on CLR 2.0.

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

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 ymmdd 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.

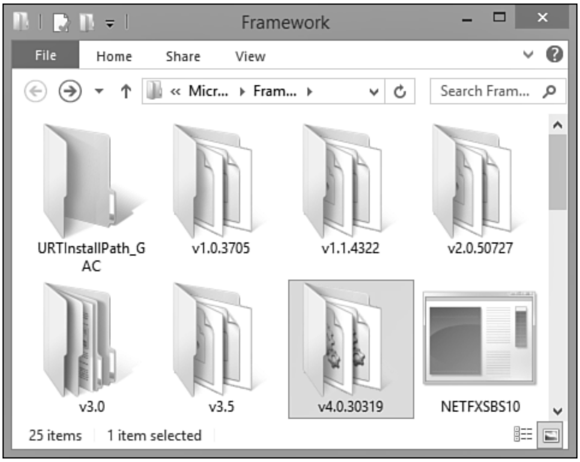


FIGURE 3.2 Side-by-side installation of .NET Framework versions.

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.

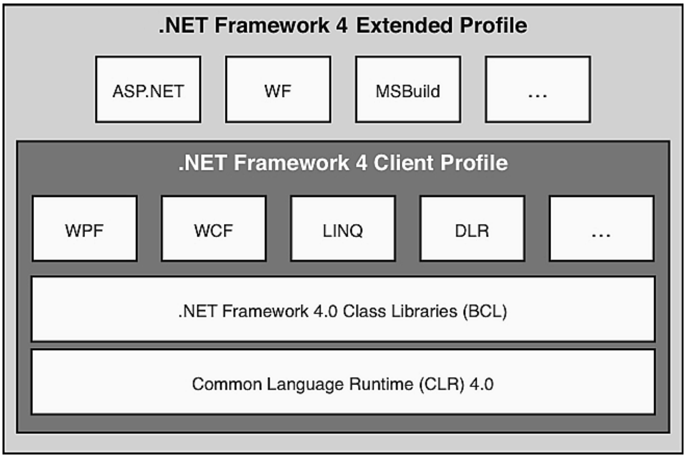


FIGURE 3.3 Client Profile subset of the .NET Framework.

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.

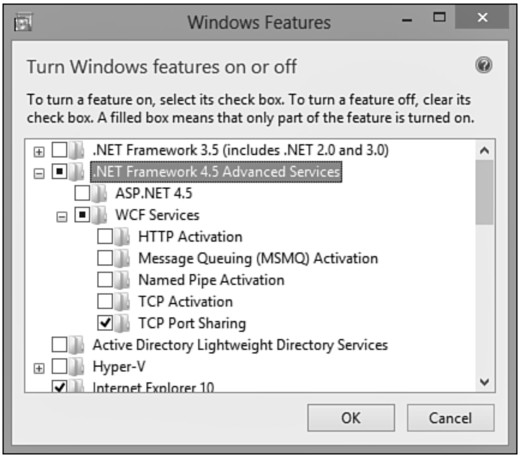
.NET 4.5 introduces a new notion of profiles though, through the new Portable Library support. When writing portable class libraries, you can select the target platforms the library can run on. For example, you could build a portable library targeting .NET Framework 4.5, .NET for Windows Store applications, 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.



Windows Features

Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- ☐ .NET Framework 3.5 (includes .NET 2.0 and 3.0)
- ■ .NET Framework 4.5 Advanced Services
  - ☐ ASP.NET 4.5
  - ■ WCF Services
    - ☐ HTTP Activation
    - ☐ Message Queuing (MSMQ) Activation
    - ☐ Named Pipe Activation
    - ☐ TCP Activation
    - ☑ TCP Port Sharing
- ☐ Active Directory Lightweight Directory Services
- ☐ Hyper-V
- ☑ Internet Explorer 10

FIGURE 3.4 .NET Framework 4.5 enabled by default on Windows 8.

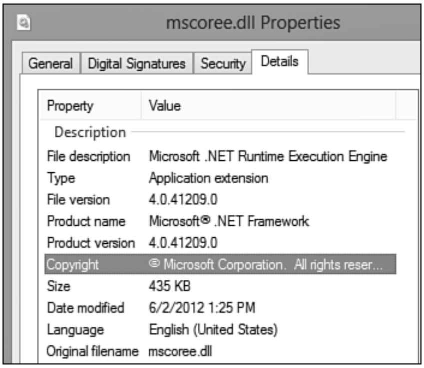
What Got Installed?

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.



Property	Value
Description	
File description	Microsoft .NET Runtime Execution Engine
Type	Application extension
File version	4.0.41209.0
Product name	Microsoft® .NET Framework
Product version	4.0.41209.0
Copyright	© Microsoft Corporation. All rights reser...
Size	435 KB
Date modified	6/2/2012 1:25 PM
Language	English (United States)
Original filename	mscoree.dll

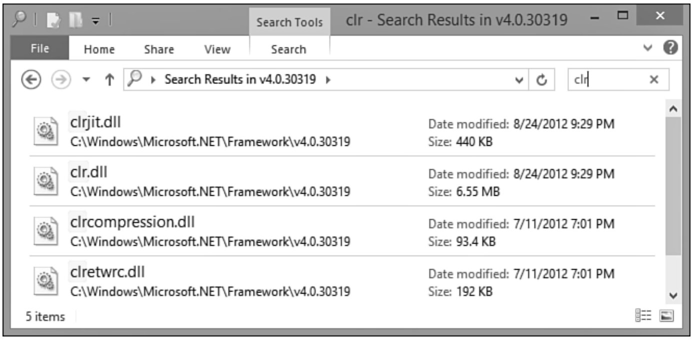
FIGURE 3.5 The version of the CLR runtime shim.

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.



clrjit.dll
C:\Windows\Microsoft.NET\Framework\v4.0.30319
Date modified: 8/24/2012 9:29 PM
Size: 440 KB

clr.dll
C:\Windows\Microsoft.NET\Framework\v4.0.30319
Date modified: 8/24/2012 9:29 PM
Size: 6.55 MB

clrcompression.dll
C:\Windows\Microsoft.NET\Framework\v4.0.30319
Date modified: 7/11/2012 7:01 PM
Size: 93.4 KB

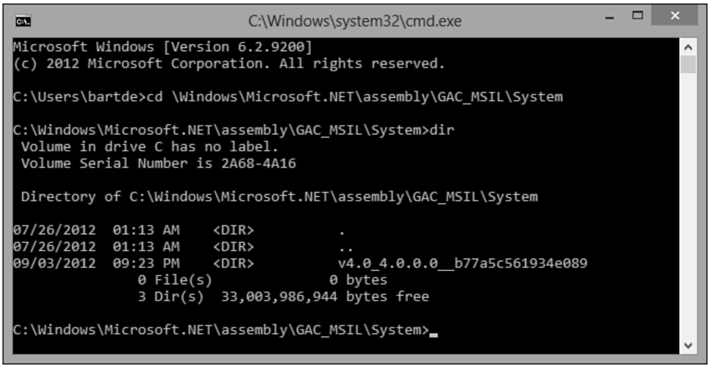
clretwrc.dll
C:\Windows\Microsoft.NET\Framework\v4.0.30319
Date modified: 7/11/2012 7:01 PM
Size: 192 KB

FIGURE 3.6 The CLR itself.

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.



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\bartde>cd \Windows\Microsoft.NET\assembly\GAC_MSIL\System

C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System>dir
 Volume in drive C has no label.
 Volume Serial Number is 2A68-4A16

 Directory of C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System

07/26/2012  01:13 AM    <DIR>          .
07/26/2012  01:13 AM    <DIR>          ..
09/03/2012  09:23 PM    <DIR>          v4.0_4.0.0.0__b77a5c561934e089
               0 File(s)              0 bytes
               3 Dir(s)  33,003,986,944 bytes free

C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System>_
```

FIGURE 3.7 Inside the GAC.

NOTE: GAC SPLITTING

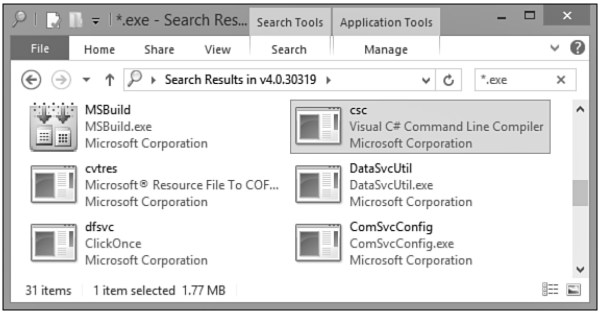
Notice the v4.0 prefix in the name of the folder containing the .NET 4.0 version of System.dll? This is an artifact of the “GAC splitting” carried out in .NET 4.0. This simple naming trick hides assemblies targeting different versions of the runtime so that a specific version of the CLR doesn’t try to 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.



MSBuild
MSBuild.exe
Microsoft Corporation

csc
Visual C# Command Line Compiler
Microsoft Corporation

cvtres
Microsoft® Resource File To COF...
Microsoft Corporation

DataSvcUtil
DataSvcUtil.exe
Microsoft Corporation

dfsvc
ClickOnce
Microsoft Corporation

ComSvcConfig
ComSvcConfig.exe
Microsoft Corporation

31 items 1 item selected 1.77 MB

FIGURE 3.8 One of the .NET Framework tools: the C# compiler.

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:

```
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.

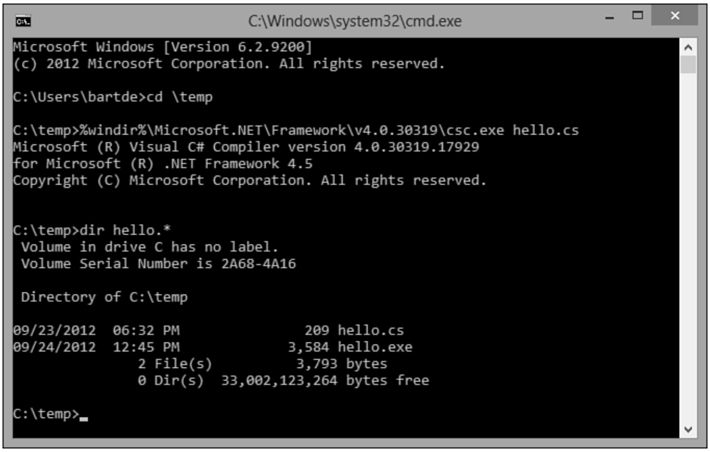
Finally, inside the `Main` method we've written a couple of statements. The first one makes a method call to the `Write` method on the `Console` type, printing the string `Enter your name:` to the screen. In the second line, we read the user's name from the console input and assign it to a local variable 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.



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\bartde>cd \temp

C:\temp>%windir%\Microsoft.NET\Framework\v4.0.30319\csc.exe hello.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.


C:\temp>dir hello.*
 Volume in drive C has no label.
 Volume Serial Number is 2A68-4A16

 Directory of C:\temp

09/23/2012  06:32 PM               209 hello.cs
09/24/2012  12:45 PM             3,584 hello.exe
               2 File(s)          3,793 bytes
               0 Dir(s)  33,002,123,264 bytes free

C:\temp>
```

FIGURE 3.9 Running the C# compiler.

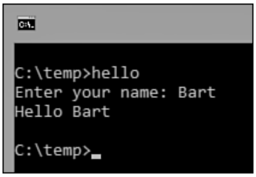
If the user has installed Visual Studio 2012, a more convenient way to invoke the compiler is from the Visual Studio 2012 command prompt. This specialized command prompt has search paths configured properly such that tools like csc.exe will be found.

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.



```
C:\temp>hello
Enter your name: Bart
Hello Bart

C:\temp>_
```

FIGURE 3.10 Our program in action.

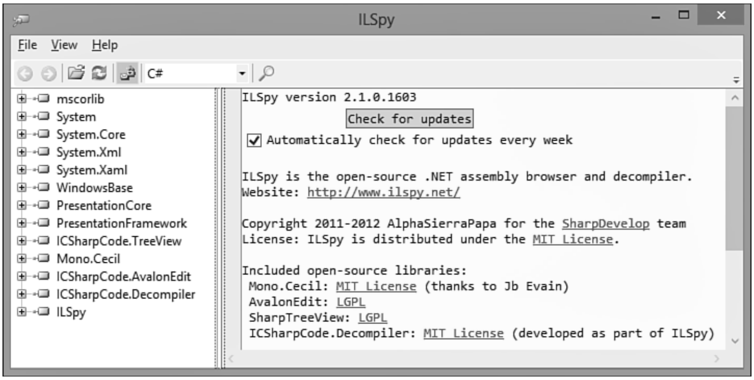
That wasn't too hard, was it? To satisfy our technical curiosity, let's take a look at the produced assembly.

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.

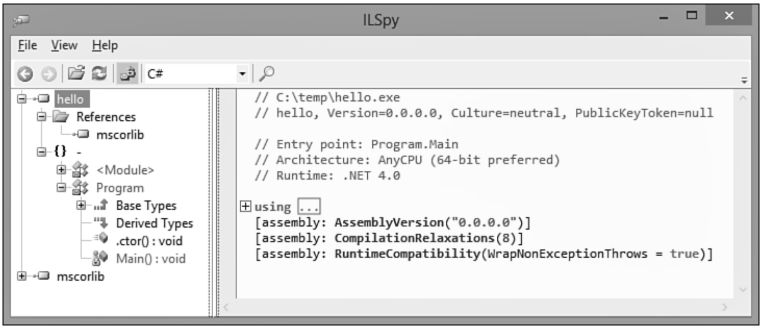
When you run the tool for the first time, it loads the ILSpy assembly as the assembly to inspect, including all of its dependencies, as shown in Figure 3.11.



```
ILSpy version 2.1.0.1603
            Check for updates
 ☑ Automatically check for updates every week

ILSpy is the open-source .NET assembly browser and decompiler.
Website: http://www.ilspy.net/

Copyright 2011-2012 AlphaSierraPapa for the SharpDevelop team
License: ILSpy is distributed under the MIT License.

Included open-source libraries:
 Mono.Cecil: MIT License (thanks to Jb Evain)
 AvalonEdit: LGPL
 SharpTreeView: LGPL
 ICSharpCode.Decompiler: MIT License (developed as part of ILSpy)
```

FIGURE 3.11 ILSpy disassembling itself.

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.



```
// C:\temp\hello.exe
// hello, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

// Entry point: Program.Main
// Architecture: AnyCPU (64-bit preferred)
// Runtime: .NET 4.0

using ...
[assembly: AssemblyVersion("0.0.0.0")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
```

FIGURE 3.12 Inspecting the assembly structure in ILSpy.

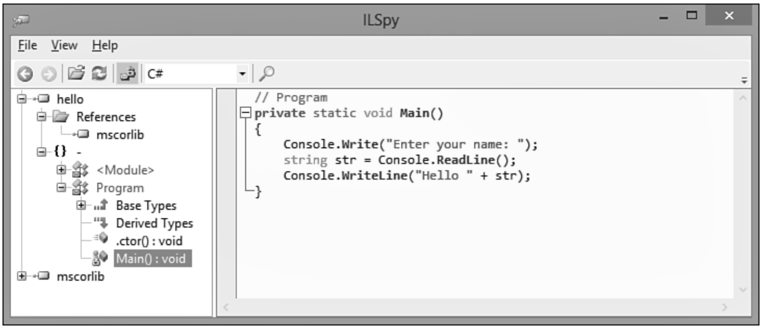
Looking at this structure gives us a good opportunity to explain a few concepts briefly. As we drill down in the tree view, we start from the "hello" *assembly* we just compiled. Assemblies are just CLR concepts by themselves and don't have direct affinity to file-based storage. Indeed, it's possible 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.

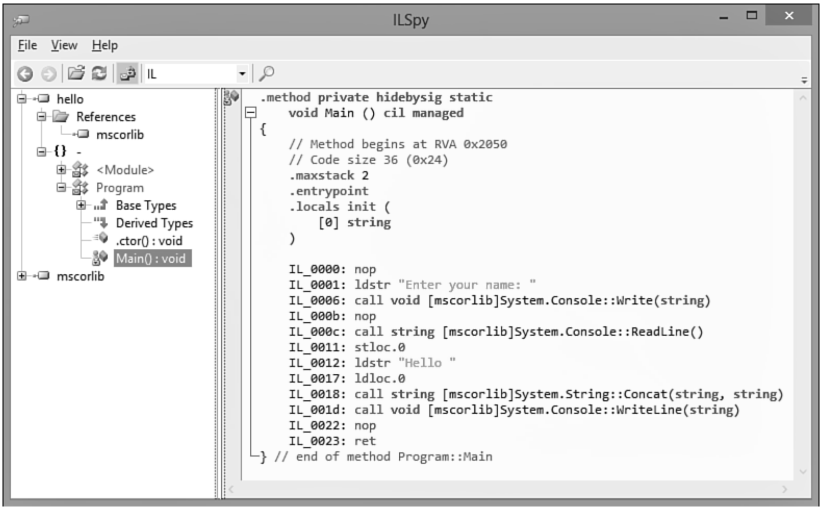
The pane on the right shows the decompiled code back in C#. It's important to realize this didn't use the hello.cs source code file at all. The hello.exe assembly doesn't have any link back to the source files from which it was compiled. "All" ILSpy does is reconstruct the C# code from the IL code inside 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.



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

FIGURE 3.13 Disassembling the `Main` method.



```
.method private hidebysig static
    void Main () cil managed
{
    // Method begins at RVA 0x2050
    // Code size 36 (0x24)
    .maxstack 2
    .entrypoint
    .locals init (
        [0] string
    )

    IL_0000: nop
    IL_0001: ldstr "Enter your name: "
    IL_0006: call void [mscorlib]System.Console::Write(string)
    IL_000b: nop
    IL_000c: call string [mscorlib]System.Console::ReadLine()
    IL_0011: stloc.0
    IL_0012: ldstr "Hello "
    IL_0017: ldloc.0
    IL_0018: call string [mscorlib]System.String::Concat(string, string)
    IL_001d: call void [mscorlib]System.Console::WriteLine(string)
    IL_0022: nop
    IL_0023: ret
} // end of method Program::Main
```

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.

Optional features to install:

- ☑ Blend for Visual Studio
- ☑ LightSwitch
- ☑ Microsoft Foundation Classes for C++
- ☑ Microsoft Office Developer Tools
- ☑ Microsoft SharePoint Developer Tools
- ☑ Microsoft SQL Server Data Tools
- ☑ Silverlight Development Kits
- ☑ Microsoft Web Developer Tools

☑ Select All

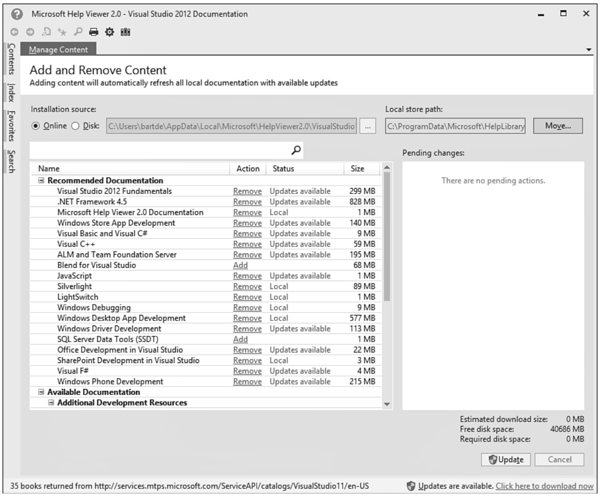
Reset Defaults

Setup requires 337 MB on C:

FIGURE 3.16 Visual Studio 2012 Professional installation options.

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.



Microsoft Help Viewer 2.0 - Visual Studio 2012 Documentation

Add and Remove Content

Adding content will automatically refresh all local documentation with available updates

Installation source: ● Online ○ Disk: C:\Users\bartde\AppData\Local\Microsoft\HelpViewer2.0\VisualStudio

Local store path: C:\ProgramData\Microsoft\HelpLibrary

Name	Action	Status	Size
⊟ **Recommended Documentation**			
Visual Studio 2012 Fundamentals	Remove	Updates available	299 MB
.NET Framework 4.5	Remove	Updates available	828 MB
Microsoft Help Viewer 2.0 Documentation	Remove	Local	1 MB
Windows Store App Development	Remove	Updates available	140 MB
Visual Basic and Visual C#	Remove	Updates available	9 MB
Visual C++	Remove	Updates available	59 MB
ALM and Team Foundation Server	Remove	Updates available	195 MB
Blend for Visual Studio	Add		68 MB
JavaScript	Remove	Updates available	1 MB
Silverlight	Remove	Local	89 MB
LightSwitch	Remove	Local	1 MB
Windows Debugging	Remove	Local	9 MB
Windows Desktop App Development	Remove	Local	577 MB
Windows Driver Development	Remove	Updates available	113 MB
SQL Server Data Tools (SSDT)	Add		1 MB
Office Development in Visual Studio	Remove	Updates available	22 MB
SharePoint Development in Visual Studio	Remove	Local	3 MB
Visual F#	Remove	Updates available	4 MB
Windows Phone Development	Remove	Updates available	215 MB
⊟ **Available Documentation**			
⊟ **Additional Development Resources**			

Pending changes:

There are no pending actions.

Estimated download size:	0 MB
Free disk space:	40686 MB
Required disk space:	0 MB

35 books returned from http://services.mtps.microsoft.com/ServiceAPI/catalogs/VisualStudio11/en-US

Updates are available. Click here to download now

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.

FIGURE 3.18 Visual Studio 2012 Start screen entries.

TIP

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*.

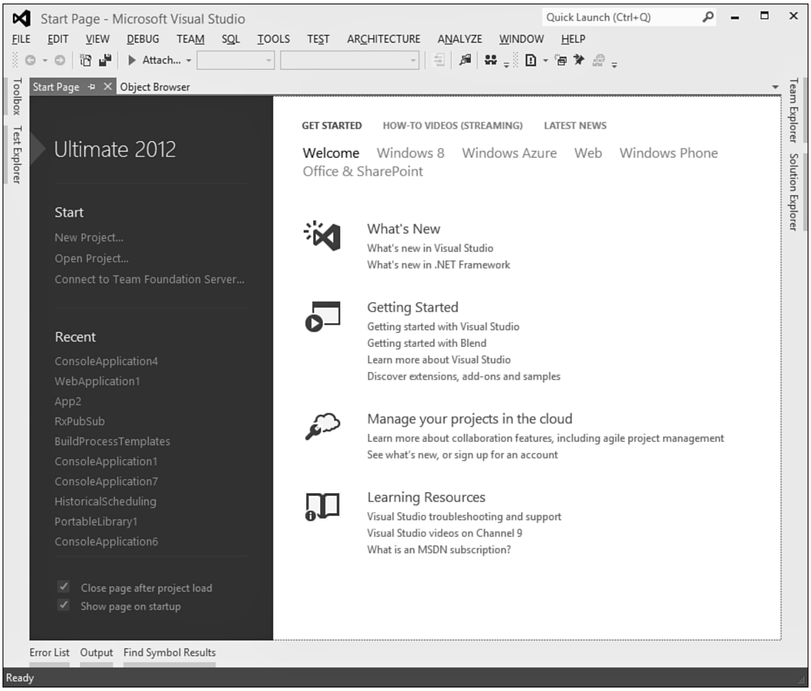
The first thing you'll see in Visual Studio is the Start page shown in Figure 3.20. It provides links to various tasks (for example, to reload recently used projects). An RSS feed shows news items from the MSDN website.

This program is protected by U.S. and international copyright laws as described in Help/About.

© 2012 Microsoft Corporation.
All rights reserved.

This product is licensed to:
Bart

FIGURE 3.19 Visual Studio 2012 splash screen.



Ultimate 2012

Start
New Project...
Open Project...
Connect to Team Foundation Server...

Recent
ConsoleApplication4
WebApplication1
App2
RxPubSub
BuildProcessTemplates
ConsoleApplication1
ConsoleApplication7
HistoricalScheduling
PortableLibrary1
ConsoleApplication6

☑ Close page after project load
☑ Show page on startup

GET STARTED HOW-TO VIDEOS (STREAMING) LATEST NEWS

Welcome Windows 8 Windows Azure Web Windows Phone Office & SharePoint

What's New
What's new in Visual Studio
What's new in .NET Framework

Getting Started
Getting started with Visual Studio
Getting started with Blend
Learn more about Visual Studio
Discover extensions, add-ons and samples

Manage your projects in the cloud
Learn more about collaboration features, including agile project management
See what's new, or sign up for an account

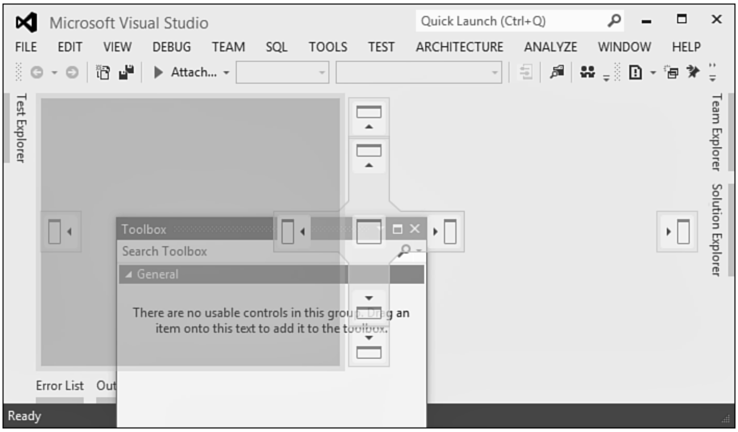
Learning Resources
Visual Studio troubleshooting and support
Visual Studio videos on Channel 9
What is an MSDN subscription?

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.



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

FIGURE 3.21 Customizing the look and feel by docking panels.

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.

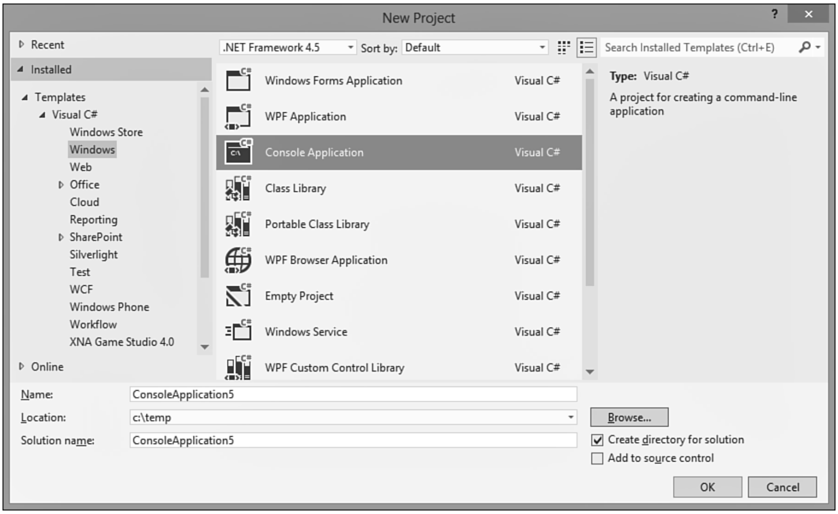
Because I've selected Visual C# as my preferred language at the first start of Visual Studio 2012, the C# templates are shown immediately. (For other languages, scroll down to the Other Languages entry on the left.) Subcategories are used to organize the various templates. Under the Windows category,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



.NET Framework 4.5

Sort by: Default

Type: Visual C#

A project for creating a command-line application

Name: ConsoleApplication5

Location: c:\temp

Solution name: ConsoleApplication5

☑ Create directory for solution

☐ Add to source control

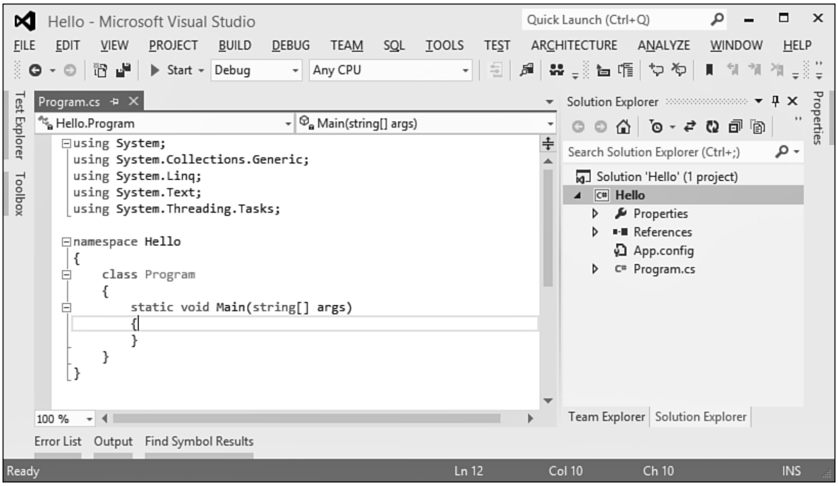
FIGURE 3.22 The New Project dialog.

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.



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

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

FIGURE 3.23 A new Console Application project.

Solution Explorer

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: VWORRIED 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.

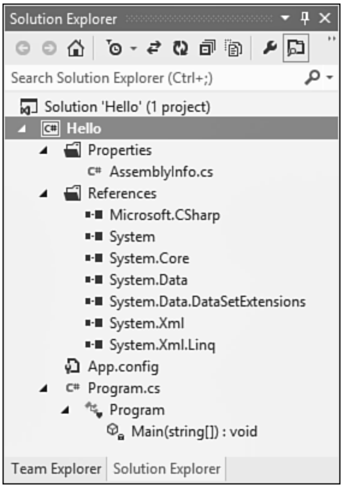


FIGURE 3.24 Solution Explorer.

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.

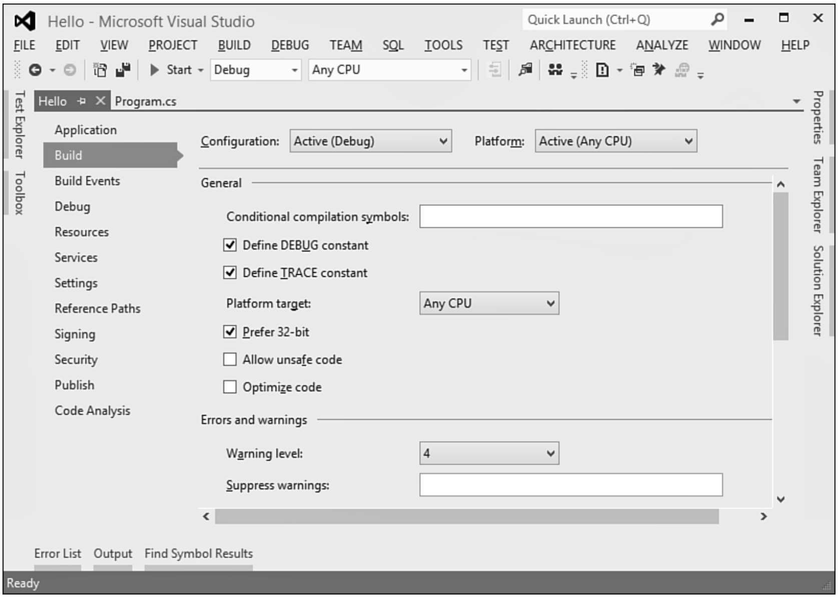
NOTE: SOURCE CONTROL

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/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.



Configuration: Active (Debug)
Platform: Active (Any CPU)

General

Conditional compilation symbols:
☑ Define DEBUG constant
☑ Define TRACE constant
Platform target: Any CPU
☑ Prefer 32-bit
☐ Allow unsafe code
☐ Optimize code

Errors and warnings

Warning level: 4
Suppress warnings:

FIGURE 3.25 Project properties.

Code Editor

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:

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

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

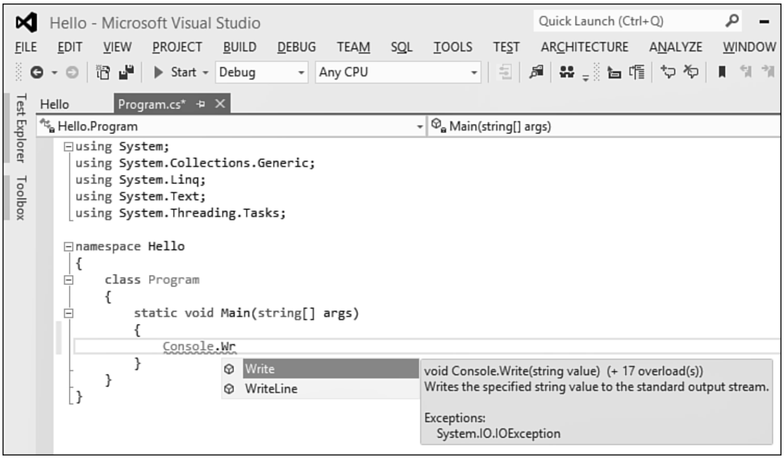
There are a few differences with the code we started from when writing our little console application manually.

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:

```
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.



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

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Wr
        }
    }
}
```

void Console.Write(string value) (+ 17 overload(s))
Writes the specified string value to the standard output stream.

Exceptions:
System.IO.IOException

FIGURE 3.26 IntelliSense while typing code.

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.

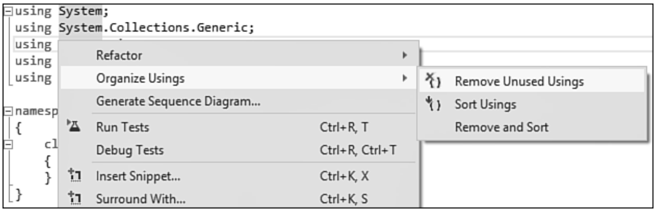
IntelliSense will help you with the next two lines of code in a similar way as it did for the first. As you type, notice different tokens get colorized differently. Built-in language keywords are marked with blue, type names (like `Console`) have a color that I don’t know the name of but that looks kind 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.

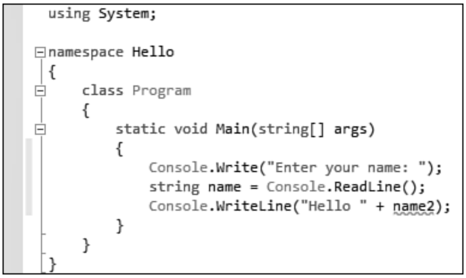


```
using System;
using System.Collections.Generic;
using
using
using

namesp
{
    cl
    {
    }
}
```

FIGURE 3.27 Reducing the clutter of excessive imported namespaces.

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.



```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter your name: ");
            string name = Console.ReadLine();
            Console.WriteLine("Hello " + name2);
        }
    }
}
```

FIGURE 3.28 The background compiler detecting a typo.

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.

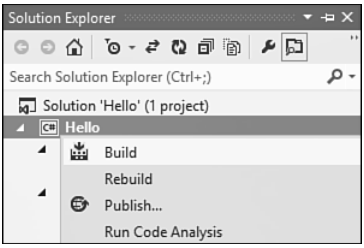


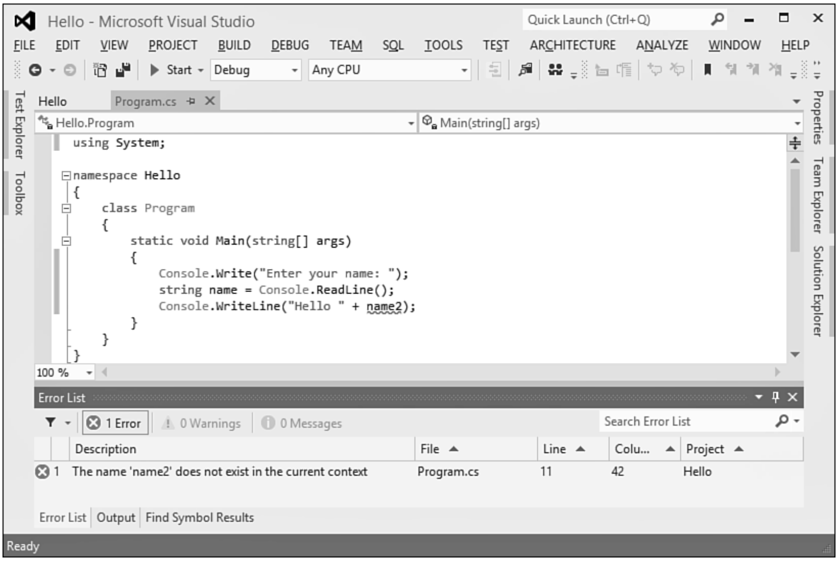
FIGURE 3.29 Starting a build from the project node context menu.

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.



```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter your name: ");
            string name = Console.ReadLine();
            Console.WriteLine("Hello " + name2);
        }
    }
}
```

	Description	File	Line	Colu...	Project
1	The name 'name2' does not exist in the current context	Program.cs	11	42	Hello

FIGURE 3.30 The Error List pane showing a build error.

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.

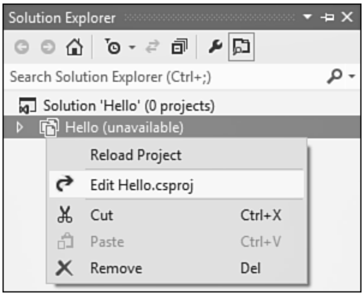
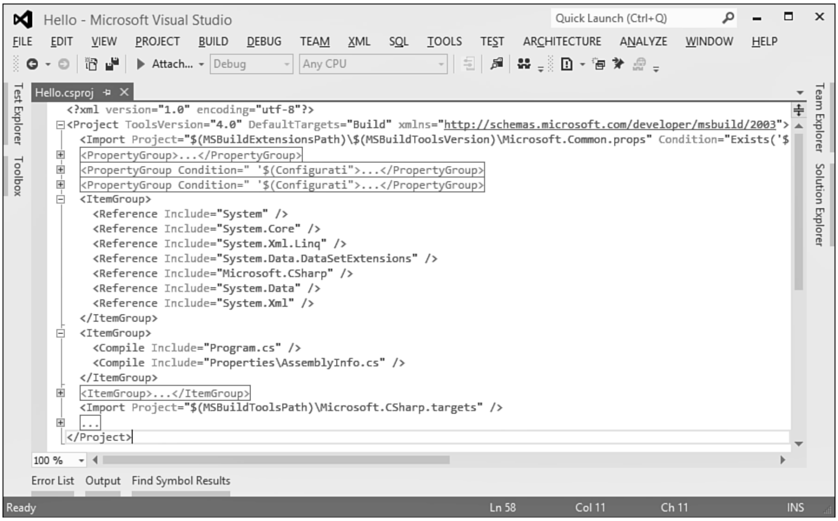


FIGURE 3.31 Showing the project definition file.



```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" Condition="Exists('$
  <PropertyGroup>...</PropertyGroup>
  <PropertyGroup Condition=" '$(Configurati">...</PropertyGroup>
  <PropertyGroup Condition=" '$(Configurati">...</PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="System.Xml.Linq" />
    <Reference Include="System.Data.DataSetExtensions" />
    <Reference Include="Microsoft.CSharp" />
    <Reference Include="System.Data" />
    <Reference Include="System.Xml" />
  </ItemGroup>
  <ItemGroup>
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
  </ItemGroup>
  <ItemGroup>...</ItemGroup>
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
  ...
</Project>
```

FIGURE 3.32 Project file in the XML editor.

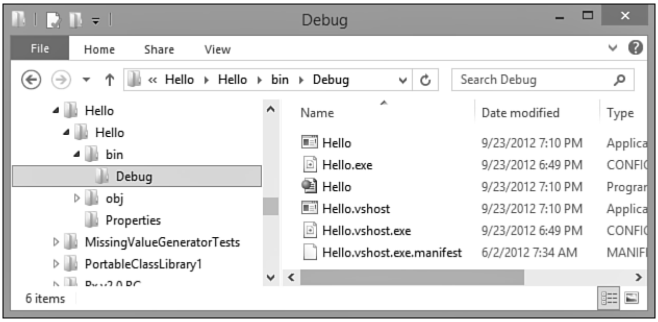
From this, we can see that MSBuild projects are XML files that describe the structure of the project being built: what the source files are, required dependencies, and so forth. Visual Studio uses MSBuild files to store a project's structure and to drive its build. Notable entries in this file include 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.

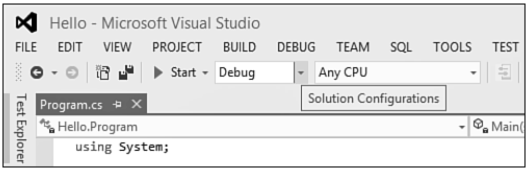
Now that you know how to inspect the MSBuild project file, go ahead and choose Reload Project from the project's node context menu in Solution Explorer. Assuming a successful build (correct the silly typo illustrated before), where can the resulting binaries be found? Have a look at the project's folder, 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.



Name	Date modified	Type
Hello	9/23/2012 7:10 PM	Applica
Hello.exe	9/23/2012 6:49 PM	CONFI
Hello	9/23/2012 7:10 PM	Progra
Hello.vshost	9/23/2012 7:10 PM	Applica
Hello.vshost.exe	9/23/2012 6:49 PM	CONFI
Hello.vshost.exe.manifest	6/2/2012 7:34 AM	MANIF

FIGURE 3.33 Build output folder.

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.



```
using System;
```

FIGURE 3.34 Changing the solution configuration.

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.

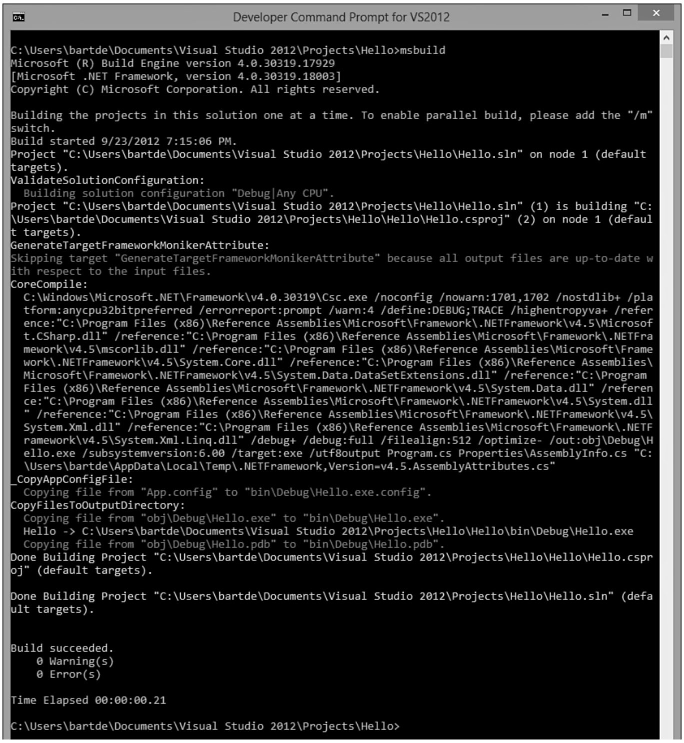
To see a command-line build in action, open a Developer Command Prompt for VS2012 from the Start menu, change the directory to the location of the Hello.csproj file, and invoke msbuild.exe (see Figure 3.35). The fact there's only one recognized project file extension causes MSBuild to invoke the build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



```
C:\Users\bartde\Documents\Visual Studio 2012\Projects\Hello>msbuild
Microsoft (R) Build Engine version 4.0.30319.17929
[Microsoft .NET Framework, version 4.0.30319.18003]
Copyright (C) Microsoft Corporation. All rights reserved.

Building the projects in this solution one at a time. To enable parallel build, please add the "/m"
switch.
Build started 9/23/2012 7:15:06 PM.
Project "C:\Users\bartde\Documents\Visual Studio 2012\Projects\Hello\Hello.sln" on node 1 (default
targets).
ValidateSolutionConfiguration:
  Building solution configuration "Debug|Any CPU".
Project "C:\Users\bartde\Documents\Visual Studio 2012\Projects\Hello\Hello.sln" (1) is building "C:
\Users\bartde\Documents\Visual Studio 2012\Projects\Hello\Hello\Hello.csproj" (2) on node 1 (defaul
t targets).
GenerateTargetFrameworkMonikerAttribute:
Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date w
ith respect to the input files.
CoreCompile:
  C:\Windows\Microsoft.NET\Framework\v4.0.30319\Csc.exe /noconfig /nowarn:1701,1702 /nostdlib+ /pla
  tform:anycpu32bitpreferred /errorreport:prompt /warn:4 /define:DEBUG;TRACE /highentropyva+ /refer
  ence:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\Microsof
  t.CSharp.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFra
  mework\v4.5\mscorlib.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Frame
  work\.NETFramework\v4.5\System.Core.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\
  Microsoft\Framework\.NETFramework\v4.5\System.Data.DataSetExtensions.dll" /reference:"C:\Program
  Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.Data.dll" /referen
  ce:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.dll
  " /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\
  System.Xml.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETF
  ramework\v4.5\System.Xml.Linq.dll" /debug+ /debug:full /filealign:512 /optimize- /out:obj\Debug\H
  ello.exe /subsystemversion:6.00 /target:exe /utf8output Program.cs Properties\AssemblyInfo.cs "C:
  \Users\bartde\AppData\Local\Temp\.NETFramework,Version=v4.5.AssemblyAttributes.cs"
_CopyAppConfigFile:
  Copying file from "App.config" to "bin\Debug\Hello.exe.config".
CopyFilesToOutputDirectory:
  Copying file from "obj\Debug\Hello.exe" to "bin\Debug\Hello.exe".
  Hello -> C:\Users\bartde\Documents\Visual Studio 2012\Projects\Hello\Hello\bin\Debug\Hello.exe
  Copying file from "obj\Debug\Hello.pdb" to "bin\Debug\Hello.pdb".
Done Building Project "C:\Users\bartde\Documents\Visual Studio 2012\Projects\Hello\Hello\Hello.cspr
oj" (default targets).

Done Building Project "C:\Users\bartde\Documents\Visual Studio 2012\Projects\Hello\Hello.sln" (defa
ult targets).


Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.21

C:\Users\bartde\Documents\Visual Studio 2012\Projects\Hello>
```

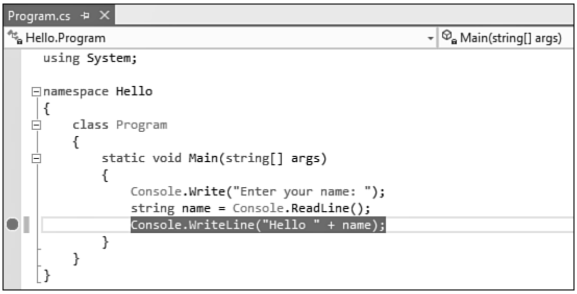
FIGURE 3.35 MSBuild invoked from the command line.

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.

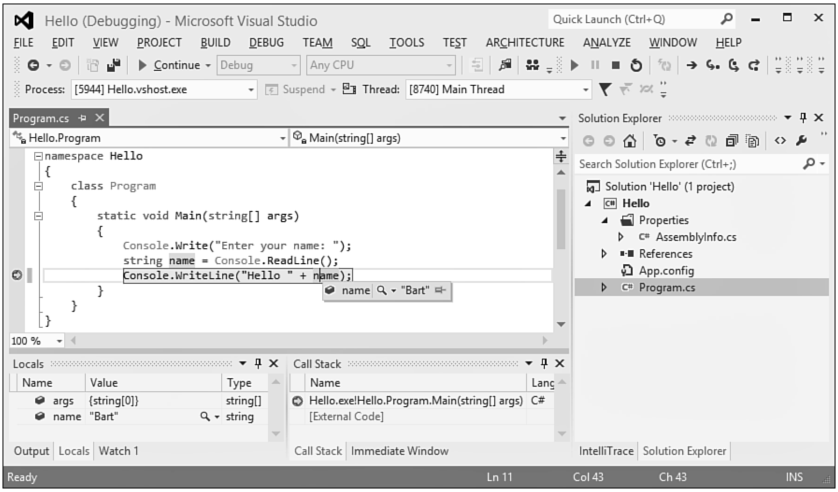
The most commonly used debugging technique is to run the code with breakpoints set at various places of interest, as shown in Figure 3.36. Doing so right inside a source code file is easy by putting the cursor on the line of interest and pressing F9. Alternative approaches include clicking in the gray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



```
using System;

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

FIGURE 3.36 Code editor with a breakpoint set.



Process: [5944] Hello.vshost.exe

Thread: [8740] Main Thread

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

name "Bart"

Locals

Name	Value	Type
args	{string[0]}	string[]
name	"Bart"	string

Call Stack

Name	Lang
Hello.exe!Hello.Program.Main(string[] args)	C#
[External Code]	

FIGURE 3.37 Hitting a breakpoint in the debugger.

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.

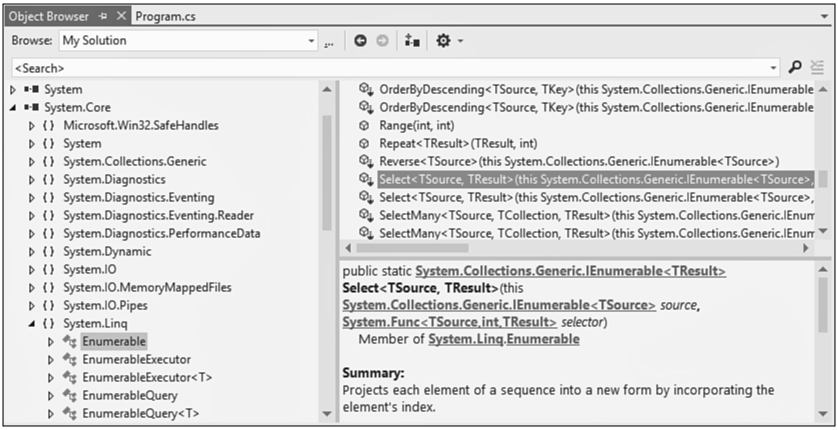
Although the original target audience for CDB and WinDbg consists of Win32 native code developers and driver writers, an extension for managed code ships right with the .NET Framework. This debugger extension is known as SOS, which stands for Son of Strike, with Strike being an old code name for the 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).



Browse: My Solution

public static System.Collections.Generic.IEnumerable<TResult>
Select<TSource, TResult>(this
System.Collections.Generic.IEnumerable<TSource> *source*,
System.Func<TSource,int,TResult> *selector*)
Member of System.Linq.Enumerable

Summary:
Projects each element of a sequence into a new form by incorporating the element's index.

FIGURE 3.38 Object Browser visualizing the `System.Core` assembly.

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.

You can find more information on the Microsoft Shared Source Initiative for the .NET Framework site at http://referencesource.microsoft.com/netframework.aspx.

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.

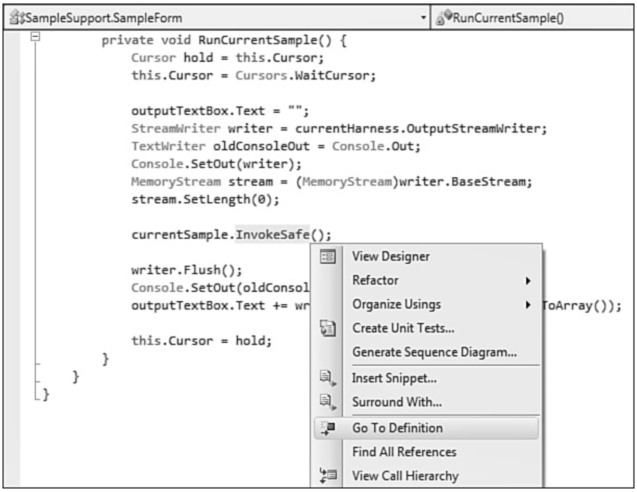
NOTE

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.

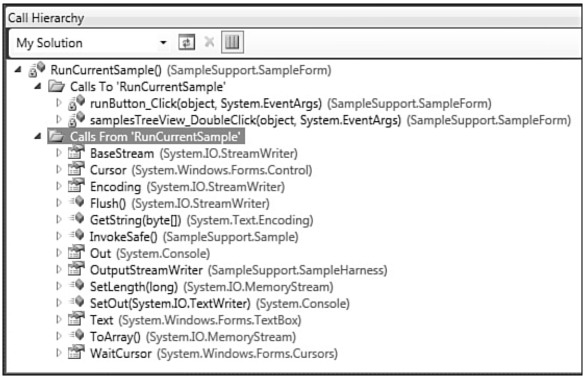
Find All References is similar in nature but performs the opposite operation: Instead of finding the definition site for the selection, it looks for all use sites of it. For example, when considering changing the implementation of some method, you better find out who's using it and what the impact of any change might be.



```
        private void RunCurrentSample() {
            Cursor hold = this.Cursor;
            this.Cursor = Cursors.WaitCursor;

            outputTextBox.Text = "";
            StreamWriter writer = currentHarness.OutputStreamWriter;
            TextWriter oldConsoleOut = Console.Out;
            Console.SetOut(writer);
            MemoryStream stream = (MemoryStream)writer.BaseStream;
            stream.SetLength(0);

            currentSample.InvokeSafe();

            writer.Flush();
            Console.SetOut(oldConsol
            outputTextBox.Text += wr                                     ToArray());

            this.Cursor = hold;
        }
    }
}
```

FIGURE 3.39 Code navigation options.

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.



```
RunCurrentSample() (SampleSupport.SampleForm)
  Calls To 'RunCurrentSample'
    runButton_Click(object, System.EventArgs) (SampleSupport.SampleForm)
    samplesTreeView_DoubleClick(object, System.EventArgs) (SampleSupport.SampleForm)
  Calls From 'RunCurrentSample'
    BaseStream (System.IO.StreamWriter)
    Cursor (System.Windows.Forms.Control)
    Encoding (System.IO.StreamWriter)
    Flush() (System.IO.StreamWriter)
    GetString(byte[]) (System.Text.Encoding)
    InvokeSafe() (SampleSupport.Sample)
    Out (System.Console)
    OutputStreamWriter (SampleSupport.SampleHarness)
    SetLength(long) (System.IO.MemoryStream)
    SetOut(System.IO.TextWriter) (System.Console)
    Text (System.Windows.Forms.TextBox)
    ToArray() (System.IO.MemoryStream)
    WaitCursor (System.Windows.Forms.Cursors)
```

FIGURE 3.40 Call Hierarchy analyzing some code.

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.

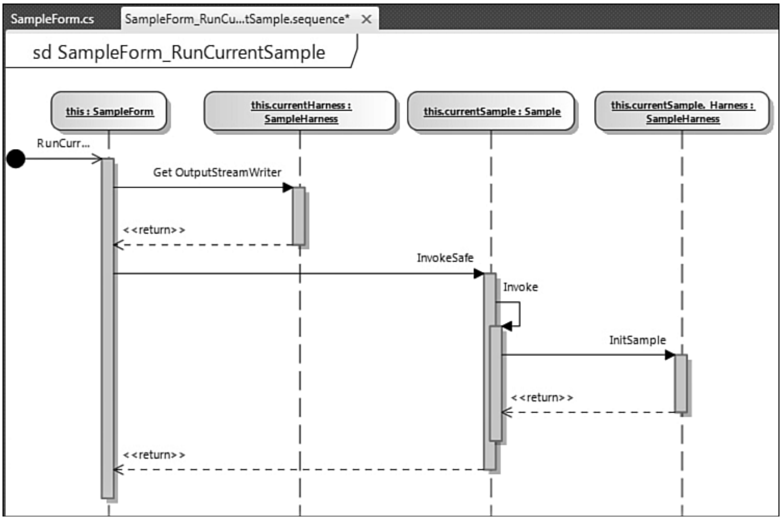
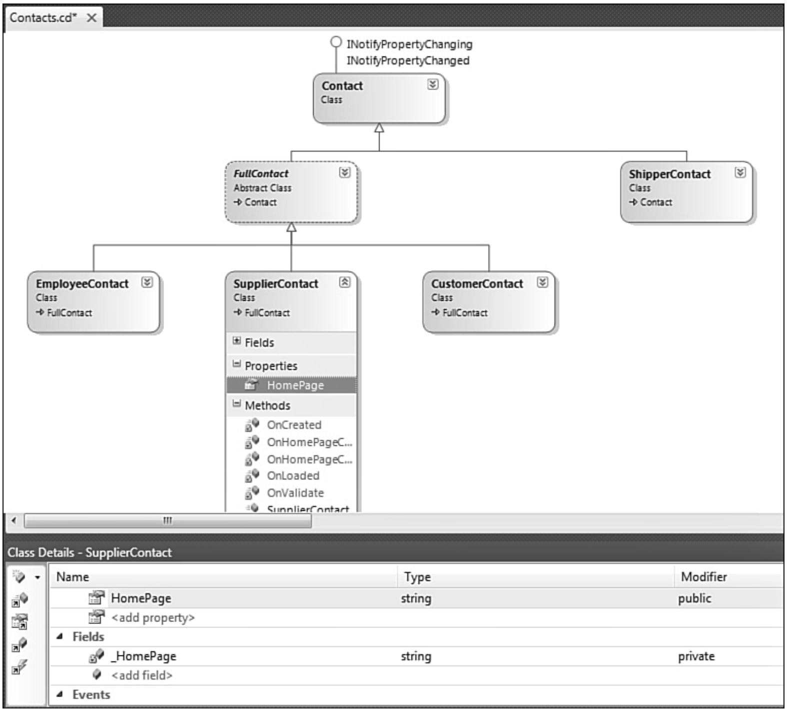


FIGURE 3.41 A simple sequence diagram.

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.



Class Details - SupplierContact

Name	Type	Modifier
HomePage	string	public
<add property>		
Fields		
_HomePage	string	private
<add field>		
Events		

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.

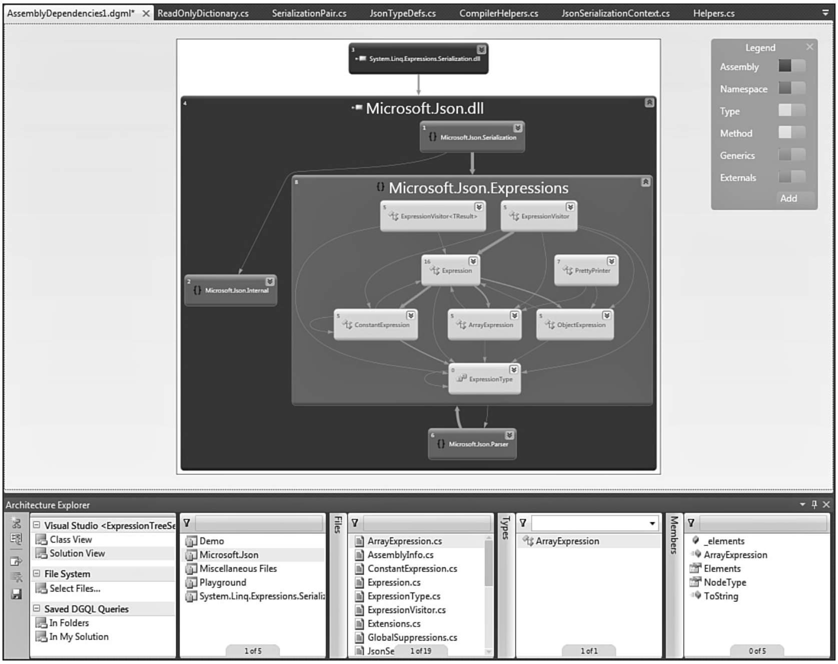


FIGURE 3.43 Graph view for the solution, project, a code file item, and some types.

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 application. 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

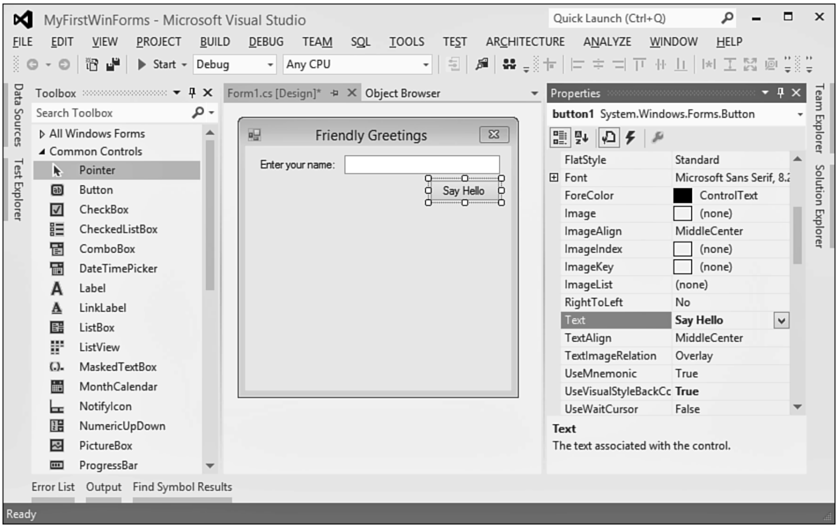
Since the introduction of Visual Basic 1.0 (as early as 1991), Rapid Application Development (RAD) has been a core theme of the Microsoft tools for developers. Rich designers for UI development are huge time savers over a coding approach to accomplish the same task. This was true in the world of pure 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.

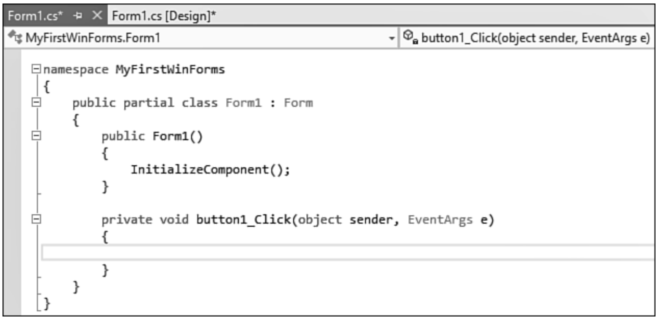


button1 System.Windows.Forms.Button

FlatStyle	Standard
Font	Microsoft Sans Serif, 8.2
ForeColor	ControlText
Image	(none)
ImageAlign	MiddleCenter
ImageIndex	(none)
ImageKey	(none)
ImageList	(none)
RightToLeft	No
Text	**Say Hello**
TextAlign	MiddleCenter
TextImageRelation	Overlay
UseMnemonic	True
UseVisualStyleBackCc	**True**
UseWaitCursor	False

Text
The text associated with the control.

FIGURE 3.44 The Windows Forms designer.



```
namespace MyFirstWinForms
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {

        }
    }
}
```

FIGURE 3.45 An empty event handler ready for implementation.

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:

```
#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.

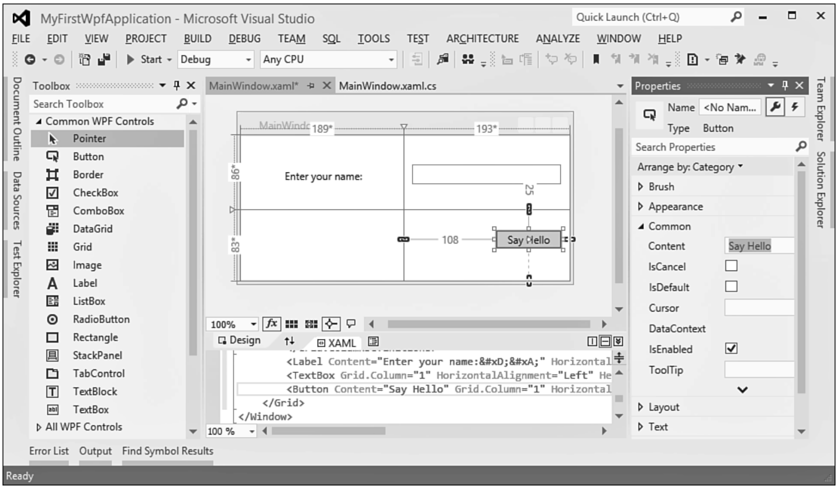
With this, we finish our discussion of Windows Forms for now and redirect our attention to its modern successor: WPF.

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).



Name <No Nam...
Type Button
Content Say Hello
IsCancel ☐
IsDefault ☐
IsEnabled ☑

```
        <Label Content="Enter your name:&#xD;&#xA;" Horizontal
        <TextBox Grid.Column="1" HorizontalAlignment="Left" He
        <Button Content="Say Hello" Grid.Column="1" Horizontal
    </Grid>
</Window>
```

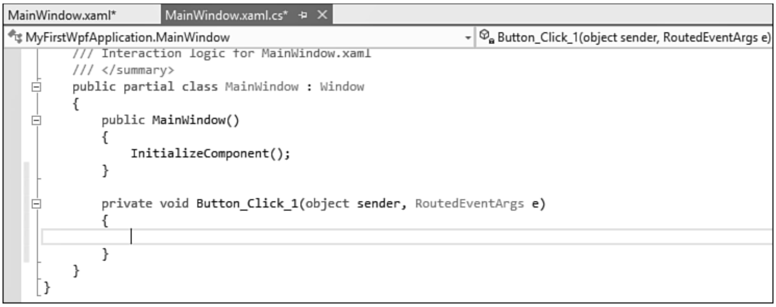
FIGURE 3.46 The integrated WPF designer.

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 type. 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.

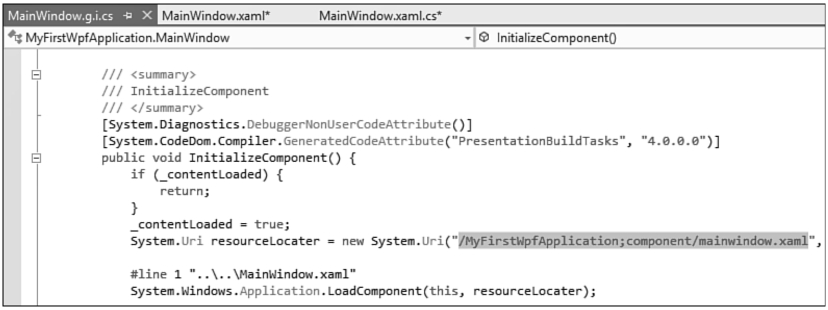


```
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            
        }
    }
}
```

FIGURE 3.47 Code skeleton for an event handler in WPF.

Notice, though, that there's still a call to `InitializeComponent` in `theWindow1` 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.



```
        /// <summary>
        /// InitializeComponent
        /// </summary>
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
        public void InitializeComponent() {
            if (_contentLoaded) {
                return;
            }
            _contentLoaded = true;
            System.Uri resourceLocater = new System.Uri("/MyFirstWpfApplication;component/mainwindow.xaml",

            #line 1 "..\..\MainWindow.xaml"
            System.Windows.Application.LoadComponent(this, resourceLocater);
```

FIGURE 3.48 Generated code for a WPF window definition.

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.

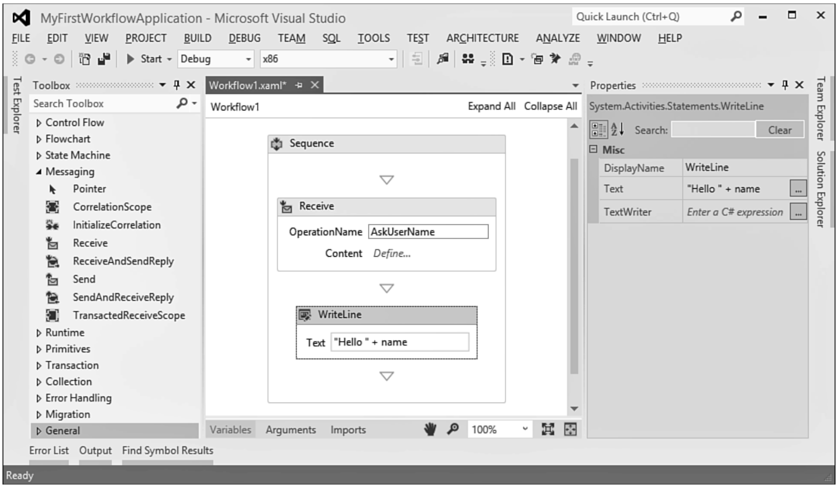
Windows Workflow Foundation

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.



OperationName AskUserName
Content Define...

Text "Hello " + name

DisplayName	WriteLine
Text	"Hello " + name
TextWriter	Enter a C# expression

FIGURE 3.49 A simple 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.

ASP.NET

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.

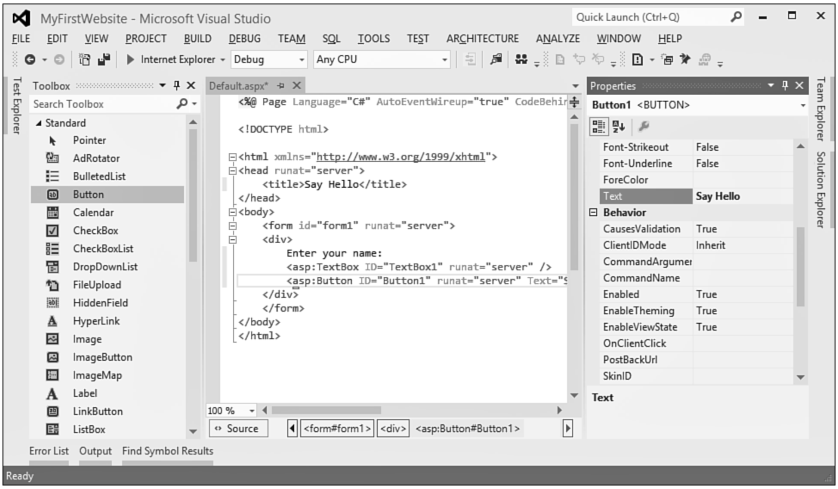
From the Visual Studio point of view, ASP.NET has rich project support with a built-in designer and deployment facilities. Figure 3.50 shows ASP.NET's page designer.

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:

```
<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.



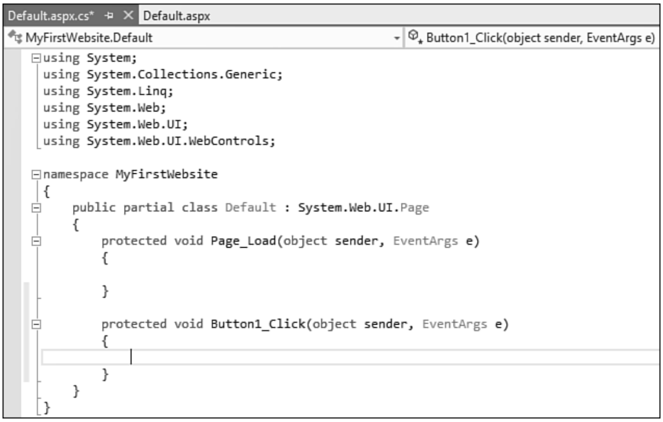
```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehir

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Say Hello</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        Enter your name:
        <asp:TextBox ID="TextBox1" runat="server" />
        <asp:Button ID="Button1" runat="server" Text="S
    </div>
    </form>
</body>
</html>
```

Button1 <BUTTON>

Font-Strikeout	False
Font-Underline	False
ForeColor	
Text	Say Hello
Behavior	
CausesValidation	True
ClientIDMode	Inherit
CommandArgumer	
CommandName	
Enabled	True
EnableTheming	True
EnableViewState	True
OnClientClick	
PostBackUrl	
SkinID	

Text

FIGURE 3.50 ASP.NET's page designer.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace MyFirstWebsite
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void Button1_Click(object sender, EventArgs e)
        {

        }
    }
}
```

FIGURE 3.51 Event handler code in ASP.NET.

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).

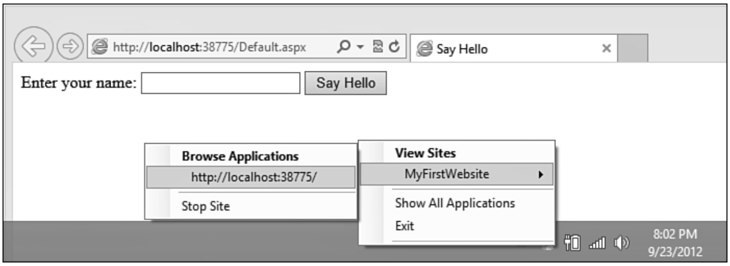


FIGURE 3.52 The Development Server has started.

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.

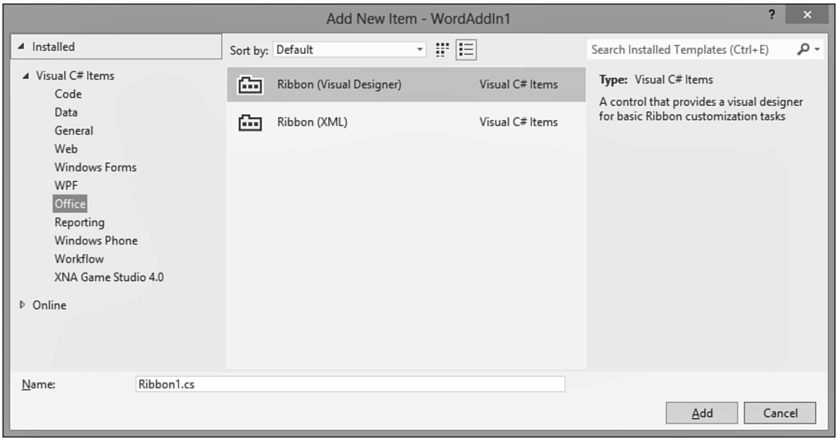
One of the designer-related innovations in Visual Studio 2012 is built-in support to create Office ribbon extensions, as shown in Figure 3.53.

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.



Sort by: Default

Type: Visual C# Items

A control that provides a visual designer for basic Ribbon customization tasks

Name: Ribbon1.cs

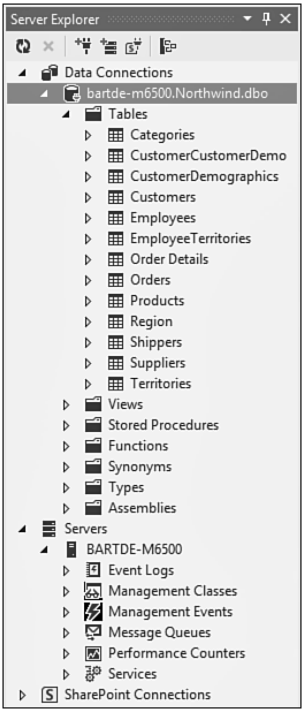
FIGURE 3.53 Ribbon designer support in Visual Studio 2012.

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.



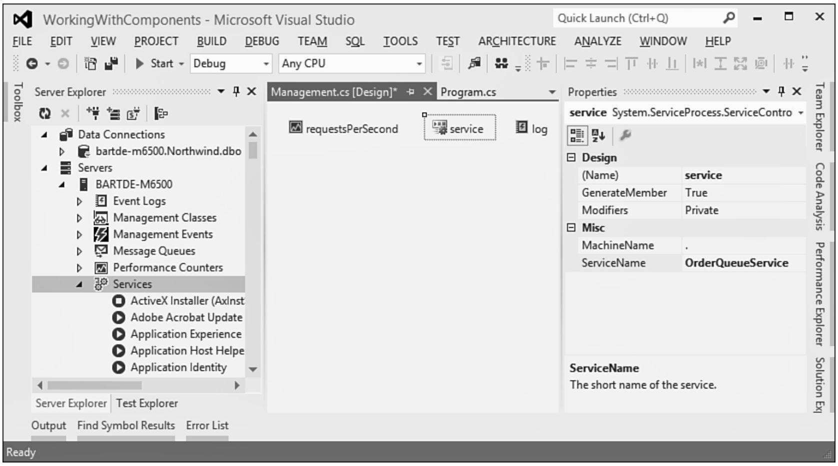
Server Explorer

- Data Connections
  - bartde-m6500.Northwind.dbo
    - Tables
      - Categories
      - CustomerCustomerDemo
      - CustomerDemographics
      - Customers
      - Employees
      - EmployeeTerritories
      - Order Details
      - Orders
      - Products
      - Region
      - Shippers
      - Suppliers
      - Territories
    - Views
    - Stored Procedures
    - Functions
    - Synonyms
    - Types
    - Assemblies
- Servers
  - BARTDE-M6500
    - Event Logs
    - Management Classes
    - Management Events
    - Message Queues
    - Performance Counters
    - Services
- SharePoint Connections

FIGURE 3.54 Server Explorer with an active database connection.

- 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.



service System.ServiceProcess.ServiceContro

Design	
(Name)	**service**
GenerateMember	True
Modifiers	Private
Misc	
MachineName	.
ServiceName	**OrderQueueService**

ServiceName
The short name of the service.

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.

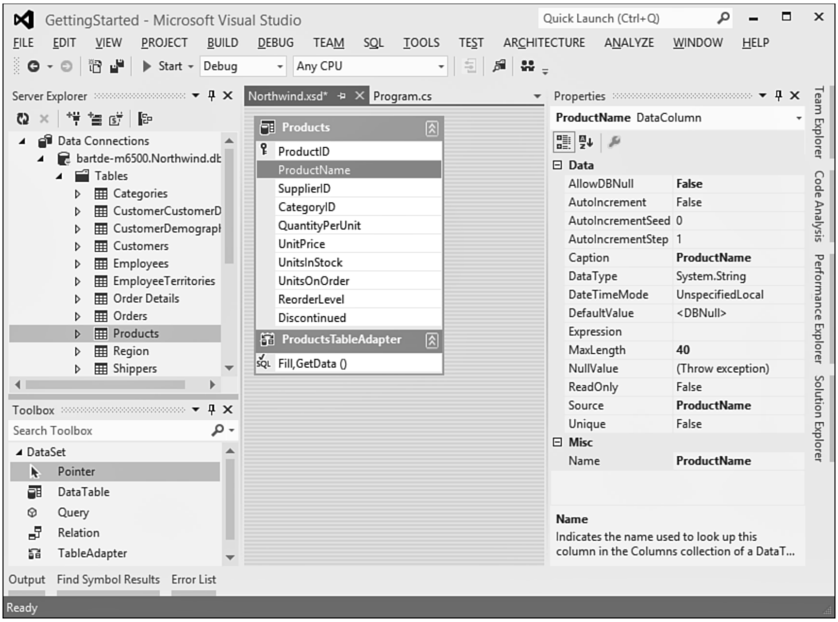
To create a strongly typed DataSet, Visual Studio provides a designer that can interact with Server Explorer. This makes it incredibly easy to generate a mapping just by carrying out a few drag-and-drop operations. Figure 3.56 shows the result of creating such a mapping.

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.



Data	
AllowDBNull	**False**
AutoIncrement	False
AutoIncrementSeed	0
AutoIncrementStep	1
Caption	**ProductName**
DataType	System.String
DateTimeMode	UnspecifiedLocal
DefaultValue	<DBNull>
Expression	
MaxLength	**40**
NullValue	(Throw exception)
ReadOnly	False
Source	**ProductName**
Unique	False
Misc	
Name	**ProductName**

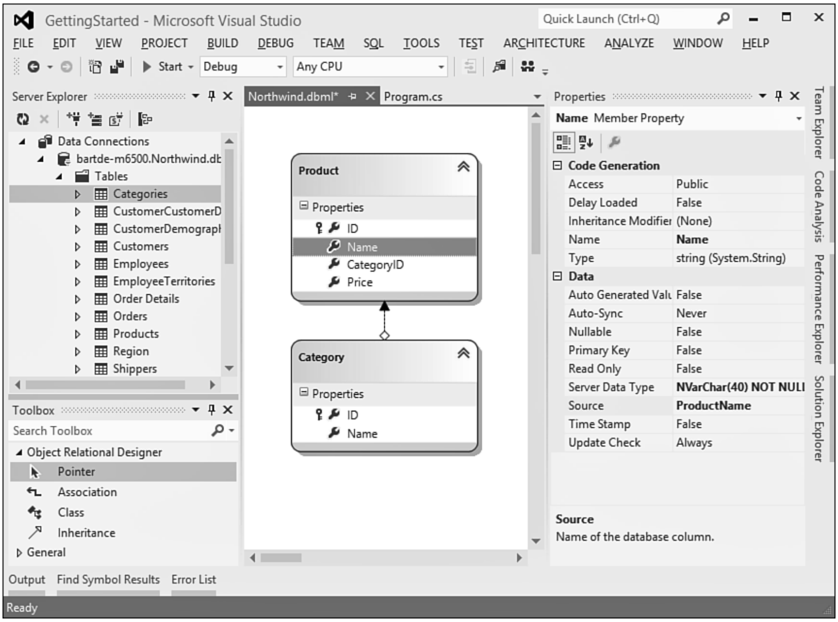
Name
Indicates the name used to look up this column in the Columns collection of a DataT...

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.



Code Generation	
Access	Public
Delay Loaded	False
Inheritance Modifier	(None)
Name	**Name**
Type	string (System.String)
Data	
Auto Generated Valu	False
Auto-Sync	Never
Nullable	False
Primary Key	False
Read Only	False
Server Data Type	**NVarChar(40) NOT NUL**
Source	**ProductName**
Time Stamp	False
Update Check	Always

Source
Name of the database column.

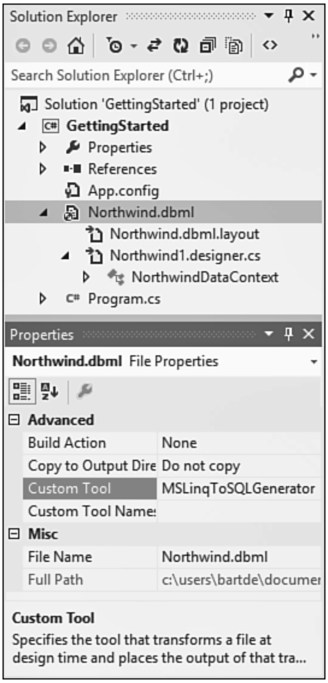
FIGURE 3.57 LINQ to SQL designer.

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.



- Solution 'GettingStarted' (1 project)
  - GettingStarted
    - Properties
    - References
    - App.config
    - Northwind.dbml
      - Northwind.dbml.layout
      - Northwind1.designer.cs
        - NorthwindDataContext
    - Program.cs

Northwind.dbml File Properties

Advanced

Build Action	None
Copy to Output Dire	Do not copy
Custom Tool	MSLinqToSQLGenerator
Custom Tool Name:	

Misc

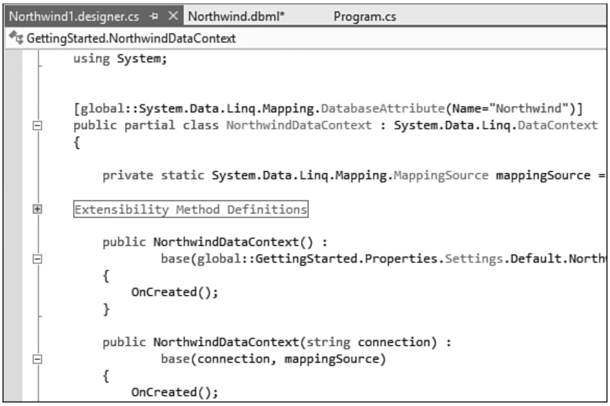
File Name	Northwind.dbml
Full Path	c:\users\bartde\documer

Custom Tool

Specifies the tool that transforms a file at design time and places the output of that tra...

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

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?



```
using System;


[global::System.Data.Linq.Mapping.DatabaseAttribute(Name="Northwind")]
public partial class NorthwindDataContext : System.Data.Linq.DataContext
{

    private static System.Data.Linq.Mapping.MappingSource mappingSource =

Extensibility Method Definitions

    public NorthwindDataContext() :
            base(global::GettingStarted.Properties.Settings.Default.North
    {
        OnCreated();
    }

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

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`:

```
public partial class NorthwindDataContext : System.Data.Linq.DataContext
{
    #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.

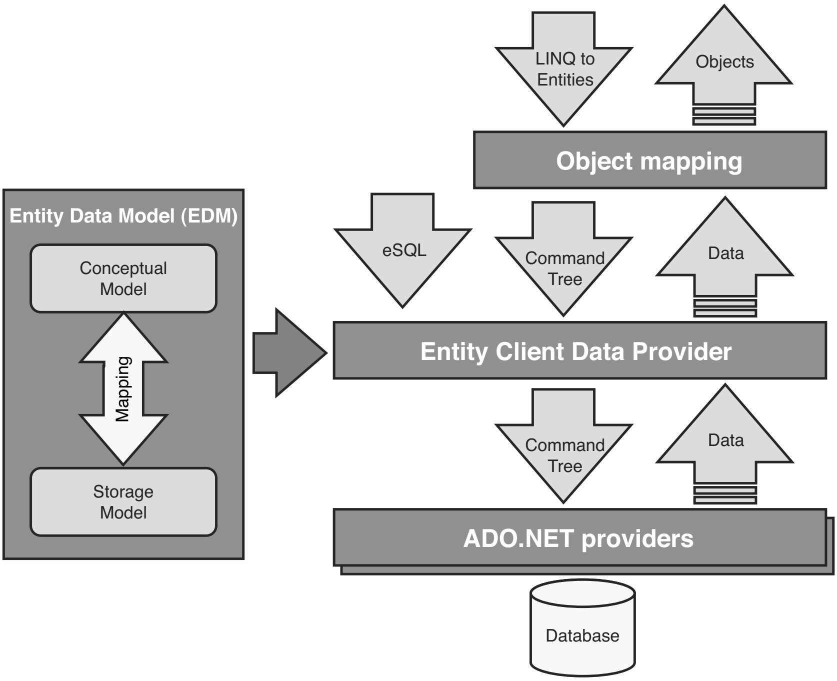
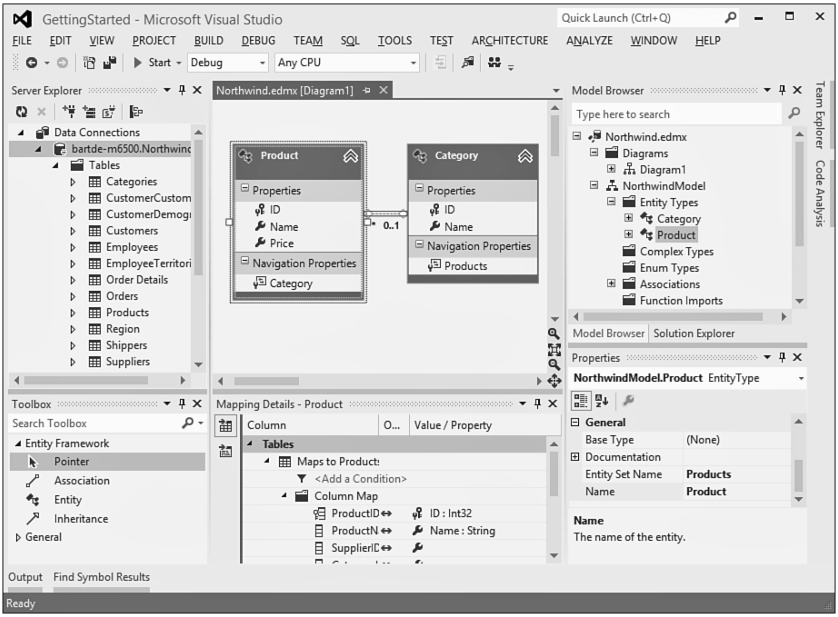


FIGURE 3.60 Entity Framework overview.

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.



Base Type (None)
Entity Set Name Products
Name Product

Name
The name of the entity.

FIGURE 3.61 ADO.NET Entity Framework designer.

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:

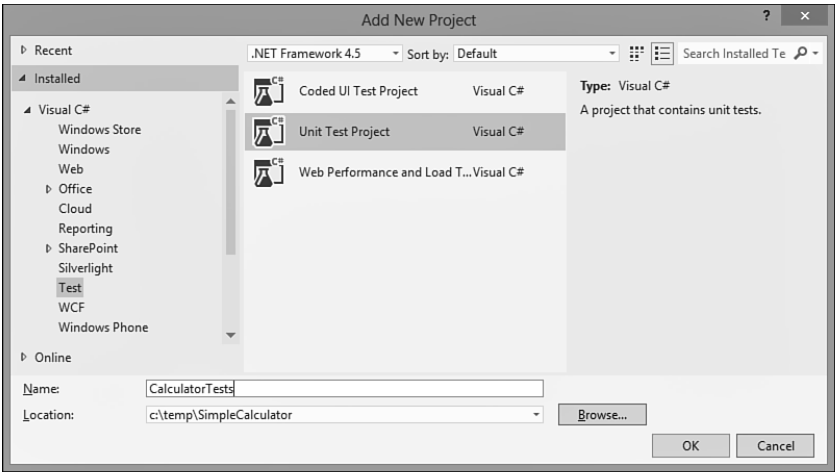
```
public static class Calculator
{
    public static int Add(int a, int b)
    {
        return a + b;
    }

    public static int Subtract(int a, int b)
    {
        return a - b;
    }
```

```
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    public static int Divide(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).



.NET Framework 4.5

Sort by: Default

Type: Visual C#

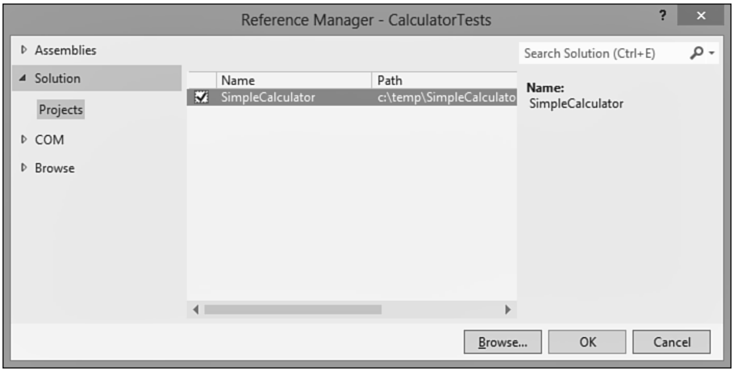
A project that contains unit tests.

Name: CalculatorTests

Location: c:\temp\SimpleCalculator

FIGURE 3.62 Creating a new unit test project.

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).



☑ SimpleCalculator c:\temp\SimpleCalculato

Name:
SimpleCalculator

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:

```
[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:

```
[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.

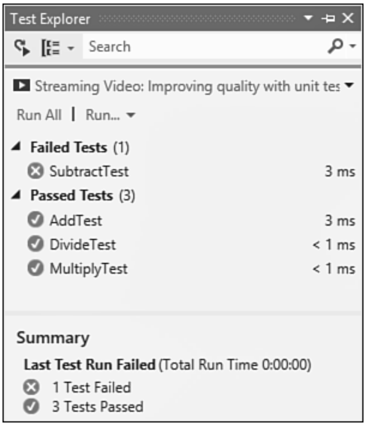
If all of this sounds like a wonderful dream, wake up now. With Pex, Microsoft Research has created such a toolkit that plugs in to Visual Studio. Pex stands for Program Exploration, reflecting its automated test case generation powers based on reasoning about the program. If you care about test coverage (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.



Failed Tests (1)
SubtractTest 3 ms
Passed Tests (3)
AddTest 3 ms
DivideTest < 1 ms
MultiplyTest < 1 ms

Summary
Last Test Run Failed (Total Run Time 0:00:00)
1 Test Failed
3 Tests Passed

FIGURE 3.64 Test results.

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.

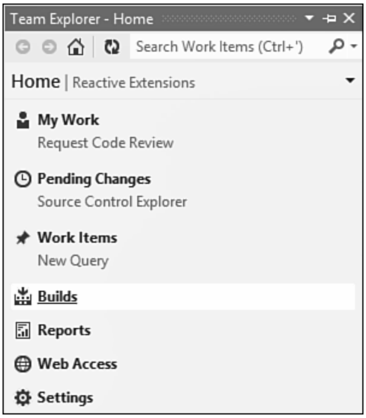


FIGURE 3.65 Team Explorer in Visual Studio 2012.

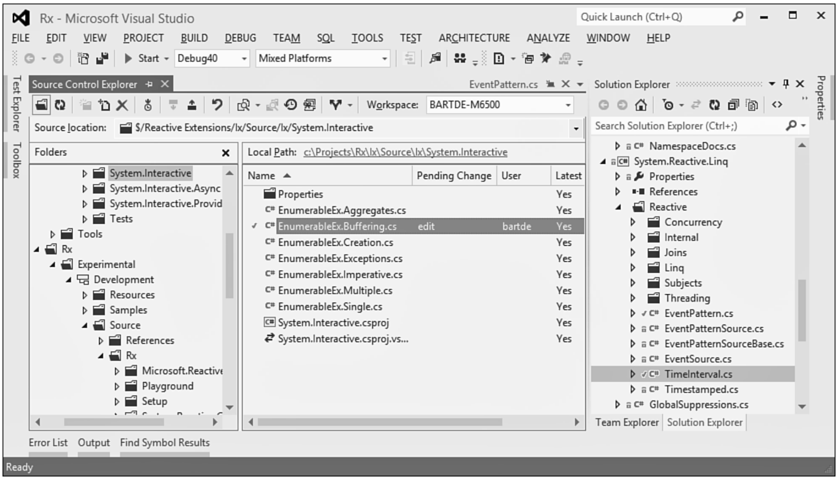
- Documents displays all sorts of documentation—Word documents, Visio diagrams, plain old text files, and such—that accompany the project. Those are also available through a SharePoint web interface.
- Reports leverages the SQL Server Reporting Services technology to display information about various 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.



Workspace: BARTDE-M6500

Source location: $/Reactive Extensions/Ix/Source/Ix/System.Interactive

Local Path: c:\Projects\Rx\Ix\Source\Ix\System.Interactive

Name	Pending Change	User	Latest
Properties			Yes
EnumerableEx.Aggregates.cs			Yes
✓ EnumerableEx.Buffering.cs	edit	bartde	Yes
EnumerableEx.Creation.cs			Yes
EnumerableEx.Exceptions.cs			Yes
EnumerableEx.Imperative.cs			Yes
EnumerableEx.Multiple.cs			Yes
EnumerableEx.Single.cs			Yes
System.Interactive.csproj			Yes
System.Interactive.csproj.vs...			Yes

FIGURE 3.66 Source control integrated in Visual Studio 2012.

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.

This page intentionally left blank

Index

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

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

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

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

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

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