Microsoft

Full Coverage of Multicore Programming

CLR via C#

3

THIRD EDITION

Jeffrey Richter

Wintellect
Know how.

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2009943026

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 WCT 5 4 3 2 1 0

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to msinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental Editor: Devon Musgrave
Project Editor: Valerie Woolley
Editorial Production: Custom Editorial Productions, Inc.
Technical Reviewer: Christophe Nasarre; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X16-61995

Table of Contents

Foreword

At first, when Jeff asked me to write the foreword for his book, I was so flattered! He must really respect me, I thought. Ladies, this is a common thought process error—trust me, he doesn't respect you. It turns out that I was about #14 on his list of potential foreword writers and he had to settle for me. Apparently, none of the other candidates (Bill Gates, Steve Ballmer, Catherine Zeta-Jones, . . .) were that into him. At least he bought me dinner.

But no one can tell you more about this book than I can. I mean, Catherine could give you a mobile makeover, but I know all kinds of stuff about reflection and exceptions and C# language updates because he has been talking on and on about it for years. This is standard dinner conversation in our house! Other people talk about the weather or stuff they heard at the water cooler, but we talk about .NET. Even Aidan, our six-year-old, asks questions about Jeff's book. Mostly about when he will be done writing it so they can play something "cool." Grant (age 2) doesn't talk yet, but his first word will probably be "Sequential."

In fact, if you want to know how this all started, it goes something like this. About 10 years ago, Jeff went to a "Secret Summit" at Microsoft. They pulled in a bunch of industry experts (Really, how do you get this title? Believe me, this isn't Jeff's college degree), and unveiled the new COM. Late that night in bed (in our house, this is what we discuss in bed), he talked about how COM is dead. And he was enchanted. Lovestruck, actually. In a matter of days he was hanging around the halls of Building 42 on Microsoft's Redmond campus, hoping to learn more about this wonderful .NET. The affair hasn't ended, and this book is what he has to show for it.

For years, Jeff has told me about threading. He really likes this topic. One time, in New Orleans, we went on a two-hour walk, alone, holding hands, and he spoke the whole time about how he had enough content for a threading book: The art of threading. How misunderstood threading in Windows is. It breaks his heart, all those threads out there. Where do they all go? Why were they created if no one had a plan for them? These are the questions of the universe to Jeff, the deeper meanings in life. Finally, in this book, he has written it down. It is all here. Believe me folks, if you want to know about threading, no one has thought about it more or worked with it more than Jeff has. And all those wasted hours of his life (he can't get them back) are here at your disposal. Please read it. Then send him an e-mail about how that information changed your life. Otherwise, he is just another tragic literary figure whose life ended without meaning or fulfillment. He will drink himself to death on diet soda.

This edition of the book even includes a new chapter about the runtime serializer. Turns out, this is not a new breakfast food for kids. When I figured out it was more computer talk and not something to put on my grocery list, I tuned it out. So I don't know what it says, but it is in here and you should read it (with a glass of milk).

My hope is that now he is finished talking about garbage collection in theory and can get on with actually collecting our garbage and putting it on the curb. Seriously people, how hard is that?

Folks, here is the clincher—this is Jeffrey Richter's magnum opus. This is it. There will be no more books. Of course, we say this every time he finishes one, but this time we really mean it. So, 13 books (give or take) later, this is the best and the last. Get it fast, because there are only a limited number and once they are gone—poof. No more. Just like QVC or something. Back to real life for us, where we can discuss the important things, like what the kids broke today and whose turn is it to change the diapers.

Kristin Trace (Jeffrey's wife)

November 24, 2009

A typical family breakfast at the Richter household

Introduction

It was October 1999 when some people at Microsoft first demonstrated the Microsoft .NET Framework, the common language runtime (CLR), and the C# programming language to me. The moment I saw all of this, I was impressed and I knew that it was going to change the way I wrote software in a very significant way. I was asked to do some consulting for the team and immediately agreed. At first, I thought that the .NET Framework was an abstraction layer over the Win32 API and COM. As I invested more and more of my time into it, however, I realized that it was much bigger. In a way, it is its own operating system. It has its own memory manager, its own security system, its own file loader, its own error handling mechanism, its own application isolation boundaries (AppDomains), its own threading models, and more. This book explains all these topics so that you can effectively design and implement software applications and components for this platform.

I have spent a good part of my life focusing on threading, concurrent execution, parallelism, synchronization, and so on. Today, with multicore computers becoming so prevalent, these subjects are becoming increasingly important. A few years ago, I decided to create a book dedicated to threading topics. However, one thing led to another and I never produced the book. When it came time to revise this book, I decided to incorporate all the threading information in here. So this book covers the .NET Framework's CLR and the C# programming language, and it also has my threading book embedded inside it (see Part V, "Threading").

It is October 2009 as I write this text, making it 10 years now that I've worked with the .NET Framework and C#. Over the 10 years, I have built all kinds of applications and, as a consultant to Microsoft, have contributed quite a bit to the .NET Framework itself. As a partner in my own company, Wintellect (*http://Wintellect.com*), I have worked with numerous customers to help them design software, debug software, performance-tune software, and solve issues they have with the .NET Framework. All these experiences have really helped me learn the spots that people have trouble with when trying to be productive with the .NET Framework. I have tried to sprinkle knowledge from these experiences through all the topics presented in this book.

Who This Book Is For

The purpose of this book is to explain how to develop applications and reusable classes for the .NET Framework. Specifically, this means that I intend to explain how the CLR works and the facilities that it offers. I'll also discuss various parts of the Framework Class Library (FCL). No book could fully explain the FCL—it contains literally thousands of types now, and this number continues to grow at an alarming rate. Therefore, here I'm concentrating on the core types that every developer needs to be aware of. And while this book isn't specifically about Windows Forms, Windows Presentation Foundation (WPF), Silverlight, XML Web services,

Web Forms, and so on, the technologies presented in the book are applicable to *all* these application types.

The book addresses Microsoft Visual Studio 2010, .NET Framework version 4.0, and version 4.0 of the C# programming language. Since Microsoft tries to maintain a large degree of backward compatibility when releasing a new version of these technologies, many of the things I discuss in this book apply to earlier versions as well. All the code samples use the C# programming language as a way to demonstrate the behavior of the various facilities. But, since the CLR is usable by many programming languages, the book's content is still quite applicable for the non-C# programmer.

Note You can download the code shown in the book from Wintellect's Web site (*http://Wintellect.com*). In some parts of the book, I describe classes in my own Power Threading Library. This library is available free of charge and can also be downloaded from Wintellect's Web site.

Today, Microsoft offers several versions of the CLR. There is the desktop/server version, which runs on 32-bit x86 versions of Microsoft Windows as well as 64-bit x64 and IA64 versions of Windows. There is the Silverlight version, which is produced from the same source code base as the desktop/server version of the .NET Framework's CLR. Therefore, everything in this book applies to building Silverlight applications, with the exception of some differences in how Silverlight loads assemblies. There is also a "lite" version of the .NET Framework called the .NET Compact Framework, which is available for Windows Mobile phones and other devices running the Windows CE operating system. Much of the information presented in this book is applicable to developing applications for the .NET Compact Framework, but this platform is not the primary focus of this book.

On December 13, 2001, ECMA International (*http://www.ecma-international.org/*) accepted the C# programming language, portions of the CLR, and portions of the FCL as standards. The standards documents that resulted from this have allowed other organizations to build ECMA-compliant versions of these technologies for other CPU architectures, as well as other operating systems. In fact, Novell produces Moonlight (*http://www.mono-project.com/Moonlight*), an open-source implementation of Silverlight (*http://Silverlight.net*) that is primarily for Linux and other UNIX/X11-based operating systems. Moonlight is based on the ECMA specifications. Much of the content in this book is about these standards; therefore, many will find this book useful for working with any runtime/library implementation that adheres to the ECMA standard.

Note My editors and I have worked hard to bring you the most accurate, up-to-date, in-depth, easy-to-read, painless-to-understand, bug-free information. Even with this fantastic team assembled, however, things inevitably slip through the cracks. If you find any mistakes in this book (especially bugs) or have some constructive feedback, I would greatly appreciate it if you would contact me at *JeffreyR@Wintellect.com*.

Dedication

To Kristin Words cannot express how I feel about our life together. I cherish our family and all our adventures. I'm filled each day with love for you.

To Aidan (age 6) and Grant (age 2) You both have been an inspiration to me and have taught me to play and have fun. Watching the two of you grow up has been so rewarding and enjoyable for me. I am lucky to be able to partake in your lives. I love and appreciate you more than you could ever know.

Acknowledgments

I couldn't have written this book without the help and technical assistance of many people. In particular, I'd like to thank my family. The amount of time and effort that goes into writing a book is hard to measure. All I know is that I could not have produced this book without the support of my wife, Kristin, and my two sons, Aidan and Grant. There were many times when we wanted to spend time together but were unable to due to book obligations. Now that the book project is completed, I really look forward to adventures we will all share together.

For this book revision, I truly had some fantastic people helping me. Christophe Nasarre, who I've worked with on several book projects, has done just a phenomenal job of verifying my work and making sure that I'd said everything the best way it could possibly be said. He has truly had a significant impact on the quality of this book. As always, the Microsoft Press editorial team is a pleasure to work with. I'd like to extend a special thank you to Ben Ryan, Valerie Woolley, and Devon Musgrave. Also, thanks to Jean Findley and Sue McClung for their editing and production support.

Support for This Book

Every effort has been made to ensure the accuracy of this book. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article accessible via the Microsoft Help and Support site. Microsoft Press provides support for books, including instructions for finding Knowledge Base articles, at the following Web site:

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

If you have questions regarding the book that are not answered by visiting the site above or viewing a Knowledge Base article, send them to Microsoft Press via e-mail to mspinput@microsoft.com.

Please note that Microsoft software product support is not offered through these addresses.

We Want to Hear from You

We welcome your feedback about this book. Please share your comments and ideas via the following short survey:

http://www.microsoft.com/learning/booksurvey

Your participation will help Microsoft Press create books that better meet your needs and standards.

Note We hope that you will give us detailed feedback via our survey. If you have questions about our publishing program, upcoming titles, or Microsoft Press in general, we encourage you to interact with us via Twitter at *http://twitter.com/MicrosoftPress*. For support issues, use only the e-mail address shown above.

Chapter 1

The CLR's Execution Model

In this chapter:

The Microsoft .NET Framework introduces many new concepts, technologies, and terms. My goal in this chapter is to give you an overview of how the .NET Framework is designed, introduce you to some of the new technologies the framework includes, and define many of the terms you'll be seeing when you start using it. I'll also take you through the process of building your source code into an application or a set of redistributable components (files) that contain types (classes, structures, etc.) and then explain how your application will execute.

Compiling Source Code into Managed Modules

OK, so you've decided to use the .NET Framework as your development platform. Great! Your first step is to determine what type of application or component you intend to build. Let's just assume that you've completed this minor detail; everything is designed, the specifications are written, and you're ready to start development.

Now you must decide which programming language to use. This task is usually difficult because different languages offer different capabilities. For example, in unmanaged C/C++, you have pretty low-level control of the system. You can manage memory exactly the way you want to, create threads easily if you need to, and so on. Microsoft Visual Basic 6, on the other hand, allows you to build UI applications very rapidly and makes it easy for you to control COM objects and databases.

The common language runtime (CLR) is just what its name says it is: a runtime that is usable by different and varied programming languages. The core features of the CLR (such as memory

management, assembly loading, security, exception handling, and thread synchronization) are available to any and all programming languages that target it—period. For example, the runtime uses exceptions to report errors, so all languages that target the runtime also get errors reported via exceptions. Another example is that the runtime also allows you to create a thread, so any language that targets the runtime can create a thread.

In fact, at runtime, the CLR has no idea which programming language the developer used for the source code. This means that you should choose whatever programming language allows you to express your intentions most easily. You can develop your code in any programming language you desire as long as the compiler you use to compile your code targets the CLR.

So, if what I say is true, what is the advantage of using one programming language over another? Well, I think of compilers as syntax checkers and "correct code" analyzers. They examine your source code, ensure that whatever you've written makes some sense, and then output code that describes your intention. Different programming languages allow you to develop using different syntax. Don't underestimate the value of this choice. For mathematical or financial applications, expressing your intentions by using APL syntax can save many days of development time when compared to expressing the same intention by using Perl syntax, for example.

Microsoft has created several language compilers that target the runtime: C++/CLI, C# (pronounced "C sharp"), Visual Basic, F# (pronounced "F sharp"), Iron Python, Iron Ruby, and an Intermediate Language (IL) Assembler. In addition to Microsoft, several other companies, colleges, and universities have created compilers that produce code to target the CLR. I'm aware of compilers for Ada, APL, Caml, COBOL, Eiffel, Forth, Fortran, Haskell, Lexico, LISP, LOGO, Lua, Mercury, ML, Mondrian, Oberon, Pascal, Perl, Php, Prolog, RPG, Scheme, Smalltalk, and Tcl/Tk.

Figure 1-1 shows the process of compiling source code files. As the figure shows, you can create source code files written in any programming language that supports the CLR. Then you use the corresponding compiler to check the syntax and analyze the source code. Regardless of which compiler you use, the result is a *managed module*. A managed module is a standard 32-bit Microsoft Windows portable executable (PE32) file or a standard 64-bit Windows portable executable (PE32+) file that requires the CLR to execute. By the way, managed assemblies always take advantage of Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) in Windows; these two features improve the security of your whole system.

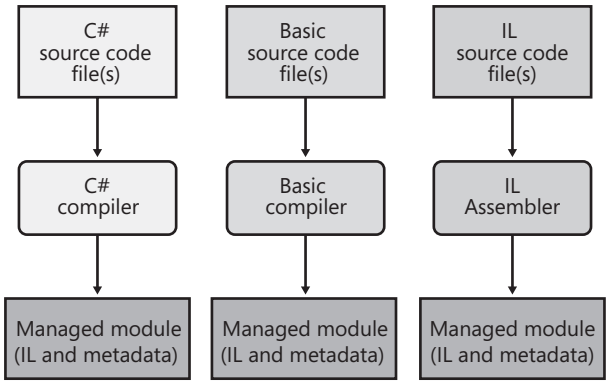

FIGURE 1-1 Compiling source code into managed modules

Table 1-1 describes the parts of a managed module.

TABLE 1-1 Parts of a Managed Module

Part	Description
PE32 or PE32+ header	The standard Windows PE file header, which is similar to the Common Object File Format (COFF) header. If the header uses the PE32 format, the file can run on a 32-bit or 64-bit version of Windows. If the header uses the PE32+ format, the file requires a 64-bit version of Windows to run. This header also indicates the type of file: GUI, CUI, or DLL, and contains a timestamp indicating when the file was built. For modules that contain only IL code, the bulk of the information in the PE32(+) header is ignored. For modules that contain native CPU code, this header contains information about the native CPU code.
CLR header	Contains the information (interpreted by the CLR and utilities) that makes this a managed module. The header includes the version of the CLR required, some flags, the **`MethodDef`** metadata token of the managed module's entry point method (**`Main`** method), and the location/size of the module's metadata, resources, strong name, some flags, and other less interesting stuff.
Metadata	Every managed module contains metadata tables. There are two main types of tables: tables that describe the types and members defined in your source code and tables that describe the types and members referenced by your source code.
IL code	Code the compiler produced as it compiled the source code. At runtime, the CLR compiles the IL into native CPU instructions.

Native code compilers produce code targeted to a specific CPU architecture, such as x86, x64, or IA64. All CLR-compliant compilers produce IL code instead. (I'll go into more detail about IL code later in this chapter.) IL code is sometimes referred to as *managed code* because the CLR manages its execution.

In addition to emitting IL, every compiler targeting the CLR is required to emit full *metadata* into every managed module. In brief, metadata is a set of data tables that describe what is defined in the module, such as types and their members. In addition, metadata also has tables indicating what the managed module references, such as imported types and their members. Metadata is a superset of older technologies such as COM's Type Libraries and Interface Definition Language (IDL) files. The important thing to note is that CLR metadata is far more complete. And, unlike Type Libraries and IDL, metadata is always associated with the file that contains the IL code. In fact, the metadata is always embedded in the same EXE/DLL as the code, making it impossible to separate the two. Because the compiler produces the metadata and the code at the same time and binds them into the resulting managed module, the metadata and the IL code it describes are never out of sync with one another.

Metadata has many uses. Here are some of them:

- Metadata removes the need for native C/C++ header and library files when compiling because all the information about the referenced types/members is contained in the file that has the IL that implements the type/members. Compilers can read metadata directly from managed modules.
- Microsoft Visual Studio uses metadata to help you write code. Its IntelliSense feature parses metadata to tell you what methods, properties, events, and fields a type offers, and in the case of a method, what parameters the method expects.
- The CLR's code verification process uses metadata to ensure that your code performs only "type-safe" operations. (I'll discuss verification shortly.)
- Metadata allows an object's fields to be serialized into a memory block, sent to another machine, and then deserialized, re-creating the object's state on the remote machine.
- Metadata allows the garbage collector to track the lifetime of objects. For any object, the garbage collector can determine the type of the object and, from the metadata, know which fields within that object refer to other objects.

In Chapter 2, "Building, Packaging, Deploying, and Administering Applications and Types," I'll describe metadata in much more detail.

Microsoft's C#, Visual Basic, F#, and the IL Assembler always produce modules that contain managed code (IL) and managed data (garbage-collected data types). End users must have the CLR (presently shipping as part of the .NET Framework) installed on their machine in order to execute any modules that contain managed code and/or managed data in the same way that they must have the Microsoft Foundation Class (MFC) library or Visual Basic DLLs installed to run MFC or Visual Basic 6 applications.

By default, Microsoft's C++ compiler builds EXE/DLL modules that contain unmanaged (native) code and manipulate unmanaged data (native memory) at runtime. These modules don't require the CLR to execute. However, by specifying the **/CLR** command-line switch, the C++ compiler produces modules that contain managed code, and of course, the CLR must

then be installed to execute this code. Of all of the Microsoft compilers mentioned, C++ is unique in that it is the only compiler that allows the developer to write both managed and unmanaged code and have it emitted into a single module. It is also the only Microsoft compiler that allows developers to define both managed and unmanaged data types in their source code. The flexibility provided by Microsoft's C++ compiler is unparalleled by other compilers because it allows developers to use their existing native C/C++ code from managed code and to start integrating the use of managed types as they see fit.

Combining Managed Modules into Assemblies

The CLR doesn't actually work with modules, it works with assemblies. An *assembly* is an abstract concept that can be difficult to grasp initially. First, an assembly is a logical grouping of one or more modules or resource files. Second, an assembly is the smallest unit of reuse, security, and versioning. Depending on the choices you make with your compilers or tools, you can produce a single-file or a multifile assembly. In the CLR world, an assembly is what we would call a *component*.

In Chapter 2, I'll go over assemblies in great detail, so I don't want to spend a lot of time on them here. All I want to do now is make you aware that there is this extra conceptual notion that offers a way to treat a group of files as a single entity.

Figure 1-2 should help explain what assemblies are about. In this figure, some managed modules and resource (or data) files are being processed by a tool. This tool produces a single PE32(+) file that represents the logical grouping of files. What happens is that this PE32(+) file contains a block of data called the *manifest*. The manifest is simply another set of metadata tables. These tables describe the files that make up the assembly, the publicly exported types implemented by the files in the assembly, and the resource or data files that are associated with the assembly.

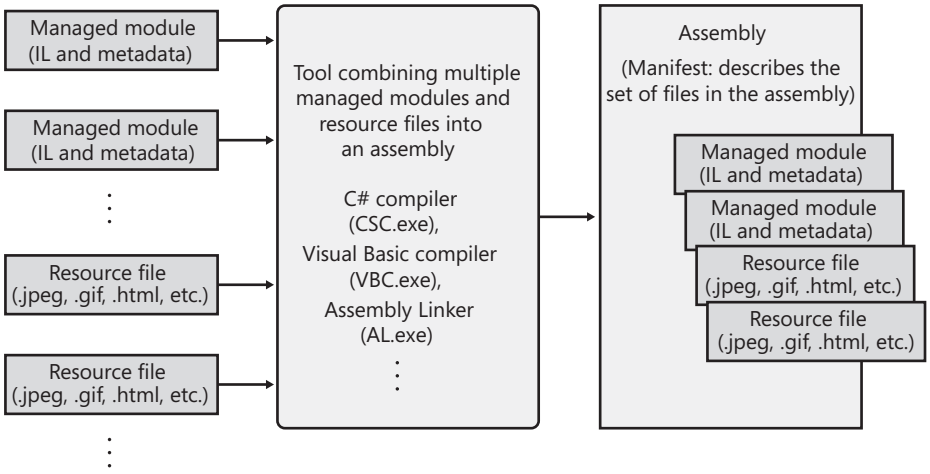

FIGURE 1-2 Combining managed modules into assemblies

By default, compilers actually do the work of turning the emitted managed module into an assembly; that is, the C# compiler emits a managed module that contains a manifest. The manifest indicates that the assembly consists of just the one file. So, for projects that have just one managed module and no resource (or data) files, the assembly will be the managed module, and you don't have any additional steps to perform during your build process. If you want to group a set of files into an assembly, you'll have to be aware of more tools (such as the assembly linker, AL.exe) and their command-line options. I'll explain these tools and options in Chapter 2.

An assembly allows you to decouple the logical and physical notions of a reusable, securable, versionable component. How you partition your code and resources into different files is completely up to you. For example, you could put rarely used types or resources in separate files that are part of an assembly. The separate files could be downloaded on demand from the Web as they are needed at runtime. If the files are never needed, they're never downloaded, saving disk space and reducing installation time. Assemblies allow you to break up the deployment of the files while still treating all of the files as a single collection.

An assembly's modules also include information about referenced assemblies (including their version numbers). This information makes an assembly *self-describing*. In other words, the CLR can determine the assembly's immediate dependencies in order for code in the assembly to execute. No additional information is required in the registry or in Active Directory Domain Services (AD DS). Because no additional information is needed, deploying assemblies is much easier than deploying unmanaged components.

Loading the Common Language Runtime

Each assembly you build can be either an executable application or a DLL containing a set of types for use by an executable application. Of course, the CLR is responsible for managing the execution of code contained within these assemblies. This means that the .NET Framework must be installed on the host machine. Microsoft has created a redistribution package that you can freely ship to install the .NET Framework on your customers' machines. Some versions of Windows ship with the .NET Framework already installed.

You can tell if the .NET Framework has been installed by looking for the MSCorEE.dll file in the %SystemRoot%\System32 directory. The existence of this file tells you that the .NET Framework is installed. However, several versions of the .NET Framework can be installed on a single machine simultaneously. If you want to determine exactly which versions of the .NET Framework are installed, examine the subkeys under the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP
```

The .NET Framework SDK includes a command-line utility called CLRVer.exe that shows all of the CLR versions installed on a machine. This utility can also show which version of the CLR is