Joe Mayo

MVP
Microsoft Most Valuable Professional

C# 3.0

UNLEASHED

With the .NET Framework 3.5

SAMS

C# 3.0 Unleashed
With the .NET Framework 3.5

ISBN-13: 978-0-672-32981-4
ISBN-10: 0-672-32981-6

Library of Congress Cataloging-in-Publication Data
Mayo, Joseph.
C# 3.0 unleashed : with the .NET Framework 3.5 / Joe Mayo. — 1st ed.
p. cm.
ISBN 978-0-672-32981-4
1. C# (Computer program language) 2. Microsoft .NET Framework. I. Title.
QA76.73.C154M38 2008
006.7'882—dc22

2008026117

Printed in the United States of America

First Printing June 2008

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

Warning and Disclaimer
Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The 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@pearson.com

Editor-in-Chief
Karen Gettman

Executive Editor
Neil Rowe

Acquisitions Editor
Brook Farling

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Andrew Beaster

Copy Editor
Keith Cline

Indexer
Brad Herriman

Proofreader
San Dee Phillips

Technical Editors
Tony Gravagno
Todd Meister
J. Boyd Nolan

Publishing Coordinator
Cindy Teeters

Cover Designer
Gary Adair

Composition
Jake McFarland

Introduction

Welcome to *C# 3.0 Unleashed*, a programmer's guide and reference to the C# (pronounced "C sharp") programming language. C# is primarily an object-oriented programming language, created at Microsoft, which emphasizes a component-based approach to software development. In its third version, C# is still evolving, and this book guides you on a journey of learning how that evolution helps you accomplish more in your software engineering endeavors.

C# is one of several languages of the .NET (pronounced "dot net") platform, which includes a runtime engine called the Common Language Runtime (CLR) and a huge class library. The runtime is a virtual machine that manages code and provides several other services. The class library includes literally thousands of reusable objects and supports several user interface technologies for both desktop and Web Application development.

C# is evolving as a programming language. It began life as an object-oriented, component-based language but now is growing into areas that were once considered the domain of functional programming languages. Throughout this book, you'll see examples of objects and components being used as building blocks for applications. You'll also see many examples that include Language Integrated Query (LINQ), which is a declarative way to query data sources, whether the data source is in the form of objects, relational, XML, or any other format.

Just as C# (and the .NET platform) has evolved, so has this book. *C# Unleashed* began as a language-centric learning guide and reference for applying the C# programming language. The audience was varied because C# was new and developers from all types of backgrounds were

programming with it. All the applications compiled on the command line, and all you needed was the .NET Framework SDK and an editor to do everything.

At its essence, the same concepts driving the first version of this book made it into this version. For example, you don't need to already know .NET before getting started. If you've programmed with any programming language, *C# 3.0 Unleashed* should be an easy on-ramp for you. This book contains a few command-line examples, especially in the beginning, because I believe that using the command line is a skill that is still necessary and useful. However, I quickly move to the Visual Studio 2008 (VS2008) Integrated Development Environment (IDE) for the largest share of the rest of the book. You aren't required to use VS2008, however; I show you right away how to build your applications without it, and Appendix A, "Compiling Programs," is a guide to command-line options with examples (just like the first version of *C# Unleashed*). However, VS2008 is an incredible tool for increasing productivity, and I provide tips throughout this book for cranking out algorithms with code-focused RAD.

In addition to coverage of VS2008, I've included several new chapters for the newest technologies, such as Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), and AJAX. If you like the cutting edge, there are chapters on the ADO.NET Entity Framework and ADO.NET Data Services. Speaking of data, I've added an entire part of this book with multiple chapters on working with data.

Since July 2000, when I cracked open the first public pre-beta release of .NET, I've been hooked, with C# as my language of choice. I've made a good living and found my C# skills in demand, even in a difficult economy. Most of all, I've gained an enormous amount of experience in both teaching, as a formal course instructor, and as a developer, delivering value to customers with an awesome toolset. I hope that all the gotchas, tips, and doses of reality that I've encountered and shared in this book will help you learn and thrive as I have.

Why This Book Is for You

If you've developed software in any other computer programming language, you will be able to understand the contents of this book with no trouble. You already know how to make logical decisions and construct iterative code. You also understand variables and basic number systems such as hexadecimal. Novices may want to start with something at the introductory level, such as *Sams Teach Yourself C# in 21 Days*. Honestly, ambitious beginners could do well with this book if they're motivated.

This is a book written for any programmer who wants to learn C# and .NET. It's basic enough for you to see every aspect of C# that's possible, yet it's sufficiently advanced to provide insight into the modern enterprise-level tasks you deal with every day.

Organization and Goals

C# 3.0 Unleashed is divided into eight parts. To promote learning from the beginning, it starts with the simpler material and those items strictly related to the C# language itself. Later, the book moves into other C#-related areas, showing how to use data, user interface technologies, web services, and other useful .NET technologies.

Part 1 is the beginning, covering basic C# language syntax and other essentials. Chapter 1 starts you off by discussing the .NET platform. This is an important chapter because you need to know the environment that you are building applications for. It permeates everything else you do as a C# developer and should be a place you return to on occasion to remind yourself of the essential ingredients of being a successful C# developer. In Chapter 2, you learn how to build a simple C# application using both the command line and VS2008. It is just the beginning of much VS2008 coverage to come. Chapter 3 is another essential milestone for success in developing .NET applications with C#, learning the type system. Chapters 4 and 5 show you how to work with strings and arrays, respectively. By the time you reach Chapter 7, you'll have enough skills necessary to write a simple application and encounter bugs. So, I hope you find my tips on using the VS2008 debugger helpful before moving on to more complexity with object-oriented programming in Part 2.

Part 2 covers object and component programming in C#. In the first version of *C# Unleashed*, I dedicated an entire chapter to basic object-oriented programming concepts. What changed in *C# 3.0 Unleashed* is that I weaved some of those concepts into other chapters. This way, developers who already know object-oriented programming don't have to skip over an entire chapter, but those who don't aren't completely left out. Mostly, I concentrate on how C# implements object-oriented programming, explaining those nuances that are of interest to existing object-oriented programmers and necessary for any C# developer.

Part 3 teaches you some of the more advanced features of C#. With an understanding of objects from Part 2, you learn about object lifetime—when objects are first instantiated and when they are cleaned up from memory. An entire body of knowledge builds upon earlier chapters, leading to where you need to be to understand .NET memory management, the Garbage Collector, what it means for you as a C# developer, and mostly, what you can do to ensure that your objects and the resources they work with are properly managed.

Part 4 gives you five chapters of data. Feedback from the first version of this book indicated that you wanted more. So, now you can learn about LINQ to Objects, LINQ to SQL, ADO.NET, LINQ to DataSet, XML, LINQ to XML, ADO.NET Entity Framework, LINQ to Entities, ADO.NET Data Services, and LINQ to Data Services. Really, five chapters aren't the end of the story, and there is good reason why I moved data earlier in the book: I use LINQ throughout the rest of the book. In addition to learning how to use all of these data access technologies, you'll see many examples in the whole book.

Part 5 demonstrates how to use various desktop user interface technologies. You have choices, console applications, which were beefed up in .NET 2.0, Windows Forms, and WPF. By the way, if you are interested in Silverlight, you'll want to read the WPF chapter

first because both technologies use XAML, the same layout, and the same control set. Not only does it help me bring more information to you on these new technologies, but it also should be comforting that what you learn with one technology is useful with another, expanding your skill set as a .NET developer.

Part 6 teaches you how to build web user interfaces. ASP.NET is the primary web UI technology for .NET today, and I provide a fair amount of coverage to help you get up-to-speed with it. You'll want to pay attention to the discussion of the difference between desktop and web applications because it affects how you develop ASP.NET applications. In recent years, Asynchronous JavaScript and XML (AJAX) has become a hot topic. I show you how to use ASP.NET AJAX, which ships with VS2008, to make your ASP.NET pages more responsive to the user. The newest web UI technology is Silverlight, which enables you to build interactive websites that were once only possible with desktop UI technologies. A couple of the new capabilities of Silverlight are easier ways to play audio and video on the web and animation; these new capabilities allow you to build web experiences similar to Adobe Flash.

Part 7 brings you in touch with various communications technologies. In a connected world, these chapters teach you how to use essential tools. You learn how to use TCP/IP, HTTP, and FTP, and send email using .NET Framework libraries. The remoting chapter is still there, as is the web services chapter. However, an additional chapter covers the new WCF web services.

Part 8 covers topics in architecture and design. Many programmers learn C# and all the topics discussed previously and then find their own way to build applications with what they've learned. If they find an effective way to build applications, then that is positive. However, it's common for people to want to know what the best way is for putting together all of these objects, components, and services to build a usable application. I don't have all the answers because architecture and design is a big topic, and there are as many opinions about it as there are questions. However, I've taken a quick foray into the subject, showing you some of the techniques that have worked for me. You learn how C# and .NET support common design patterns and make it easy for you to use these patterns. I show you how to build an n-layered application and describe a couple more ways that you can take what I've presented and use it in your own way. I also show you how to use a couple .NET tools, including the Class Designer, and introduce you to Windows Workflow (WF), which has a graphical design surface for building applications graphically.

Part 9 is a grab bag of technologies that could be important to your development, depending on what you want to do. For example, multithreading is something that most programmers will do on occasion. However, multithreading is a skill that most programmers will need as multiprocessing and multicore CPUs become more common, meaning that I added more multiprocessing/multithreaded information in this version of the book. Depending on where you are in the world, localization and globalization could be very important, so I explain the essentials of resources and satellite assemblies for localization

purposes. There is still a lot of legacy code that people need to communicate with, depending on the needs of the project you are working on. To help out, the chapter on Interop covers P/Invoke for interoperating with Win32 DLLs and COM Interop for working with COM. There's also some information on working with COM+. For those of you who like a solution out of the box, I explain how to use the .NET trace facilities for instrumenting and logging. There's also a section on how to use existing performance counters and how to instrument your own code with a custom performance counter for diagnostics through the Windows Performance Monitor.

Part 10 helps you with your ultimate goal: deploying code. This is a series of quick chapters to help you build setup programs and deploy desktop or web applications. Before that, I give you some more information about assemblies and what they are made of. The Security chapter will help you learn how the .NET Code Access Security (CAS) system works. Along the way, I throw in several tips to ensure that your deployment endeavors go more smoothly than if you would have had to do it alone.

That's what this book is all about. I wish you luck in learning C# and hope that you find *C# 3.0 Unleashed* a helpful learning tool and useful reference.

CHAPTER 1

Introducing the .NET Platform

IN THIS CHAPTER

- What Is .NET?
- The Common Language Runtime (CLR)
- The .NET Framework Class Library (FCL)
- C# and Other .NET Languages
- The Common Type System (CTS)
- The Common Language Specification (CLS)

As a C# developer, it's important to understand the environment you are building applications on: Microsoft .NET (pronounced "Dot Net"). After all, your design and development decisions will often be influenced by code-compilation practicalities, the results of compilation, and the behavior of applications in the runtime environment. The foundation of all .NET development begins here, and throughout this book I occasionally refer back to this chapter when explaining concepts that affect the practical implementation of C#.

By learning about the .NET environment, you can gain an understanding of what .NET is and what it means to you. You learn about the parts of .NET, including the Common Language Runtime (CLR), the .NET Framework Class Library, and how .NET supports multiple languages. Along the way, you see how the parts of .NET tie together, their relationships, and what they do for you. First, however, you need to know what .NET is, which is explained in the next section.

What Is .NET?

Microsoft .NET, which I refer to as just .NET, is a platform for developing "managed" software. The word *managed* is key here—a concept setting the .NET platform apart from many other development environments. I'll explain what the word *managed* means and why it is an integral capability of the .NET platform.

When referring to other development environments, as in the preceding paragraph, I'm focusing on the traditional

practice of compiling to an executable file that contains machine code and how that file is loaded and executed by the operating system. Figure 1.1 shows what I mean about the traditional compilation-to-execution process.

FIGURE 1.1 Traditional compilation.

In the traditional compilation process, the executable file is binary and can be executed by the operating system immediately. However, in the managed environment of .NET, the file produced by the compiler (the C# compiler in our case) is not an executable binary. Instead, it is an assembly, shown in Figure 1.2, which contains metadata and intermediate language code.

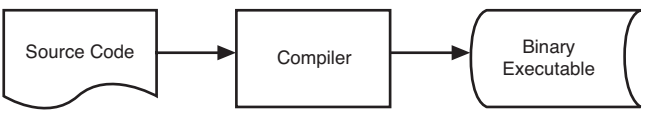

FIGURE 1.2 Managed compilation.

As mentioned in the preceding paragraph, an assembly contains intermediate language and metadata rather than binary code. This intermediate language is called Microsoft Intermediate Language (MSIL), which is commonly referred to as IL. IL is a high-level, component-based assembly language. In later sections of this chapter, you learn how IL supports a common type system and multiple languages in the same platform.

.NET STANDARDIZATION

.NET has been standardized by both the European Computer Manufacturers Association (ECMA) and the Open Standards Institute (OSI). The standard is referred to as the Common Language Infrastructure (CLI). Similarly, the standardized term for IL is Common Intermediate Language (CIL).

In addition to .NET, there are other implementations of CIL—the two most well known by Microsoft and Novell. Microsoft's implementation is an open source offering for the purposes of research and education called the Shared Source Common Language Infrastructure (SSCLI). The Novell offering is called Mono, which is also open source.

Beyond occasional mention, this book focuses mainly on the Microsoft .NET implementation of the CLI standard.

The other part of an assembly is metadata, which is extra information about the code being used in the assembly. Figure 1.3 shows the contents of an assembly.

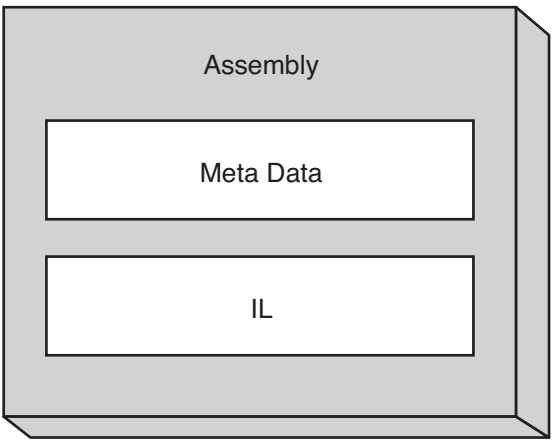

FIGURE 1.3 Assembly contents.

Figure 1.3 is a simplified version of an assembly, showing only those parts pertaining to the current discussion. Assemblies have other features that illustrate the difference between an assembly and an executable file. Specifically, the role of an assembly is to be a unit of deployment, execution, identity, and security in the managed environment. In Part X, Chapters 43 and 44 explain more about the role of the assembly in deployment, identity, and security. The fact that an assembly contains metadata and IL, instead of only binary code, has a significant advantage, allowing execution in a managed environment. The next section explains how the CLR uses the features of an assembly to manage code during execution.

The Common Language Runtime (CLR)

As introduced in the preceding section, C# applications are compiled to IL, which is executed by the CLR. This section highlights several features of the CLR. You'll also see how the CLR manages your application during execution.

Why Is the CLR Important?

In many traditional execution environments of the past, programmers needed to perform a lot of the low-level work (plumbing) that applications needed to support. For example, you had to build custom security systems, implement error handling, and manage memory.

The degree to which these services were supported on different language platforms varied considerably. Visual Basic (VB) programmers had built-in memory management and an error-handling system, but they didn't always have easy access to all the features of COM+, which opened up more sophisticated security and transaction processing. C++ programmers have full access to COM+ and exception handling, but memory management is a totally manual process. In a later section, you learn about how .NET supports multiple

languages, but knowing just a little about a couple of popular languages and a couple of the many challenges they must overcome can help you to understand why the CLR is such a benefit for a C# developer.

The CLR solves many problems of the past by offering a feature-rich set of plumbing services that all languages can use. The features described in the next section further highlight the value of the CLR.

CLR Features

This section describes, more specifically, what the CLR does for you. Table 1.1 summarizes CLR features with descriptions and chapter references (if applicable) in this book where you can find more detailed information.

TABLE 1.1 CLR Features

Feature	Description
.NET Framework Class Library support	Contains built-in types and libraries to manage assemblies, memory, security, threading, and other runtime system support
Debugging	Facilities for making it easier to debug code. (Chapter 7)
Exception management	Allows you to write code to create and handle exceptions. (Chapter 11)
Execution management	Manages the execution of code
Garbage collection	Automatic memory management and garbage collection (Chapter 15)
Interop	Backward-compatibility with COM and Win32 code. (Chapter 41)
Just-In-Time (JIT) compilation	An efficiency feature for ensuring that the CLR only compiles code just before it executes
Security	Traditional role-based security support, in addition to Code Access Security (CAS) (Chapter 44)
Thread management	Allows you to run multiple threads of execution (Chapter 39)
Type loading	Finds and loads assemblies and types
Type safety	Ensures references match compatible types, which is very useful for reliable and secure code (Chapter 4)

In addition to the descriptions provided in Table 1.1, the following sections expand upon a few of the CLR features. These features are included in the CLR execution process.

The CLR Execution Process

Beyond just executing code, parts of the execution process directly affect your application design and how a program behaves at runtime. Many of these subjects are handled throughout this book, but this section highlights specific additional items you should know about.

From the time you or another process selects a .NET application for execution, the CLR executes a special process to run your application, shown in Figure 1.4.

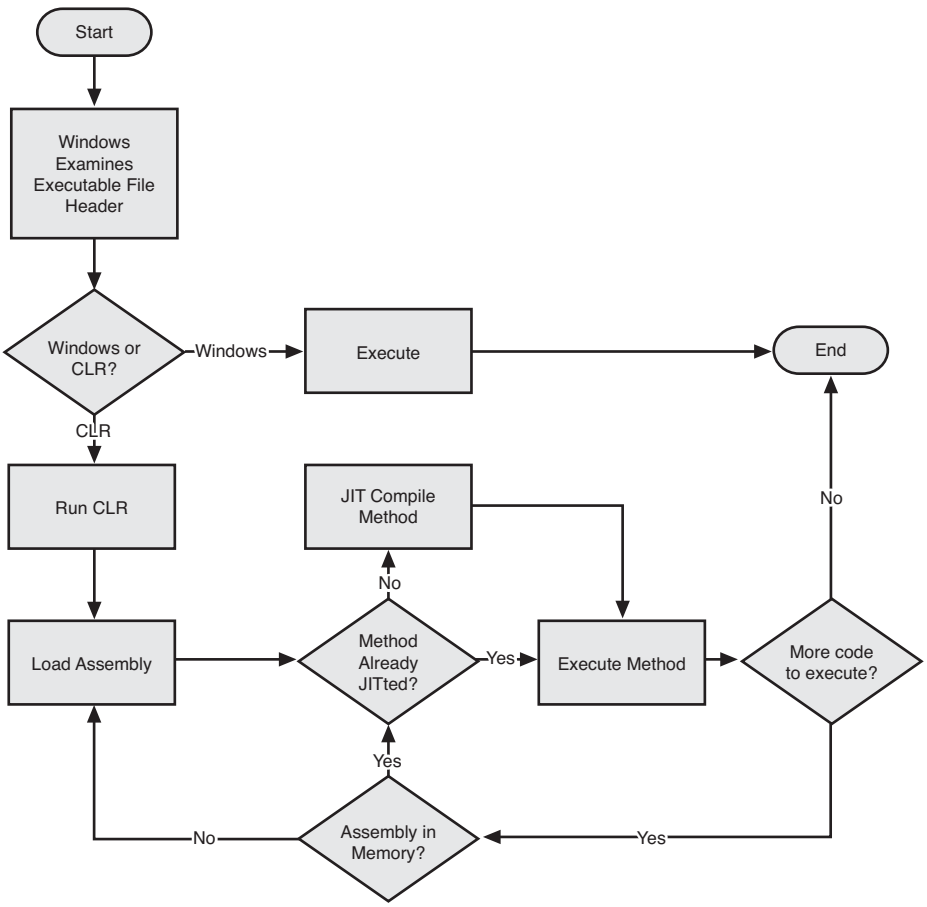

FIGURE 1.4 The CLR execution process (summarized).

As illustrated in Figure 1.4, Windows (the OS) will be running at Start; the CLR won't begin execution until Windows starts it. When an application executes, OS inspects the file to see whether it has a special header to indicate that it is a .NET application. If not, Windows continues to run the application.

If an application is for .NET, Windows starts up the CLR and passes the application to the CLR for execution. The CLR loads the executable assembly, finds the entry point, and begins its execution process.

The executable assembly could reference other assemblies, such as dynamic link libraries (DLLs), so the CLR will load those. However, this is on an as-needed basis. An assembly won't be loaded until the CLR needs access to the assembly's code. It's possible that the

code in some assemblies won't be executed, so there isn't a need to use resources unless absolutely necessary.

As mentioned previously, the C# compiler produces IL as part of an assembly's output. To execute the code, the CLR must translate the IL to binary code that the operating system understands. This is the responsibility of the JIT compiler.

As its name implies, the JIT compiler only compiles code before the first time that it executes. After the IL is compiled to machine code by the JIT compiler, the CLR holds the compiled code in a working set. The next time that the code must execute, the CLR checks its working set and runs the code directly if it is already compiled. It is possible that the working set could be paged out of memory during program execution, for various reasons that are necessary for efficient operation of the CLR on the particular machine it is running on. If more memory is available than the size of the working set, the CLR can hold on to the code. Additionally, in the case of Web applications where scalability is an issue, the working set can be swapped out due to periodic recycling or heavier load on the server, resulting in additional load time for subsequent requests.

The JIT compiler operates at the method level. If you aren't familiar with the term *method*, it is essentially the same as a function or procedure in other languages. Therefore, when the CLR begins execution, the JIT compiler compiles the entry point (the `Main` method in C#). Each subsequent method is JIT compiled just before execution. If a method being JIT compiled contains calls to methods in another assembly, the CLR loads that assembly (if not already loaded).

This process of checking the working set, JIT compilation, assembly loading, and execution continues until the program ends.

The meaning to you in the CLR execution process is in the form of application design and understanding performance characteristics. In the case of assembly loading, you have some control over when certain code is loaded. For example, if you have code that is seldom used or necessary only in specialized cases, you could separate it into its own DLL, which will keep the CLR from loading it when not in use. Similarly, separating seldomly executed logic into a separate method ensures the code doesn't JIT until it's called.

Another detail you might be concerned with is application performance. As described earlier, code is loaded and JIT compiled. Another DLL adds load time, which may or may not make a difference to you, but it is certainly something to be aware of. By the way, after code has been JIT compiled, it executes as fast as any other binary code in memory.

One of the CLR features listed in Table 1.1 is .NET Framework Class Library (FCL) support. The next section goes beyond FCL support for the CLR and gives an overview of what else the FCL includes.

The .NET Framework Class Library (FCL)

.NET has an extensive library, offering literally thousands of reusable types. Organized into namespaces, the FCL contains code supporting all the .NET technologies, such as Windows Forms, Windows Presentation Foundation, ASP.NET, ADO.NET, Windows

Workflow, and Windows Communication Foundation. In addition, the FCL has numerous cross-language technologies, including file I/O, networking, text management, and diagnostics. As mentioned earlier, the FCL has CLR support in the areas of built-in types, exception handling, security, and threading. Table 1.2 shows some common FCL libraries.

WHAT IS A TYPE?

Types are used to define the meaning of variables in your code. They could be built-in types such as int, double, or string. You can also have custom types such as `Customer`, `Employee`, or `BankAccount`. Each type has optional data/behavior associated with it.

Much of this book is dedicated to explaining the use of types, whether built-in, custom, or those belonging to the .NET FCL. Chapter 4, "Understanding Reference Types and Value Types," includes a more in-depth discussion on how C# supports the .NET type system.

TABLE 1.2 Common .NET Framework Class Library Namespaces

`System`	`System.Runtime`
`System.Collections`	`System.Security`
`System.Configuration`	`System.ServiceModel`
`System.Data`	`System.Text`
`System.Diagnostics`	`System.Threading`
`System.Drawing`	`System.Web`
`System.IO`	`System.Windows`
`System.Linq`	`System.Workflow.*`
`System.Net`	`System.Xml`

The namespaces in Table 1.2 are a sampling from the many available in the .NET Framework. They're representative of the types they contain. For example, you can find Windows Presentation Foundation (WPF) libraries in the `System.Windows` namespace, Windows Communication Foundation (WCF) is in the `System.ServiceModel` namespace, and Language Integrated Query (LINQ) types can be found in the `System.Linq` namespace.

Another aspect of Table 1.2 is that I included only two levels in the namespace hierarchy, `System.*`. In fact, there are multiple namespace levels, depending on which technology you view. For example, if you want to write code using the Windows Workflow (WF) runtime, you look in the `System.Workflow.Runtime` namespace. Generally, you can find the more common types at the higher namespace levels.

One of the benefits you should remember about the FCL is the amount of code reuse it offers. As you read through this book, you'll see many examples of how the FCL forms the basis for code you can write. For example, you learn how to create your own exception

object in Chapter 13, "Naming and Organizing Types with Namespaces," which requires that you use the Exception types from the FCL. Even if you encounter situations that don't require your use of FCL code, you can still use it. An example of when you would want to reuse FCL code is in Chapter 17, "Parameterizing Type with Generics and Writing Iterators," where you learn how to use existing generic collection classes. The FCL was built and intended for reuse, and you can often be much more productive by using FCL types rather than building your own from scratch.

Another important feature of the FCL is language neutrality. Just like the CLR, it doesn't matter which .NET language you program in—the FCL is reusable by all .NET programming languages, which are discussed in the next section.

C# and Other .NET Languages

.NET supports multiple programming languages, which are assisted by both the CLR and the FCL. Literally dozens of languages target the .NET CLR as a platform. Table 1.3 lists some of these languages.

TABLE 1.3 Languages Targeting the .NET CLR

A#	Fortran	Phalanger (PHP)
APL	IronPython	Python
C++	IronRuby	RPG
C#	J#	Silverfrost FTN95
COBOL	Jscript	Scheme
Component Pascal	LSharp	SmallScript
Delphi	Mercury	Smalltalk
Delta Forth	Mondrian	TMT Pascal
Eiffel.NET	Oberon	VB.NET
F#	Perl	Zonnon

Table 1.3 is not a comprehensive list because there are new languages being created for .NET on a regular basis. An assumption one could make from this growing list is that .NET is a successful multilanguage platform.

As you learned earlier in this chapter, the C# compiler emits IL. However, the C# compiler is not alone—all compilers for languages in Table 1.2 emit IL, too. By having a CLR that consumes IL, anyone can build a compiler that emits IL and join the .NET family of languages.

In the next section, you learn how the CLR supports multiple languages via a Common Type System (CTS), the relationship of the languages via a Common Language Specification (CLS), and how languages are supported via the FCL.

The Common Type System (CTS)

To support multiple programming languages on a single CLR and have the ability to reuse the FCL, the types of each programming language must be compatible. This binary compatibility between language types is called the Common Type System (CTS).

The built-in types are represented as types in the FCL. This means that a C# int is the same as a VB.NET Integer type and their .NET type is `System.Int32`, which is a 32-bit integer named `Int32` in the `System` namespace of the FCL. You'll learn more about C# types, type classification, and how C# types map to the CTS in Chapter 4.

The Common Language Specification (CLS)

Although the CLR understands all types in the CTS, each language targeting the CLR will not implement all types. Languages must often be true to their origins and will not lose their features or add new features that aren't compatible with how they are used.

However, one of the benefits of having a CLR with a CTS that understands IL, and an FCL that supports all languages, is the ability to write code in one language that is consumable by other languages. Imagine you are a third-party component vendor and your language of choice is C#. It would be desirable that programmers in any .NET language (for example, IronRuby or Delphi) would be able to purchase and use your components.

For programming languages to communicate effectively, targeting IL is not enough. There must be a common set of standards to which every .NET language must adhere. This common set of language features is called the Common Language Specification (CLS).

Most .NET compilers can produce both CLS-compliant and non-CLS-compliant code, and it is up to the developer to choose which language features to use. For example, C# supports unsigned types, which are non-CLS compliant. For CLS compliance, you can still use unsigned types within your code so long as you don't expose them in the public interface of your code, where code written in other languages can see.

Summary

.NET is composed of a CLR and the .NET FCL, and supports multiple languages. The CLR offers several features that free you from the low-level plumbing work required in other environments. The FCL is a large library of code that supports additional technologies such as Windows Presentation Foundation, Windows Communication Foundation, Windows Workflow, ASP.NET, and many more. The FCL also contains much code that you can reuse in your own applications. Through its support of IL, a CTS, and a CLS, many languages target the .NET platform. Therefore, you can write a reusable library with C# code that can be consumed by code written in other programming languages.

Remember that understanding the .NET platform, which includes CLR, FCL, and multiple-language support, has implications in the way you design and write your code. Throughout this book, you'll encounter many instances where the concepts in this chapter lay the foundation of the tasks you need to accomplish. You might want to refer back to this chapter for an occasional refresher.

This chapter has been purposefully as short as possible to cover only the platform issues most essential to building C# applications. If you're like me, you'll be eager to jump into some code. The next chapter does that by introducing you to essential syntax of the C# programming language.

Index

A

B

C

D

E

F

G

H

I

J–K

L

M

N

O

P

Q

R

S

T

U

V

W

X–Z