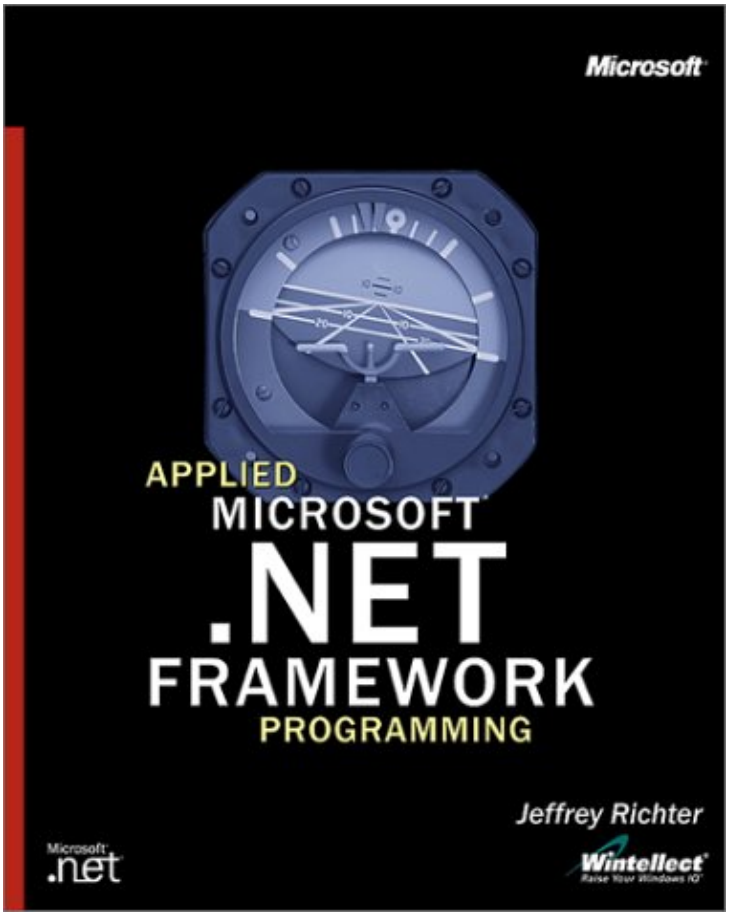



# Applied Microsoft .NET Framework Programming

# Table of Contents

**Applied Microsoft .NET Framework Programming....1**

**Introduction....4**
What Makes Up the Microsoft .NET Initiative....7
An Underlying Operating System: Windows....7
Helpful Products: The .NET Enterprise Servers....7
Microsoft XML Web Services: .NET My Services....8
The Development Platform: The .NET Framework....8
The Development Environment: Visual Studio .NET....11
Goal of This Book....12
System Requirements....13
This Book Has No Mistakes....13
Support....13

**Part I: Basics of the Microsoft .NET Framework....15**
Chapter List....15

**Chapter 1: The Architecture of the .NET Framework Development Platform....16**
Compiling Source Code into Managed Modules....16
Combining Managed Modules into Assemblies....18
Loading the Common Language Runtime....19
Executing Your Assembly's Code....22
IL and Verification....27
The .NET Framework Class Library....29
The Common Type System....31
The Common Language Specification....33
Interoperability with Unmanaged Code....37

**Chapter 2: Building, Packaging, Deploying, and Administering Applications and Types....40**
Overview....40
.NET Framework Deployment Goals....40
Building Types into a Module....41
Combining Modules to Form an Assembly....47
Adding Assemblies to a Project Using the Visual Studio .NET IDE....52
Using the Assembly Linker....53
Including Resource Files in the Assembly....55
Assembly Version Resource Information....55
Version Numbers....59
Culture....60
Simple Application Deployment (Privately Deployed Assemblies)....61
Simple Administrative Control (Configuration)....62

**Chapter 3: Shared Assemblies....66**
Overview....66
Two Kinds of Assemblies, Two Kinds of Deployment....67
Giving an Assembly a Strong Name....67
The Global Assembly Cache....71
The Internal Structure of the GAC....76
Building an Assembly That References a Strongly Named Assembly....78
Strongly Named Assemblies Are Tamper-Resistant....80

# Table of Contents

**Chapter 3: Shared Assemblies**
Delayed Signing....81
Privately Deploying Strongly Named Assemblies....84
Side–by–Side Execution....85
How the Runtime Resolves Type References....86
Advanced Administrative Control (Configuration)....88
Publisher Policy Control....93
Repairing a Faulty Application....95

**Part II: Working with Types and the Common Language Runtime....99**
Chapter List....99

**Chapter 4: Type Fundamentals....100**
All Types Are Derived from System.Object....100
Casting Between Types....101
Casting with the C# is and as Operators....103
Namespaces and Assemblies....105

**Chapter 5: Primitive, Reference, and Value Types....109**
Programming Language Primitive Types....109
Checked and Unchecked Primitive Type Operations....112
Reference Types and Values Types....114
Boxing and Unboxing Value Types....118

**Chapter 6: Common Object Operations....128**
Object Equality and Identity....128
Implementing Equals for a Reference Type Whose Base Classes Don't Override Object's Equals....129
Implementing Equals for a Reference Type When One or More of Its Base Classes Overrides Object's Equals....130
Implementing Equals for a Value Type....131
Summary of Implementing Equals and the ==/!= Operators....133
Identity....134
Object Hash Codes....134
Object Cloning....136

**Part III: Designing Types....139**
Chapter List....139

**Chapter 7: Type Members and Their Accessibility....140**
Type Members....140
Accessibility Modifiers and Predefined Attributes....142
Type Predefined Attributes....144
Field Predefined Attributes....144
Method Predefined Attributes....145

**Chapter 8: Constants and Fields....147**
Constants....147
Fields....148

# Table of Contents

**Chapter 9: Methods ..... 150**
Instance Constructors ..... 150
Type Constructors ..... 155
Operator Overload Methods ..... 157
Operators and Programming Language Interoperability ..... 159
Conversion Operator Methods ..... 161
Passing Parameters by Reference to a Method ..... 164
Passing a Variable Number of Parameters to a Method ..... 168
How Virtual Methods Are Called ..... 170
Virtual Method Versioning ..... 171

**Chapter 10: Properties ..... 176**
Parameterless Properties ..... 176
Parameterful Properties ..... 179

**Chapter 11: Events ..... 184**
Overview ..... 184
Designing a Type That Exposes an Event ..... 185
Designing a Type That Listens for an Event ..... 189
Explicitly Controlling Event Registration ..... 191
Designing a Type That Defines Lots of Events ..... 192
Designing the EventHandlerSet Type ..... 196

**Part IV: Essential Types ..... 199**
Chapter List ..... 199

**Chapter 12: Working with Text ..... 200**
Characters ..... 200
The System.String Type ..... 202
Constructing Strings ..... 202
Strings Are Immutable ..... 204
Comparing Strings ..... 205
String Interning ..... 210
String Pooling ..... 213
Examining a String's Characters ..... 213
Other String Operations ..... 216
Dynamically Constructing a String Efficiently ..... 217
Constructing a StringBuilder Object ..... 217
StringBuilder's Members ..... 218
Obtaining a String Representation for an Object ..... 220
Specific Formats and Cultures ..... 221
Formatting Multiple Objects into a Single String ..... 224
Providing Your Own Custom Formatter ..... 226
Parsing a String to Obtain an Object ..... 228
Encodings: Converting Between Characters and Bytes ..... 232
Encoding/Decoding Streams of Characters and Bytes ..... 238
Base-64 String Encoding and Decoding ..... 239

# Table of Contents

**Chapter 13: Enumerated Types and Bit Flags....240**
Enumerated Types....240
Bit Flags....244

**Chapter 14: Arrays....247**
Overview....247
All Arrays Are Implicitly Derived from System.Array....249
Casting Arrays....251
Passing and Returning Arrays....252
Creating Arrays That Have a Nonzero Lower Bound....253
Fast Array Access....254
Redimensioning an Array....257

**Chapter 15: Interfaces....259**
Interfaces and Inheritance....259
Designing an Application That Supports Plug-In Components....263
Changing Fields in a Boxed Value Type Using Interfaces....264
Implementing Multiple Interfaces That Have the Same Method....266
Explicit Interface Member Implementations....268

**Chapter 16: Custom Attributes....273**
Using Custom Attributes....273
Defining Your Own Attribute....276
Attribute Constructor and Field/Property Data Types....278
Detecting the Use of a Custom Attribute....279
Matching Two Attribute Instances Against Each Other....283
Pseudo-Custom Attributes....285

**Chapter 17: Delegates....287**
A First Look at Delegates....287
Using Delegates to Call Back Static Methods....289
Using Delegates to Call Back Instance Methods....290
Demystifying Delegates....291
Some Delegate History: System.Delegate and System.MulticastDelegate....294
Comparing Delegates for Equality....296
Delegate Chains....296
C#'s Support for Delegate Chains....300
Having More Control over Invoking a Delegate Chain....301
Delegates and Reflection....303

**Part V: Managing Types....306**
Chapter List....306

**Chapter 18: Exceptions....307**
Overview....307
The Evolution of Exception Handling....307
The Mechanics of Exception Handling....309
The try Block....310
The catch Block....310
The finally Block....312

# Table of Contents

**Chapter 18: Exceptions**

What Exactly Is an Exception? ... 312
The System.Exception Class ... 316
FCL–Defined Exception Classes ... 317
Defining Your Own Exception Class ... 319
How to Use Exceptions Properly ... 322
You Can't Have Too Many finally Blocks ... 323
Don't Catch Everything ... 324
Gracefully Recovering from an Exception ... 325
Backing Out of a Partially Completed Operation When an Unrecoverable Exception Occurs ... 326
Hiding an Implementation Detail ... 327
What's Wrong with the FCL ... 329
Performance Considerations ... 330
Catch Filters ... 333
Unhandled Exceptions ... 335
Controlling What the CLR Does When an Unhandled Exception Occurs ... 339
Unhandled Exceptions and Windows Forms ... 340
Unhandled Exceptions and ASP.NET Web Forms ... 342
Unhandled Exceptions and ASP.NET XML Web Services ... 342
Exception Stack Traces ... 342
Remoting Stack Traces ... 344
Debugging Exceptions ... 345
Telling Visual Studio What Kind of Code to Debug ... 349

**Chapter 19: Automatic Memory Management (Garbage Collection) ... 351**

Understanding the Basics of Working in a Garbage–Collected Platform ... 351
The Garbage Collection Algorithm ... 354
Finalization ... 357
What Causes Finalize Methods to Get Called ... 362
Finalization Internals ... 363
The Dispose Pattern: Forcing an Object to Clean Up ... 365
Using a Type That Implements the Dispose Pattern ... 370
C#'s using Statement ... 373
An Interesting Dependency Issue ... 374
Weak References ... 375
Weak Reference Internals ... 377
Resurrection ... 378
Designing an Object Pool Using Resurrection ... 379
Generations ... 381
Programmatic Control of the Garbage Collector ... 385
Other Garbage Collector Performance Issues ... 387
Synchronization–Free Allocations ... 388
Scalable Parallel Collections ... 388
Concurrent Collections ... 389
Large Objects ... 390
Monitoring Garbage Collections ... 391

# Table of Contents

**Chapter 20: CLR Hosting, AppDomains, and Reflection....392**
Metadata: The Cornerstone of the .NET Framework....392
CLR Hosting....393
AppDomains....394
Accessing Objects Across AppDomain Boundaries....396
AppDomain Events....397
Applications and How They Host the CLR and Manage AppDomains....398
"Yukon"....399
The Gist of Reflection....400
Reflecting Over an Assembly's Types....401
Reflecting Over an AppDomain's Assemblies....403
Reflecting Over a Type's Members: Binding....404
Explicitly Loading Assemblies....405
Loading Assemblies as "Data Files"....407
Building a Hierarchy of Exception–Derived Types....408
Explicitly Unloading Assemblies: Unloading an AppDomain....410
Obtaining a Reference to a System.Type Object....412
Reflecting Over a Type's Members....415
Creating an Instance of a Type....417
Calling a Type's Method....418
Bind Once, Invoke Multiple Times....422
Reflecting Over a Type's Interfaces....426
Reflection Performance....428

**List of Figures....429**

**List of Tables....432**

# Applied Microsoft .NET Framework Programming

**Jeffrey Richter**

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

Copyright © 2002 by Jeffrey Richter

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging−in−Publication Data
Richter, Jeffrey.
Applied Microsoft .NET Framework Programming / Jeffrey Richter.
p. cm.
Includes index.
ISBN 0−7356−1422−9
1. Microsoft .NET Framework. 2. Internet programming. I. Title.
QA76.625 .R53 2002
005.2'76—dc21 2001056250

Printed and bound in the United States of America.

3 4 5 6 7 8 9 QWT 7 6 5 4 3 2

Distributed in Canada by Penguin Books Canada Limited.

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 *mspinput@microsoft.com*.

Active Directory, ActiveX, Authenticode, DirectX, IntelliSense, JScript, Microsoft, Microsoft Press, MSDN, the .NET logo, PowerPoint, Visual Basic, Visual C++, Visual Studio, Win32, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e−mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e−mail address, logo, person, place, or event is intended or should be inferred.

**Acquisitions Editor:** Anne Hamilton
**Project Editor:** Sally Stickney

***To Kristin***

*I want to tell you how much you mean to me.*

*Your energy and exuberance always lift me higher.*

*Your smile brightens my every day.*

*Your zest makes my heart sing.*

*I love you.*

**Jeffrey Richter**

Jeffrey Richter is a co–founder of Wintellect (*http://www.Wintellect.com/*), a training, design, and debugging company dedicated to helping companies produce better software faster. Jeff has written many books, including *Programming Applications for Microsoft Windows* (Microsoft Press, 1999) and *Programming Server–Side Applications for Microsoft Windows 2000* (Microsoft Press, 2000). Jeff is also a contributing editor for *MSDN Magazine*, where he has written several feature articles and is the .NET columnist. Jeff also speaks at various trade conferences worldwide, including VSLive!, WinSummit, and Microsoft's TechEd and PDC.

Jeff has consulted for many companies, including AT&T, DreamWorks, General Electric, Hewlett–Packard, IBM, and Intel. Jeff's code has shipped in many Microsoft products, among them Visual Studio, Microsoft Golf, Windows Sound System, and various versions of Windows, from Windows 95 to Windows XP and the Windows .NET Server Family. Since October 1999, Jeff has consulted with the .NET Framework team and has used the .NET Framework to produce the XML Web service front end to Microsoft's very popular TerraServer Web property (*http://www.TerraServer.net/*).

On the personal front, Jeff holds both airplane and helicopter pilot licenses, though he never gets to fly as often as he'd like. He is also a member of the International Brotherhood of Magicians and enjoys showing friends slight–of–hand card tricks from time to time. Jeff's other hobbies include music, drumming, and model railroading. He also enjoys traveling and the theater. He lives near Bellevue, Washington, with his wife, Kristin, and their cat, Max. He doesn't have any children yet, but he has the feeling that kids may be a part of his life soon.

**Acknowledgments**

I couldn't have written this book without the help and technical assistance of many people. In particular, I'd like to thank the following people:

- **Members of the Microsoft Press editorial team**: Sally Stickney, project editor and manuscript editor; Devon Musgrave, manuscript editor; Jim Fuchs, technical editing consultant; Carl Diltz and Katherine Erickson, compositors; Joel Panchot, artist; and Holly M. Viola, copy editor.
- **Members of the Microsoft .NET Framework team**: Fred Aaron, Brad Abrams, Mark Anders, Chris Anderson, Dennis Angeline, Keith Ballinger, Sanjay Bhansali, Mark Boulter, Christopher Brown, Chris Brumme, Kathleen Carey, Ian Carmichael, Rajesh Chandrashekaran, Yann Christensen, Suzanne Cook, Krzysztof Cwalina, Shajan Dasan,

Peter de Jong, Blair Dillaway, Patrick Dussud, Erick Ellis Bill Evans, Michael Fanning, Greg Fee, Kit George, Peter Golde, Will Greg, Bret Grinslade, Brian Grunkemeyer, Eric Gunnerson, Simon Hall, Jennifer Hamilton, Brian Harry, Michael Harsh, Jonathan Hawkins, Anders Hejlsberg, Jim Hogg, Paul Johns, Gopal Kakivaya, Sonja Keserovic, Abhi Khune, Loren Kornfelder, Nikhil Kothari, Tim Kurtzman, Brian LaMacchia, Sebastian Lange, Serge Lidin, Francois Liger, Yung–Shin "Bala" Lin, Mike Magruder, Rudi Martin, Erik Meijer, Gene Milener, Jim Miller, Anthony Moore, Vance Morrison, David Mortenson, Yuval Neeman, Lance Olson, Srivatsan Parthasarathy, Mahesh Prakriya, Steven Pratchner, Susan Radke–Sproul, Jayanth Rajan, Dmitry Robsman, Jay Roxe, Dario Russi, Craig Schertz, Alan Shi, Craig Sinclair, Greg Singleton, Ralph Squillace, Paul Stafford, Larry Sullivan, Dan Takacs, Ryley Taketa, David Treadwell, Sean Trowbridge, Nate Walker, Sara Williams, Jason Zander, and Eric Zinda. If I've forgotten anyone, please forgive me.

- **Reviewers**: Keith Ballinger, Tom Barclay, Lars Bergstrom, Stephen Butler, Jeffrey Cooperstein, Robert Corstanje, Tarek Dawoud, Sylvain Dechatre, Ash Dhanesha, Shawn Elliott, Chris Falter; Lakshan Fernando, Manish Godse, Eric Gunnerson, Brian Harry, Chris Hockett, Dekel Israeli, Paul Johns, Jeanine Johnson, Jim Kieley, Alex Lerner, Richard Loba, Kerry Loynd, Rob Macdonald, Darrin Massena, John Noss, Piet Obermeyer, Peter Plamondon, Keith Pleas, Mahesh Prakriya, Doug Purdy, Kent Sharkey, Alan Shi, Dan Vallejo, Scott Wadsworth, Beth Wood, and Steven Wort.
- **Wintellectuals**: Jim Bail, Francesco Balena, Doug Boling, Jason Clark, Paula Daniels, Dino Esposito, Lewis Frazer, John Lam, Jeff Prosise, John Robbins, Kenn Scribner, and Chris Shelby.

# Introduction

Over the years, our computing lifestyles have changed. Today, everyone sees the value of the Internet, and our computing lifestyle is becoming more and more dependent on Web–based services. Personally, I love to shop, get traffic conditions, compare products, buy tickets, and read product reviews all via the Internet.

However, I'm finding that there are still many things I'd like to do using the Internet that aren't possible today. For example, I'd like to find restaurants in my area that serve a particular cuisine. Furthermore, I'd like to be able to ask if the restaurant has any seating for, say, 7:00 p.m. that night. Or if I had my own business, I might like to know which vendor has a particular item in stock. If multiple vendors can supply me with the item, I'd like to be able to find out which vendor offers the least expensive price for the item or maybe which vendor can deliver the item to me the fastest.

Services like these don't exist today for two main reasons. The first reason is that no standards are in place for integrating all this information. After all, vendors today each have their own way of describing what they sell. The emerging standard for describing all types of information is Extensible Markup Language (XML). The second reason these services don't exist today is the complexity of developing the code necessary to integrate such services.

Microsoft has a vision in which selling services is the way of the future—that is, companies will offer services and interested users can consume these services. Many services will be free; others will be available through a subscription plan, and still others will be charged per use. You can think of these services as the execution of some business logic. Here are some examples of services:

- Validating a credit card purchase
- Getting directions from point A to point B
- Viewing a restaurant's menu
- Booking a flight on an airline, a hotel room, or a rental car
- Updating photos in an online photo album
- Merging your calendar and your children's calendars to plan a family vacation
- Paying a bill from a checking account
- Tracking a package being shipped to you

I could go on and on with ideas for services that any company could implement. Without a doubt, Microsoft will build some of these services and offer them in the near future. Other companies (like yours) will also produce services, some of which might compete with Microsoft in a free market.

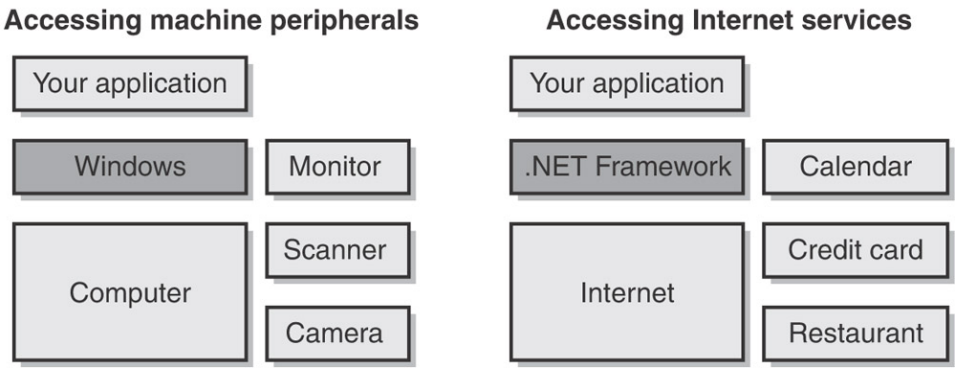
So how do we get from where we are today to a world in which all these services are easily available? And how do we produce applications—HTML–based or otherwise—that use and combine these services to produce rich features for the user? For example, if restaurants offered the service of retrieving their menu, an application could be written to query every restaurant's menu, search for a specific cuisine or dish, and then present only those restaurants in the user's own neighborhood in the application.

**Note** To create rich applications like these, businesses must offer a programmatic interface to their business logic services. This programmatic interface must be callable remotely using a network, like the Internet. This is what the Microsoft .NET initiative is all about. Simply stated, the .NET initiative is all about connecting information, people, and devices.

Let me explain it this way: Computers have peripherals—mouse, monitor, keyboard, digital cameras, and scanners—connected to them. An operating system, such as Microsoft Windows,

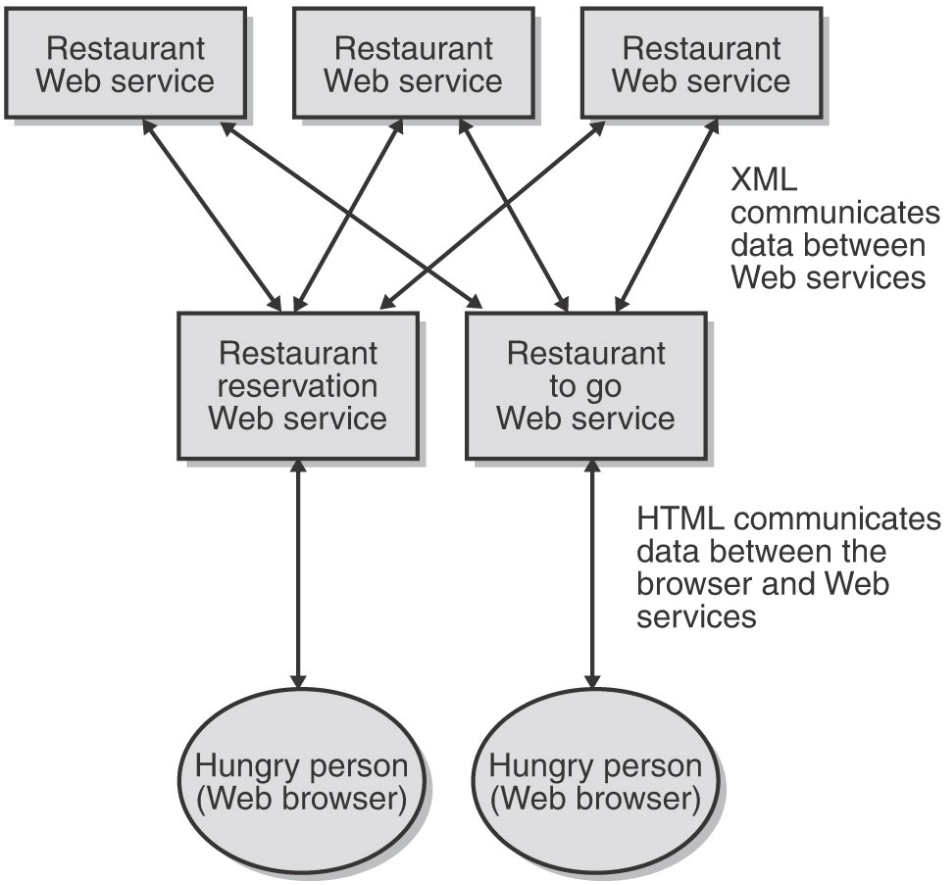
provides a development platform that abstracts the application's access to these peripherals. You can even think of these peripherals as services, in a way.

In this new world, the services (or peripherals) are now connected to the Internet. Developers want an easy way to access these services. Part of the Microsoft .NET initiative is to provide this development platform. The following diagram shows an analogy. On the left, Windows is the development platform that abstracts the hardware peripheral differences from the application developer. On the right, the Microsoft .NET Framework is the development platform that abstracts the XML Web service communication from the application developer.




Although a leader in the development and definition of the standards involved in making this new world possible, Microsoft doesn't own any of the standards. Client machines describe a server request by creating specially formatted XML and then sending it (typically using HTTP) over an intranet or the Internet. Servers know how to parse the XML data, process the client's request, and return the response as XML back to the client. *Simple Object Access Protocol* (SOAP) is the term used to describe the specially formatted XML when it is sent using HTTP.

The following figure shows a bunch of XML Web services all communicating with one another using SOAP with its XML payload. The figure also shows clients running applications that can talk to Web services and even other clients via SOAP (XML). In addition, the figure shows a client getting its results via HTML from a Web server. Here the user probably filled out a Web form, which was sent back to the Web server. The Web server processed the user's request (which involved communicating with some Web services), and the results are ultimately sent back to the user via a standard HTML page.




In addition, the computers providing the services must be running an operating system that is listening for these SOAP requests. Microsoft hopes that this operating system will be Windows, but Windows isn't a requirement. Any operating system that can listen on a TCP/IP socket port and read/write bytes to the port is good enough. In the not too distant future, mobile phones, pagers, automobiles, microwave ovens, refrigerators, watches, stereo equipment, game consoles, and all kinds of other devices will also be able to participate in this new world.

On the client or application side, an operating system must be running that can read/write to a socket port to issue service requests. The client's computer must also be capable of supporting whatever features the user's application desires. If the user's application wants to create a window or a menu, the operating system must provide this functionality or the application developer must implement it manually. Of course, Microsoft hopes that people will write applications that take advantage of the rich feature set in Windows, but again, Windows is a recommendation, not a necessity.

What I'm trying to say is that this new world will happen whether Microsoft is a part of it or not. Microsoft's .NET initiative is all about making it really easy for developers to create and access these services.

Today, we could all go write our own operating system and create our own custom Web servers to listen and manually process SOAP requests if we wanted to, but it's really hard and would take a long time. Microsoft has taken on all this hard work for us, and we can just leverage Microsoft's efforts to greatly simplify our own development efforts. Now we, as application developers, can concentrate and focus on our business logic and services, leaving all the communication protocols and plumbing to Microsoft (who has a lot of developers that just love to do this nitty–gritty stuff).

# What Makes Up the Microsoft .NET Initiative

I've been working with Microsoft and its technologies for many years now. Over the years, I've seen Microsoft introduce all kinds of new technologies and initiatives: MS–DOS, Windows, Windows CE, OLE, COM, ActiveX, COM+, Windows DNA, and so on. When I first started hearing about Microsoft's .NET initiative, I was surprised at how solid Microsoft's story seemed to be. It really seemed to me that they had a vision and a plan and that they had rallied the troops to implement the plan.

I contrast Microsoft's .NET platform to ActiveX, which was just a new name given to good old COM to make it seem more user friendly. ActiveX didn't mean much (or so many developers thought), and the term, along with ActiveX controls, never really took off. I also contrast Microsoft's .NET initiative to Windows DNA (Distributed InterNet Architecture), which was another marketing label that Microsoft tacked onto a bunch of already existing technologies. But I really believe in the Microsoft .NET initiative, and to prove it, I've written this book. So, what exactly constitutes the Microsoft .NET initiative? Well, there are several parts to it, and I'll describe each one in the following sections.

## An Underlying Operating System: Windows

Because these Web services and applications that use Web services run on computers and because computers have peripherals, we still need an operating system. Microsoft suggests that people use Windows. Specifically, Microsoft is adding XML Web service–specific features to its Windows line of operating systems, and Windows XP and the servers in the Windows .NET Server Family will be the versions best suited for this new service–driven world.

Specifically, Windows XP and the Windows .NET Server Family products have integrated support for Microsoft .NET Passport XML Web service. Passport is a service that authenticates users. Many Web services will require user authentication to access information securely. When users log on to a computer running Windows XP or one of the servers from the Windows .NET Server Family, they are effectively logging on to every Web site and Web service that uses Passport for authentication. This means that users won't have to enter usernames and passwords as they access different Internet sites. As you can imagine, Passport is a huge benefit to users: one identity and password for everything you do, and you have to enter it only once!

In addition, Windows XP and the Windows .NET Server Family products have some built–in support for loading and executing applications implementing the .NET Framework. Finally, Windows XP and the Windows .NET Server Family operating systems have a new, extensible instant messaging notification application. This application allows third–party vendors (such as Expedia, the United States Postal Service, and many others) to communicate with users seamlessly. For example, users can receive automatic notifications when their flights are delayed (from Expedia) and when a package is ready to be delivered (from the U.S. Postal Service).

I don't know about you, but I've been hoping for services like these for years—I can't wait!

## Helpful Products: The .NET Enterprise Servers

As part of the .NET initiative, Microsoft is providing several products that companies can choose to use if their business logic (services) find them useful. Here are some of Microsoft's enterprise server products:

- Microsoft Application Center 2000

- Microsoft BizTalk Server 2000
- Microsoft Commerce Server 2000
- Microsoft Exchange 2000
- Microsoft Host Integration Server 2000
- Microsoft Internet Security and Acceleration (ISA) Server 2000
- Microsoft Mobile Information Server 2002
- Microsoft SQL Server 2000

It's likely that each of these products will eventually have a ".NET" added to its name for marketing purposes. But I'm also sure that over time, these products will integrate more .NET features into them as Microsoft continues the initiative.

## Microsoft XML Web Services: .NET My Services

Certainly, Microsoft wants to do more than just provide the underlying technologies that allow others to play in this new world. Microsoft wants to play too. So, Microsoft will be building its own set of XML Web services: some will be free, and others will require some usage fee. Microsoft initially plans to offer the following .NET My Services:

- .NET Alerts
- .NET ApplicationSettings
- .NET Calendar
- .NET Categories
- .NET Contacts
- .NET Devices
- .NET Documents
- .NET FavoriteWebSites
- .NET Inbox
- .NET Lists
- .NET Locations
- .NET Presence
- .NET Profile
- .NET Services
- .NET Wallet

These consumer–oriented XML Web services are known as Microsoft's ".NET My Services." You can find out more information about them at *http://www.Microsoft.com/MyServices/*. Over time, Microsoft will add many more consumer services and will also be creating business–oriented XML Web services.

In addition to these public Web services, Microsoft will create internal services for sales data and billing. These internal services will be accessible to Microsoft employees only. I anticipate that companies will quickly embrace the idea of using Web services on their intranets to make internal company information available to employees. The implementation of publicly available Internet Web services and applications that consume them will probably proceed more slowly.

## The Development Platform: The .NET Framework

Some of the Microsoft .NET My Services (like Passport) exist today. These services run on Windows and are built using technologies such as C/C++, ATL, Win32, COM, and so on. As time goes on, these services and new services will ultimately be implemented using newer technologies, such as C# (pronounced "C sharp") and the .NET Framework.

**Important** Even though this entire introduction has been geared toward building Internet applications and Web services, the .NET Framework is capable of a lot more. All in all, the .NET Framework development platform allows developers to build the following kinds of applications: XML Web services, Web Forms, Win32 GUI applications, Win32 CUI (console UI) applications, services (controlled by the Service Control Manager), utilities, and stand–alone components. The material presented in this book is applicable to any and all of these application types.

The .NET Framework consists of two parts: the common language runtime (CLR) and the Framework Class Library (FCL). The .NET Framework is the part of the initiative that makes developing services and applications really easy. And, most important, this is what this book is all about: developing applications and XML Web services for the .NET Framework.

Initially, Microsoft will make the CLR and FCL available in the various versions of Windows, including Windows 98, Windows 98 Second Edition, and Windows Me as well as Windows NT 4, Windows 2000, and both 32–bit and 64–bit versions of Windows XP and the Windows .NET Server Family. A "lite" version of the .NET Framework, called the.NET Compact Framework, is also available for PDAs (such as Windows CE and Palm) and appliances (small devices). On December 13, 2001, the European Computer Manufacturers Association (ECMA) accepted the C# programming language, portions of the CLR, and portions of the FCL as standards. It won't be long before ECMA–compliant versions of these technologies appear on a wide variety of operating systems and CPUs.

**Note** Windows XP (both Home Edition and Professional) doesn't ship with the .NET Framework "in the box." However, the Windows .NET Server Family (Windows .NET Web Server, Windows .NET Standard Server, Windows .NET Enterprise Server, and Windows .NET Datacenter Server) will include the .NET Framework. In fact, this is how the Windows .NET Server Family got its name. The next version of Windows (code–named "Longhorn") will include the .NET Framework in all editions. For now, you'll have to redistribute the .NET Framework with your application, and your setup program will have to install it. Microsoft does make a .NET Framework redistribution file that you're allowed to freely distribute with your application: *http://go.microsoft.com/fwlink/?LinkId=5584*.

Almost all programmers are familiar with runtimes and class libraries. I'm sure many of you have at least dabbled with the C–runtime library, the standard template library (STL), the Microsoft Foundation Class library (MFC), the Active Template Library (ATL), the Visual Basic runtime library, or the Java virtual machine. In fact, the Windows operating system itself can be thought of as a runtime engine and library. Runtime engines and libraries offer services to applications, and we programmers love them because they save us from reinventing the same algorithms over and over again.

The Microsoft .NET Framework allows developers to leverage technologies more than any earlier Microsoft development platform did. Specifically, the .NET Framework really delivers on code reuse, code specialization, resource management, multilanguage development, security, deployment, and administration. While designing this new platform, Microsoft also felt it was necessary to improve on some of the deficiencies of the current Windows platform. The following list gives you just a small sampling of what the CLR and the FCL provide:

- **Consistent programming model** Unlike today, where some operating system facilities are accessed via dynamic–link library (DLL) functions and other facilities are accessed via COM objects, all application services are offered via a common object–oriented programming model.

- **Simplified programming model** The CLR seeks to greatly simplify the plumbing and arcane constructs required by Win32 and COM. Specifically, the CLR now frees the developer from having to understand any of the following concepts: the registry, globally unique identifiers (GUIDs), **IUnknown**, **AddRef**, **Release**, **HRESULT**s, and so on. The CLR doesn't just abstract these concepts away from the developer; these concepts simply don't exist, in any form, in the CLR. Of course, if you want to write a .NET Framework application that interoperates with existing, non–.NET code, you must still be aware of these concepts.
- **Run once, run always** All Windows developers are familiar with "DLL hell" versioning problems. This situation occurs when components being installed for a new application overwrite components of an old application, causing the old application to exhibit strange behavior or stop functioning altogether. The architecture of the .NET Framework now isolates application components so that an application always loads the components that it was built and tested with. If the application runs after installation, then the application should always run. This slams shut the gates of "DLL hell."
- **Simplified deployment** Today, Windows applications are incredibly difficult to set up and deploy. Several files, registry settings, and shortcuts usually need to be created. In addition, completely uninstalling an application is nearly impossible. With Windows 2000, Microsoft introduced a new installation engine that helps with all these issues, but it's still possible that a company authoring a Microsoft installer package might fail to do everything correctly. The .NET Framework seeks to banish these issues into history. The .NET Framework components (known simply as *types*) are not referenced by the registry. In fact, installing most .NET Framework applications requires no more than copying the files to a directory and adding a shortcut to the Start menu, desktop, or Quick Launch bar. Uninstalling the application is as simple as deleting the files.
- **Wide platform reach** When compiling source code for the .NET Framework, the compilers produce common intermediate language (CIL) instead of the more traditional CPU instructions. At run time, the CLR translates the CIL into native CPU instructions. Because the translation to native CPU instructions is done at run time, the translation is done for the host CPU. This means that you can deploy your .NET Framework application on any machine that has an ECMA–compliant version of the CLR and FCL running on it. These machines can be x86, IA64, Alpha, PowerPC, and so on. Users will immediately appreciate the value of this broad execution if they ever change their computing hardware or operating system.
- **Programming language integration** COM allows different programming languages to *interoperate* with one another. The .NET Framework allows languages to be *integrated* with one another so that you can use types of another language as if they are your own. For example, the CLR makes it possible to create a class in C++ that derives from a class implemented in Visual Basic. The CLR allows this because it defines and provides a Common Type System (CTS) that all programming languages that target the CLR must use. The Common Language Specification (CLS) describes what compiler implementers must do in order for their languages to integrate well with other languages. Microsoft is itself providing several compilers that produce code targeting the runtime: C++ with Managed Extensions, C#, Visual Basic .NET (which now subsumes Visual Basic Scripting Edition, or VBScript, and Visual Basic for Applications, or VBA), and JScript. In addition, companies other than Microsoft and academic institutions are producing compilers for other languages that also target the CLR.
- **Simplified code reuse** Using the mechanisms described earlier, you can create your own classes that offer services to third–party applications. This makes it extremely simple to reuse code and also creates a large market for component (type) vendors.
- **Automatic memory and management (garbage collection)** Programming requires great skill and discipline, especially when it comes to managing the use of resources such as files, memory, screen space, network connections, database resources, and so on. One of the

most common bugs is neglecting to free one of these resources, ultimately causing the application to perform improperly at some unpredictable time. The CLR automatically tracks resource usage, guaranteeing that your application never leaks resources. In fact, there is no way to explicitly "free" memory. In Chapter 19, "Automatic Memory Management (Garbage Collection)," I explain exactly how garbage collection works.

- **Type-safe verification** The CLR can verify that all your code is type-safe. Type safety ensures that allocated objects are always accessed in compatible ways. Hence, if a method input parameter is declared as accepting a 4-byte value, the CLR will detect and trap attempts to access the parameter as an 8-byte value. Similarly, if an object occupies 10 bytes in memory, the application can't coerce the object into a form that will allow more than 10 bytes to be read. Type safety also means that execution flow will transfer only to well-known locations (that is, method entry points). There is no way to construct an arbitrary reference to a memory location and cause code at that location to start executing. Together, these measures ensure type safety eliminating many common programming errors and classic system attacks (for example, exploiting buffer overruns).
- **Rich debugging support** Because the CLR is used for many programming languages, it is now much easier to implement portions of your application using the language best suited to a particular task. The CLR fully supports debugging applications that cross language boundaries.
- **Consistent method failure paradigm** One of the most aggravating aspects of Windows programming is the inconsistent style that functions use to report failures. Some functions return Win32 status codes, some functions return **HRESULT**s, and some functions throw exceptions. In the CLR, all failures are reported via exceptions—period. Exceptions allow the developer to isolate the failure recovery code from the code required to get the work done. This separation greatly simplifies writing, reading, and maintaining code. In addition, exceptions work across module and programming language boundaries. And, unlike status codes and **HRESULT**s, exceptions can't be ignored. The CLR also provides built-in stack-walking facilities, making it much easier to locate any bugs and failures.
- **Security** Traditional operating system security provides isolation and access control based on user accounts. This model has proven useful, but at its core assumes that all code is equally trustworthy. This assumption was justified when all code was installed from physical media (for example, CD-ROM) or trusted corporate servers. But with the increasing reliance on mobile code such as Web scripts, applications downloaded over the Internet, and e-mail attachments, we need ways to control the behavior of applications in a more code-centric manner. Code access security provides a means to do this.
- **Interoperability** Microsoft realizes that developers already have an enormous amount of existing code and components. Rewriting all this code to take full advantage of the .NET Framework platform would be a huge undertaking and would prevent the speedy adoption of this platform. So the .NET Framework fully supports the ability for the developer to access their existing COM components as well as call Win32 functions in existing DLLs.

Users won't directly appreciate the CLR and its capabilities, but they will certainly notice the quality and features of applications that utilize the CLR. In addition, users and your company's bottom line will appreciate how the CLR allows applications to be developed and deployed more rapidly and with less administration than Windows has ever allowed in the past.

## The Development Environment: Visual Studio .NET

The last part of the .NET initiative that I want to mention is Visual Studio .NET. Visual Studio .NET is Microsoft's development environment. Microsoft has been working on it for many years and has incorporated a lot of .NET FrameworkÐspecific features into it. Visual Studio .NET runs on Windows NT 4, Windows 2000, Windows XP, and the Windows .NET Server Family servers, and it will run on

future versions of Windows. Of course, the code produced by Visual Studio .NET will run on all these Windows platforms plus Windows 98, Windows 98 Second Edition, and Windows Me.

Like any good development environment, Visual Studio .NET includes a project manager; a source code editor; UI designers; lots of wizards, compilers, linkers, tools, and utilities; documentation; and debuggers. It supports building applications for both the 32–bit and 64–bit Windows platforms as well as for the new .NET Framework platform. Another important improvement is that there is now just one integrated development environment for all programming languages.

Microsoft also provides a .NET Framework SDK. This free SDK includes all the language compilers, a bunch of tools, and a lot of documentation. Using this SDK, you can develop applications for the .NET Framework without using Visual Studio .NET. You'll just have to use your own editor and project management system. You also don't get drag–and–drop Web Forms and Windows Forms building. I use Visual Studio .NET regularly and will refer to it throughout this book. However, this book is mostly about programming in general, so Visual Studio .NET isn't required to learn, use, and understand the concepts I present in each chapter.

## Goal of This Book

The purpose of this book is to explain how to develop applications for the .NET Framework. Specifically, this means that I intend to explain how the CLR works and the facilities it offers. I'll also discuss various parts of the FCL. No book could fully explain the FCL—it contains literally thousands of types, and this number is growing at an alarming rate. So, 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, XML Web services, Web Forms, and so on, the technologies presented in the book are applicable to *all* these application types.

With this book, I'm not attempting to teach you any particular programming language. I'm assuming that you're familiar with a programming language such as C++, C#, Visual Basic, or Java. I also assume that you're familiar with object–oriented programming concepts such as data abstraction, inheritance, and polymorphism. A good understanding of these concepts is critical because all .NET Framework features are offered via an object–oriented paradigm. If you're not familiar with these concepts, I strongly suggest you first find a book that teaches these concepts.

Although I don't intend to teach basic programming, I will spend time on various programming topics that are specific to the .NET Framework. All .NET Framework developers must be aware of these topics, which I explain and use throughout this book.

Finally, because this is a book about the .NET Framework's common language runtime, it's not about programming in any one specific programming language. However, I provide lots of code examples in the book to show how things really work. To remain programming language agnostic, the best language for me to use for these examples would be IL (intermediate language) assembly language. IL is the only programming language that the CLR understands. All language compilers compile source code to IL, which is later processed by the CLR. Using IL, you can access every feature offered by the CLR.

However, using IL assembly language is a pretty low–level way to write programs and isn't an ideal way to demonstrate programming concepts. So I decided to use C# as my programming language of choice throughout this entire book. I chose C# because it is the language Microsoft designed specifically for developing code for the .NET Framework. If you've decided not to use C# for your programming projects, that's OK—I'll just assume that you can read C# even if you're not

programming in it.

## System Requirements

The .NET Framework will install on Windows 98, Windows 98 Second Edition, Windows Me, Windows NT 4 (all editions), Windows 2000 (all editions), Windows XP (all editions), and the Windows .NET Server Family servers. You can download it from *http://go.microsoft.com/fwlink/?LinkId=5584*.

The .NET Framework SDK and Visual Studio .NET require Windows NT 4 (all editions), Windows 2000 (all editions), Windows XP (all editions), and the servers in the Windows .NET Server Family. You can download the .NET Framework SDK from *http://go.microsoft.com/fwlink/?LinkId=77*. You have to buy Visual Studio .NET, of course.

You can download the code associated with this book from *http://www.Wintellect.com*.

## This Book Has No Mistakes

This section's title clearly states what I want to say. But we all know that it is a flat–out lie. 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 the fantastic team assembled, things inevitably slip through the cracks. If you find any mistakes in this book (especially bugs), I would greatly appreciate it if you would send the mistakes to me at *http://www.Wintellect.com*.

## Support

Every effort has been made to ensure the accuracy of this book. Microsoft Press provides corrections for books through the World Wide Web at the following address:

*http://www.microsoft.com/mspress/support/*

To connect directly to the Microsoft Press Knowledge Base and enter a query regarding a question or issue that you may have, go to:

*http://www.microsoft.com/mspress/support/search.asp*

If you have comments, questions, or ideas regarding this book, please send them to Microsoft Press using either of the following methods:

Postal Mail:

Microsoft Press
Attn: *Applied Microsoft .NET Framework Programming* Editor
One Microsoft Way
Redmond, WA 98052–6399

E–Mail:

MSPINPUT@MICROSOFT.COM

Please note that product support is not offered through the above mail addresses. For support information regarding C#, Visual Studio, or the .NET Framework, visit the Microsoft Product Standard Support Web site at:

*http://support.microsoft.com*

# Part I: Basics of the Microsoft .NET Framework

## Chapter List

*Chapter 1: The Architecture of the .NET framework Development Platform*
*Chapter 2: Building, Packaging, Deploying, and Administering Applications and Types*
*Chapter 3: Shared Assemblies*

# Chapter 1: The Architecture of the .NET Framework Development Platform

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 architected, 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 (types) and then explain how these components 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 what 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. 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 features of the CLR are available to any and all programming languages that target it—period. If the runtime uses exceptions to report errors, then all languages get errors reported via exceptions. If the runtime allows you to create a thread, then any language 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 using APL syntax can save many days of development time when compared to expressing the same intention using Perl syntax, for example.

Microsoft is creating several language compilers that target the runtime: C++ with managed extensions, C# (pronounced "C sharp"), Visual Basic, JScript, J# (a Java language compiler), and an intermediate language (IL) assembler. In addition to Microsoft, several other companies are creating compilers that produce code that targets the CLR. I'm aware of compilers for Alice, APL, COBOL, Component Pascal, Eiffel, Fortran, Haskell, Mercury, ML, Mondrian, Oberon, Perl, Python, RPG, Scheme, and Smalltalk.

Figure 1–1 shows the process of compiling source code files. As the figure shows, you can create

source code files using 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 Windows portable executable (PE) file that requires the CLR to execute. In the future, other operating systems may use the PE file format as well.

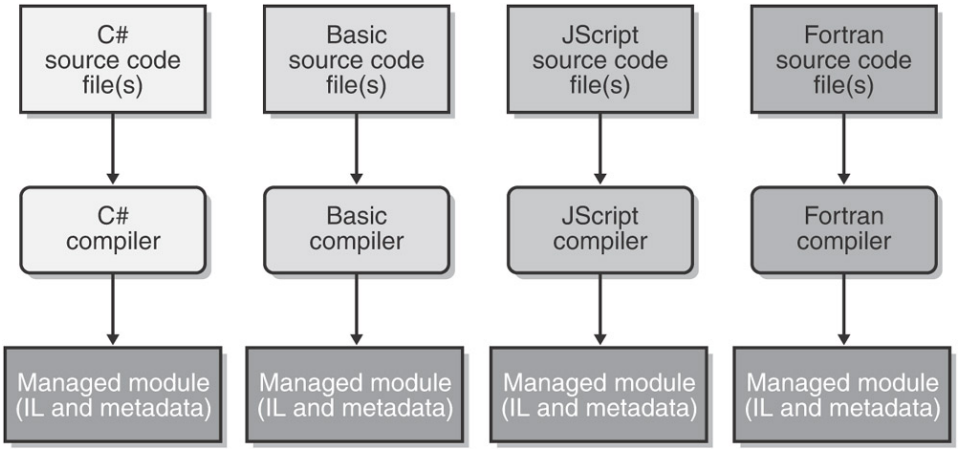



Figure 1–1 : Compiling source code into managed modules

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

Table 1–1: Parts of a Managed Module

| Part | Description |
|---|---|
| PE header | The standard Windows PE file header, which is similar to the Common Object File Format (COFF) header. This header indicates the type of file: GUI, CUI, or DLL, and it also has a timestamp indicating when the file was built. For modules that contain only IL code, the bulk of the information in the PE 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. |
| Intermediate language (IL) code | Code that the compiler produced as it compiled the source code. The CLR later compiles the IL into native CPU instructions. |

Most compilers of the past produced code targeted to a specific CPU architecture, such as x86, IA64, Alpha, or PowerPC. 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 lifetime and 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 simply 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 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 header and library files when compiling since 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.
- Visual Studio .NET uses metadata to help you write code. Its IntelliSense feature parses metadata to tell you what methods a type offers and what parameters that method expects.
- The CLR's code verification process uses metadata to ensure that your code performs only "safe" operations. (I'll discuss verification shortly.)
- Metadata allows an object's fields to be serialized into a memory block, remoted to another machine, and then deserialized, re–creating the object and its 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, I'll describe metadata in much more detail.

Microsoft's C#, Visual Basic, JScript, J#, and the IL Assembler always produce managed modules that require the CLR to execute. End–users must have the CLR installed on their machine in order to execute any managed modules, 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 unmanaged modules: the EXE or DLL files that we're all familiar with. These modules don't require the CLR in order to execute. However, by specifying a new command–line switch, the C++ compiler can produce managed modules that do require the CLR to execute. Of all the Microsoft compilers mentioned, C++ is unique in that it is the only language that allows the developer to write both managed and unmanaged code and have it emitted into a single module. This can be a great feature because it allows developers to write the bulk of their application in managed code (for type safety and component interoperability) but continue to access their existing unmanaged C++ code.

## 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 managed 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 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 PE file that represents the logical grouping of files. What happens is that this PE 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.

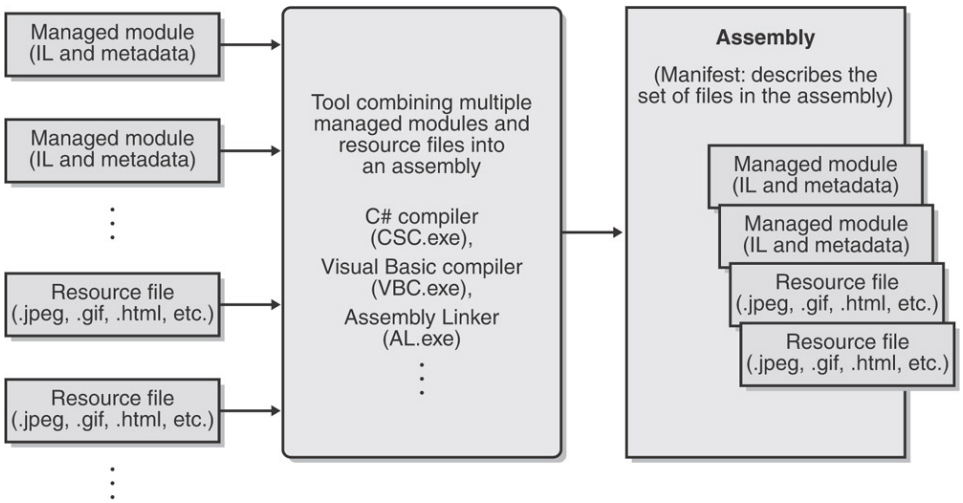



Figure 1–2 : Combining managed modules into assemblies

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, deployable, 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 from the Web as needed. 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 the files as a single collection.

An assembly's modules also include information, including version numbers, about referenced assemblies. This information makes an assembly *self–describing*. In other words, the CLR knows everything about what an assembly needs in order to execute. No additional information is required in the registry or in Active Directory. Because no additional information is needed, deploying assemblies is much easier than deploying unmanaged components.

## Loading the Common Language Runtime

Each assembly that you build can be either an executable application or a DLL containing a set of types (components) 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. Eventually, the .NET Framework will be packaged with future versions of Windows so that you won't have to ship it with your assemblies.

You can tell if the .NET Framework has been installed by looking for the MSCorEE.dll file in the %windir%\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\.NETFramework\policy
```

When you build an EXE assembly, the compiler/linker emits some special information into the resulting assembly's PE file header and the file's .**text** section. When the EXE file is invoked, this special information causes the CLR to load and initialize. The CLR then locates the application's entry point method and allows the application to start executing.

Similarly, if an unmanaged application calls **LoadLibrary** to load a managed assembly, the DLL's entry point function knows to load the CLR in order to process the code contained within the assembly.

For the most part, you don't need to know about or understand how the CLR gets loaded. For most programmers, this special information allows the application to just run, and there's nothing more to think about. For the curious, however, I'll spend the remainder of this section explaining how a managed EXE or DLL starts the CLR. If you're not interested in this subject, feel free to skip to the next section. Also, if you're interested in building an unmanaged application that hosts the CLR, see Chapter 20.

Figure 1–3 summarizes how a managed EXE loads and initializes the CLR.

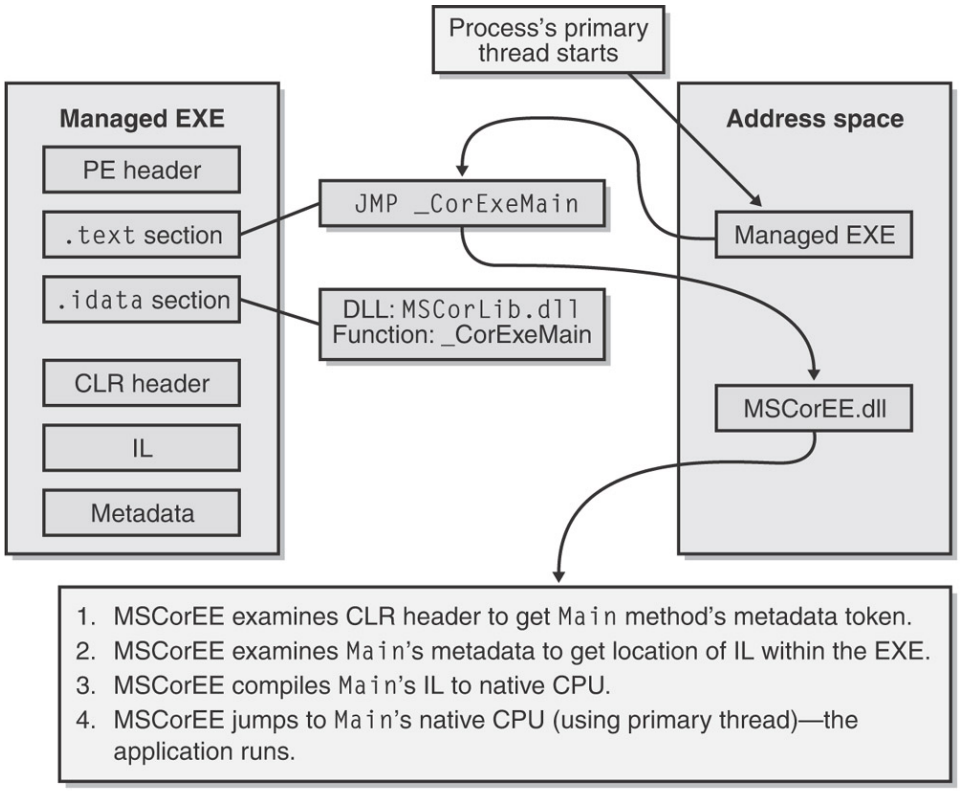



Figure 1–3 : Loading and initializing the CLR

When the compiler/linker creates an executable assembly, the following 6–byte x86 stub function is emitted into the PE file's .**text** section:

```
JMP _CorExeMain
```

Because the **_CorExeMain** function is imported from Microsoft's MSCorEE.dll dynamic–link library, MSCorEE.dll is referenced in the assembly file's import (.**idata**) section. MSCorEE.dll stands for *Microsoft Component Object Runtime Execution Engine*. When the managed EXE file is invoked, Windows treats it just like any normal (unmanaged) EXE file: the Windows loader loads the file and examines the .**idata** section to see that MSCorEE.dll should be loaded into the process's address space. Then the loader obtains the address of the **_CorExeMain** function inside MSCorEE.dll and fixes up the stub function's **JMP** instruction in the managed EXE file.

The process's primary thread begins executing this x86 stub function, which immediately jumps to **_CorExeMain** in MSCorEE.dll. **_CorExeMain** initializes the CLR and then looks at the executable assembly's CLR header to determine what managed entry point method should execute. The IL code for the method is then compiled into native CPU instructions, and the CLR jumps to the native code (using the process's primary thread). At this point, the managed application's code is running.

The situation is similar for a managed DLL. When building a managed DLL, the compiler/linker emits a similar 6–byte x86 stub function in the PE file's .**text** section for a DLL assembly:

```
JMP _CorDllMain
```

The **_CorDllMain** function is also imported from the MSCorEE.dll, causing the DLL's **.idata** section to reference MSCorEE.dll. When Windows loads the DLL, it will automatically load MSCorEE.dll (if it isn't already loaded), obtain the address of the **_CorDllMain** function, and fix up the 6–byte x86 **JMP** stub in the managed DLL. The thread that called **LoadLibrary** to load the managed DLL now jumps to the x86 stub in the managed DLL assembly, which immediately jumps to the **_CorDllMain**

function in MSCorEE.dll. **_CorDllMain** initializes the CLR (if it hasn't already been initialized for the process) and then returns so that the application can continue executing as normal.

These 6-byte x86 stub functions are required to run managed assemblies on Windows 98, Windows 98 Standard Edition, Windows Me, Windows NT 4, and Windows 2000 because all these operating systems shipped long before the CLR became available. Note that the 6-byte stub function is specifically for x86 machines. This stub doesn't work properly if the CLR is ported to run on other CPU architectures. Because Windows XP and the Windows .NET Server Family support both the x86 and the IA64 CPU architectures, Windows XP and the Windows .NET Server Family loader was modified to look specifically for managed assemblies.

On Windows XP and the Windows .NET Server Family, when a managed assembly is invoked (typically via **CreateProcess** or **LoadLibrary**), the OS loader detects that the file contains managed code by examining directory entry 14 in the PE file header. (See **IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR** in WinNT.h.) If this directory entry exists and is not 0, the loader ignores the file's import (.**idata**) section and automatically loads MSCorEE.dll into the process's address space. Once loaded, the OS loader makes the process's thread jump directly to the correct function in MSCorEE.dll. The 6-byte x86 stub functions are ignored on machines running Windows XP and the Windows .NET Server Family.

One last note on managed PE files: they always use the 32 bit PE file format, not the 64-bit PE file format. On 64-bit Windows systems, the OS loader detects the managed 32-bit PE file and automatically knows to create a 64-bit address space.

## Executing Your Assembly's Code

As mentioned earlier, managed modules contain both metadata and intermediate language (IL). IL is a CPU-independent machine language created by Microsoft after consultation with several external commercial and academic language/compiler writers. IL is much higher level than most CPU machine languages. IL understands object types and has instructions that create and initialize objects, call virtual methods on objects, and manipulate array elements directly. It even has instructions that throw and catch exceptions for error handling. You can think of IL as an object-oriented machine language.

Usually, developers will program in a high-level language, such as C# or Visual Basic. The compilers for these high-level languages produce IL. However, like any other machine language, IL can be written in assembly language, and Microsoft does provide an IL Assembler, ILAsm.exe. Microsoft also provides an IL Disassembler, ILDasm.exe.

---

IL and Protecting Your Intellectual Property

Some people are concerned that IL doesn't offer enough intellectual property protection for their algorithms. In other words, they think you could build a managed module and someone else could use a tool, such as IL Disassembler, to easily reverse engineer exactly what your application's code does.

Yes, it's true that IL code is higher level than most other assembly languages and that, in general, reverse engineering IL code is relatively simple. However, when implementing an XML Web service or a Web Forms application, your managed module resides on your server. Because no one outside your company can access the module, no one outside your company can use any tool to see the IL—your intellectual property is completely safe.

If you're concerned about any of the managed modules that you do distribute, you can obtain an obfuscator utility from a third–party vendor. These utilities "scramble" the names of all the private symbols in your managed module's metadata. It will be difficult for someone to "unscramble" the names and understand the purpose of each method. Note that these obfuscators can only provide a little protection since the IL must be available at some point in order for the CLR to process it.

If you don't feel that an obfuscator offers the kind of intellectual property protection that you desire, you can consider implementing your more sensitive algorithms in some unmanaged module that will contain native CPU instructions instead of IL and metadata. Then you can use the CLR's interoperability features to communicate between the managed and unmanaged portions of your application. Of course, this assumes that you're not worried about people reverse engineering the native CPU instructions in your unmanaged code.

---

Keep in mind that any high–level language will most likely expose only a subset of the facilities offered by the CLR. However, using IL assembly language allows a developer access to all the CLR's facilities. So, should your programming language of choice hide a facility the CLR offers that you really want to take advantage of, you can choose to write that portion of your code in IL assembly or perhaps another programming language that exposes the CLR feature you seek.

The only way for you to know what facilities the CLR offers is to read documentation specific to the CLR itself. In this book, I try to concentrate on CLR features and how they are exposed or not exposed by the C# language. I suspect that most other books and articles will present the CLR via a language perspective and that most developers will come to believe that the CLR offers only what the developer's chosen language exposes. As long as your language allows you to accomplish what you're trying to get done, this blurred perspective isn't a bad thing.

**Important** I think that this ability to switch programming languages easily with rich integration between languages is an awesome feature of the CLR. Unfortunately, I also believe that developers will often overlook this feature. Programming languages such as C# and Visual Basic are excellent languages for doing I/O operations. APL is a great language for doing advanced engineering or financial calculations. Through the CLR, you can write the I/O portions of your application using C# and then write the engineering calculations part using APL. The CLR offers a level of integration between these languages that is unprecedented and really makes mixed–language programming worthy of consideration for many development projects.

Another important point to keep in mind about IL is that it isn't tied to any specific CPU platform. This means that a managed module containing IL can run on any CPU platform as long as the operating system running on that CPU platform hosts a version of the CLR. Although the initial release of the CLR runs only on 32–bit Windows platforms, developing an application using managed IL sets up a developer to be more independent of the underlying CPU architecture.

---

### Standardizing the .NET Framework

In October 2000, Microsoft (along with Intel and Hewlett–Packard as co–sponsors) proposed a large subset of the .NET Framework to the ECMA (the European Computer Manufacturer's Association) for the purpose of standardization. The ECMA accepted this proposal and created a technical committee (TC39) to oversee the standardization process. The technical committee is charged with the following duties:

- **Technical Group 1** Develop a dynamic scripting language standard (ECMAScript). Microsoft's implementation of ECMAScript is JScript.
- **Technical Group 2** Develop a standardized version of the C# programming language.
- **Technical Group 3** Develop a Common Language Infrastructure (CLI) based on a subset of the functionality offered by the .NET Framework's CLR and class library. Specifically, the CLI will define a file format, a common type system, an extensible metadata system, an intermediate language (IL), and access to the underlying platform (P/Invoke). In addition, the CLI will define a factorable (to allow for small hardware devices) base class library designed for use by multiple programming languages.

Once the standardization is complete, these standards will be contributed to ISO/IEC JTC 1 (Information Technology). At this time, the technical committee will also investigate further directions for CLI, C#, and ECMAScript as well as entertain proposals for any complementary or additional technology. For more information about ECMA, see *http://www.ECMA.ch* and *http://MSDN.Microsoft.com/Net/ECMA*.

With the standardization of the CLI, C#, and ECMAScript, Microsoft won't "own" any of these technologies. Microsoft will simply be one company of many (hopefully) that are producing implementations of these technologies. Certainly Microsoft hopes that their implementation will be the best in terms of performance and customer–demand–driven features. This is what will help sales of Windows, since the Microsoft "best of breed" implementation will run only on Windows. However, other companies may implement these standards, compete against Microsoft, and possibly win.

---

Even though today's CPUs can't execute IL instructions directly, CPUs of the future might have this capability. To execute a method, its IL must first be converted to native CPU instructions. This is the job of the CLR's JIT (just–in–time) compiler.

Figure 1–4 shows what happens the first time a method is called.

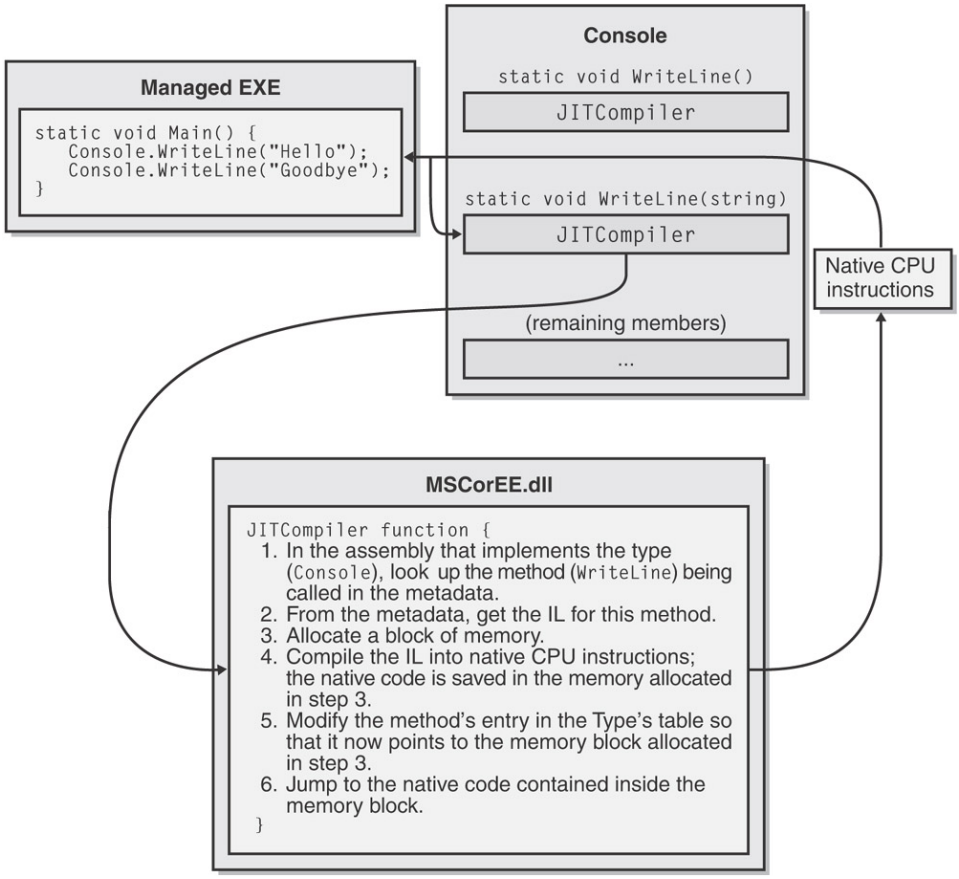



Figure 1–4 : Calling a method for the first time

Just before the **Main** method executes, the CLR detects all the types that are referenced by **Main**'s code. This causes the CLR to allocate an internal data structure that is used to manage access to the referenced type. In Figure 1–4, the **Main** method refers to a single type, **Console**, causing the CLR to allocate a single internal structure. This internal data structure contains an entry for each method defined by the type. Each entry holds the address where the method's implementation can be found. When initializing this structure, the CLR sets each entry to an internal, undocumented function contained inside the CLR itself. I call this function **JITCompiler**.

When **Main** makes its first call to **WriteLine**, the **JITCompiler** function is called. The **JITCompiler** function is responsible for compiling a method's IL code into native CPU instructions. Because the IL is being compiled "just in time," this component of the CLR is frequently referred to as a *JITter* or a *JIT compiler*.

When called, the **JITCompiler** function knows what method is being called and what type defines this method. The **JITCompiler** function then searches the defining assembly's metadata for the called method's IL. **JITCompiler** next verifies and compiles the IL code into native CPU instructions. The native CPU instructions are saved in a dynamically allocated block of memory. Then, **JITCompiler** goes back to the type's internal data structure and replaces the address of the called method with the address of the block of memory containing the native CPU instructions. Finally, **JITCompiler** jumps to the code in the memory block. This code is the implementation of the **WriteLine** method (the version that takes a **String** parameter). When this code returns, it returns to the code in **Main**, which continues execution as normal.

**Main** now calls **WriteLine** a second time. This time, the code for **WriteLine** has already been verified and compiled. So the call goes directly to the block of memory, skipping the **JITCompiler** function entirely. After the **WriteLine** method executes, it returns to **Main**. Figure 1–5 shows what the situation looks like when **WriteLine** is called the second time.

A performance hit is incurred only the first time a method is called. All subsequent calls to the method execute at the full speed of the native code: verification and compilation to native code are not performed again.

The JIT compiler stores the native CPU instructions in dynamic memory. This means that the compiled code is discarded when the application terminates. So, if you run the application again in the future or if you run two instances of the application simultaneously (in two different operating system processes), the JIT compiler will have to compile the IL to native instructions again.

For most applications, the performance hit incurred by JIT compiling isn't significant. Most applications tend to call the same methods over and over again. These methods will take the performance hit only once while the application executes. It's also likely that more time is spent inside the method than calling the method.

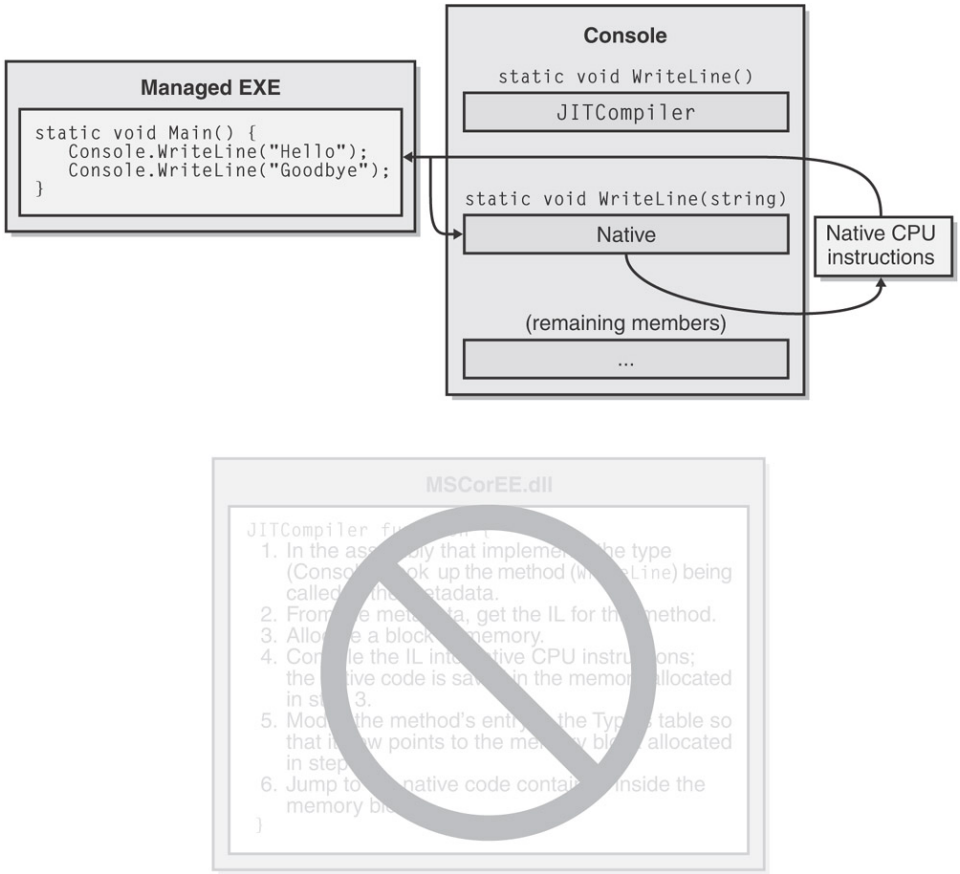



Figure 1–5 : Calling a method for the second time

You should also be aware that the CLR's JIT compiler optimizes the native code just as the back–end of an unmanaged C++ compiler does. Again, it may take more time to produce the optimized code, but the code will execute with much better performance than if it hadn't been optimized.

For those developers coming from an unmanaged C or C++ background, you're probably thinking about the performance ramifications of all this. After all, unmanaged code is compiled for a specific CPU platform and, when invoked, the code can simply execute. In this managed environment, compiling the code is accomplished in two phases. First, the compiler passes over the source code, doing as much work as possible in producing IL. But to execute the code, the IL itself must be compiled into native CPU instructions at run time, requiring more memory to be allocated and requiring additional CPU time to do the work.

Believe me, since I approached the CLR from a C/C++ background myself, I was quite skeptical and concerned about this additional overhead. The truth is that this second compilation stage that occurs at run time does hurt performance and it does allocate dynamic memory. However, Microsoft has done a lot of performance work to keep this additional overhead to a minimum.

If you too are skeptical, you should certainly build some applications and test the performance for yourself. In addition, you should run some nontrivial managed applications Microsoft or others have produced and measure their performance. I think you'll be surprised at how good the performance actually is.

In fact, you'll probably find this hard to believe, but many people (including me) think that managed applications could actually outperform unmanaged applications. There are many reasons to believe this. For example, when the JIT compiler compiles the IL code into native code at run time, the compiler knows more about the execution environment than an unmanaged compiler would know. Here are some ways that managed code could outperform unmanaged code:

- A JIT compiler could detect that the application is running on a Pentium 4 and produce native code that takes advantage of any special instructions offered by the Pentium 4. Usually, unmanaged applications are compiled for the lowest–common–denominator CPU and avoid using special instructions that would give the application a performance boost over newer CPUs.
- A JIT compiler could detect that a certain test is always false on the machine that it is running on. For example, consider a method with code like this:

  ```
  if (numberOfCPUs > 1) {
     Ã
  }
  ```

  This code could cause the JIT compiler not to generate any CPU instructions if the host machine has only one CPU. In this case, the native code has been fine–tuned for the host machine: the code is smaller and executes faster.
- The CLR could profile the code's execution and recompile the IL into native code while the application runs. The recompiled code could be reorganized to reduce incorrect branch predictions depending on the observed execution patterns.

These are only a few of the reasons why you should expect future managed code to execute better than today's unmanaged code. As I said, the performance is currently quite good for most applications, and it promises to improve as time goes on.

If your experiments show that the CLR's JIT compiler doesn't offer your application the kind of performance it requires, you may want to take advantage of the NGen.exe tool that ships with the .NET Framework SDK. This tool compiles all an assembly's IL code into native code and saves the resulting native code to a file on disk. At run time, when an assembly is loaded, the CLR automatically checks to see whether a precompiled version of the assembly also exists, and if it does, the CLR loads the precompiled code so that no compilation at run time is required.

## IL and Verification

IL is stack–based, which means that all its instructions push operands onto an execution stack and pop results off the stack. Because IL offers no instructions to manipulate registers, compiler developers have an easy time producing IL code; they don't have to think about managing registers, and fewer IL instructions are needed (since none exist for manipulating registers).

IL instructions are also typeless. For example, IL offers an **add** instruction that adds the last two operands pushed on the stack; there are not separate 32–bit and 64–bit **add** instructions. When the **add** instruction executes, it determines the types of the operands on the stack and performs the appropriate operation.

In my opinion, the biggest benefit of IL isn't that it abstracts away the underlying CPU. The biggest benefit is application robustness. While compiling IL into native CPU instructions, the CLR performs a process called *verification*. Verification examines the high–level IL code and ensures that everything it does is "safe." For example, verification checks that no memory is read from without having previously been written to, that every method is called with the correct number of parameters and that each parameter is of the correct type, that every method's return value is used properly, that every method has a return statement, and so on.

The managed module's metadata includes all the method and type information used by the verification process. If the IL code is determined to be "unsafe," then a **System.Security.VerifierException** exception is thrown, preventing the method from executing.

---

Is Your Code Safe?

By default, the Microsoft C# and Visual Basic compilers produce *safe* code. Safe code is code that is verifiably safe. However, using the **unsafe** keyword in C# or other languages (such as C++ with Managed Extensions or IL assembly language), it's possible to produce code that can't be verifiably safe. The code might, in fact, be safe, but verification is unable to prove this.

To ensure that all your managed module's methods contain verifiably safe IL, you can use the PEVerify utility (PEVerify.exe) that ships with the .NET Framework SDK. When Microsoft tests their C# and Visual Basic compilers, they run the resulting module through PEVerify to ensure that the compiler always produces verifiably safe code. If PEVerify detects unsafe code, Microsoft fixes the compiler.

You may want to consider running PEVerify on your own modules before you package and ship them. If PEVerify detects a problem, then there is a bug in the compiler and you should report it to Microsoft (or whatever company produces the compiler you're using). If PEVerify doesn't detect any unverifiable code, you know that your code will run without throwing a **VerifierException** on the end–user's machine.

You should be aware that verification requires access to the metadata contained in any dependant assemblies. So, when you use PEVerify to check an assembly, it must be able to locate and load all referenced assemblies. Because PEVerify uses the CLR to locate the dependant assemblies, the assemblies are located using the same binding and probing rules that would normally be used when executing the assembly. (I'll discuss these binding and probing rules in Chapters 2 and 3.)

Note that an administrator can elect to turn off verification (using the Microsoft .NET Framework Configuration administrative tool). With verification off, the JIT compiler will compile unverifiable IL into native CPU instructions; however, the administrator is taking full responsibility for the code's behavior.

---

In Windows, each process has its own virtual address space. Separate address spaces are necessary because you can't trust the application's code. It is entirely possible (and unfortunately, all too common) that an application will read from or write to an invalid memory address. By placing

each Windows process in a separate address space, you gain robustness: one process can't adversely affect another process.

By verifying the managed code, however, you know that the code doesn't improperly access memory that it shouldn't and you know that the code can't adversely affect another application's code. This means that you can run multiple managed applications in a single Windows virtual address space.

Because Windows processes require a lot of operating system resources, having many of them can hurt performance and limit available resources. Reducing the number of processes by running multiple applications in a single OS process can improve performance, require fewer resources, and be just as robust. This is another benefit of managed code as compared to unmanaged code.

The CLR does, in fact, offer the ability to execute multiple managed applications in a single OS process. Each managed application is called an *AppDomain*. By default, every managed EXE will run in its own, separate address space that has just the one AppDomain. However, a process hosting the CLR (such as Internet Information Services [IIS] or a future version of SQL Server) can decide to run AppDomains in a single OS process. I'll devote part of Chapter 20 to a discussion of AppDomains.

## The .NET Framework Class Library

Included with the .NET Framework is a set of .NET Framework Class Library (FCL) assemblies that contains several thousand type definitions, where each type exposes some functionality. All in all, the CLR and the FCL allow developers to build the following kinds of applications:

- **XML Web services** Methods that can be accessed over the Internet very easily. XML Web services are, of course, the main thrust of Microsoft's .NET initiative.
- **Web Forms** HTML–based applications (Web sites). Typically, Web Forms applications will make database queries and Web service calls, combine and filter the returned information, and then present that information in a browser using a rich HTML–based user interface. Web Forms provides a Visual Basic 6 and Visual InterDev style development environment for Web applications written in any CLR language.
- **Windows Forms** Rich Windows GUI applications. Instead of using a Web Forms page to create your application's UI, you can use the more powerful, higher performance functionality offered by the Windows desktop. Windows Forms applications can take advantage of controls, menus, and mouse and keyboard events, and they can talk directly to the underlying operating system. Like Web Forms applications, Windows Forms applications also make database queries and call XML Web services. Windows Forms provides a Visual Basic 6Ðlike development environment for GUI applications written in any CLR language.
- **Windows console applications** For applications with very simple UI demands, a console application provides a quick and easy way to build an application. Compilers, utilities, and tools are typically implemented as console applications.
- **Windows services** Yes, it is possible to build service applications controllable via the Windows Service Control Manager (SCM) using the .NET Framework.
- **Component library** The .NET Framework allows you to build stand–alone components (types) that can be easily incorporated into any of the previously mentioned application types.

Because the FCL contains literally thousands of types, a set of related types is presented to the developer within a single namespace. For example, the **System** namespace (which you should

become most familiar with) contains the **Object** base type, from which all other types ultimately derive. In addition, the **System** namespace contains types for integers, characters, strings, exception handling, and console I/O as well as a bunch of utility types that convert safely between data types, format data types, generate random numbers, and perform various math functions. All applications will use types from the **System** namespace.

To access any of the platform's features, you need to know which namespace contains the types that expose the facilities you're after. If you want to customize any type's behavior, you can simply derive your own type from the desired FCL type. The object–oriented nature of the platform is how the .NET Framework presents a consistent programming paradigm to software developers. Also, developers can easily create their own namespaces containing their own types. These namespaces and types merge seamlessly into the programming paradigm. Compared to Win32 programming paradigms, this new approach greatly simplifies software development.

Most of the namespaces in the FCL present types that can be used for any kind of application. Table 1–2 lists some of the more general namespaces and briefly describes what the types in that namespace are used for.

Table 1–2: Some General FCL Namespaces

| Namespace | Description of Contents |
|---|---|
| **System** | All the basic types used by every application |
| **System.Collections** | Types for managing collections of objects; includes the popular collection types, such as stacks, queues, hash tables, and so on |
| **System.Diagnostics** | Types to help instrument and debug applications |
| **System.Drawing** | Types for manipulating 2–D graphics; typically used for Windows Forms applications and for creating images that are to appear in a Web Forms page |
| **System.EnterpriseServices** | Types for managing transactions, queued components, object pooling, JIT activation, security, and other features to make the use of managed code more efficient on the server |
| **System.Globalization** | Types for National Language Support (NLS), such as string compares, formatting, and calendars |
| **System.IO** | Types for doing stream I/O, walking directories and files |
| **System.Management** | Types used for managing other computers in the enterprise via Windows Management Instrumentation (WMI) |
| **System.Net** | Types that allow for network communications |
| **System.Reflection** | Types that allow the inspection of metadata and late binding to types and their members |
| **System.Resources** | Types for manipulating external data resources |
| **System.Runtime.InteropServices** | Types that allow managed code to access unmanaged OS platform facilities such as COM components and functions in Win32 DLLs |

| | |
|---|---|
| **System.Runtime.Remoting** | Types that allow for types to be accessed remotely |
| **System.Runtime.Serialization** | Types that allow for instances of objects to be persisted and regenerated from a stream |
| **System.Security** | Types used for protecting data and resources |
| **System.Text** | Types to work with text in different encodings, such as ASCII or Unicode |
| **System.Threading** | Types used for asynchronous operations and synchronizing access to resources |
| **System.Xml** | Types used for processing XML schemas and data |

This book is about the CLR and about the general types that interact closely with the CLR (which are most of the namespaces listed in Table 1–2). So the content of this book is applicable to all .NET Framework programmers, regardless of the type of application they're building.

In addition to the more general namespaces, the FCL also offers namespaces whose types are used for building specific application types. Table 1–3 lists some of the application–specific namespaces in the FCL.

Table 1–3: Some Application–Specific FCL Namespaces

| Namespace | Application Type |
|---|---|
| **System.Web.Services** | Types used to build XML Web services |
| **System.Web.UI** | Types used to build Web Forms |
| **System.Windows.Forms** | Types used to build Windows GUI applications |
| **System.ServiceProcess** | Types used to build a Windows service controllable by the SCM |

I expect many good books will be published that explain how to build specific application types (such as Windows services, Web Forms, and Windows Forms). These books will give you an excellent start at helping you build your application. I tend to think of these application–specific books as helping you learn from the top down because they concentrate on the application type and not on the development platform. In this book, I'll offer information that will help you learn from the bottom up. After reading this book and an application–specific book, you should be able to easily and proficiently build any kind of .NET Framework application you desire.

## The Common Type System

By now, it should be obvious to you that the CLR is all about types. Types expose functionality to your applications and components. Types are the mechanism by which code written in one programming language can talk to code written in a different programming language. Because types are at the root of the CLR, Microsoft created a formal specification—the Common Type System (CTS)—that describes how types are defined and how they behave.

The CTS specification states that a type can contain zero or more members. In Part III, I'll cover all these members in great detail. For now, I just want to give you a brief introduction to them:

- **Field** A data variable that is part of the object's state. Fields are identified by their name and type.
- **Method** A function that performs an operation on the object, often changing the object's state. Methods have a name, a signature, and modifiers. The signature specifies the calling convention, the number of parameters (and their sequence), the types of the parameters, and the type of value returned by the method.
- **Property** To the caller, this member looks like a field. But to the type implementer, it looks like a method (or two). Properties allow an implementer to validate input parameters and object state before accessing the value and/or calculate a value only when necessary. They also allow a user of the type to have simplified syntax. Finally, properties allow you to create read–only or write–only "fields."
- **Event** An event allows a notification mechanism between an object and other interested objects. For example, a button could offer an event that notifies other objects when the button is clicked.

The CTS also specifies the rules for type visibility and for access to the members of a type. For example, marking a type as *public* (called **public**) exports the type, making it visible and accessible to any assembly. On the other hand, marking a type as *assembly* (called **internal** in C#) makes the type visible and accessible to code within the same assembly only. Thus, the CTS establishes the rules by which assemblies form a boundary of visibility for a type, and the CLR enforces the visibility rules.

Regardless of whether a type is visible to a caller, the type gets to control whether the caller has access to its members. The following list shows the valid options for controlling access to a method or a field:

- **Private** The method is callable only by other methods in the same class type.
- **Family** The method is callable by derived types, regardless of whether they are within the same assembly. Note that many languages (such as C++ and C#) refer to family as **protected**.
- **Family and assembly** The method is callable by derived types, but only if the derived type is defined in the same assembly. Many languages (such as C# and Visual Basic) don't offer this access control. Of course, IL Assembly language makes it available.
- **Assembly** The method is callable by any code in the same assembly. Many languages refer to *assembly* as **internal**.
- **Family or assembly** The method is callable by derived types in any assembly. The method is also callable by any types in the same assembly. C# refers to *family or assembly* as **protected internal**.
- **Public** The method is callable by any code in any assembly.

In addition, the CTS defines the rules governing type inheritance, virtual functions, object lifetime, and so on. These rules have been designed to accommodate the semantics expressible in modern–day programming languages. In fact, you won't even need to learn the CTS rules per se since the language you choose will expose its own language syntax and type rules in the same way you're familiar with today and will map the language–specific syntax into the "language" of the CLR when it emits the managed module.

When I first started working with the CLR, I soon realized that it is best to think of the language and the behavior of your code as two separate and distinct things. Using C++, you can define your own types with their own members. Of course, you could have used C# or Visual Basic to define the same type with the same members. Sure, the syntax you use for defining this type is different depending on the language you choose, but the behavior of the type will be absolutely identical

regardless of the language because the CLR's CTS defines the behavior of the type.

To help clarify this idea, let me give you an example. The CTS supports single inheritance only. So, while the C++ language supports types that inherit from multiple base types, the CTS can't accept and operate on any such type. To help the developer, the Visual C++ compiler reports an error if it detects that you're attempting to create managed code that includes a type inherited from multiple base types.

Here's another CTS rule. All types must (ultimately) inherit from a predefined type: **System.Object**. As you can see, **Object** is the name of a type defined in the **System** namespace. This **Object** is the root of all other types and therefore guarantees every type instance has a minimum set of behaviors. Specifically, the **System.Object** type allows you to do the following:

- Compare two instances for equality
- Obtain a hash code for the instance
- Query the true type of an instance
- Perform a shallow (bitwise) copy of the instance
- Obtain a string representation of the instance's object's current state

## The Common Language Specification

COM allows objects created in different languages to communicate with one another. On the other hand, the CLR now integrates all languages and allows objects created in one language to be treated as equal citizens by code written in a completely different language. This integration is possible because of the CLR's standard set of types, self–describing type information (metadata), and common execution environment.

While this language integration is a fantastic goal, the truth of the matter is that programming languages are very different from one another. For example, some languages don't treat symbols with case–sensitivity or don't offer unsigned integers, operator overloading, or methods that support a variable number of parameters.

If you intend to create types that are easily accessible from other programming languages, you need to use only features of your programming language that are guaranteed to be available in all other languages. To help you with this, Microsoft has defined a *Common Language Specification* (CLS) that details for compiler vendors the minimum set of features that their compilers must support if these compilers are to target the CLR.

The CLR/CTS supports a lot more features than the subset defined by the CLS, so if you don't care about interlanguage operability, you can develop very rich types limited only by the language's feature set. Specifically, the CTS defines rules that externally visible types and methods must adhere to if they are to be accessible from any CLR–compliant programming language. Note that the CLS rules don't apply to code that is accessible only within the defining assembly. Figure 1–6 summarizes the ideas expressed in this paragraph.

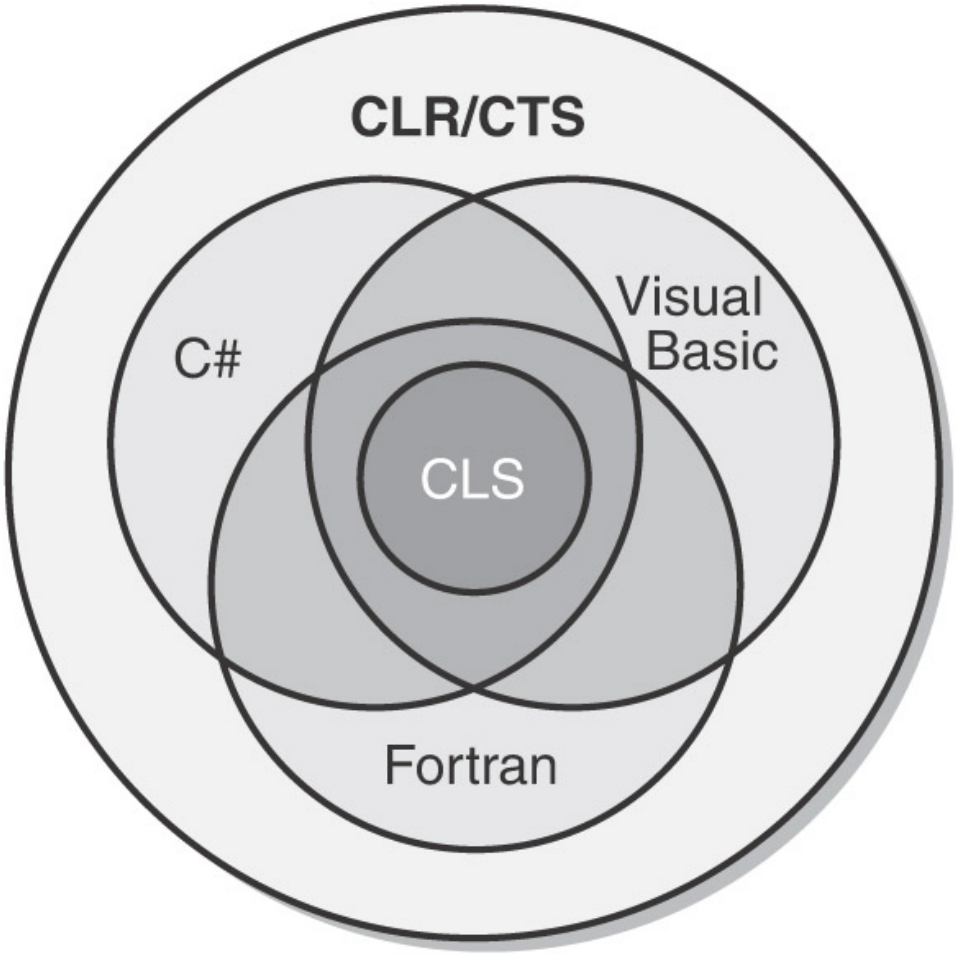



Figure 1–6 : Languages offer a subset of the CLR/CTS and a superset of the CLS (but not necessarily the same superset)

As Figure 1–6 shows, the CLR/CTS offers a set of features. Some languages expose a large subset of the CLR/CTS. A programmer willing to write in IL assembly language, for example, is able to use all the features the CLR/CTS offers. Most other languages, such as C#, Visual Basic, and Fortran, expose a subset of the CLR/CTS features to the programmer. The CLS defines the minimum set of features that all languages must support.

If you're designing a type in one language and you expect that type to be used by another language, you shouldn't take advantage of any features that are outside the CLS. Doing so would mean that your type's members might not be accessible by programmers writing code in other programming languages.

In the following code, a CLS–compliant type is being defined in C#. However, the type has a few non–CLS–compliant constructs causing the C# compiler to complain about the code.

```
using System;

// Tell compiler to check for CLS compliance
[assembly:CLSCompliant(true)]

// Errors appear because the class is public
public class App {

   // Error: Return type of ÔApp.Abc()' is not CLS-compliant
   public  UInt32 Abc() { return 0; }
```

```
   // Error: Identifier ÔApp.abc()' differing
   // only in case is not CLS-compliant
   public  void   abc() { }

   // No error: Method is private
   private UInt32 ABC() { return 0; }
}
```

In this code, the **[assembly:CLSCompliant(true)]** attribute is applied to the assembly. This attribute tells the compiler to ensure that any publicly exposed type doesn't have any construct that would prevent the type from being accessed from any other programming language. When this code is compiled, the C# compiler emits two errors. The first error is reported because the method **Abc** returns an unsigned integer; Visual Basic and some other languages can't manipulate unsigned integer values. The second error is because this type exposes two public methods that differ only by case: **Abc** and **abc**. Visual Basic and some other languages can't call both these methods.

Interestingly, if you were to delete **public** from in front of **Ôclass App'** and recompile, both errors would go away. The reason is that the **App** type would default to **internal** and would therefore no longer be exposed outside the assembly. For a complete list of CLS rules, refer to the "Cross–Language Interoperability" section in the .NET Framework SDK documentation.

Let me distill the CLS rules to something very simple. In the CLR, every member of a type is either a field (data) or a method (behavior). This means that every programming language must be able to access fields and call methods. Certain fields and certain methods are used in special and common ways. To ease programming, languages typically offer additional abstractions to make coding these common programming patterns easier. For example, languages expose concepts such as enums, arrays, properties, indexers, delegates, events, constructors, destructors, operator overloads, conversion operators, and so on. When a compiler comes across any of these things in your source code, it must translate these constructs into fields and methods so that the CLR and any other programming language can access the construct.

Consider the following type definition, which contains a constructor, a destructor, some overloaded operators, a property, an indexer, and an event. Note that the code shown is there just to make the code compile; it doesn't show the correct way to implement a type.

```
using System;

class Test {
   // Constructor
   public Test() {}

   // Destructor
   ~Test() {}

   // Operator overload
   public static Boolean operator == (Test t1, Test t2) {
      return true;
   }
   public static Boolean operator != (Test t1, Test t2) {
      return false;
   }

   // An operator overload
   public static Test operator + (Test t1, Test t2) { return null; }

   // A property
```

```
   public String AProperty {
      get { return null; }
      set { }
   }

   // An indexer
   public String this[Int32 x] {
      get { return null; }
      set { }
   }

   // An event
   event EventHandler AnEvent;
}
```

When the compiler compiles this code, the result is a type that has a number of fields and methods defined in it. You can easily see this using the IL Disassembler tool (ILDasm.exe) provided with the .NET Framework SDK to examine the resulting managed module, which is shown in Figure 1–7.

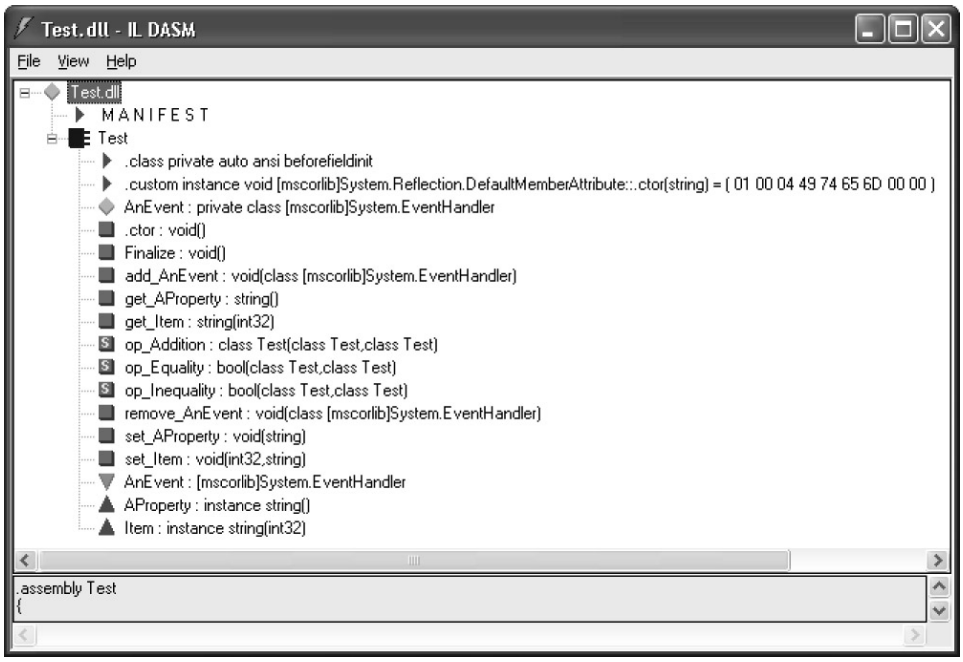



Figure 1–7 : ILDasm showing Test type's fields and methods (obtained from metadata)

Table 1–4 shows how the programming language constructs got mapped to the equivalent CLR fields and methods.

Table 1–4: Test Type's Fields and Methods (obtained from metadata)

| Type Member | Member Type | Equivalent Programming Language Construct |
|---|---|---|
| **AnEvent** | Field | Event; the name of the field is **AnEvent** and its type is **System.EventHandler** |
| **.ctor** | Method | Constructor |
| **Finalize** | Method | Destructor |
| **add_AnEvent** | Method | Event **add** accessor method |
| **get_AProperty** | Method | Property **get** accessor method |
| **get_Item** | Method | Indexer **get** accessor method |

| | | |
|---|---|---|
| **op_Addition** | Method | **+** operator |
| **op_Equality** | Method | **==** operator |
| **op_Inequality** | Method | **!=** operator |
| **remove_AnEvent** | Method | Event **remove** accessor method |
| **set_AProperty** | Method | Property **set** accessor method |
| **set_Item** | Method | Indexer **set** accessor method |

The additional nodes under the **Test** type that aren't mentioned in Table 1–4—**.class**, **.custom**, **AnEvent**, **AProperty**, and **Item**—identify additional metadata about the type. These nodes don't map to fields or methods; they just offer some additional information about the type that the CLR, programming languages, or tools can get access to. For example, a tool can see that the **Test** type offers an event, called **AnEvent**, which is exposed via the two methods (**add_AnEvent** and **remove_AnEvent**).

## Interoperability with Unmanaged Code

The .NET Framework offers a ton of advantages over other development platforms. However, very few companies can afford to redesign and reimplement all of their existing code. Microsoft realizes this and has constructed the CLR so that it offers mechanisms that allow an application to consist of both managed and unmanaged parts. Specifically, the CLR supports three interoperability scenarios:

- **Managed code can call an unmanaged function in a DLL** Managed code can easily call functions contained in DLLs using a mechanism called P/Invoke (for Platform Invoke). After all, many of the types defined in the FCL internally call functions exported from Kernel32.dll, User32.dll, and so on. Many programming languages will expose a mechanism that makes it easy for managed code to call out to unmanaged functions contained in DLLs. For example, a C# or Visual Basic application can call the **CreateSemaphore** function exported from Kernel32.dll.
- **Managed code can use an existing COM component (server)** Many companies have already implemented a number of unmanaged COM components. Using the type library from these components, a managed assembly can be created that describes the COM component. Managed code can access the type in the managed assembly just like any other managed type. See the TlbImp.exe tool that ships with the .NET Framework SDK for more information. At times, you might not have a type library or you might want to have more control over what TlbImp.exe produces. In these cases, you can manually build a type in source code that the CLR can use to achieve the proper interoperability. For example, you could use DirectX COM components from a C# or Visual Basic application.
- **Unmanaged code can use a managed type (server)** A lot of existing unmanaged code requires that you supply a COM component for the code to work correctly. It's much easier to implement these components using managed code so that you can avoid all the code having to do with reference counting and interfaces. For example, you could create an ActiveX control or a shell extension in C# or Visual Basic. See the TlbExp.exe and RegAsm.exe tools that ship with the .NET Framework SDK for more information.

In addition to these three scenarios, Microsoft's Visual C++ compiler (version 13) supports a new **/clr** command–line switch. This switch tells the compiler to emit IL code instead of native x86 instructions. If you have a large amount of existing C++ code, you can recompile the code using this new compiler switch. The new code will require the CLR to execute, and you can now modify the code over time to take advantage of the CLR–specific features.

The /**clr** switch can't compile to IL any methods that contain inline assembly language (via the **__asm** keyword), accept a variable number of arguments, call **setjmp**, or contain intrinsic routines (such as **__enable**, **__disable**, **_ReturnAddress**, and **_AddressOfReturnAddress**). For a complete list of the constructs that the C++ compiler can't compile into IL, refer to the documentation for the Visual C++ compiler. When the compiler can't compile the method into IL, it compiles the method into x86 so that the application still runs.

Keep in mind that although the IL code produced is managed, the data is not; that is, data objects are not allocated from the managed heap and they are not garbage collected. In fact, the data types don't have metadata produced for them, and the types' method names are mangled.

The following C code calls the standard C runtime library's **printf** function and also calls the **System.Console WriteLine** method. The **System.Console** type is defined in the FCL. So, C/C++ code can use libraries available to C/C++ as well as managed types.

```
#include <stdio.h>        // For printf

#using <mscorlib.dll>     // For managed types defined in this assembly
using namespace System;   // Easily access System namespace types

// Implement a normal C/C++ main function
void main() {

   // Call the C runtime library's printf function.
   printf("Displayed by printf.\r\n");

   // Call the FCL's System.Console's WriteLine method.
   Console::WriteLine("Displayed by Console::WriteLine.");
}
```

Compiling this code couldn't be easier. If this code were in a MgdCApp.cpp file, you'd compile it by executing the following line at the command prompt:

```
cl /clr MgdCApp.cpp
```

The result is a MgdCApp.exe assembly file. If you run MgdCApp.exe, you'll see the following output:

```
C:\>MgdCApp
Displayed by printf.
Displayed by Console::WriteLine.
```

If you use ILDasm.exe to examine this file, you'll see the output shown in Figure 1–8.

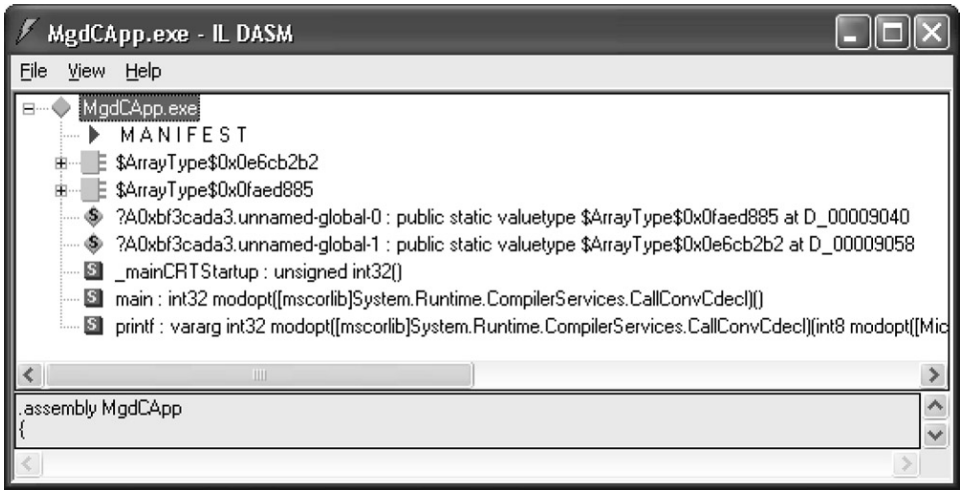


Figure 1–8 : ILDasm showing MgdCApp.exe assembly's metadata

In Figure 1–8, you see that ILDasm shows all the global functions and global fields defined within the assembly. Obviously, the compiler has generated a lot of stuff automatically. If you double–click the **Main** method, ILDasm will show you the IL code:

```
.method public static int32
        modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
        main() cil managed
{
  .vtentry 1 : 1
  // Code size       28 (0x1c)
  .maxstack  1
  IL_0000:  ldsflda    valuetype
                       $ArrayType$0x0faed885 Ô?A0x44d29f64.unnamed-global-0'
  IL_0005:  call       vararg int32
     modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
     printf(int8
     modopt([Microsoft.VisualC]Microsoft.VisualC.NoSignSpecifiedModifier)
     modopt([Microsoft.VisualC]Microsoft.VisualC.IsConstModifier)*)
  IL_000a:  pop
  IL_000b:  ldsflda    valuetype
                       $ArrayType$0x0e6cb2b2 Ô?A0x44d29f64.unnamed-global-1'
  IL_0010:  newobj     instance void [mscorlib]System.String::.ctor(int8*)
  IL_0015:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_001a:  ldc.i4.0
  IL_001b:  ret
} // end of method ÔGlobal Functions'::main
```

What we see here isn't pretty because the compiler generates a lot of special code to make all this work. However, from this IL code, you can see that **printf** and the **Console**.**WriteLine** method are both called.

# Chapter 2: Building, Packaging, Deploying, and Administering Applications and Types

## Overview

Before we get into the chapters that explain how to develop programs for the Microsoft .NET Framework, let's discuss the steps required to build, package, and deploy your applications and their types. In this chapter, I'll focus on the basics of how to build components that are for your application's sole use. In Chapter 3, I'll cover the more advanced concepts you'll need to understand, including how to build and use assemblies containing components that are to be shared by multiple applications. In both chapters, I'll also talk about the ways an administrator can use information to affect the execution of an application and its types.

Today, applications consist of several types. (In the .NET Framework, a *type* is called a *component*, but in this book, I'll avoid the term *component* and use *type* instead.) Applications typically consist of types created by you and Microsoft as well as several other organizations. If these types are developed using any language that targets the common language runtime (CLR), they can all work together seamlessly; a type can even use another type as its base class, regardless of what languages the types are developed in.

In this chapter, I'll also explain how these types are built and packaged into files for deployment. In the process, I'll take you on a brief historical tour of some of the problems that the .NET Framework is solving.

## .NET Framework Deployment Goals

Over the years, Windows has gotten a reputation for being unstable and complicated. This reputation, whether deserved or not, is the result of many different factors. First, all applications use dynamic–link libraries (DLLs) from Microsoft or other vendors. Because an application executes code from various vendors, the developer of any one piece of code can't be 100 percent sure how someone else is going to use it. This situation can potentially cause all kinds of trouble, but in practice, problems don't typically arise from this kind of interaction because applications are tested and debugged before they are deployed.

Users, however, frequently run into problems when one company decides to update its code and ships new files to them. These new files are supposed to be "backward compatible" with the previous files, but who knows for sure? In fact, when one vendor updates its code, retesting and debugging all the already shipped applications to ensure that the changes have had no undesirable effect is usually impossible.

I'm sure that everyone reading this book has experienced some variation of this problem: when installing a new application, you discover that it has somehow corrupted an already installed application. This predicament is known as "DLL hell." This type of instability puts fear into the hearts and minds of the typical computer user. The end result is that users have to carefully consider whether to install new software on their machines. Personally, I've decided not to try out certain applications for fear of some application I really rely on being adversely affected.

The second reason that has contributed to the reputation of Windows is installation complexities. Today, when most applications are installed, they affect all parts of the system. For example,

installing an application causes files to be copied to various directories, updates registry settings, and installs shortcuts links on your desktop, Start menu, and Quick Launch toolbar. The problem with this is that the application isn't isolated as a single entity. You can't easily back up the application since you must copy the application's files and also the relevant parts of the registry. In addition, you can't easily move the application from one machine to another; you must run the installation program again so that all files and registry settings are set properly. Finally, you can't easily uninstall or remove the application without having this nasty feeling that some part of the application is still lurking on your machine.

The third reason has to do with security. When applications are installed, they come with all kinds of files, many of them written by different companies. In addition, "Web applications" frequently have code that is downloaded over the wire in such a way that users don't even realize that code is being installed on their machine. Today, this code can perform any operation, including deleting files or sending e–mail. Users are right to be terrified of installing new applications because of the potential damage they can cause. To make users comfortable, security must be built into the system so that users can explicitly allow or disallow code developed by various companies to access the system resources.

The .NET Framework addresses the "DLL hell" issue in a big way, as you'll see as you read this chapter and Chapter 3. It also goes a long way toward fixing the problem of having an application's state scattered all over a user's hard disk. For example, unlike COM, components no longer require settings in the registry. Unfortunately, applications still require shortcut links, but future versions of Windows may solve this problem. As for security, the .NET Framework includes a new security model called *code access security*. Whereas Windows security is based around a user's identity, code access security is based around an assembly's identity. So a user could decide to trust all assemblies published by Microsoft or not to trust any assemblies downloaded from the Internet. As you'll see, the .NET Framework enables users to control their machines, what gets installed, and what runs more than Windows ever did.

## Building Types into a Module

In this section, I'll show you how to turn your source file, containing various types, into a file that can be deployed. Let's start by examining the following simple application:

```
public class App {
   static public void Main(System.String[] args) {
      System.Console.WriteLine("Hi");
   }
}
```

This application defines a type, called **App**. This type has a single static, public method called **Main**. Inside **Main** is a reference to another type called **System.Console**. **System.Console** is a type implemented by Microsoft, and the IL code that implements this type's methods are in the MSCorLib.dll file. So, our application defines a type and also uses another company's type.

To build this sample application, put the preceding code into a source code file, say App.cs, and then execute the following command line:

```
csc.exe /out:App.exe /t:exe /r:MSCorLib.dll App.cs
```

This command line tells the C# compiler to emit an executable file called App.exe (/**out:App.exe**). The type of file produced is a Win32 console application (/**t[arget]:exe**).

When the C# compiler processes the source file, it sees that the code references the **System.Console** type's **WriteLine** method. At this point, the compiler wants to ensure that this type exists somewhere, that it has a **WriteLine** method, and that it checks that the types of the arguments that **WriteLine** expects match up with what the program is supplying. To make the C# compiler happy, you must give it a set of assemblies that it can use to resolve references to external types. (I'll define assemblies shortly, but for now you can think of an assembly as a set of one or more DLL files.) In the command line above, I've included the /**r[eference]:MSCorLib.dll** switch telling the compiler to look for external types in the assembly identified by the MSCorLib.dll file.

MSCorLib.dll is a special file in that it contains all the core types, such as bytes, integers, characters, strings, and so on. In fact, these types are so frequently used that the C# compiler automatically references this assembly. In other words, the following command line (with the /**r** switch omitted) gives the same results as the line shown earlier:

```
csc.exe /out:App.exe /t:exe App.cs
```

Furthermore, because the /**out:App.exe** and the /**t:exe** command–line switches also match what the C# compiler would choose as defaults, the following command line gives the same results too:

```
csc.exe App.cs
```

If, for some reason, you really don't want the C# compiler to reference the MSCorLib.dll assembly, you can use the /**nostdlib** switch. For example, the following command line will generate an error when compiling the App.cs file since the **System.Console** type is defined in MSCorLib.dll:

```
csc.exe /out:App.exe /t:exe /nostdlib App.cs
```

Now, let's take a closer look at the App.exe file produced by the C# compiler. What exactly is this file? Well, for starters, it is a standard PE (portable executable) file. This means that a machine running 32–bit or 64–bit Windows should be able to load this file and do something with it. Windows supports two types of applications, those with a console user interface (CUI) and those with graphical user interface (GUI). Because I specified the /**t:exe** switch, the C# compiler produced a CUI application. You'd use the /**t:winexe** switch to have the C# compiler produce a GUI application.

Now we know what kind of PE file we've created. But what exactly is in the App.exe file? A managed PE file has four main parts: the PE header, the CLR header, the metadata, and the intermediate language (IL). The PE header is the standard information that Windows expects. The CLR header is a small block of information that is specific to modules that require the CLR (managed modules). The header includes the major and minor version number of the metadata that the module was built with, some flags, a MethodDef token (described later) indicating the module's entry point method if this module is a CUI or GUI executable, and an optional strong name digital signature (discussed in Chapter 3). Finally, the header contains the size and offsets of certain metadata tables contained within the module. You can see the exact format of the CLR header by examining the **IMAGE_COR20_HEADER** defined in the CorHdr.h header file.

The metadata is a block of binary data that consists of several tables. There are three categories of tables: definition tables, reference tables, and manifest tables. Table 2–1 describes some of the more common definition tables that exist in a module's metadata block.

Table 2–1: Common Definition Metadata Tables

| Metadata Definition Table Name | Description |
|---|---|

| | |
|---|---|
| ModuleDef | Always contains one entry that identifies the module. The entry includes the module's filename and extension (without path) and a module version ID (in the form of a GUID created by the compiler). This allows the file to be renamed while keeping a record of its original name. However, renaming a file is strongly discouraged and can prevent the CLR from locating an assembly at runtime—don't do this. |
| TypeDef | Contains one entry for each type defined in the module. Each entry includes the type's name, base type, flags (i.e., public, private, etc.) and points to the methods it owns in the MethodDef table, the fields it owns in the FieldDef table, the properties it owns in the PropertyDef table, and the event it owns in the EventDef table. |
| MethodDef | Contains one entry for each method defined in the module. Each entry includes the method's name, flags (private, public, virtual, abstract, static, final, etc), signature, and offset within the module where IL code can be found. Each entry can also refer to a ParamDef table entry where more information about the method's parameters can be found. |
| FieldDef | Contains one entry for every field defined in the module. Each entry includes a name, flags (i.e., private, public, etc.), and type. |
| ParamDef | Contains one entry for each parameter defined in the module. Each entry includes a name and flags (in, out, retval, etc). |
| PropertyDef | Contains one entry for each property defined in the module. Each entry includes a name, flags, type, and backing field (which can be null). |
| EventDef | Contains one entry for each event defined in the module. Each entry includes a name and flags. |

As a compiler compiles your source code, everything that your code defines causes an entry to be created in one of the tables described in Table 2–1. As the compiler compiles the source code, it also detects the types, fields, methods, properties, and events that the source code references. The metadata includes a set of reference tables that keep a record of this stuff. Table 2–2 shows some of the more common reference metadata tables.

Table 2–2: Common Reference Metadata Tables

| Metadata Reference Table Name | Description |
|---|---|
| AssemblyRef | Contains one entry for each assembly referenced by the module. Each entry includes the information necessary to bind to the assembly: the assembly's name (without path and extension), version number, culture, and public key token (normally a small hash value, generated from the publisher's public key, identifying the referenced assembly's publisher). Each entry also contains some flags and a hash value. This hash value was intended to be a checksum of the referenced assembly's bits. The CLR completely ignores this hash value and will probably continue to do so in the future. |
| ModuleRef | Contains one entry for each PE module that implements types referenced by this module. Each entry includes the module's filename and extension (without path). This table is used to bind to types that are implemented in different modules of the calling assembly's module. |
| TypeRef | |

| | |
|---|---|
| | Contains one entry for each type referenced by the module. Each entry includes the type's name and a reference to where the type can be found. If the type is implemented within another type, then the reference indicates a TypeRef entry. If the type is implemented in the same module, then the reference indicates a ModuleDef entry. If the type is implemented in another module within the calling assembly, then the reference indicates a ModuleRef entry. If the type is implemented in a different assembly, then the reference indicates an AssemblyRef entry. |
| MemberRef | Contains one entry for each member (fields and methods, as well as property and event methods) referenced by the module. Each entry includes the member's name and signature, and points to the TypeRef entry for the type that defines the member. |

There are many more tables than what I list in Tables 2–1 and 2–2, but I just wanted to give you a sense of the kind of information that the compiler emits to produce the metadata information. Earlier I mentioned that there is also a set of manifest metadata tables; I'll discuss these a little later in the chapter.

Various tools allow you to examine the metadata within a managed PE file. My personal favorite is ILDasm.exe, the IL disassembler. To see the metadata tables, execute the following command line:

```
ILDasm /Adv App.exe
```

This causes ILDasm.exe to run, loading the App.exe assembly. The /**Adv** switch tells ILDasm to make some "advanced" menu items available. These advanced menu items can be found on the View menu. To see the metadata in a nice, human–readable form, select the View.MetaInfo.Show! menu item (or press Ctrl+M). This causes the following information to appear:

```
ScopeName : App.exe
MVID      : {ED543DFC-44DD-4D14-9849-F7EC1B840BD0}
=============================================================
Global functions
-------------------------------

Global fields
-------------------------------

Global MemberRefs
-------------------------------

TypeDef #1
-------------------------------
    TypDefName: App  (02000002)
    Flags     : [Public] [AutoLayout] [Class] [AnsiClass]  (00100001)
    Extends   : 01000001 [TypeRef] System.Object
    Method #1 [ENTRYPOINT]
    -------------------------------
        MethodName: Main (06000001)
        Flags     : [Public] [Static] [HideBySig] [ReuseSlot]  (00000096)
        RVA       : 0x00002050
        ImplFlags : [IL] [Managed]  (00000000)
        CallCnvntn: [DEFAULT]
        ReturnType: Void
        1 Arguments
            Argument #1:  SZArray String
```

```
        1 Parameters
            (1) ParamToken : (08000001) Name : args flags: [none] (00000000)

    Method #2
    -------------------------
        MethodName: .ctor (06000002)
        Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]
            [RTSpecialName] [.ctor]  (00001886)
        RVA       : 0x00002068
        ImplFlags : [IL] [Managed]  (00000000)
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        No arguments.


TypeRef #1 (01000001)
-------------------------
Token:             0x01000001
ResolutionScope:   0x23000001
TypeRefName:       System.Object
    MemberRef #1
    -------------------------
        Member: (0a000003) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        No arguments.

TypeRef #2 (01000002)
-------------------------
Token:             0x01000002
ResolutionScope:   0x23000001
TypeRefName:       System.Diagnostics.DebuggableAttribute
    MemberRef #1
    -------------------------
        Member: (0a000001) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        2 Arguments
            Argument #1:  Boolean
            Argument #2:  Boolean

TypeRef #3 (01000003)
-------------------------
Token:             0x01000003
ResolutionScope:   0x23000001
TypeRefName:       System.Console
    MemberRef #1
    -------------------------
        Member: (0a000002) WriteLine:
        CallCnvntn: [DEFAULT]
        ReturnType: Void
        1 Arguments
            Argument #1:  String

Assembly
-------------------------
    Token: 0x20000001
    Name : App
    Public Key    :
    Hash Algorithm : 0x00008004
```

```
    Major Version: 0x00000000
    Minor Version: 0x00000000
    Build Number: 0x00000000
    Revision Number: 0x00000000
    Locale: <null>
    Flags : [SideBySideCompatible]  (00000000)
    CustomAttribute #1 (0c000001)
    -------------------------------
        CustomAttribute Type: 0a000001
        CustomAttributeName: System.Diagnostics.DebuggableAttribute ::
            instance void .ctor(bool,bool)
        Length: 6
        Value : 01 00 00 01 00 00                              >                <
        ctor args: ( <can not decode> )


AssemblyRef #1
-------------------------------
    Token: 0x23000001
    Public Key or Token: b7 7a 5c 56 19 34 e0 89
    Name: mscorlib
    Major Version: 0x00000001
    Minor Version: 0x00000000
    Build Number: 0x00000c1e
    Revision Number: 0x00000000
    Locale: <null>
    HashValue Blob:
    Flags: [none] (00000000)


User Strings
-------------------------------
70000001 : ( 2) L"Hi"
```

Fortunately, ILDasm processes the metadata tables and combines information where appropriate so that you don't have to parse the raw table information. For example, in the dump above, you see that when ILDasm shows a TypeDef entry, the corresponding member definition information is shown with it before the first TypeRef entry is displayed.

You don't need to fully understand everything you see here. The important thing to remember is that App.exe contains a TypeDef whose name is **App**. This type identifies a public class that is derived from **System.Object** (a type referenced from another assembly). The **App** type also defines two methods: **Main** and **.ctor** (a constructor).

**Main** is a static, public method whose code is IL (vs. native CPU code, such as x86). **Main** has a **void** return type and takes a single argument, **args**, which is an array of **String**. The constructor method (always shown with a name of **.ctor**) is public and its code is also IL. The constructor has a **void** return type and has no arguments but has a **this** pointer, which refers to the object's memory that is to be constructed when the method is called.

I strongly encourage you to experiment using ILDasm. It can show you a wealth of information, and the more you understand what you're seeing, the better you'll understand the common language runtime and its capabilities. As you'll see, I use ILDasm quite a bit more in this book.

Just for fun, let's look at some statistics about the App.exe assembly. When you select ILDasm's View.Statistics menu item, the following information is displayed:

```
File size             : 3072
```

```
PE header size       : 512 (496 used)     (16.67%)
PE additional info   : 923                (30.05%)
Num.of PE sections   : 3
CLR header size      : 72                 ( 2.34%)
CLR meta-data size   : 520                (16.93%)
CLR additional info  : 0                  ( 0.00%)
CLR method headers   : 24                 ( 0.78%)
Managed code         : 18                 ( 0.59%)
Data                 : 828                (26.95%)
Unaccounted          : 175                ( 5.70%)

Num.of PE sections   : 3
  .text    - 1024
  .rsrc    - 1024
  .reloc   - 512

CLR meta-data size   : 520
  Module          -    1 (10 bytes)
  TypeDef         -    2 (28 bytes)     0 interfaces, 0 explicit layout
  TypeRef         -    3 (18 bytes)
  MethodDef       -    2 (28 bytes)     0 abstract, 0 native, 2 bodies
  MemberRef       -    3 (18 bytes)
  ParamDef        -    1 (6 bytes)
  CustomAttribute-    1 (6 bytes)
  Assembly        -    1 (22 bytes)
  AssemblyRef     -    1 (20 bytes)
  Strings         -  128 bytes
  Blobs           -   40 bytes
  UserStrings     -    8 bytes
  Guids           -   16 bytes
  Uncategorized   -  172 bytes

CLR method headers : 24
  Num.of method bodies  - 2
  Num.of fat headers    - 2
  Num.of tiny headers   - 0

Managed code : 18
  Ave method size - 9
```

Here you can see the size (in bytes) of the file and the size (in bytes and percentages) of the various parts that make up the file. For this very small App.cs application, the PE header and the metadata occupy the bulk of the file's size. In fact, the IL code occupies just 18 bytes. Of course, as an application grows, it will reuse most of its types and references to other types and assemblies, causing the metadata and header information to shrink considerably as compared to the overall size of the file.

## Combining Modules to Form an Assembly

The App.exe file discussed in the previous section is more than just a PE file with metadata; it is also an *assembly*. An assembly is a collection of one or more files containing type definitions and resource files. One of the assembly's files is chosen to hold a *manifest*. The manifest is another set of metadata tables that basically contain the name of the files that are part of the assembly. They also describe the assembly's version, culture, publisher, publicly exported types, and all the files that comprise the assembly.

The CLR operates on assemblies; that is, the CLR always loads the file that contains the manifest

metadata tables first and then uses the manifest to get the names of the other files that are in the assembly. Here are some characteristics of assemblies that you should remember:

- An assembly defines the reusable types.
- An assembly is marked with a version number.
- An assembly can have security information associated with it.

An assembly's individual files don't have these attributes—except for the file that contains the manifest metadata tables.

To package, version, secure, and use types, you must place them in modules that are part of an assembly. In most cases, an assembly consists of a single file, as the preceding App.exe example does. However, an assembly can also consist of multiple files: some PE files with metadata and some resource files such as .gif or .jpg files. It might help you to think of an assembly as a logical EXE or a DLL.

I'm sure that many of you reading this are wondering why Microsoft has introduced this new assembly concept. The reason is that an assembly allows you to decouple the logical and physical notions of reusable types. For example, an assembly can consist of several types. You could put the frequently used types in one file and the less frequently used types in another file. If your assembly is deployed by downloading it via the Internet, the file with the infrequently used types might not ever have to be downloaded to the client if the client never accesses the types. For example, an ISV specializing in UI controls might choose to implement Active Accessibility types in a separate module (to satisfy Microsoft's Logo requirements). Only users who require the additional accessibility features would require that this module be downloaded.

You configure an application to download assembly files by specifying a **codeBase** element (discussed in Chapter 3) in the application's configuration file. The **codeBase** element identifies a URL where all of an assembly's files can be found. When attempting to load an assembly's file, the CLR obtains the **codeBase** element's URL and checks the machine's download cache to see if the file is present. If it is, the file is loaded. If the file isn't in the cache, the CLR downloads the file from the URL into the cache. If the file can't be found, the CLR throws a **FileNotFoundException** exception at runtime.

I've identified three reasons to use multifile assemblies:

- You can partition your types among separate files, allowing for files to be incrementally downloaded as described in the Internet download scenario. Partitioning the types into separate files also allows for partial or piecemeal packaging and deployment for "shrink–wrapped" scenarios.
- You can add resource or data files to your assembly. For example, you could have a type that calculates some insurance information. This type might require access to some actuarial tables to make its computations. Instead of embedding the actuarial tables in your source code, you could use a tool (such as the assembly linker (AL.exe), discussed later) so that the data file is considered to be part of the assembly. By the way, this data file can be in any format: a text file, a Microsoft Excel spreadsheet, a Microsoft Word table, or whatever you like—as long as your application knows how to parse the file's contents.
- You can create assemblies consisting of types implemented in different programming languages. When you compile C# source code, the compiler produces a module. When you compile Visual Basic source code, the compiler produces a separate module. You can implement some types in C#, some types in Visual Basic, and other types in other languages. You can then use a tool to combine all these modules into a single assembly. To

developers using the assembly, the assembly just contains a bunch of types; developers won't even know that different programming languages were used. By the way, if you prefer, you can run ILDasm.exe on each of the modules to obtain an IL source code file. Then you can run ILAsm.exe and pass it all the IL source code files. ILAsm.exe will produce a single file containing all the types. This technique requires that your source code compiler produces IL–only code, so you can't use this technique with Visual C++, for example.

**Important** To summarize, an assembly is a unit of reuse, versioning, and security. It allows you to partition your types and resources into separate files so that you and consumers of your assembly get to determine which files to package together and deploy. Once the CLR loads the file containing the manifest, it can determine which of the assembly's other files contain the types and resources that the application is referencing. Anyone consuming the assembly is required to know only the name of the file containing the manifest; the file partitioning is then abstracted away from the consumer and may change in the future without breaking the application's behavior.

To build an assembly, you must select one of your PE files to be the keeper of the manifest. Or you can create a separate PE file that contains nothing but the manifest. Table 2–3 shows the manifest metadata tables that turn a managed module into an assembly.

Table 2–3: Manifest Metadata Tables

| Manifest Metadata Table Name | Description |
|---|---|
| AssemblyDef | Contains a single entry if this module identifies an assembly. The entry includes the assembly's name (without path and extension), version (major, minor, build, and revision), culture, flags, hash algorithm, and the publisher's public key (which can be **null**). |
| FileDef | Contains one entry for each PE and resource file that is part of the assembly. The entry includes the file's name and extension (without path), hash value, and flags. If this assembly consists only of its own file, the FileDef table has no entries. |
| ManifestResourceDef | Contains one entry for each resource that is part of the assembly. The entry includes the resource's name, flags (public, private), and an index into the FileDef table indicating the file that contains the resource file or stream. If the resource isn't a stand–alone file (such as .jpeg or a .gif), the resource is a stream contained within a PE file. For an embedded resource, the entry also includes an offset indicating the start of the resource stream within the PE file. |
| ExportedTypesDef | Contains one entry for each public type exported from all the assembly's PE modules. The entry includes the type's name, an index into the FileDef table (indicating which of this assembly's files implements the type), and an index into the TypeDef table. *Note*: To save file space, types |

| | |
|---|---|
| | exported from the file containing the manifest are not repeated in this table because the type information is available using the metadata's TypeDef table. |

The existence of a manifest provides a level of indirection between consumers of the assembly and the partitioning details of the assembly and makes assemblies self–describing. Also, note that the file containing the manifest knows which files are part of the assembly, but the individual files themselves aren't aware that they are part of an assembly.

**Note** The assembly file that contains the manifest also has an AssemblyRef table in it. This table contains an entry for all the assemblies referenced by all the assembly's files. This allows tools to open an assembly's manifest and see its set of referenced assemblies without having to open the assembly's other files. Again, the entries in the AssemblyRef table exist to make an assembly self–describing.

The C# compiler produces an assembly when you specify any of the following command–line switches: /**t[arget]:exe**, /**t[arget]:winexe**, or /**t[arget]:library**. All these switches cause the compiler to generate a single PE file that contains the manifest metadata tables. The resulting file is either a CUI executable, a GUI executable, or a DLL, respectively.

In addition to these switches, the C# compiler supports the /**t[arget]:module** switch. This switch tells the compiler to produce a PE file that doesn't contain the manifest metadata tables. The PE file produced is always a DLL PE file, and this file must be added to an assembly before the types within it can be accessed. When you use the /**t:module** switch, the C# compiler, by default, names the output file with an extension of .netmodule.

**Important** Unfortunately, the Visual Studio .NET integrated development environment (IDE) doesn't natively support the ability for you to create multifile assemblies. If you want to create multifile assemblies, you must resort to using command–line tools.

There are many ways to add a module to an assembly. If you're using the C# compiler to build a PE file with a manifest, you can use the /**addmodule** switch. To understand how to build a multifile assembly, let's assume that we have two source code files:

- RUT.cs, which contains rarely used types
- FUT.cs, which contains frequently used types

Let's compile the rarely used types into their own module so that users of the assembly won't need to deploy this module if they never access the rarely used types:

```
csc /t:module RUT.cs
```

This line causes the C# compiler to create a RUT.netmodule file. This file is a standard DLL PE file, but, by itself, the CLR can't load it.

Next let's compile the frequently used types into their own module. We'll make this module the keeper of the assembly's manifest because the types are used so often. In fact, because this module will now represent the entire assembly, I'll change the name of the output file to JeffTypes.dll instead of calling it FUT.dll:

```
csc /out:JeffTypes.dll /t:library /addmodule:RUT.netmodule FUT.cs
```

This line tells the C# compiler to compile the FUT.cs file to produce the JeffTypes.dll file. Because **/t:library** is specified, a DLL PE file containing the manifest metadata tables is emitted into the JeffTypes.dll file. The **/addmodule:RUT.netmodule** switch tells the compiler that RUT.netmodule is a file that should be considered part of the assembly. Specifically, the **/addmodule** switch tells the compiler to add the file to the FileDef manifest metadata table and to add RUT.netmodule's publicly exported types to the ExportedTypesDef manifest metadata table.

Once the compiler has finished all its processing, the two files shown in Figure 2–1 are created. The module on the right contains the manifest.

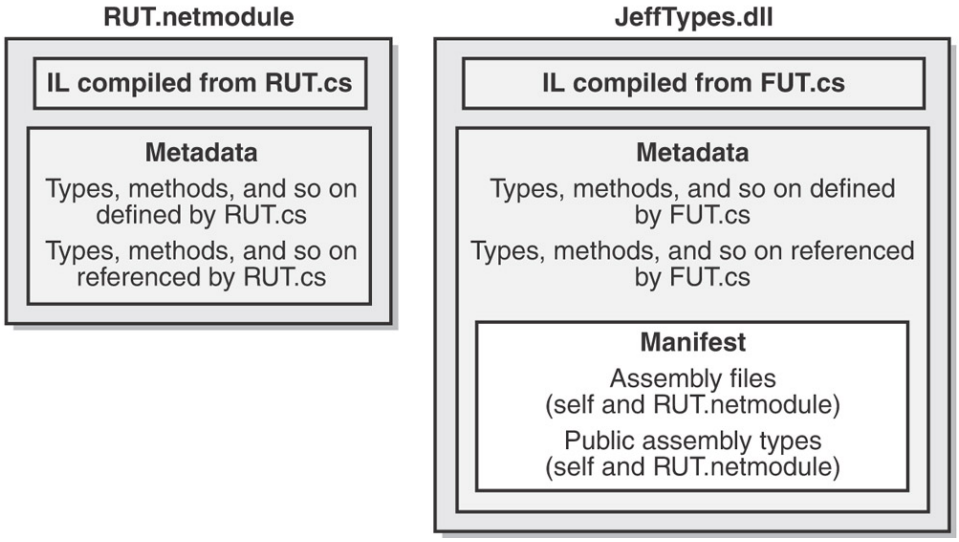



Figure 2–1 : A multifile assembly consisting of two managed modules, one with a manifest

The RUT.netmodule file contains the IL code generated by compiling RUT.cs. This file also contains metadata tables that describe the types, methods, fields, properties, events, and so on that are defined by RUT.cs. The metadata tables also describe the types, methods, and so on that are referenced by RUT.cs. The JeffTypes.dll is a separate file. Like RUT.netmodule, this file includes the IL code generated by compiling FUT.cs and also includes similar definition and reference metadata tables. However, JeffTypes.dll contains the additional manifest metadata tables, making JeffTypes.dll an assembly. The additional manifest metadata tables describe all the files that make up the assembly (the JeffTypes.dll file itself and the RUT.netmodule file). The manifest metadata tables also include all the public types exported from JeffTypes.dll and RUT.netmodule.

**Note** In reality, the manifest metadata tables don't actually include the types that are exported from the PE file that contains the manifest. The purpose of this optimization is to reduce the number of bytes required by the manifest information in the PE file. So statements like, "The manifest metadata tables also include all the public types exported from JeffTypes.dll and RUT.netmodule" aren't 100 percent accurate. However, this statement does accurately reflect what the manifest is logically exposing.

Once the JeffTypes.dll assembly is built, you can use ILDasm.exe to examine the metadata's manifest tables to verify that the assembly file does in fact have references to the RUT.netmodule file's types. If you build this project and then use ILDasm.exe to examine the metadata, you'll see the FileDef and ExportedTypesDef tables included in the output. Here's what those tables look like:

```
File #1
-------------------------

    Token: 0x26000001
    Name : RUT.netmodule
```

```
    HashValue Blob : 03 d4 09 ef 2d ac d3 4b  64 75 d7 81 cc 8e 88 7d  51
                     67 e2 5b
    Flags : [ContainsMetaData]  (00000000)


ExportedType #1
-------------------------

    Token: 0x27000001
    Name: ARarelyUsedType
    Implementation token: 0x26000001
    TypeDef token: 0x02000002
    Flags     : [Public] [AutoLayout] [Class] [AnsiClass]  (00100001)
```

From this, you can see that RUT.netmodule is a file considered to be part of the assembly. From the ExportedType table, you can see that there is a publicly exported type, **ARarelyUsedType**. The implementation token for this type is 0x26000001, which indicates that the type's IL code is contained in the RUT.netmodule file.

**Note** For the curious, metadata tokens are 4–byte values. The high byte indicates the type of token (0x01=TypeRef, 0x02=TypeDef, 0x26=FileRef, 0x27=ExportedType). For the complete list, see the **CorTokenType** enumerated type in the CorHdr.h file included with the .NET Framework SDK. The low three bytes of the token simply identify the row in the corresponding metadata table. For example, the implementation token 0x26000001 refers to the first row of the FileRef table. (Rows are numbered starting with 1, not 0.)

Any client code that consumes the JeffTypes.dll assembly's types must be built using the **/r[eference]:JeffTypes.dll** compiler switch. This switch tells the compiler to load the JeffTypes.dll assembly and all the files listed in its FileDef table. The compiler requires that all the assembly's files are installed and accessible. If you were to delete the RUT.netmodule file, the C# compiler would produce the following error: "fatal error CS0009: Metadata file 'C:\JeffTypes.dll' could not be opened—'Error importing module 'rut.netmodule' of assembly 'C:\JeffTypes.dll'—The system cannot find the file specified." This means that to build a new assembly, all the files from a referenced assembly *must* be present.

As the client code executes, it calls methods. When a method is called for the first time, the CLR detects what types the method references. The CLR then attempts to load the referenced assembly's file that contains the manifest. If the type being accessed is in this file, the CLR performs its internal bookkeeping, allowing the type to be used. If the manifest indicates that the referenced type is in a different file, the CLR attempts to load the necessary file, performs its internal bookkeeping, and allows the type to be accessed. The CLR loads assembly files only when a method referencing a type is called. This means that to run an application, all the files from a referenced assembly *do not* need to be present.

## Adding Assemblies to a Project Using the Visual Studio .NET IDE

If you're using the Visual Studio .NET IDE to build your project, you'll have to add any assemblies you want to reference to your project. To do so, open the Solution Explorer window, right–click on the project you want to add a reference to, and select the Add Reference menu item. This causes the Add Reference dialog box, shown in Figure 2–2, to appear.

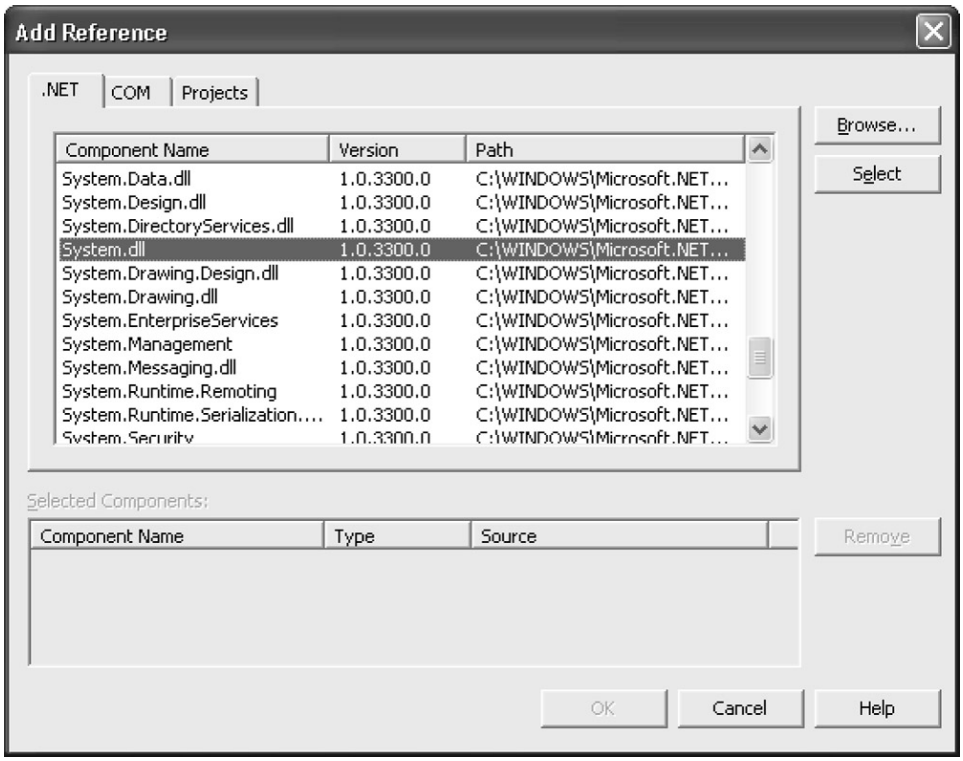


Figure 2–2 : Add Reference dialog box in Visual Studio .NET

To have your project reference a managed assembly, select the desired assembly from the list. If the assembly you want isn't in the list, select the Browse button to navigate to the desired assembly (file containing a manifest) to add the assembly reference. The COM tab on the Add Reference dialog box allows an unmanaged COM server to be accessed from within managed source code. The Projects tab allows the current project to reference an assembly that is created by another project in the same solution.

To make your own assemblies appear in the .NET tab's list, add the following subkey to the registry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\
AssemblyFolders\MyLibName
```

**MyLibName** is a unique name that you create—Visual Studio doesn't display this name. After creating the subkey, change its default string value so that it refers to a directory path (such as "C:\Program Files\MyLibPath") containing your assembly's files.

## Using the Assembly Linker

Instead of using the C# compiler, you might want to create assemblies using the Assembly Linker utility, AL.exe. The Assembly Linker is useful if you want to create an assembly consisting of modules built from different compilers (if your compiler doesn't support the equivalent of C#'s **/addmodule** switch) or perhaps if you just don't know your assembly packaging requirements at build time. You can also use AL.exe to build resource–only assemblies (called *satellite* assemblies, which I'll talk about again later in the chapter), which are typically used for localization purposes.

The AL.exe utility can produce an EXE or a DLL PE file that contains nothing but a manifest describing the types in other modules. To understand how AL.exe works, let's change the way the JeffTypes.dll assembly is built:

```
csc /t:module RUT.cs
csc /t:module FUT.cs
```

```
al  /out:JeffTypes.dll /t:library FUT.netmodule RUT.netmodule
```

Figure 2–3 shows the files that result from executing these statements.

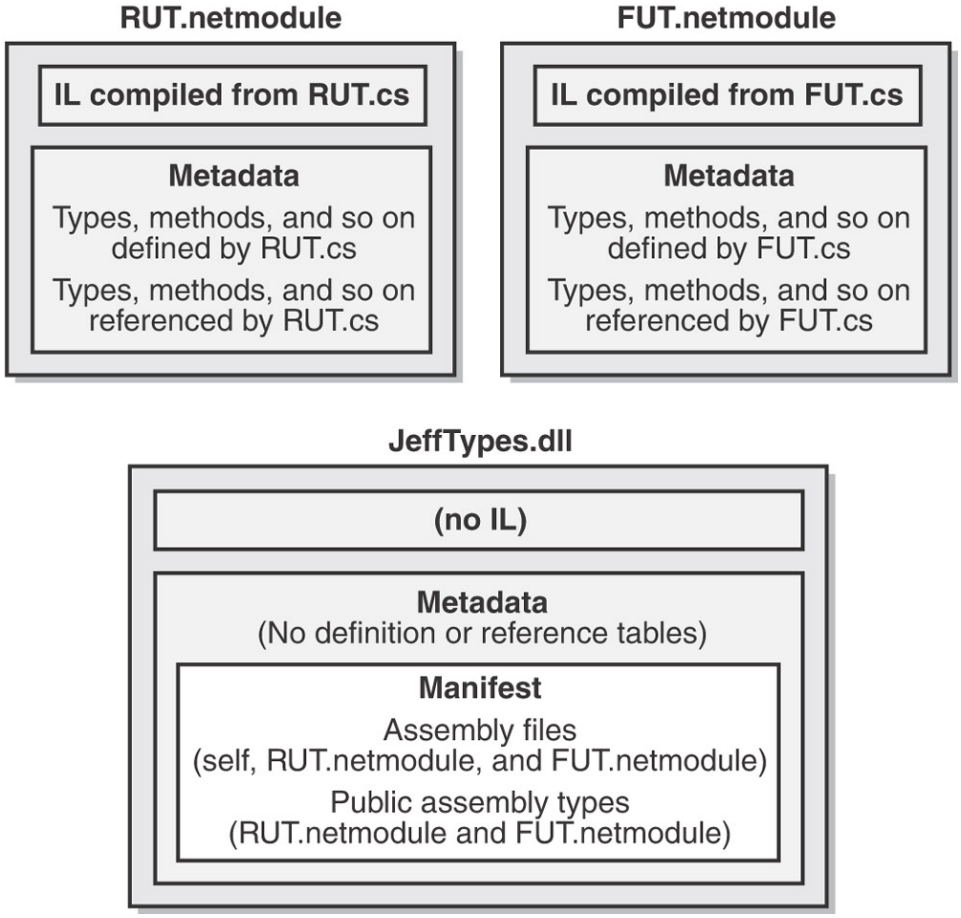



Figure 2–3 : A multifile assembly consisting of three managed modules, one with a manifest
In this example, two separate modules, RUT.netmodule and FUT.netmodule, are created that are not themselves assemblies (because they don't contain manifest metadata tables). Then a third file is produced: JeffTypes.dll, which is a small DLL PE file (because of the /**t[arget]:library** switch) that contains no IL code but has manifest metadata tables indicating that RUT.netmodule and FUT.netmodule are part of the assembly. The resulting assembly consists of three files: JeffTypes.dll, RUT.netmodule, and FUT.netmodule. The Assembly Linker has no way to combine multiple files into a single file.

The AL.exe utility can also produce CUI and GUI PE files (using the /**t[arget]:exe** or /**t[arget]:winexe** command–line switch), but this is very unusual since it would mean that you'd have an EXE PE file with just enough IL code in it to call a method in another module. The Assembly Linker generates this IL code when you call AL.exe using the /**main** command–line switch.

```
csc /t:module /r:JeffTypes.dll App.cs
al  /out:App.exe /t:exe /main:App.Main app.netmodule
```

Here the first line builds the App.cs file into a module. The second line produces a small App.exe PE file that contains the manifest metadata tables. In addition, there is a small global function emitted by AL.exe because of the /**main:App.Main** switch. This function, **__EntryPoint**, contains the following IL code:

```
.method privatescope static void __EntryPoint() il managed
{
  .entrypoint
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  tail.
  IL_0002:  call       void [.module 'App.mod']App::Main()
  IL_0007:  ret
} // end of method 'Global Functions::__EntryPoint'
```

As you can see, this code simply calls the **Main** method contained in the **App** type defined in the App.netmodule file.

The /**main** switch in AL.exe isn't that useful because it's unlikely that you'd ever create an assembly for an application where the application's entry point isn't in the PE file that contains the manifest metadata tables. I mention the switch here only to make you aware of its existence.

### Including Resource Files in the Assembly

When using AL.exe to create an assembly, you can add resource files (non–PE files) to the assembly by using the /**embed[resource]** switch. This switch takes a file (any file) and embeds the file's contents into the resulting PE file.The manifest's ManifestResourceDef table is updated to reflect the existence of the resources.

AL.exe also supports a /**link[resource]** switch, which also takes a file containing resources. However, the /**link[resource]** switch updates the manifest's ManifestResourceDef and FileDef tables, indicating that the resource exists and identifying which of the assembly's files contain it. The resource file is not embedded into the assembly PE file; it remains separate and must be packaged and deployed with the other assembly files.

Like AL.exe, CSC.exe also allows you to combine resources into an assembly produced by the C# compiler. The C# compiler's /**resource** switch embeds the specified resource file into the resulting assembly PE file, updating the ManifestResourceDef table. The compiler's /**linkresource** switch adds an entry to the ManifestResourceDef and the FileDef manifest tables to refer to a stand–alone resource file.

One last note about resources: it's possible to embed standard Win32 resources into an assembly. You can do this easily by specifying the pathname of a .res file with the /**win32res** switch when using either AL.exe or CSC.exe. In addition, you can quickly and easily embed a standard Win32 icon resource into an assembly file by specifying the pathname of an .ico file with the /**win32icon** switch when using either AL.exe or CSC.exe. The typical reason that an icon is embedded is so that Explorer can show an icon for a managed executable file.

## Assembly Version Resource Information

When AL.exe or CSC.exe produces a PE file assembly, it also embeds into the PE file a standard Win32 Version resource. Users can examine this resource by viewing the file's properties. Figure 2–4 shows the Version page of the JeffTypes.dll Properties dialog box.

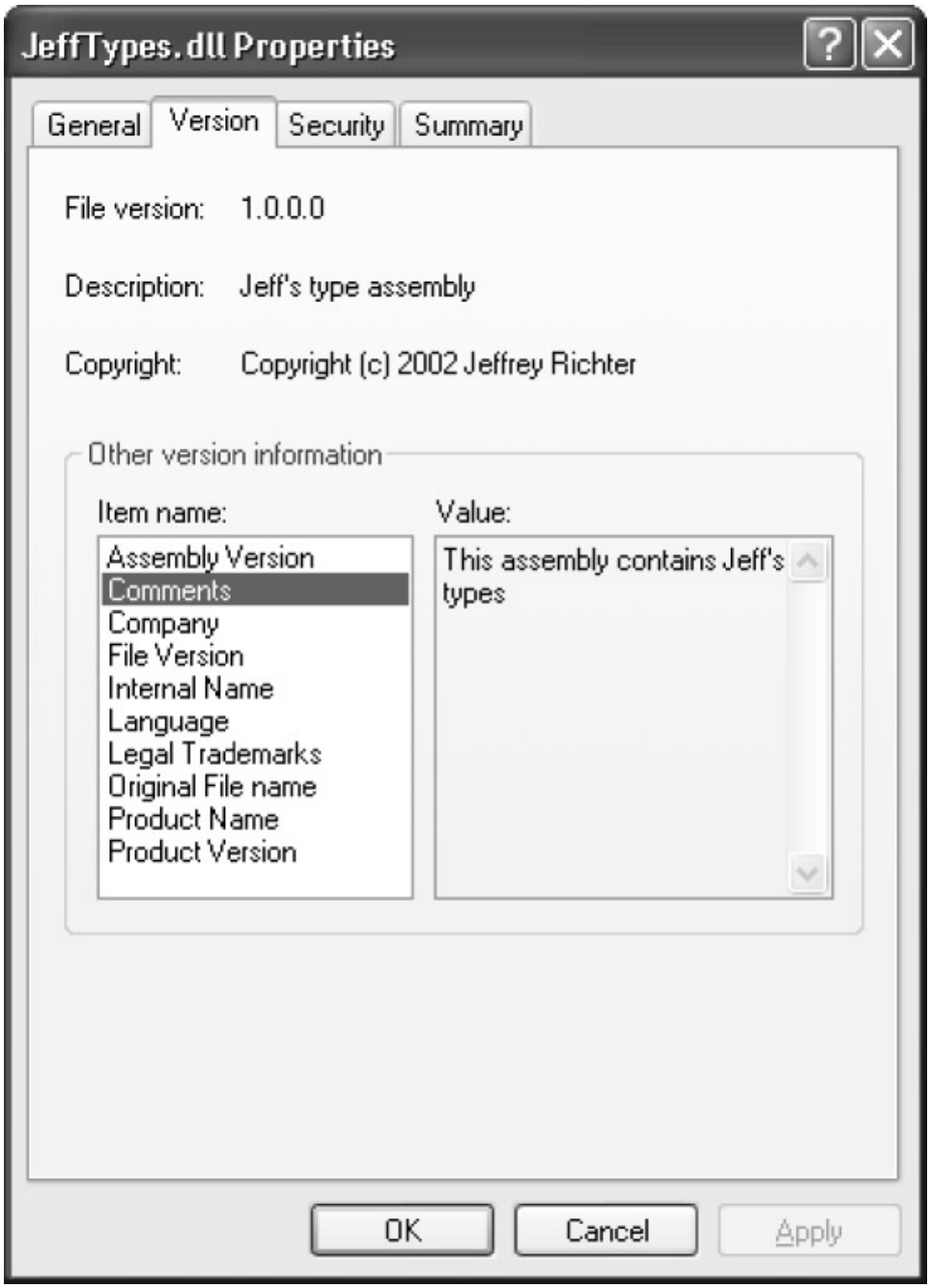



Figure 2–4 : Version tab of the JeffTypes.dll Properties dialog box

In addition, you can use the resource editor in Visual Studio .NET, shown in Figure 2–5, to view/modify the version resource fields.

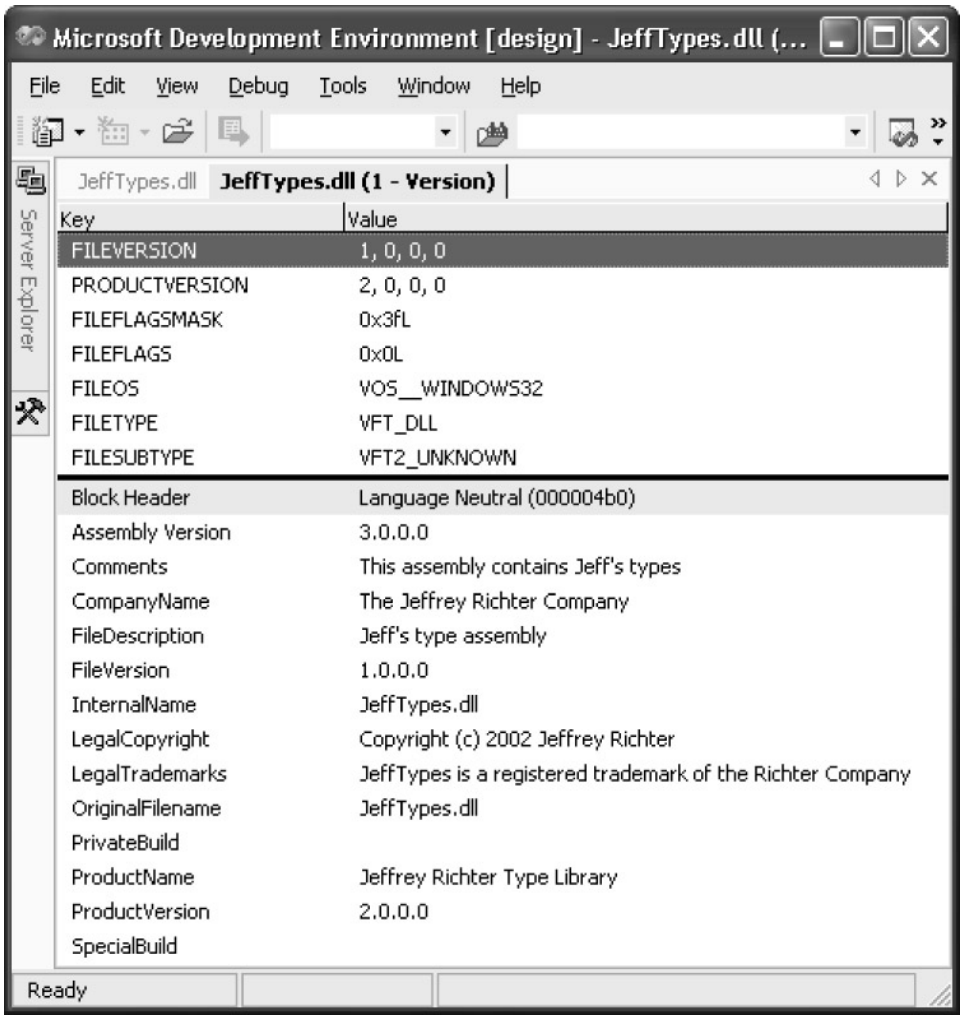


Figure 2–5 : Resource editor in Visual Studio .NET

When building an assembly, you should set the version resource fields using custom attributes that you apply at the assembly level in your source code. Here's what the code that produced the version information in Figure 2–5 looks like:

```
using System.Reflection;

// Set the version CompanyName, LegalCopyright & LegalTrademarks fields
[assembly:AssemblyCompany("The Jeffrey Richter Company")]
[assembly:AssemblyCopyright("Copyright (c) 2002 Jeffrey Richter")]
[assembly:AssemblyTrademark(
   "JeffTypes is a registered trademark of the Richter Company")]

// Set the version ProductName & ProductVersion fields
[assembly:AssemblyProduct("Jeffrey Richter Type Library")]
[assembly:AssemblyInformationalVersion("2.0.0.0")]

// Set the version FileVersion, AssemblyVersion,
// FileDescription, and Comments fields
[assembly:AssemblyFileVersion("1.0.0.0")]
[assembly:AssemblyVersion("3.0.0.0")]
[assembly:AssemblyTitle("Jeff's type assembly")]
[assembly:AssemblyDescription("This assembly contains Jeff's types")]

// Set the culture (discussed later in the "Culture" section)
[assembly:AssemblyCulture("")]
```

Table 2–4 shows the version resource fields and the custom attributes that correspond to them. If you're using AL.exe to build your assembly, you can use command–line switches to set this information instead of using the custom attributes. The second column in Table 2–4 shows the AL.exe command–line switch that corresponds to each version resource field. Note that the C# compiler doesn't offer these command–line switches and that, in general, using custom attributes is the preferred way to set this information.

Table 2–4: Version Resource Fields and Their Corresponding AL.exe Switches and Custom Attributes

| Version Resource | AL.exe Switch | Custom Attribute/Comment |
|---|---|---|
| FILEVERSION | **/fileversion** | **System.Reflection.AssemblyFileVersionAttribute–FileVersionAttribute** |
| PRODUCTVERSION | /productversion | System.Reflection.AssemblyInformationalVersionAttribute |
| FILEFLAGSMASK | (none) | Always set to VS_FFI_FILEFLAGSMASK (defined in WinVer.h as 0x0000003F) |
| FILEFLAGS | (none) | Always 0 |
| FILEOS | (none) | Currently always **VOS__WINDOWS32** |
| FILETYPE | **/target** | Set to **VFT_APP** if **/target:exe** or **/target:winexe** is specified; set to **VFT_DLL if /target:library** is specified |
| FILESUBTYPE | (none) | Always set to **VFT2_UNKNOWN** (This field has no meaning for **VFT_APP** and **VFT_DLL**.) |
| AssemblyVersion | **/version** | System.Reflection.Assembly–VersionAttribute |
| Comments | **/description** | **System.Reflection.Assembly–DescriptionAttribute** |
| CompanyName | **/company** | System.Reflection.AssemblyCompanyAttribute |
| FileDescription | **/title** | **System.Reflection.AssemblyTitleAttribute** |
| FileVersion | **/version** | **System.Reflection.AssemblyVersionAttribute** |
| InternalName | **/out** | Set to the name of the output file specified (without the extension) |
| LegalCopyright | **/copyright** | **System.Reflection.AssemblyCopyrightAttribute** |
| LegalTrademarks | **/trademark** | **System.Reflection.AssemblyTrademarkAttribute** |
| OriginalFilename | /out | Set to the name of the output file (without a path) |
| PrivateBuild | (none) | Always blank |
| ProductName | **/product** | **System.Reflection.AssemblyProductAttribute** |
| ProductVersion | **/productversion** | **System.Reflection.AssemblyInformationalVersionAttribute** |
| SpecialBuild | (none) | Always blank |

**Important** When you create a new C# project in Visual Studio .NET, an AssemblyInfo.cs file is automatically created for you. This file contains all the assembly attributes described in this section plus a few additional attributes that I'll cover in Chapter 3. You can simply open the AssemblyInfo.cs file and modify your assembly–specific information. The file that Visual Studio .NET creates for you has some problems that I'll go over later in this chapter. In a real production project, you *must* modify the contents of this file.

## Version Numbers

In the previous section, you saw that several version numbers can be applied to an assembly. All these version numbers have the same format: each consists of four period–separated parts, as shown in Table 2–5.

Table 2–5: Format of Version Numbers

| Part | Major Number | Minor Number | Build Number | Revision Number |
|---|---|---|---|---|
| Example: | 2 | 5 | 719 | 2 |

Table 2–5 shows an example of a version number: 2.5.719.2. The first two numbers make up the "public perception" of the version. The public will think of this example as version 2.5 of the assembly. The third number, 719, indicates the build of the assembly. If your company builds its assembly every day, you should increment the build number each day as well. The last number, 2, indicates the revision of the build. If for some reason your company has to build an assembly twice in one day, maybe to resolve a hot bug that is halting other work, then the revision number should be incremented.

Microsoft uses this version–numbering scheme, and it's simply a recommendation; you're welcome to devise your own number–versioning scheme if you prefer. The only assumption the CLR makes is that bigger numbers indicate later versions.

You'll notice that an assembly has three version numbers associated with it. This is very unfortunate and leads to a lot of confusion. Let me explain each version number's purpose and how it is expected to be used:

- **AssemblyFileVersion** This version number is stored in the Win32 version resource. This number is informational only; the CLR doesn't examine or care about this version number in any way. Typically, you set the major and minor parts to represent the version you want the public to see. Then you increment the build and revision parts each time a build is performed. Ideally, Microsoft's tool (such as CSC.exe or AL.exe) would automatically update the build and revision numbers for you (based on the data/time when the build was performed), but unfortunately they don't. This version number can be seen when using Windows Explorer and is used to determine exactly when an assembly file was built.
- **AssemblyInformationalVersionAttribute** This version number is also stored in the Win32 version resource, and again, this number is informational only; the CLR doesn't examine or care about it in any way. This version number exists to indicate the version of the product that includes this assembly. For example, Version 2.0 of MyProduct might contain several assemblies; one of these assemblies is marked as Version 1.0 since it's a new assembly that didn't ship in Version 1.0 of MyProduct. Typically, you set the major and minor parts of this version number to represent the public version of your product. Then you increment the build and revision parts each time you package a complete product with all its assemblies.
- **AssemblyVersion** This version number is stored in the AssemblyDef manifest metadata table. The CLR uses this version number when binding to strongly named assemblies (discussed in Chapter 3). This number is extremely important and is used to uniquely identify an assembly. When starting to develop an assembly, you should set the major, minor, build, and revision numbers and shouldn't change them until you're ready to begin work on the next deployable version of your assembly. When you build an assembly, this version number in the referenced assembly is embedded in the AssemblyRef table's entry. This means that an assembly is tightly bound to a specific version of a reference assembly.

**Important** The CSC.exe and AL.exe tools support the ability to automatically increment the assembly version number with each build. This feature is a bug and shouldn't be used because changing the assembly version number will break any assemblies that reference this assembly. The AssemblyInfo.cs file that Visual Studio .NET automatically creates for you when you create a new project is in error: it sets the **AssemblyVersion** attribute so that its major and minor parts are 1.0 and that the build and revision parts are automatically updated by the compiler. You should definitely modify this file and hard–code all four parts of the assembly version number.

## Culture

Like version numbers, assemblies also have a culture as part of their identity. For example, I could have an assembly that is strictly for German, another assembly for Swiss German, another assembly for U.S. English, and so on. Cultures are identified via a string that contains a primary and a secondary tag (as described in RFC1766). Table 2–6 shows some examples.

Table 2–6: Examples of Assembly Culture Tags

| Primary Tag | Secondary Tag | Culture |
|---|---|---|
| de | (none) | German |
| de | AT | Austrian German |
| de | CH | Swiss German |
| en | (none) | English |
| en | GB | British English |
| en | US | U.S. English |

In general, if you create an assembly that contains code, you don't assign a culture to it. This is because code doesn't usually have any culture–specific assumptions built into it. An assembly that isn't assigned a culture is referred to as being *culture neutral*.

If you're designing an application that has some culture–specific resources to it, Microsoft highly recommends that you create one assembly that contains your code and your application's default (or fallback) resources. When building this assembly, don't specify a specific culture. This is the assembly that other assemblies will reference to create and manipulate types.

Now you can create one or more separate assemblies that contain only culture–specific resources—no code at all. Assemblies that are marked with a culture are called *satellite assemblies*. For these satellite assemblies, assign a culture that accurately reflects the culture of the resources placed in the assembly. You should create one satellite assembly for each culture you intend to support.

You'll usually use the AL.exe tool to build a satellite assembly. You won't use a compiler because the satellite assembly should have no code contained within it. When using AL.exe, you specify the desired culture using the /**c[ulture]:text** switch, where text is a string such as "en–US" representing U.S. English. When you deploy a satellite assembly, you should place it in a subdirectory whose name matches the culture text. For example, if the application's base directory is C:\MyApp, then the U.S. English satellite assembly should be placed in the C:\MyApp\en–US subdirectory. At runtime, you access a satellite assembly's resources using the **System.Resources.ResourceManager** class.

**Note** Although discouraged, it is possible to create a satellite

assembly that contains code. If you prefer, you can specify the culture using the **System.Reflection.AssemblyCultureAttribute** custom attribute instead of using AL.exe's /**culture** switch, for example, as shown here:

```
// Set assembly's culture to Swiss German
[assembly:AssemblyCulture("de-CH")]
```

Normally, you shouldn't build an assembly that references a satellite assembly. In other words, an assembly's AssemblyRef entries should all refer to culture-neutral assemblies. If you want to access types or members contained in a satellite assembly, you should use reflection techniques as discussed in Chapter 20.

## Simple Application Deployment (Privately Deployed Assemblies)

Throughout this chapter, I've explained how you build modules and how you combine those modules into an assembly. At this point, I'm ready to package and deploy all the assemblies so that users can run the application.

Assemblies don't dictate or require any special means of packaging. The easiest way to package a set of assemblies is simply to copy all the files directly. For example, you could put all the assembly files on a CD-ROM disk and ship it to the user with a batch file setup program that just copies the files from the CD to a directory on the user's hard drive. Because the assemblies include all the dependant assembly references and types, the user can just run the application and the runtime will look for referenced assemblies in the application's directory. No modifications to the registry or to Active Directory are necessary for the application to run. To uninstall the application, just delete all the files—that's it!

Of course, you can package and install the assembly files using other mechanisms, such as .cab files (typically used for Internet download scenarios to compress files and reduce download times). You can also package the assembly files into an MSI file for use by the Windows Installer service (MSIExec.exe). Using MSI allows assemblies to be installed on demand the first time the CLR attempts to load the assembly. This feature isn't new to MSI; it can perform the same demand-load functionality for unmanaged EXE and DLL files as well.

**Note** Using a batch file or some other simple "installation software" will get an application onto the user's machine; however, you'll need more sophisticated installation software to create shortcut links on the user's desktop, Start menu, and Quick Launch toolbar. Also, you can easily back up and restore the application or move it from one machine to another, but the various shortcut links will require special handling. Future versions of Windows may improve this story.

Assemblies that are deployed to the same directory as the application are called *privately deployed assemblies* because the assembly files aren't shared with any other application (unless it's also deployed to the same directory). Privately deployed assemblies are a big win for developers, end-users, and administrators because they can simply be copied to an application's base directory and the CLR will load them and execute the code in them. In addition, an application can be uninstalled by simply deleting the assemblies in its directory. This allows simple backup and restore

to work as well.

This simple install/move/uninstall story is possible because each assembly has metadata indicating which referenced assembly should be loaded; no registry settings or Active Directory settings are required.

In addition, the referencing assembly scopes every type. This means that an application always binds to the exact type that it was built and tested with; the CLR can't load a different assembly that just happens to provide a type with the same name. This is different from COM, where types are recorded in the registry, making them available to any application running on the machine.

In Chapter 3, I'll discuss how to deploy shared assemblies that are accessible by multiple applications.

## Simple Administrative Control (Configuration)

The user or the administrator can best determine some aspects of an application's execution. For example, an administrator might decide to move an assembly's files on the user's hard disk or to override information contained in the assembly's manifest. Other scenarios also exist related to versioning and remoting; I'll talk about some of these in Chapter 3.

To allow administrative control over an application, a configuration file can be placed in the application's directory. An application's publisher can create and package this file. The setup program would then install this configuration file in the application's base directory. In addition, the machine's administrator or an end–user could create or modify this file. The CLR interprets the content of this file to alter its policies for locating and loading assembly files.

These configuration files contain XML and can be associated with an application or with the machine. Using a separate file (vs. registry settings) allows the file to be easily backed up and also allows the administrator to copy the application to another machine: just copy the necessary files and the administrative policy is copied too.

In Chapter 3, we'll explore this configuration file in more detail. But I want to give you a taste of it now. Let's say that the publisher of an application wants its application deployed with the JeffTypes assembly files in a different directory than the application's assembly file. The desired directory structure looks like this:

```
AppDir directory (contains the application's assembly files)
   App.exe
   App.exe.config (discussed below)

   AuxFiles subdirectory (contains JeffTypes' assembly files)
      JeffTypes.dll
      FUT.netmodule
      RUT.netmodule
```

Since the JeffTypes files are no longer in the application's base directory, the CLR won't be able to locate and load these files; running the application will cause a **System.IO.FileNotFoundException** exception to be thrown. To fix this, the publisher creates an XML configuration file and deploys it to the application's base directory. The name of this file must be the name of the application's main assembly file with a .config extension: App.exe.config, for this example. This configuration file should look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="AuxFiles" />
      </assemblyBinding>
   </runtime>
</configuration>
```

Whenever the CLR attempts to locate an assembly file, it always looks in the application's directory first, and if it can't find the file there, it looks in the AuxFiles subdirectory. You can specify multiple semicolon–delimited paths for the probing element's **privatePath** attribute. Each path is considered relative to the application's base directory. You can't specify an absolute or a relative path identifying a directory that is outside the application's base directory. The idea is that an application can control its directory and its subdirectories but has no control over other directories.

By the way, you can write code that opens and parses the information contained in a configuration file. This allows your application to define settings that an administrator or a user can create and persist in the same file as all the application's other settings. You use the classes defined in the **System.Configuration** namespace to manipulate a configuration file at runtime.

The name and location of this XML configuration is different depending on the application type.

---

Probing for Assembly Files

When the CLR needs to locate an assembly, it scans several subdirectories. Here is the order in which directories are probed for a culture–neutral assembly:

```
AppBase\AsmName.dll
AppBase\AsmName\AsmName.dll
AppBase\privatePath1\AsmName.dll
AppBase\privatePath1\AsmName\AsmName.dll
AppBase\privatePath2\AsmName.dll
AppBase\privatePath2\AsmName\AsmName.dll
Ã
```

In the example above, no configuration file would be needed if the JeffTypes assembly files were deployed to a subdirectory called JeffTypes since the CLR would automatically scan for a subdirectory whose name matches the name of the assembly being searched for.

If the assembly can't be found in any of the preceding subdirectories, the CLR starts all over, using an .exe extension instead of a .dll extension. If the assembly still can't be found, a **FileNotFoundException** is thrown.

For satellite assemblies, the same rules are followed except that the assembly is expected to be in a subdirectory of the application base directory whose name matches the culture. For example, if AsmName.dll has a culture of "en–US" applied to it, the following directories are probed:

```
AppBase\en-US\AsmName.dll
AppBase\en-US\AsmName\AsmName.dll
AppBase\en-US\privatePath1\AsmName.dll
AppBase\en-US\privatePath1\AsmName\AsmName.dll
AppBase\en-US\privatePath2\AsmName.dll
AppBase\en-US\privatePath2\AsmName\AsmName.dll
Ã
```

Again, if the assembly can't be found in any of the subdirectories listed here, the CLR checks the same set of assemblies looking for an .exe file instead of a .dll file.

---

- For executable applications (EXEs), the configuration file must be in the application's base directory and it must be the name of the EXE file with ".config" appended to it.
- For ASP.NET Web Forms and XML Web service applications, the file must be in the Web application's virtual root directory and is always named Web.config. In addition, subdirectories can also contain their own Web.config file and the configuration settings are inherited. For example, a Web application located at *http://www.Wintellect.com/Training* would use the settings in the Web.config files contained in the virtual root directory and in its Training subdirectory.
- For assemblies containing client–side controls hosted by Microsoft Internet Explorer, the HTML page must contain a link tag whose **rel** attribute is set to "Configuration" and whose **href** attribute is set to the URL of the configuration file, which can be given any name. Here's an example: <LINK REL=Configuration HREF=http://www.Wintellect.com/Controls.config> For more information see the .NET Framework documentation.

As mentioned at the beginning of this section, configuration settings apply to a particular application and to the machine. When you install the .NET Framework, it creates a Machine.config file. There is one Machine.config file per version of the CLR you have installed on the machine. In the future, it will be possible to have multiple versions of the .NET Framework installed on a single machine simultaneously.

The Machine.config file is located in the following directory:

```
C:\WINDOWS\Microsoft.NET\Framework\version\CONFIG
```

Of course, C:\WINDOWS identifies your Windows directory, and *version* is a version number identifying a specific version of the .NET Framework.

Settings in the Machine.config file override settings in an application–specific configuration file. An administrator can create a machine–wide policy by modifying a single file. Normally, administrators and users should avoid modifying the Machine.config file because this file has many settings related to various things, making it much more difficult to navigate. Plus, you want the application's settings to be backed up and restored, and keeping an application's settings in the application–specific configuration file enables this.

Because editing an XML configuration file is a little unwieldy, Microsoft's .NET Framework team produced a GUI tool to help. The GUI tool is implemented as a Microsoft Management Console (MMC) snap–in, which means that it isn't available when running on a Windows 98, Windows 98 Standard Edition, or Windows Me machine. You can find the tool by opening Control Panel, selecting Administrative Tools, and then selecting the Microsoft .NET Framework Configuration tool. In the window that appears, you can traverse the tree's nodes until you get to the Applications node, as shown in Figure 2–6.

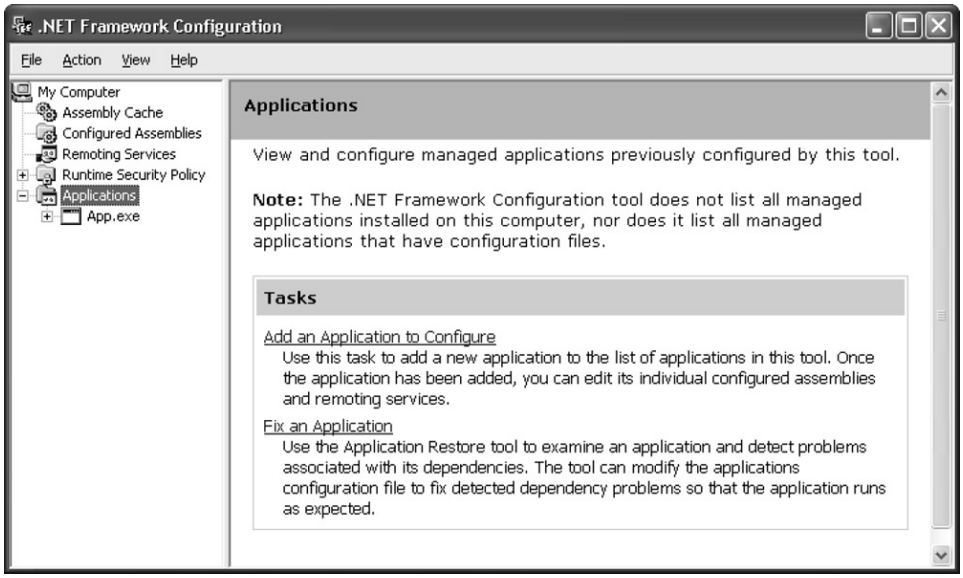



Figure 2–6 : Applications node of the Microsoft .NET Framework Configuration tool

From the Applications node, you can select the Add An Application To Configure link that appears in the right–hand pane. This will invoke a wizard that will prompt you for the pathname of the executable file you want to create an XML configuration file for. After you've added an application, you can also use this to alter its configuration file. Figure 2–7 shows the tasks that you can perform to an application.

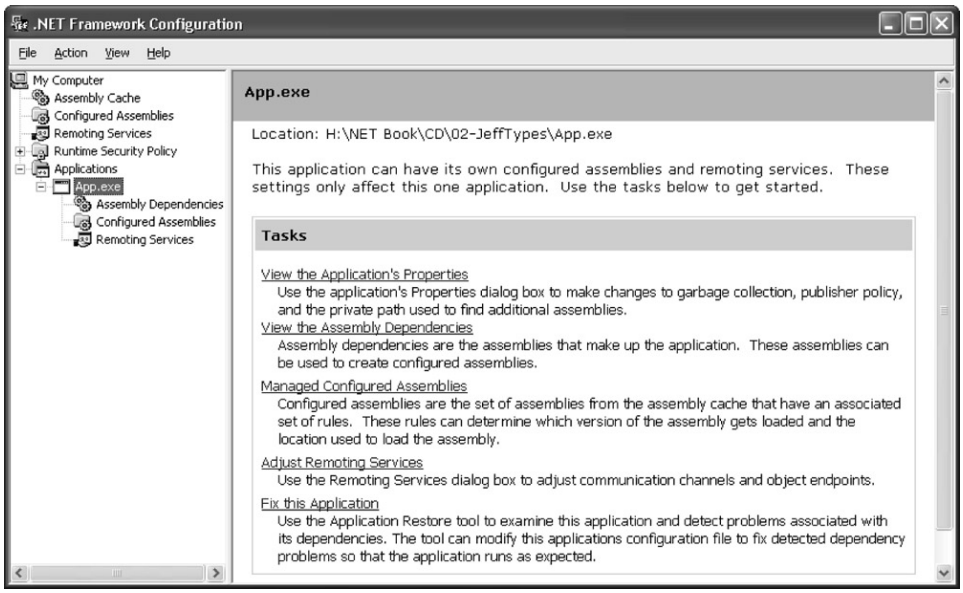



Figure 2–7 : Configuring an application using the Microsoft .NET Framework Configuration tool

I'll discuss configuration files a bit more in Chapter 3.

# Chapter 3: Shared Assemblies

## Overview

In Chapter 2, I talked about the steps required to build, package, and deploy an assembly. I focused on what's called *private deployment,* where assemblies are placed in the application's base directory (or a subdirectory thereof) for the application's sole use. Deploying assemblies privately gives a company a large degree of control over the naming, versioning, and behavior of the assembly.

In this chapter, I'll concentrate on creating assemblies that multiple applications can access. The assemblies that ship with the Microsoft .NET Framework are an excellent example of globally deployed assemblies because almost all managed applications use types defined by Microsoft in the .NET Framework Class Library (FCL).

As I mentioned in Chapter 2, Windows has a reputation for being unstable. The main reason for this reputation is the fact that applications are built and tested using code implemented by someone else. After all, when you write an application for Windows, your application is calling into code written by Microsoft developers. Also, a large number of companies out there make controls that application developers can incorporate into their own applications. In fact, the .NET Framework encourages this, and lot more control vendors will likely pop up over time.

As time marches on, Microsoft developers and control developers modify their code: they fix bugs, add features, and so on. Eventually, the new code makes its way onto the user's hard disk. The user's applications that were previously installed and working fine are no longer using the same code that the applications were built and tested with. As a result, the applications' behavior is no longer predictable, which contributes to the instability of Windows.

File versioning is a very difficult problem to solve. In fact, I assert that if you take a file and change just one bit in the file from a 0 to a 1 or from a 1 to a 0, there's absolutely no way to guarantee that code that used the original file will now work just as well if it uses the new version of the file. One of the reasons why this statement is true is that a lot of applications exploit bugs—either knowingly or unknowingly. If a later version of a file fixes a bug, the application no longer runs as expected.

So here's the problem: how do you fix bugs and add features to a file and also guarantee that you don't break some application? I've given this question a lot of thought and have come to one conclusion: it's just not possible. But, obviously, this answer isn't good enough. Files will ship with bugs, and developers will always want to add new features. There must be a way to distribute new files with the hope that the applications will work just fine. And if the application doesn't work fine, there has to be an *easy* way to restore the application to its "last–known good state."

In this chapter, I'll explain the infrastructure that the .NET Framework has in place to deal with versioning problems. Let me warn you: what I'm about to describe is complicated. I'm going to talk about a lot of algorithms, rules, and policies that are built into the common language runtime (CLR). I'm also going to mention a lot of tools and utilities that the application developer must use. This stuff is complicated because, as I've mentioned, the versioning problem is difficult to address and to solve.

## Two Kinds of Assemblies, Two Kinds of Deployment

The .NET Framework supports two kinds of assemblies: *weakly named assemblies* and *strongly named assemblies*.

**Important** By the way, you won't find the term *weakly named assembly* in any of the .NET Framework documentation. Why? Because I made it up. In fact, the documentation has no term to identify a weakly named assembly. I decided to coin the term so that I can talk about assemblies without any ambiguity as to what kind of assembly I'm referring to.

Weakly named assemblies and strongly named assemblies are structurally identical—that is, they use the same portable executable (PE) file format, PE header, CLR header, metadata, and manifest tables that we examined in Chapter 2. And you use the same tools, such as the C# compiler and AL.exe, to build both kinds of assemblies. The real difference between weakly named and strongly named assemblies is that a strongly named assembly is signed with a publisher's public/private key pair that uniquely identifies the assembly's publisher. This key pair allows the assembly to be uniquely identified, secured, and versioned, and it allows the assembly to be deployed anywhere on the user's hard disk or even on the Internet. This ability to uniquely identify an assembly allows the CLR to enforce certain "known to be safe" policies when an application tries to bind to a strongly named assembly. This chapter is dedicated to explaining what strongly named assemblies are and what policies the CLR applies to them.

An assembly can be deployed in two ways: privately or globally. A privately deployed assembly is an assembly that is deployed in the application's base directory or one of its subdirectories. A weakly named assembly can be deployed only privately. I talked about privately deployed assemblies in Chapter 2. A globally deployed assembly is an assembly that is deployed into some well–known location that the CLR knows to look in when it's searching for an assembly. A strongly named assembly can be deployed privately or globally. I'll explain how to create and deploy strongly named assemblies in this chapter. Table 3–1 summarizes the kinds of assemblies and the ways that they can be deployed.

Table 3–1: How Weakly and Strongly Named Assemblies Can Be Deployed

| Kind of Assembly | Can Be Privately Deployed? | Can Be Globally Deployed? |
|---|---|---|
| Weakly named | Yes | No |
| Strongly named | Yes | Yes |

## Giving an Assembly a Strong Name

If multiple applications are going to access an assembly, the assembly must be placed in a well–known directory and the CLR must know to look in this directory automatically when a reference to the assembly is detected. However, we have a problem: two (or more) companies could produce assemblies that have the same filename. Then, if both of these assemblies get copied into the same well–known directory, the last one installed wins and all the applications that were using the old assembly no longer function as desired. (This is exactly why DLL hell exists today in Windows.)

Obviously, differentiating assemblies simply by using a filename isn't good enough. The CLR needs to support some mechanism that allows assemblies to be uniquely identified. This is what the term

*strongly named assembly* refers to. A strongly named assembly consists of four attributes that uniquely identify the assembly: a filename (without an extension), a version number, a culture identity, and a public key token (a value derived from a public key). The following strings identify four completely different assembly files:

```
"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

"MyTypes, Version=1.0.8123.0, Culture="en-US", PublicKeyToken=b77a5c561934e089"

"MyTypes, Version=2.0.1234.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
```

The first string identifies an assembly file called MyTypes.dll. The company producing the assembly is creating version 1.0.8123.0 of this assembly, and nothing in the assembly is sensitive to any one culture because **Culture** is set to **neutral**. Of course, any company could produce a MyTypes.dll assembly that is marked with a version number of 1.0.8123.0 and a neutral culture.

There must be a way to distinguish this company's assembly from another company's assembly that happens to have the same attributes. For several reasons, Microsoft chose to use standard public/private key cryptographic technologies instead of any other unique identification technique, such as GUIDs, URLs, or URNs. Specifically, cryptographic techniques provide a way to check the integrity of the assembly's bits as they are installed on a hard drive, and they also allow permissions to be granted on a per–publisher basis. I'll discuss these techniques more later in this chapter.

So, a company that wants to uniquely mark its assemblies must acquire a public/private key pair. Then the public key can be associated with the assembly. No two companies should have the same public/private key pair, and this distinction is what allows two companies to create assemblies that have the same name, version, and culture without causing any conflict.

**Note** The **System.Reflection.AssemblyName** class is a helper class that makes it easy for you to build an assembly name and to obtain the various parts of an assembly's name. The class offers several public instance properties, such as **CultureInfo**, **FullName**, **KeyPair**, **Name**, and **Version**. The class also offers a few public instance methods, such as **GetPublicKey**, **GetPublicKeyToken**, **SetPublicKey**, and **SetPublicKeyToken**.

In Chapter 2, I showed you how to name an assembly file and how to apply an assembly version number and a culture. A weakly named assembly can have assembly version and culture attributes embedded in the manifest metadata; however, the CLR always ignores the version number and uses only the culture information when it's probing subdirectories looking for the satellite assembly. Because weakly named assemblies are always privately deployed, the CLR simply uses the name of the assembly (tacking on a .dll or an .exe extension) when searching for the assembly's file in the application's base directory or any of the subdirectories specified in the XML configuration file's probing element's **privatePath** attribute.

A strongly named assembly has a filename, an assembly version, and a culture. In addition, a strongly named assembly is signed with the publisher's private key.

The first step in creating a strongly named assembly is to obtain a key by using the Strong Name Utility, SN.exe, that ships with the .NET Framework SDK and Visual Studio .NET. This utility offers a whole slew of features depending on the command–line switch you specify. Note that all SN.exe's command–line switches are case–sensitive. To generate a public/private key pair, you run SN.exe as follows:

```
SN Đk MyCompany.keys
```

This line tells SN.exe to create a file called MyCompany.keys. This file will contain the public and private key numbers persisted in a binary format.

Public key numbers are very big. If you want to see them, you can execute this command line:

```
SN Đtp MyCompany.keys
```

When I execute this line, I get the following output:

```
Microsoft (R) .NET Framework Strong Name Utility  Version 1.0.3210.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Public key is
0702000000240000525341320004000001000100031f38d3b2e55454ed52c5d246911011be59543
878d99e7da35c8bca8b714a96010572ca8ad63b9a1ea20f62036d79f250c86bbb3b85eb52785a8
7b543a068d9563c9b6db5bbc33898248d8a8cd7476a006b1977ce0c41ba502147d53e51ce06104
836dd392b85aac991d36884e20409de4aa362de46bd00ff043e012b57878c981647e3deec439c5
087d60e978d972663e7c7b28ab7236aab2ae686bfc7c1eda062d4027bdfed92ef5cc93d1138047
20c91abbe5a88ccca87f8b6751cafecee8b17657cdaef038568a9bf59ccefd056d971f9e839564
3849384688ebeb6b6b4fda9e8dc95606af700244923c822bafcee7dfe6606580bb125277fff941
4e8add01daacc5189209437cf2df24f5a3b8b463d37f059aa1dca6183460103912f25bc5304f01
4bcecff1bf1f50ca24c57f42eb885ed18834be32317357f33e8809abd1cd820847d365b7bf62c6
f1799fd1f3fa726e355a7eccf111f0f7a64a3d2e8cd83375a523d5fb99eb55c4abf59ec5ce571c
c6d4eb0eafa9891e19a94b3a264b64f83fa8dd3dbb3ffbfa2798c0f07c76d624a0d31f2ac0e536
80b021356f575ae4bf6f2ed794805ef29723261dcd5faace2f42f821f5b1fb6fad1331d30c621e
01187fce0b3067f409239f8b40fca884793b47bade292c1509c1169bb09c96803f270bdad9c8a8
ff8b9a6cf10025b53509b615623accd7a5f90641dd234b6537f7bb6215236639d8116569755817
308efaf043a627060191d0072a1eacadcb646ca23c13bef498cff88b3c0f49298446acaaabe62e
8b95326fea73ef1783b073

Public key token is 4da24326b8a214c7
```

The size of public keys makes them difficult to work with. To make things easier for the developer (and for end–users too), *public key tokens* were created. A public key token is a 64–bit hash of the public key. SN.exe's **Đtp** switch shows the public key token that corresponds to the complete public key at the end of its output.

Now that you know how to create a public/private key pair, creating a strongly named assembly is simple. You just apply an instance of the **System.Reflection.AssemblyKeyFileAttribute** attribute to your source code:

```
[assembly:AssemblyKeyFile("MyCompany.keys")]
```

When a compiler sees this attribute in your source code, the compiler opens the specified file (MyCompany.Keys), signs the assembly with the private key, and embeds the public key in the manifest. Note that you sign only the assembly file that contains the manifest; the assembly's other files can't be signed explicitly.

Here's what it means to sign a file: When you build a strongly named assembly, the assembly's FileDef manifest metadata table includes the list of all the files that make up the assembly. As each file's name is added to the manifest, the file's contents are hashed and this hash value is stored along with the file's name in the FileDef table. You can override the default hash algorithm used with AL.exe's /**algid** switch or the assembly level **System.Reflection.Assembly–AlgIDAttribute** custom attribute. By default, a SHA–1 algorithm is used, and this should be sufficient for almost all applications.

After the PE file containing the manifest is built, the PE file's entire contents are hashed, as shown in Figure 3–1. The hash algorithm used here is always SHA–1 and can't be overridden. This hash value—typically around 100 or 200 bytes in size—is signed with the publisher's private key, and the resulting RSA digital signature is stored in a reserved section (not included in the hash) within the PE file. The CLR header of the PE file is updated to reflect where the digital signature is embedded within the file.

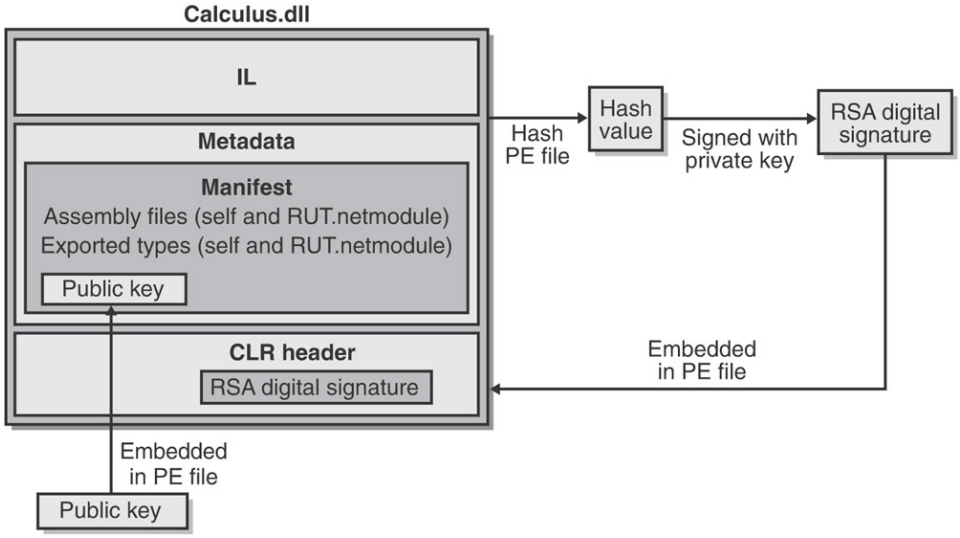



Figure 3–1 : Signing an assembly

The publisher's public key is also embedded into the AssemblyDef manifest metadata table in this PE file. The combination of the filename, the assembly version, the culture, and the public key gives this assembly a strong name, which is guaranteed to be unique. There is no way that two companies could produce a "Calculus" assembly with the same public key (assuming that the companies don't share this key pair with each other).

At this point, the assembly and all its files are ready to be packaged and distributed.

As described in Chapter 2, when you compile your source code, the compiler detects the types and members that your code references. You must specify the referenced assemblies to the compiler. For the C# compiler, you use the /**reference** command–line switch. Part of the compiler's job is to emit an AssemblyRef metadata table inside the resulting managed module. Each entry in the AssemblyRef metadata table indicates the referenced assembly's name (without path and extension), version number, culture, and public key information.

**Important** Because public keys are such large numbers and a single assembly might reference many assemblies, a large percentage of the resulting file's total size would be occupied with public key information. To conserve storage space, Microsoft hashes the public key and takes the last 8 bytes of the hashed value. This reduced value has been determined to be statistically unique and is therefore safe to pass around the system. These reduced public key values—known as public key tokens—are what are actually stored in an AssemblyRef table. In general, developers and end–users will see public key token values much more frequently than full public key values.

Following is the AssemblyRef metadata information for the JeffTypes.dll file that I discussed in Chapter 2:

```
AssemblyRef #1
-------------------------
    Token: 0x23000001
```

```
    Public Key or Token: b7 7a 5c 56 19 34 e0 89
    Name: mscorlib
    Major Version: 0x00000001
    Minor Version: 0x00000000
    Build Number: 0x00000c1e
    Revision Number: 0x00000000
    Locale: <null>
    HashValue Blob: 3e 10 f3 95 e3 73 0b 33  1a 4a 84 a7 81 76 eb 32  4b
                    36 4d a5
    Flags: [none] (00000000)
```

From this, you can see that JeffTypes.dll references a type that is contained in an assembly matching the following attributes:

```
"MSCorLib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Unfortunately, ILDasm.exe uses the term *Locale* when it really should be using *Culture* instead. Microsoft says that they'll fix the term in a future version of the tool.

If you look at JeffTypes.dll's AssemblyDef metadata table, you see the following:

```
Assembly
_____________________________

    Token: 0x20000001
    Name : JeffTypes
    Public Key    :
    Hash Algorithm : 0x00008004
    Major Version: 0x00000001
    Minor Version: 0x00000000
    Build Number: 0x00000253
    Revision Number: 0x00005361
    Locale: <null>
    Flags : [SideBySideCompatible]  (00000000)
```

This is equivalent to the following:

```
"JeffTypes, Version=1.0.595.21345, Culture=neutral,
PublicKeyToken=null"
```

In this line, no public key token is specified because in Chapter 2 the JeffTypes.dll assembly wasn't signed with a public key, making it a weakly named assembly.

If I had used SN.exe to create a key file, added the **AssemblyKeyFile−Attribute** to the source code, and then recompiled, the resulting assembly would be signed. If you're using AL.exe to build the assembly, you specify its /**keyfile** switch instead of using the **AssemblyKeyFileAttribute**. If I had used ILDasm.exe to explore the new assembly's metadata, the AssemblyDef entry would have bytes appearing after the Public Key field and the assembly would be strongly named. By the way, the AssemblyDef entry always stores the full public key, not the public key token. The full public key is necessary to ensure that the file hasn't been tampered with. I'll explain the tamper resistance of strongly named assemblies later in this chapter.

## The Global Assembly Cache

Now that you know how to create a strongly named assembly, it's time to learn how to deploy this assembly and how the CLR uses the information to locate and load the assembly.

If an assembly is to be accessed by multiple applications, the assembly must be placed into a well–known directory and the CLR must know to look in this directory automatically when a reference to the assembly is detected. This well–known location is called the global assembly cache (GAC), which can usually be found in the following directory:

```
C:\Windows\Assembly\GAC
```

The GAC directory is structured: it contains many subdirectories, and an algorithm is used to generate the names of these subdirectories. You should never manually copy assembly files into the GAC; instead, you should use tools to accomplish this task. These tools know the GAC's internal structure and how to generate the proper subdirectory names.

While developing and testing, the most common tool for installing a strongly named assembly into the GAC is GACUtil.exe. Running this tool without any command–line arguments yields the following usage:

```
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.0.3415.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Usage: Gacutil <option> [<parameters>]
 Options:
  /i
    Installs an assembly to the global assembly cache. Include the
    name of the file containing the manifest as a parameter.
    Example:  /i myDll.dll

  /if
    Installs an assembly to the global assembly cache and forces
    overwrite if assembly already exists in cache. Include the
    name of the file containing the manifest as a parameter.
    Example:  /if myDll.dll

  /ir
    Installs an assembly to the global assembly cache with traced
    reference. Include the name of file containing manifest,
    reference scheme, ID and description as parameters.
    Example:  /ir myDll.dll FILEPATH c:\apps\myapp.exe MyApp

  /u[ngen]
    Uninstalls an assembly. Include the name of the assembly to
    remove as a parameter. If ngen is specified, the assembly is
    removed from the cache of ngen'd files, otherwise the assembly
    is removed from the global assembly cache.
    Examples:.
      /ungen myDll
      /u myDll,Version=1.1.0.0,Culture=en,PublicKeyToken=874e23ab874e23ab

  /ur
    Uninstalls an assembly reference. Include the name of the
    assembly, type of reference, ID and data as parameters.
    Example:  /ur myDll,Version=1.1.0.0,Culture=en,PublicKeyToken=874e23ab874e23ab
                        FILEPATH c:\apps\myapp.exe MyApp

  /uf
    Forces uninstall of an assembly by removing all install references.
    Include the full name of the assembly to remove as a parameter.
    Assembly will be removed unless referenced by Windows Installer.
    Example:  /uf myDll,Version=1.1.0.0,Culture=en,PublicKeyToken=874e23ab874e23ab

  /l
```

```
    Lists the contents of the global assembly cache. Allows optional
    assembly name parameter to list matching assemblies only.

  /lr
    Lists the contents of the global assembly cache with traced
    reference information. Allows optional assembly name parameter
    to list matching assemblies only.

  /cdl
    Deletes the contents of the download cache

  /ldl
    Lists the contents of the downloaded files cache

  /nologo
    Suppresses display of the logo banner

  /silent
    Suppresses display of all output
```

As you can see, you can invoke GACUtil.exe specifying the /**i** switch to install an assembly into the GAC, and you can use GACUtil.exe's /**u** switch to uninstall an assembly from the GAC. Note that you can't ever place a weakly named assembly into the GAC. If you pass the filename of a weakly named assembly to GACUtil.exe, it displays the following error message:

> "Failure adding assembly to the cache: Attempt to install an assembly without a strong name."

**Note** By default, the GAC can be manipulated only by a user belonging to the Windows Administrators group. GACUtil.exe will fail to install or uninstall an assembly if the user invoking the execution of the utility isn't a member of this group.

Using GACUtil.exe's /**i** switch is very convenient for developer testing. However, if you use GACUtil.exe to deploy an assembly in a production environment, it's recommended that you use GACUtil.exe's /**ir** switch instead and its /**ur** switch to uninstall the assembly. The /**ir** switch integrates the installation with the Windows install and uninstall engine. Basically, it tells the system which application requires the assembly and then ties the application and the assembly together.

**Note** If a strongly named assembly is packaged in a cabinet (.cab) file or is compressed in some way, the assembly's file must first be decompressed to temporary file(s) before you use GACUtil.exe to install the assembly's files into the GAC. Once the assembly's files have been installed, the temporary file(s) can be deleted.

The GACUtil.exe tool doesn't ship with the end-user .NET Framework redistributable package. If your application includes some assemblies that you want deployed into the GAC, you must use the Windows Installer (MSI) version 2 or later because MSI is the only tool that is guaranteed to be on end-user machines and capable of installing assemblies into the GAC. (You can determine which version of the Windows Installer is installed by running MSIExec.exe.)

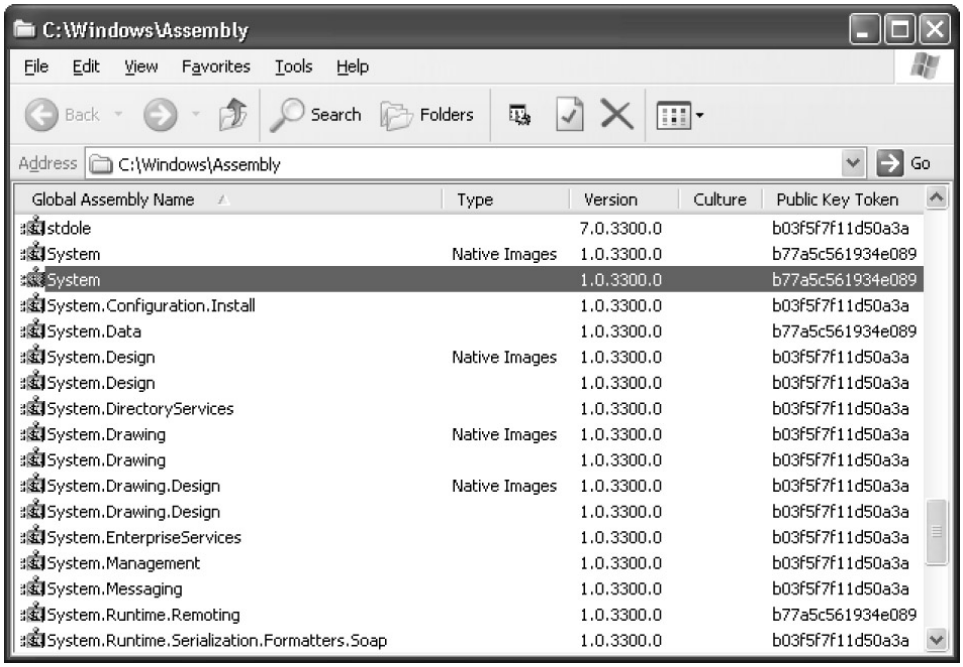
**Important** Globally deploying assembly files into the GAC is a form of registering the assembly, although the actual Windows registry isn't affected in any way. Installing assemblies into the GAC breaks the goal of simple application installation, backup, restore, moving, and uninstall. So really, you get only the "simple" story when you avoid global deployment and use private deployment exclusively.

What is the purpose of "registering" an assembly in the GAC? Well, say two companies each produce a Calculus assembly consisting of one file: Calculus.dll. Obviously, both these files can't go

in the same directory because the last one installed would overwrite the first one, surely breaking some application. When you use a tool to install an assembly into the GAC, the tool creates subdirectories under the C:\Windows\Assembly\GAC directory and copies the assembly files into this subdirectory.

Normally, no one examines the GAC's subdirectories, so the structure of the GAC shouldn't really matter to you. As long as the tools and the CLR know the structure, all is good. Just for fun, I'll describe the internal structure of the GAC in the next section.

When you install the .NET Framework, an Explorer shell extension (ShFusion.dll) is installed. This shell extension also knows the structure of the GAC, and it displays the GAC's contents in a nice, user–friendly fashion. When I use Explorer and navigate to my C:\Windows\Assembly directory, I see what's shown in Figure 3–2: the assemblies installed into the GAC. Each row shows the assembly's name, type, version number, culture (if any), and public key token.




| Global Assembly Name | Type | Version | Culture | Public Key Token |
|---|---|---|---|---|
| stdole | | 7.0.3300.0 | | b03f5f7f11d50a3a |
| System | Native Images | 1.0.3300.0 | | b77a5c561934e089 |
| System | | 1.0.3300.0 | | b77a5c561934e089 |
| System.Configuration.Install | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Data | | 1.0.3300.0 | | b77a5c561934e089 |
| System.Design | Native Images | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Design | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.DirectoryServices | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Drawing | Native Images | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Drawing | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Drawing.Design | Native Images | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Drawing.Design | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.EnterpriseServices | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Management | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Messaging | | 1.0.3300.0 | | b03f5f7f11d50a3a |
| System.Runtime.Remoting | | 1.0.3300.0 | | b77a5c561934e089 |
| System.Runtime.Serialization.Formatters.Soap | | 1.0.3300.0 | | b03f5f7f11d50a3a |

Figure 3–2 : Using Explorer's shell extension to see the assemblies installed into the GAC

You can select an entry and click the secondary mouse button to display a context menu. The context menu contains Delete and Properties menu items. Obviously, the Delete menu item deletes the selected assembly's files from the GAC and fixes up the GAC's internal structure. Selecting the Properties menu item displays a property dialog box that looks like the one shown in Figure 3–3. The Last Modified timestamp indicates when the assembly was added to the GAC. Selecting the Version tab reveals the property dialog box shown in Figure 3–4.

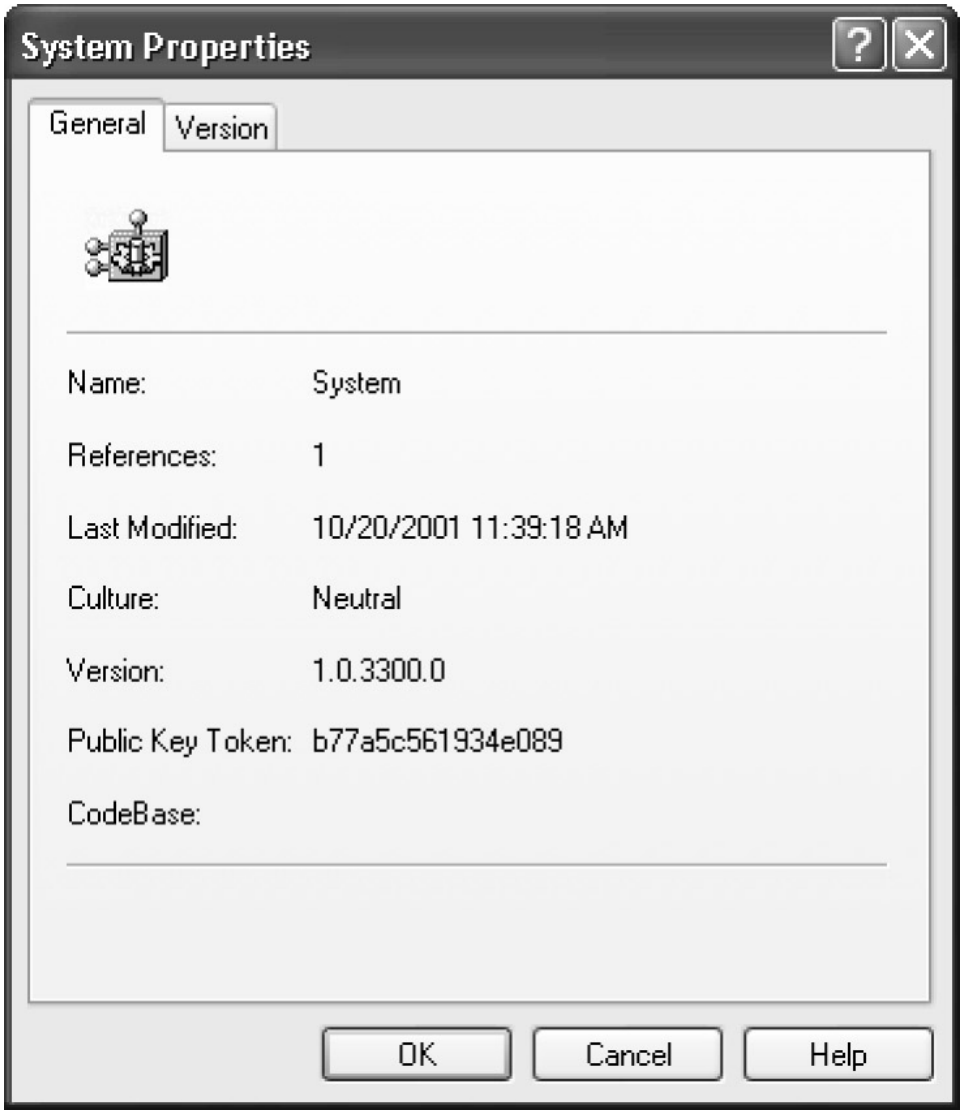


Figure 3–3 : General tab of the System Properties dialog box

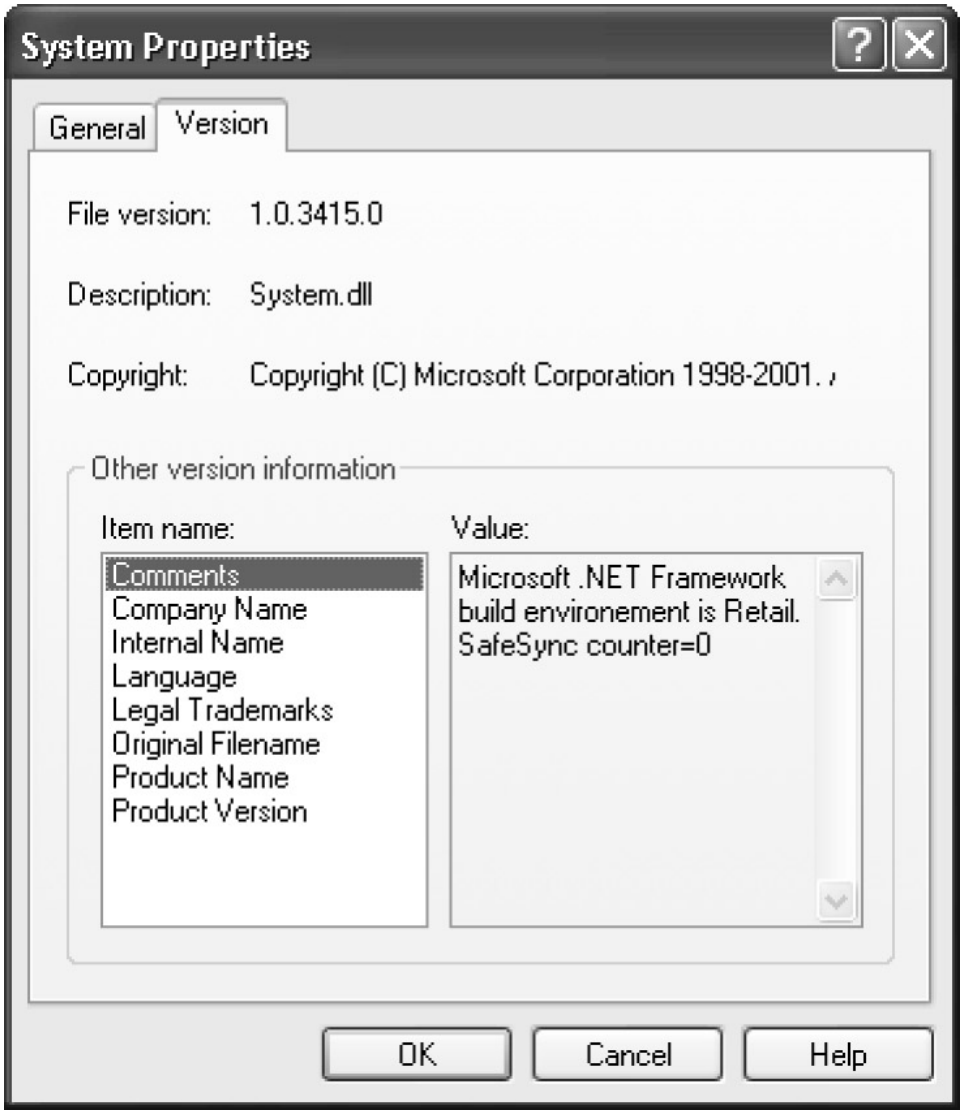


Figure 3–4 : Version tab of the System Properties dialog box

Last but not least, you can drag an assembly file that contains a manifest and drop the file in Explorer's window. When you do this, the shell extension installs the assembly's files into the GAC. For some developers, this is an easier way to install assemblies into the GAC for testing instead of using the GACUtil.exe tool.

## The Internal Structure of the GAC

Simply stated, the purpose of the GAC is to maintain a relationship between a strongly named assembly and a subdirectory. Basically, the CLR has an internal function that takes an assembly's name, version, culture, and public key token. This function then returns the path of a subdirectory where the specified assembly's files can be found.

If you go to a command prompt and change into the C:\Windows\Assembly\GAC directory, you'll see several subdirectories, one for each assembly that has been installed into the GAC. Here's what my GAC directory looks like (with some of the directories deleted to save trees):

```
Volume in drive C has no label.
 Volume Serial Number is 94FA-5DE7

 Directory of C:\WINDOWS\assembly\GAC
```

```
08/15/2001  05:07 PM    <DIR>          .
08/15/2001  05:07 PM    <DIR>          ..
08/15/2001  03:09 PM    <DIR>          Accessibility
08/15/2001  05:06 PM    <DIR>          ADODB
07/03/2001  04:54 PM    <DIR>          CRVsPackageLib
08/15/2001  05:06 PM    <DIR>          Microsoft.ComCtl2
08/15/2001  05:06 PM    <DIR>          Microsoft.ComctlLib
08/15/2001  05:05 PM    <DIR>          Microsoft.JScript
08/15/2001  05:07 PM    <DIR>          Microsoft.mshtml
08/15/2001  05:06 PM    <DIR>          Microsoft.MSMAPI
08/15/2001  05:06 PM    <DIR>          Microsoft.MSMask
08/15/2001  05:07 PM    <DIR>          Microsoft.MSRDC
08/15/2001  05:07 PM    <DIR>          Microsoft.MSWinsockLib
08/15/2001  05:07 PM    <DIR>          Microsoft.MSWLess
08/15/2001  05:07 PM    <DIR>          Microsoft.PicClip
08/15/2001  05:07 PM    <DIR>          Microsoft.RichTextLib
08/15/2001  05:06 PM    <DIR>          Microsoft.StdFormat
08/15/2001  05:07 PM    <DIR>          Microsoft.SysInfoLib
08/15/2001  05:07 PM    <DIR>          Microsoft.TabDlg
08/15/2001  05:05 PM    <DIR>          Microsoft.VisualBasic
08/15/2001  03:09 PM    <DIR>          System
08/15/2001  03:09 PM    <DIR>          System.Configuration.Install
08/15/2001  03:09 PM    <DIR>          System.Data
08/15/2001  03:09 PM    <DIR>          System.Design
08/15/2001  03:09 PM    <DIR>          System.DirectoryServices
08/15/2001  03:09 PM    <DIR>          System.Drawing
08/15/2001  03:09 PM    <DIR>          System.Drawing.Design
08/15/2001  03:08 PM    <DIR>          System.EnterpriseServices
08/15/2001  03:09 PM    <DIR>          System.Management
08/15/2001  03:09 PM    <DIR>          System.Messaging
08/15/2001  03:09 PM    <DIR>          System.Runtime.Remoting
08/15/2001  03:08 PM    <DIR>          System.Security
08/15/2001  03:09 PM    <DIR>          System.ServiceProcess
08/15/2001  03:09 PM    <DIR>          System.Web
08/15/2001  03:09 PM    <DIR>          System.Web.RegularExpressions
08/15/2001  03:09 PM    <DIR>          System.Web.Services
08/15/2001  03:09 PM    <DIR>          System.Windows.Forms
08/15/2001  03:09 PM    <DIR>          System.Xml
               0 File(s)              0 bytes
              95 Dir(s)  14,798,938,112 bytes free
```

If you change into one of these directories, you'll see one or more additional subdirectories. My System directory looks like this:

```
Volume in drive C has no label.
 Volume Serial Number is 94FA-5DE7

 Directory of C:\WINDOWS\assembly\GAC\System

08/15/2001  03:09 PM    <DIR>          .
08/15/2001  03:09 PM    <DIR>          ..
08/15/2001  03:09 PM    <DIR>          1.0.3300.0__b77a5c561934e089
               0 File(s)              0 bytes
               3 Dir(s)  14,798,929,920 bytes free
```

The System directory contains one subdirectory for every System.dll assembly installed on the machine. In my case, just one version of the System.dll assembly is installed:

```
"System, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
```

The attributes are separated by underscore characters and are in the form of "*(Version)_(Culture)_(PublicKeyToken)*". In this example, there is no culture information, making the culture neutral. Inside this subdirectory are the files (such as System.dll) making up this strongly named version of the System assembly.

**Important** It should be obvious that the whole point of the GAC is to hold multiple versions of an assembly. For example, the GAC can contain version 1.0.0.0 and version 2.0.0.0 of Calculus.dll. If an application is built and tested using version 1.0.0.0 of Calculus.dll, the CLR will load version 1.0.0.0 of Calculus.dll for that application even though a later version of the assembly exists and is installed into the GAC. This is the CLR's default policy in regards to loading assembly versions, and the benefit of this policy is that installing a new version of an assembly won't affect an already installed application. You can affect this policy in a number of ways, and I'll discuss them later in this chapter.

## Building an Assembly That References a Strongly Named Assembly

Whenever you build an assembly, the assembly will have references to other strongly named assemblies. This is true if only because **System.Object** is defined in MSCorLib.dll. However, it's likely that an assembly will reference types in other strongly named assemblies published either by Microsoft, by a third party, or by your own organization.

In Chapter 2, I showed you how to use CSC.exe's /**reference** command–line switch to specify the assembly filenames you want to reference. If the filename is a full path, CSC.exe loads the specified file and uses its metadata information to build the assembly. If you specify a filename without a path, CSC.exe attempts to find the assembly by looking in the following directories (in order of their presentation here):

1. The working directory.
2. The directory that contains the CLR that the compiler itself is using to produce the resulting assembly. MSCorLib.dll is always obtained from this directory. This directory has a path similar to the following:

   C:\WINDOWS\Microsoft.NET\Framework\v1.0.3427.
3. Any directories specified using CSC.exe's /**lib** command–line switch.
4. Any directories specified by the **LIB** environment variable.

So if you're building an assembly that references Microsoft's System.Drawing.dll, you can specify the /**reference:System.Drawing.dll** switch when invoking CSC.exe. The compiler will examine the directories shown earlier and will find the System.Drawing.dll file in the directory that contains the CLR that the compiler itself is using to produce the assembly. Even though this is the directory where the assembly is found at compile time, this isn't the directory where the assembly will be loaded from at run time.

You see, when you install the .NET Framework, it installs two copies of Microsoft's assembly files. One set is installed into the CLR directory, and another set is installed into the GAC. The files in the CLR directory exist so that you can easily build your assembly. The copies in the GAC exist so that they can be loaded at run time.

The reason that CSC.exe doesn't look in the GAC for referenced assemblies is because you'd have to specify a long, ugly path to the assembly file—something like C:\WINDOWS\Assembly\GAC\System.Drawing\1.0.3300.0__b03f5f7f11d50a3a\System.Drawing.dll. Alternatively, CSC.exe could allow you to specify a still long but slightly nicer-looking string, such as "System.Drawing, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a". Both of these solutions were deemed worse than having the assembly files installed twice on the user's hard drive.

Before leaving this section, I should also talk about *response files*. A response file is a text file that contains a set of compiler command-line switches. When you execute CSC.exe, the compiler opens response files and uses any switches that are specified in them as though the switches were passed on the command line. You instruct the compiler to use a response file by specifying its name on the command line prepended by an @ sign. For example, you could have a response file called MyProject.rsp that contains the following text:

```
/out:MyProject.exe
/target:winexe
```

To have CSC.exe use these settings, you'd invoke it as follows:

```
csc.exe @MyProject.rsp CodeFile1.cs CodeFile2.cs
```

This tells the C# compiler what to name the output file and what kind of target to create. As you can see, response files are very convenient because you don't have to manually express the desired command-line arguments each time you want to compile your project.

The C# compiler supports multiple response files. In addition to the files you explicitly specify on the command line, the compiler automatically looks for files called CSC.rsp. When you run CSC.exe, it looks in the current directory for a local CSC.rsp file—you should place any project-specific settings in this file. The compiler also looks in CSC.exe's own directory for a global CSC.rsp file. Settings that you want applied to all your projects should go in this file. The compiler uses the settings in all these response files. If you have conflicting settings in the local and global response files, the settings in the local file override the settings in the global file. Likewise, any settings explicitly passed on the command line override the settings taken from a local response file.

When you install the .NET Framework, it installs a default global CSC.rsp file. This file contains the following switches:

```
# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.Vsa.dll
/r:System.Configuration.Install.dll
/r:System.Data.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
```

```
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceProcess.dll
/r:System.Web.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.Dll
/r:System.XML.dll
```

When building a project, the compiler will assume you want to reference all the assemblies listed. Don't worry: an AssemblyRef entry is created only if your source code refers to a type or member defined by one of these assemblies. This response file is a big convenience for developers: it allows you to use types and namespaces defined in various Microsoft–published assemblies without having to specify any /**reference** command–line switches when compiling. Of course, you're welcome to add your own switches to the global CSC.rsp file if you want to make your life even easier.

**Note** You can tell the compiler to ignore both local and global CSC.rsp files by specifying the /**noconfig** command–line switch.

## Strongly Named Assemblies Are Tamper–Resistant

Signing a file with a private key ensures that the holder of the corresponding public key produced the assembly. When the assembly is installed into the GAC, the system hashes the contents of the file containing the manifest and compares the hash value with the RSA digital signature value embedded within the PE file (after unsigning it with the public key). If the values are identical, the file's contents haven't been tampered with and you know that you have the public key that corresponds to the publisher's private key. In addition, the system hashes the contents of the assembly's other files and compares the hash values with the hash values stored in the manifest file's FileDef table. If any of the hash values don't match, at least one of the assembly's files has been tampered with and the assembly will fail to install into the GAC.

**Important** This mechanism ensures only that a file's contents haven't been tampered with; the mechanism doesn't allow you to tell who the publisher is unless you're absolutely positive that the publisher produced the public key you have and you're sure that the publisher's private key was never compromised. If the publisher wants to associate its identity with the assembly, the publisher must use Microsoft's Authenticode technology in addition.

When an application needs to bind to an assembly, the CLR uses the referenced assembly's properties (name, version, culture, and public key) to locate the assembly in the GAC. If the referenced assembly can be found, its containing subdirectory is returned and the file holding the manifest is loaded. Finding the assembly this way assures the caller that the assembly loaded at run time came from the same publisher that built the assembly the code was compiled against. This assurance comes because the public key token in the referencing assembly's AssemblyRef table corresponds to the public key in the referenced assembly's AssemblyDef table. If the referenced assembly isn't in the GAC, the CLR looks in the application's base directory and then in any of the private paths identified in the application's configuration file; then, if the application was installed using MSI, the CLR asks MSI to locate the assembly. If the assembly can't be found in any of these locations, the bind fails and a **System.IO.FileNotFoundException** exception is thrown.

When strongly named assembly files are loaded from a location other than the GAC (via a **codeBase** element in a configuration file), the CLR compares hash values when the assembly is loaded. In other words, a hash of the file is performed every time an application executes. This performance hit is required to be certain that the assembly file's content hasn't been tampered with. When the CLR detects mismatched hash values at run time, it throws a **System.IO.File–LoadException** exception.

## Delayed Signing

Earlier in this chapter, I discussed how the SN.exe tool can produce public/private key pairs. This tool generates the keys by making calls into the Crypto API provided by Windows. These keys can be stored in files or other "storage devices." For example, large organizations (such as Microsoft) will maintain the returned private key in a hardware device that stays locked in a safe; only a few people in the company have access to the private key. This precaution prevents the private key from being compromised and ensures the key's integrity. The public key is, well, public and freely distributed.

When you're ready to package your strongly named assembly, you'll have to use the secure private key to sign it. However, while developing and testing your assembly, gaining access to the secure private key can be a hassle. For this reason, the .NET Framework supports *delayed signing,* sometimes referred to as *partial signing*.

Delayed signing allows you to build an assembly using only your company's public key; the private key isn't necessary. Using the public key allows assemblies that reference your assembly to embed the correct public key value in their AssemblyRef metadata entries. It also allows the assembly to be placed in the GAC's internal structure appropriately. If you don't sign the file with your company's private key, you lose all the tampering protection afforded to you because the assembly's files won't be hashed and a digital signature won't be embedded in the file. This loss of protection shouldn't be a problem, however, because you use delayed signing only while developing your own assembly, not when you're ready to package and deploy the assembly.

Basically, you get your company's public key value in a file and pass the filename to whatever utility you use to build the assembly. (You can use SN.exe's **Đp** switch to extract a public key from a file that contains a public/private key pair.) You must also tell the tool that you want the assembly to be delay signed, meaning that you're not supplying a private key. In source code, you can apply the **AssemblyKeyFileAttribute** and **DelaySignAttribute** attributes to specify these things. If you're using AL.exe, you can specify the /**keyf[ile]** and /**delay[sign]** command–line switches.

When the compiler or AL.exe detects that you're delay signing an assembly, it will emit the assembly's AssemblyDef manifest entry and this entry will contain the assembly's public key. Again, the presence of the public key allows the assembly to be placed in the GAC. It also allows you to build other assemblies that reference this assembly; the referencing assemblies will have the correct public key in their AssembyRef metadata table entries. When creating the resulting assembly, space is left in the resulting PE file for the RSA digital signature. (The utility can determine how much space is necessary from the size of the public key.) Note that the file's contents won't be hashed at this time either.

At this point, the resulting assembly doesn't have a valid signature. Attempting to install the assembly into the GAC will fail because a hash of the file's contents hasn't been done—the file appears to have been tampered with. To install the assembly into the GAC, you must prevent the system from verifying the integrity of the assembly's files. To do this, you use the SN.exe utility

specifying the **ÐVr** command–line switch. Executing SN.exe with this switch also tells the CLR to skip checking hash values for any of the assembly's files when loaded at run time.

When you're finished developing and testing the assembly, you'll need to officially sign it so that you can package and deploy it. To sign the assembly, use the SN.exe utility again, this time with the **ÐR** switch and the name of the file that contains the actual private key. The **ÐR** switch causes SN.exe to hash the file's contents, sign it with the private key, and embed the RSA digital signature in the file where the space for it had previously been reserved. After this step, you can deploy the fully signed assembly. You can also turn verification of this assembly back on by using SN.exe's **ÐVu** or **ÐVx** command–line switch.

The following list summarizes the steps discussed in this section to develop your assembly by using the delay signing technique:

1. While developing an assembly, obtain a file that contains only your company's public key and add the following two attributes to your source code:

```
[assembly:AssemblyKeyFile("MyCompanyPublicKey.keys")]
[assembly:DelaySign(true)]
```

2. After building the assembly, execute the following line so that you can install it in the GAC, build other assemblies that reference the assembly, and test the assembly. Note that you have to do this only once; it's not necessary to perform this step each time you build your assembly.

```
SN.exe ÐVr MyAssembly.dll
```

3. When ready to package and deploy the assembly, obtain your company's private key and execute the following line:

```
SN.exe -R MyAssembly.dll MyCompanyPrivateKey.keys
```

4. To test, turn verification back on by executing the following line:

```
SN ÐVu MyAssembly.dll
```

At the beginning of this section, I mentioned how organizations will keep their key pairs in a hardware device, such as a smart card. To keep these keys secure, you must make sure the key values are never persisted in a disk file. Cryptographic service providers (CSPs) offer "containers" that abstract the location of these keys. Microsoft, for example, uses a CSP that has a container that, when accessed, grabs the private key from a smart card.

If your public/private key pair is in a CSP container, don't use the **Assembly–KeyFileAttribute** attribute or AL.exe's /**keyf[ile]** switch. Instead, use the **System.Reflection.AssemblyKeyNameAttribute** attribute or AL.exe's /**keyn[ame]** switch. When using SN.exe to add the private key to the delay signed assembly, you specify the **ÐRc** switch instead of the **ÐR** switch. SN.exe offers additional switches that allow you to perform operations with a CSP.

**Important** Delay signing is useful whenever you want to perform some other operation to an assembly before you deploy it. For example, because an assembly is just a normal Windows PE file, you might want to consider rebasing the load address for the file. To do this, you can use the normal Rebase.exe tool that ships with the Microsoft Win32 Platform SDK. You can't rebase a file after it's been fully signed because the hash values will be incorrect. So, if you want to rebase an assembly file or do any other type of post–build operation, you should use

delay signing, perform the post−build operation, and then run SN.exe with the **ĐR** or **ĐRc** switch to complete the signing of the assembly with all its hashing.

Here's the AssemInfo.cs file I use for all my personal projects:

```
/******************************************************************************
Module:  AssemInfo.cs
Notices: Copyright (c) 2002 Jeffrey Richter
******************************************************************************/

using System.Reflection;

//////////////////////////////////////////////////////////////////////////////

// Set the version CompanyName, LegalCopyright, and LegalTrademarks fields.
[assembly:AssemblyCompany("The Jeffrey Richter Company")]
[assembly:AssemblyCopyright("Copyright (c) 2002 Jeffrey Richter")]
[assembly:AssemblyTrademark(
   "JeffTypes is a registered trademark of the Richter Company")]

//////////////////////////////////////////////////////////////////////////////

// Set the version ProductName and ProductVersion fields.
[assembly:AssemblyProduct("Jeffrey Richter Type Library")]
[assembly:AssemblyInformationalVersion("2.0.0.0")]

//////////////////////////////////////////////////////////////////////////////

// Set the version FileVersion, AssemblyVersion,
// FileDescription, and Comments fields.
[assembly:AssemblyFileVersion("1.0.0.0")]
[assembly:AssemblyVersion("3.0.0.0")]
[assembly:AssemblyTitle("Jeff's type assembly")]
[assembly:AssemblyDescription("This assembly contains Jeff's types")]

//////////////////////////////////////////////////////////////////////////////

// Set the assembly's culture (""=neutral).
[assembly:AssemblyCulture("")]

//////////////////////////////////////////////////////////////////////////////

#if !StronglyNamedAssembly

// Weakly named assemblies are never signed.
[assembly:AssemblyDelaySign(false)]

#else

// Strongly named assemblies are usually delay signed while building and
// completely signed using SN.exe's -R or -Rc switch.
[assembly:AssemblyDelaySign(true)]

   #if !SignedUsingACryptoServiceProvider

   // Give the name of the file that contains the public/private key pair.
   // If delay signing, only the public key is used.
   [assembly:AssemblyKeyFile("MyCompany.keys")]

   // Note: If AssemblyKeyFile and AssemblyKeyName are both specified,
   // here's what happens...
   // 1) If the container exists, the key file is ignored.
```

```
   // 2) If the container doesn't exist, the keys from the key
   //    file are copied into the container and the assembly is signed.

   #else

   // Give the name of the cryptographic service provider (CSP) container
   // that contains the public/private key pair.
   // If delay signing, only the public key is used.
   [assembly:AssemblyKeyName("")]

   #endif

#endif

////////////////////////////// End of File //////////////////////////////
```

When you create a new project with Visual Studio .NET, it automatically creates a new AssemblyInfo.cs file that is almost identical to the set of attributes shown here in my file. I prefer my file because my comments better describe what's going on and how the attributes map to the version resource information. In addition, the Visual Studio .NET AssemblyInfo.cs file initializes the **Assembly–Version** attribute incorrectly to "1.0.*", telling CSC.exe to generate build and revision version numbers automatically with each build. Having a different version for each build of your assembly prevents the CLR from loading your assembly when any previously built assemblies that reference an older version number need it.

The Visual Studio .NET AssemblyInfo.cs file also contains the **System.Reflection.AssemblyConfiguration** attribute, which is no longer used anywhere by the CLR and should have been removed completely from the .NET Framework. Leaving this attribute in the file does nothing but waste space.

## Privately Deploying Strongly Named Assemblies

Installing assemblies into the GAC offers several benefits. The GAC enables many applications to share assemblies, reducing physical memory usage on a whole. In addition, it's easy to deploy a new version of the assembly into the GAC and have all applications use the new version via a publisher policy (described later in this chapter). The GAC also provides side–by–side management for an assembly's different versions. However, the GAC is usually secured so that only an administrator can install an assembly into it. Also, installing into the GAC breaks the simple copy deployment story.

While strongly named assemblies can be installed into the GAC, they certainly don't have to be. In fact, it's recommended that you deploy assemblies into the GAC only if the assembly is intended to be shared by many applications. If an assembly isn't intended to be shared, it should be deployed privately. Deploying privately preserves the "simple" copy install deployment story and better isolates the application and its assemblies. Also, the GAC isn't intended to be the new C:\Windows\System32 dumping ground for common files. The reason is because new versions of assemblies don't overwrite each other; they are installed side by side, eating up disk space.

In addition to deploying a strongly named assembly in the GAC or privately, a strongly named assembly can be deployed to some arbitrary directory that a small set of applications know about. For example, you might be producing three applications, all of which want to share a strongly named assembly. Upon installation, you can create three directories: one for each application and an additional directory for the assembly you want shared. When you install each application into its

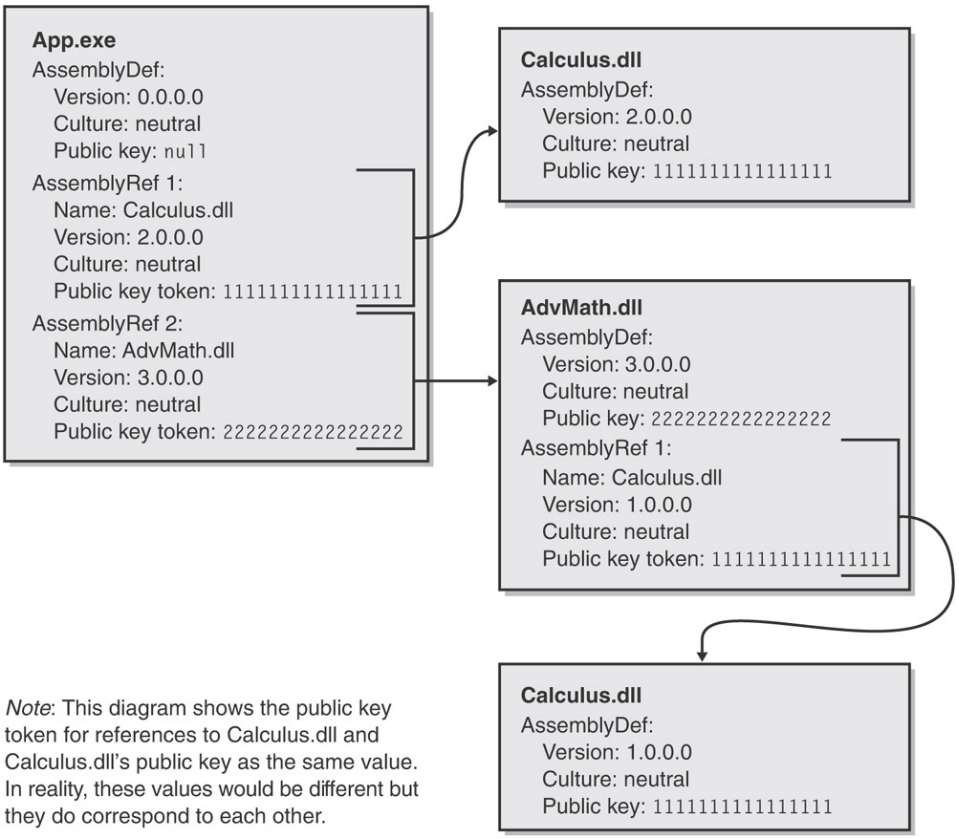
directory, also install an XML configuration file and have the shared assembly's **codeBase** element indicate the path of the shared assembly. Now, at run time, the CLR will know to look in the strongly named assembly's directory for the shared assembly. For the record, this technique is rarely used and is somewhat discouraged because no single application controls when the assembly's files should be uninstalled.

**Note** The configuration file's **codeBase** element actually identifies a URL. This URL can refer to any directory on the user's hard disk or to a Web address. In the case of a Web address, the CLR will automatically download the file and store it in the user's download cache (a subdirectory under C:\Documents and Settings\UserName\Local Settings\Application Data\Assembly. When referenced in the future, the CLR will load the assembly from this directory rather than access the URL. An example of a configuration file containing a **codeBase** element is shown later in this chapter.

**Note** When a strongly named assembly is installed into the GAC, the system ensures that the file containing the manifest hasn't been tampered with. This check occurs only once: at installation time. On the other hand, when a strongly named assembly is loaded from a directory other than the GAC, the CLR verifies the assembly's manifest file to ensure that the file's contents have not been tampered with. This additional performance hit occurs every time this file is loaded.

## Side–by–Side Execution

The strong versioning story presented here means that an assembly, App.exe, could bind to version 2.0.0.0 of a Calculus.dll assembly and version 3.0.0.0 of an AdvMath.dll assembly. The AdvMath.dll assembly could in turn bind to version 1.0.0.0 of a Calculus.dll assembly. Take a look at Figure 3–5.




*Note*: This diagram shows the public key token for references to Calculus.dll and Calculus.dll's public key as the same value. In reality, these values would be different but they do correspond to each other.

Figure 3–5 : An application that requires different versions of the Calculus.dll assembly in order to run

The CLR has the ability to load multiple files with the same name but with different paths into a single address space. This is called *side–by–side execution*, a key component for solving the Windows “DLL hell” problem.

**Important** The ability to execute DLLs side by side is awesome because it allows you to create new versions of your assembly that don’t have to maintain backward compatibility. Not having to maintain backward compatibility reduces coding and testing time for a product and allows you to get the product to market faster.

The developer must be aware of this side–by–side mechanism so that subtle bugs don’t appear. For example, an assembly could create a named Win32 file–mapping kernel object and use the storage provided by this object. Another version of the assembly could also get loaded and attempt to create a file–mapping kernel object with the same name. This second assembly won’t get new storage but will instead access the same storage allocated by the first assembly. If not carefully coded, the two assemblies will stomp all over each other’s data and the application will perform unpredictably.

## How the Runtime Resolves Type References

At the beginning of Chapter 2, we saw the following source code:

```
public class App {
   static public void Main(System.String[] args) {
      System.Console.WriteLine("Hi");
   }
}
```

This code is compiled and built into an assembly, say App.exe. When you run this application, the CLR loads and initializes. Then the CLR reads the assembly’s CLR header looking for the MethodDefToken that identifies the application’s entry point method (**Main**). From the MethodDef metadata table, the offset within the file for the method’s IL code is located and JIT–compiled into native code, which includes having the code verified for type safety, and the native code starts executing. Following is the IL code for the **Main** method. To obtain this output, I ran ILDasm.exe, selected the View menu’s Show Bytes menu item, and double–clicked the **Main** method in the tree view.

```
.method public hidebysig static void  Main(string[] args) cil managed
// SIG: 00 01 01 1D 0E
{
  .entrypoint
  // Method begins at RVA 0x2050
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  /* 72   | (70)000001       */ ldstr      "Hi"
IL_0005:  /* 28   | (0A)000002       */ call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  /* 2A   |                  */ ret
} // end of method App::Main
```

When JIT–compiling this code, the CLR detects all references to types and members and loads their defining assemblies (if not already loaded). As you can see, the IL code above has a reference to **System.Console.WriteLine**. Specifically, the IL **call** instruction references metadata token 0A000002. This token identifies an entry in the MemberRef metadata table. The CLR looks up this

MemberRef entry and sees that one of its fields refers to an entry in a TypeRef table (the **System.Console** type). From the TypeRef entry, the CLR is directed to an AssemblyRef entry: "MSCorLib, Version=1.0.3300.0, Culture="neutral", PublicKeyToken=b77a5c561934e089". At this point, the CLR knows which assembly it needs. Now the CLR must locate the assembly in order to load it.

When resolving a referenced type, the CLR can find the type in one of three places:

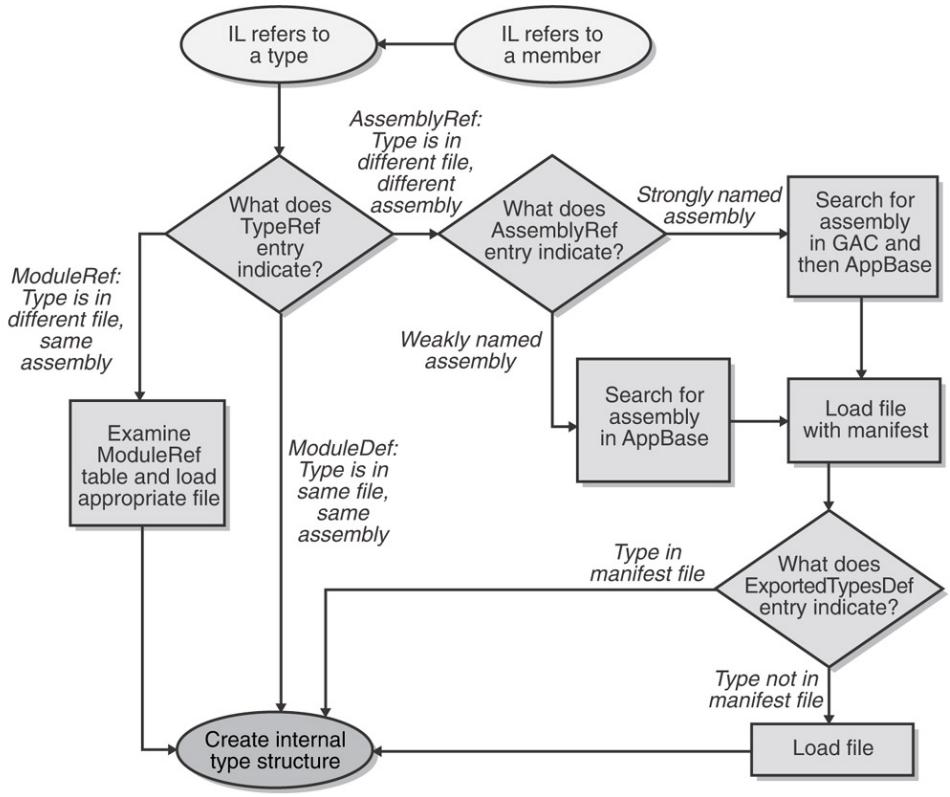
- **Same file** Access to a type that is in the same file is determined at compile time (sometimes referred to as *early bound)*. The type is loaded out of the file directly, and execution continues.
- **Different file, same assembly** The runtime ensures that the file being referenced is, in fact, in the assembly's FileRef table of the current assembly's manifest. The runtime then looks in the directory where the assembly's manifest file was loaded. The file is loaded, its hash value is checked to ensure the file's integrity, the type's member is found, and execution continues.
- **Different file, different assembly** When a referenced type is in a different assembly's file, the runtime loads the file that contains the referenced assembly's manifest. If this file doesn't contain the type, the appropriate file is loaded. The type's member is found, and execution continues.

**Note** The ModuleDef, ModuleRef, and FileDef metadata tables refer to files using the file's name and its extension. However, the AssemblyRef metadata table refers to assemblies by filename, without an extension. When binding to an assembly, the system automatically appends .dll and .exe file extensions while attempting to locate the file by probing the directories as mentioned in the section "Simple Administrative Control (Configuration)" in Chapter 2.

If any errors occur while resolving a type reference—file can't be found, file can't be loaded, hash mismatch, and so on—an appropriate exception is thrown.

In the previous example, the CLR sees that **System.Console** is implemented in a different assembly than the caller. The CLR must search for the assembly and load the PE file that contains the assembly's manifest. The manifest is then scanned to determine the PE file that implements the type. If the manifest file contains the referenced type, all is well. If the type is in another of the assembly's files, the CLR loads the other file and scans its metadata to locate the type. The CLR then creates its internal data structures to represent the type, and the JIT compiler completes the compilation for the **Main** method. Finally, the **Main** method can start executing.

Figure 3–6 illustrates how type binding occurs.




*Note*: If any operation fails, an appropriate exception is thrown.

Figure 3–6 : Flowchart showing how the CLR uses metadata to locate the proper assembly file that defines a type, given IL code that refers to a method or type

**Important** Strictly speaking, the example just described isn't 100 percent correct. For references to methods and types defined in an assembly other than MSCorLib.dll, the discussion is correct. However, MSCorLib.dll is closely tied to the version of the CLR that's running. Any assembly that references MSCorLib.dll (with the ECMA public key token of "b77a5c561934e089") always binds to the version of MSCorLib.dll that is in the same directory that contains the CLR itself. So in the previous example, the reference to **System.Console**'s **WriteLine** method binds to whatever version of MSCorLib.dll matches the version of the CLR, regardless of what version of MSCorLib.dll is referenced in the assembly's AssemblyRef metadata table.

In this section, you saw how the CLR locates an assembly when using default policies. However, an administrator or the publisher of an assembly can override the default policy. In the next two sections, I'll describe how to alter the CLR's default binding policy.

## Advanced Administrative Control (Configuration)

In the section "Simple Administrative Control (Configuration)" in Chapter 2, I gave a brief introduction to how an administrator can affect the way the CLR searches and binds to assemblies. In that section, I demonstrated how a referenced assembly's files can be moved to a subdirectory of the application's base directory and how the CLR uses the application's XML configuration file to locate the moved files.

Having discussed only the probing element's **privatePath** attribute in Chapter 2, I'm going to discuss the other XML configuration file elements in this section. Following is an XML configuration

file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="AuxFiles;bin\subdir" />

         <dependentAssembly>

            <assemblyIdentity name="JeffTypes"
               publicKeyToken="32ab4ba45e0a69a1" culture="neutral"/>

            <bindingRedirect
               oldVersion="1.0.0.0" newVersion="2.0.0.0" />

            <codeBase version="2.0.0.0"
               href="http://www.Wintellect.com/JeffTypes.dll" />

         </dependentAssembly>

         <dependentAssembly>

            <assemblyIdentity name="FredTypes"
               publicKeyToken="1f2e74e897abbcfe" culture="neutral"/>

            <bindingRedirect
               oldVersion="3.0.0.0-3.5.0.0" newVersion="4.0.0.0" />

            <publisherPolicy apply="no" />

         </dependentAssembly>

      </assemblyBinding>
   </runtime>
</configuration>
```

This XML file gives a wealth of information to the CLR. Here's what it says:

- **probing element** Look in the application base directory's AuxFiles and bin\subdir subdirectories when trying to find a weakly named assembly. For strongly named assemblies, the CLR looks in the GAC or in the URL specified by the **codeBase** element. The CLR looks in the application's private paths for a strongly named assembly only if no **codeBase** element is specified.
- **First dependentAssembly, assemblyIdentity, and bindingRedirect elements** When attempting to locate version 1.0.0.0 of the neutral culture JeffTypes assembly published by the organization that controls the **32ab4ba45e0a69a1** public key token, locate version 2.0.0.0 of the same assembly instead.
- **codeBase element** When attempting to locate version 2.0.0.0 of the neutral culture JeffTypes assembly published by the organization that controls the **32ab4ba45e0a69a1** public key token, try to find it at the following URL: *http://www.Wintellect.com/JeffTypes.dll*. Although I didn't mention it in Chapter 2, a **codeBase** element can also be used with weakly named assemblies. In this case, the assembly's version number is ignored and should be omitted from the XML's **codeBase** element. Also, the **codeBase** URL must refer to a directory under the application's base directory.
- **Second dependentAssembly, assemblyIdentity, and bindingRedirect elements** When attempting to locate version 3.0.0.0 through version 3.5.0.0 inclusive of the neutral culture FredTypes assembly published by the organization that controls the **1f2e74e897abbcfe**

public key token, locate version 4.0.0.0 of the same assembly instead.

- **publisherPolicy element** If the organization that produces the FredTypes assembly has deployed a publisher policy file (described in the next section), the CLR should ignore this file.

When compiling a method, the CLR determines the types and members being referenced. Using this information, the runtime determines—by looking in the referencing assembly's AssemblyRef table—what assembly was originally referenced when the calling assembly was built. The CLR then looks up the assembly in the application's configuration file and applies any version number redirections.

If the **publisherPolicy** element's **apply** attribute is set to **yes**—or if the element is omitted—the CLR examines the GAC and applies any version number redirections that the publisher of the assembly feels is necessary. I'll talk more about publisher policy in the next section.

The CLR then looks up the assembly in the machine's Machine.config file and applies any version number redirections there. Finally, the CLR knows the version of the assembly that it should load, and it attempts to load the assembly from the GAC. If the assembly isn't in the GAC and if there is no **codeBase** element, the CLR probes for the assembly as I described in Chapter 2. If the configuration file that performs the last redirection also contains a **codeBase** element, the CLR attempts to load the assembly from the **codeBase** element's specified URL.

Using these configuration files, an administrator can really control what assembly the CLR decides to load. If an application is experiencing a bug, the administrator can contact the publisher of the errant assembly. The publisher can send the administrator a new assembly that the administrator can install. By default, the CLR won't load this new assembly since the already built assemblies don't reference the new version. However, the administrator can modify the application's XML configuration file to instruct the CLR to load the new assembly.

If the administrator wants all applications on the machine to pick up the new assembly, the administrator can modify the machine's Machine.config file instead and the CLR will load the new assembly whenever an application refers to the old assembly.

If the new assembly doesn't fix the original bug, the administrator can delete the binding redirection lines from the configuration file and the application will behave as it did before. It's important to note that the system allows the use of an assembly that doesn't exactly match the assembly version recorded in the metadata. This extra flexibility is very handy. Later in this chapter, I'll go into more detail about how an administrator can easily repair an application.

---

The .NET Framework Configuration Tool

If you don't like manually editing XML text files—and who does?—you can use the .NET Framework Configuration tool, which ships with the .NET Framework. Open Control Panel, select Administrative Tools, and then select the Microsoft .NET Framework Configuration tool. While in the tool, you can select Configure An Assembly, which causes an assembly's Properties dialog box to pop up. From within this dialog box, you can set all of the XML configuration information. Figures 3–7, 3–8, and 3–9 show different pages of an assembly's properties dialog box.

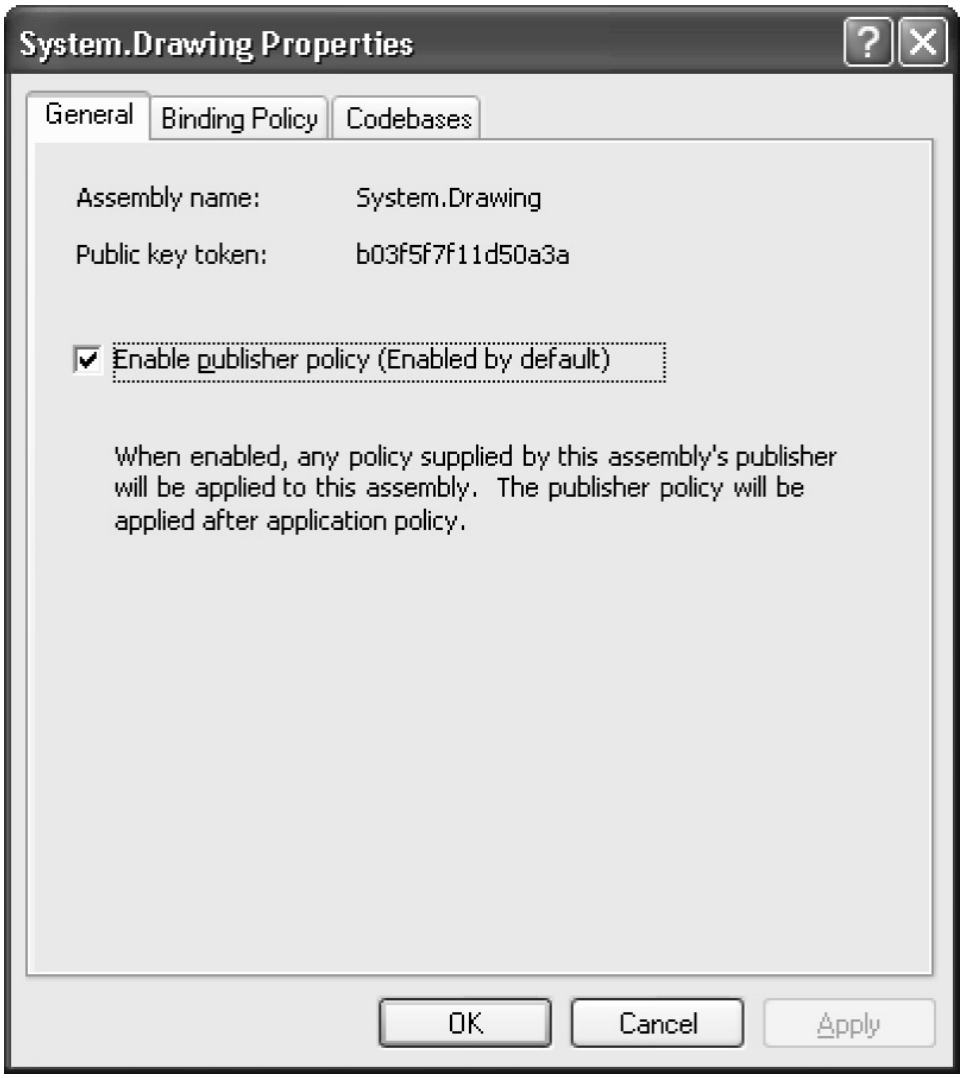


Figure 3–7 : General tab of the System.Drawing Properties dialog box

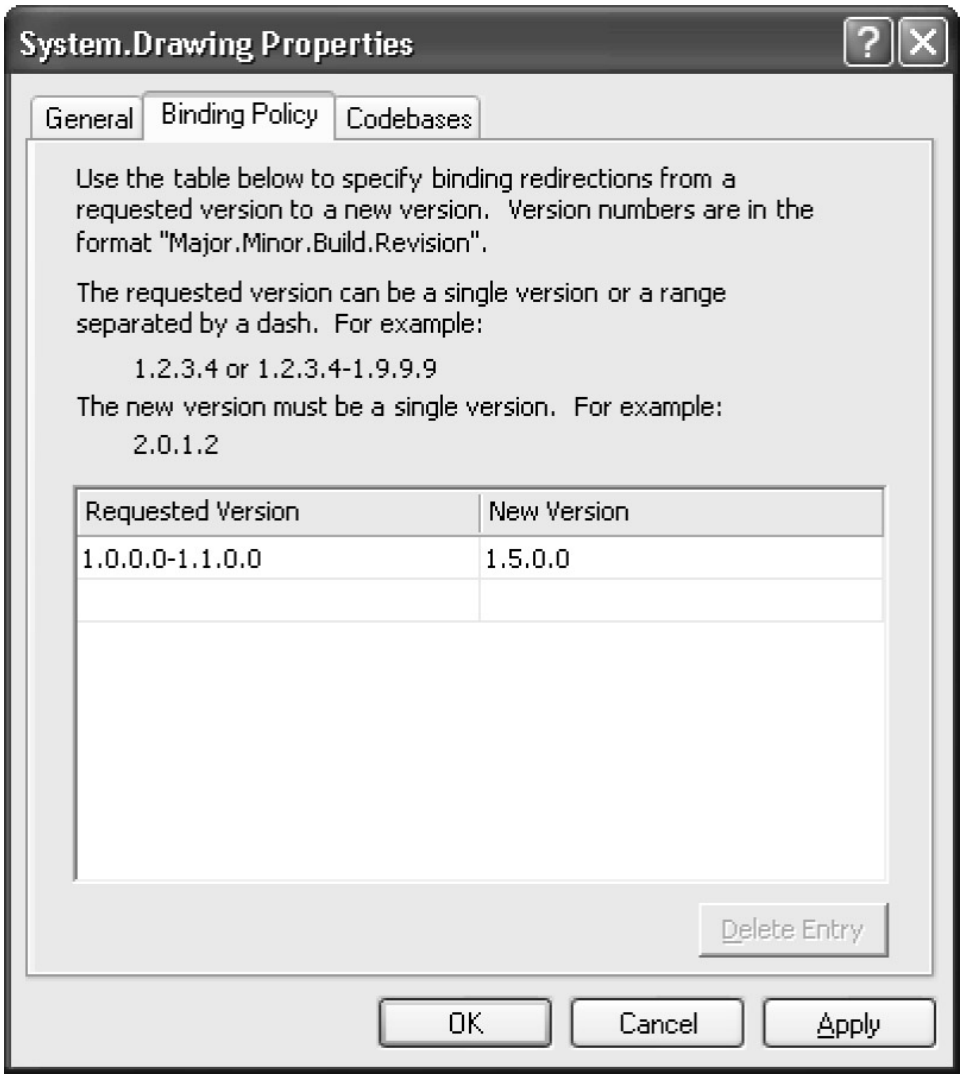
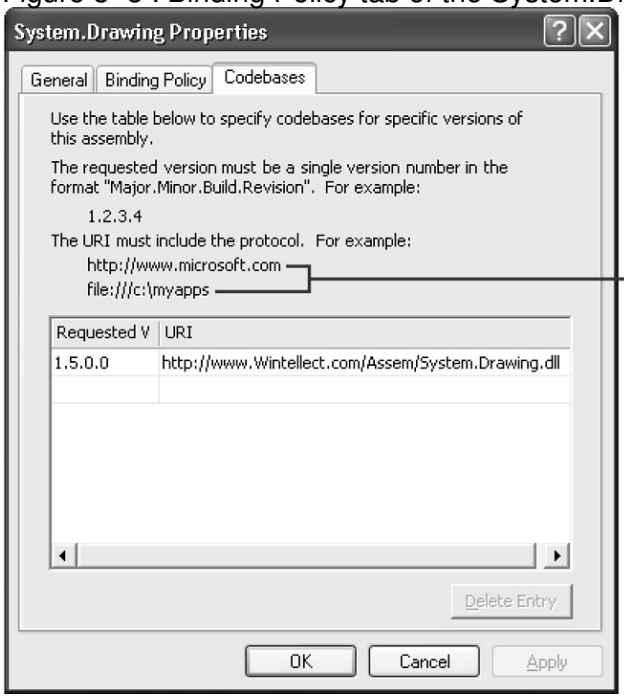



Figure 3–8 : Binding Policy tab of the System.Drawing Properties dialog box




These two examples are incorrect. The URI must include the filename and the extension, as shown in the pane below. Microsoft will fix these examples in a future version.

Figure 3–9 : Codebases tab of the System.Drawing Properties dialog box

## Publisher Policy Control

In the scenario described in the previous section, the publisher of an assembly simply sent a new version of the assembly to the administrator, who installed the assembly and manually edited the application's or machine's XML configuration files. In general, when a publisher fixes a bug in an assembly, she would like an easy way to package and distribute the new assembly to all the users. But she also needs a way to tell each user's CLR to use the new assembly version instead of the old assembly version. Sure, each user could modify his application's or machine's XML configuration file, but this is terribly inconvenient and is error–prone. What the publisher needs is a way to create "policy information" that is installed on the user's computer when the new assembly is installed. In this section, I'll show how an assembly's publisher can create this policy information.

Let's say you're a publisher of an assembly and that you've just created a new version of your assembly that fixes some bugs. When you package your new assembly to send out to all your users, you should also create an XML configuration file. This configuration file looks just like the configuration files we've been talking about. Here's an example file (called JeffTypes.config) for the JeffTypes.dll assembly:

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>

            <assemblyIdentity name="JeffTypes"
               publicKeyToken="32ab4ba45e0a69a1" culture="neutral"/>

            <bindingRedirect
               oldVersion="1.0.0.0" newVersion="2.0.0.0" />

            <codeBase version="2.0.0.0"
               href="http://www.Wintellect.com/JeffTypes.dll" />

         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Of course, a publisher can set policy only for the assemblies that it itself creates. In addition, the elements shown here are the only elements that can be specified in a publisher policy configuration file; you can't specify the **probing** or **publisherPolicy** elements, for example.

This configuration file tells the CLR to load version 2.0.0.0 of the JeffTypes assembly whenever version 1.0.0.0 of the assembly is referenced. Now you, the publisher, can create an assembly that contains this publisher policy configuration file. You create the publisher policy assembly by running AL.exe as follows:

```
AL.exe /out:policy.1.0.JeffTypes.dll
       /version:1.0.0.0
       /keyfile:MyCompany.keys
       /linkresource:JeffTypes.config
```

Let me explain the meaning of AL.exe's command–line switches:

- **The /out switch** This switch tells AL.exe to create a new PE file, called Policy.1.0.MyAsm.dll, which contains nothing but a manifest. The name of this assembly is very important. The first part of the name, Policy, tells the CLR that this assembly contains

publisher policy information. The second and third parts of the name, 1.0, tell the CLR that this publisher policy assembly is for any version of the JeffTypes assembly that has a major and minor version of 1.0. Publisher policies apply to the major and minor version numbers of an assembly only; you can't create a publisher policy that is specific to individual builds or revisions of an assembly. The fourth part of the name, JeffTypes, indicates the name of the assembly that this publisher policy corresponds to. The fifth and last part of the name, dll, is simply the extension given to the resulting assembly file.

- **The /version switch** This switch identifies the version of the publisher policy assembly; this version number has nothing to do with the JeffTypes assembly itself. You see, publisher policy assemblies can also be versioned. Today, the publisher might create a publisher policy redirecting version 1.0.0.0 of JeffTypes to version 2.0.0.0. In the future, the publisher might want to direct version 1.0.0.0 of JeffTypes to version 2.5.0.0. The CLR uses this version number so that it knows to pick up the latest version of the publisher policy assembly.
- **The /keyfile switch** This switch causes AL.exe to sign the publisher policy assembly using the publisher's public/private key pair. This key pair must also match the key pair used for all versions of the JeffTypes assembly. After all, this is how the CLR knows that the same publisher created both the JeffTypes assembly and this publisher policy file.
- **The /linkresource switch** This switch tells AL.exe that the XML configuration file is to be considered a separate file of the assembly. The resulting assembly consists of two files, both of which must be packaged and deployed to the users along with the new version of the JeffTypes assembly. By the way, you can't use AL.exe's /**embedresource** switch to embed the XML configuration file into the assembly file, making a single file assembly, because the CLR requires that the XML file be contained in its own, separate file.

Once this publisher policy assembly is built, it can be packaged together with the new JeffTypes.dll assembly file and deployed to users. The publisher policy assembly must be installed into the GAC. While the JeffTypes assembly can also be installed into the GAC, it doesn't have to be. It could be deployed into an application's base directory or some other directory identified by a **codeBase** URL.

**Important** A publisher should create a publisher policy assembly only when deploying a bug fix or a service pack version of an assembly. When installing an application "out of the box," no publisher policy assemblies should be installed.

I want to make one last point about publisher policy. Say that a publisher distributes a publisher policy assembly and for some reason the new assembly introduces more bugs than it fixes. If this happens, the administrator would like to tell the CLR to ignore the publisher policy assembly. To have the runtime do this, the administrator can edit the application's configuration file and add the following **publisherPolicy** element:

```
<publisherPolicy apply="no"/>
```

This element can be placed in the application's configuration file so that it applies to all assemblies, or it can be placed in the application's configuration file to have it apply to a specific assembly. When the CLR processes the application's configuration file, it will see that the GAC shouldn't be examined for the publisher policy assembly. So, the CLR will continue to operate using the older version of the assembly. Note, however, that the CLR will still examine and apply any policy specified in the Machine.config file.

**Important** A publisher policy assembly is a way for a publisher to make a statement about the compatibility of different versions of an assembly. If a new version of an assembly isn't intended to be compatible with an earlier version, the publisher shouldn't create a publisher policy assembly. In general, use a publisher policy assembly when you build

a new version of your assembly that fixes a bug. You should test the new version of the assembly for backward compatibility. On the other hand, if you're adding new features to your assembly, you should consider the assembly to have no relationship to a previous version and you shouldn't ship a publisher policy assembly. In addition, there's no need to do any backward compatibility testing with such an assembly.

## Repairing a Faulty Application

When a console or Windows Forms application is running under a user account, the CLR keeps a record of the assemblies that the application actually loads; a record isn't kept for ASP.NET Web Forms or XML Web services applications. This assembly load information is accumulated in memory and is written to disk when the application terminates. The files that contain this information are written to the following directory:

```
C:\Documents and Settings\UserName\Local Settings\
   Application Data\ApplicationHistory
```

*UserName* identifies the name of the logged–on user.

If you look in this directory, you'll see files like this:

```
Volume in drive C has no label.
 Volume Serial Number is 94FA-5DE7

 Directory of C:\Documents and Settings\v-jeffrr\Local Settings\ApplicationHistory

08/23/2001  10:46 AM    <DIR>          .
08/23/2001  10:46 AM    <DIR>          ..
08/22/2001  04:14 PM             1,014 App.exe.c4bc1771.ini
08/23/2001  10:46 AM             2,845 ConfigWizards.exe.c4c8182.ini
08/14/2001  05:51 PM             9,815 devenv.exe.49453f8d.ini
08/22/2001  02:25 PM             3,226 devenv.exe.7dc18209.ini
08/23/2001  10:46 AM             3,368 mmc.exe.959a7e97.ini
08/15/2001  03:06 PM             2,248 RegAsm.exe.18b34bd3.ini
               6 File(s)         22,516 bytes
               2 Dir(s)  14,698,717,184 bytes free
```

Each file identifies a particular application. The hexadecimal number is a hash value identifying the file's path and is needed to distinguish two files with the same name residing in different subdirectories.

As an application runs, the CLR maintains a "snapshot" of the set of assemblies loaded by the application. When the application terminates, this information is compared with the information in the application's corresponding .ini file. If the application loaded the same set of assemblies that it loaded previously, the information in the .ini file matches the information in memory and the in–memory information is discarded. If on the other hand the in–memory information differs from the information in the .ini file, the CLR appends the in–memory information to the .ini file. By default, the .ini file is able to store up to five snapshots.

Basically, the CLR is keeping a record of the assemblies that an application used. Now, let's say that you install some new assemblies and maybe some new publisher policy assembly gets installed too. A week later, you run an application and all of a sudden, the application isn't performing correctly. What can you do? Historically, in Windows, the best thing to do would be to reinstall the failing application and hope that the reinstall doesn't break some other application

(which would be likely).

Fortunately for the end–user, the CLR keeps a historical record of the assemblies that an application uses. All you have to do is build an XML configuration file for the application where the elements tell the CLR to use the same assemblies that were loaded when the application was in a last–known good state.

To make creating or modifying an application configuration file easy, you can use the .NET Framework Configuration tool. Run the tool, right–click on the Application node in the tree pane, and select the Fix An Application menu item. This causes the dialog box in Figure 3–10 to appear.

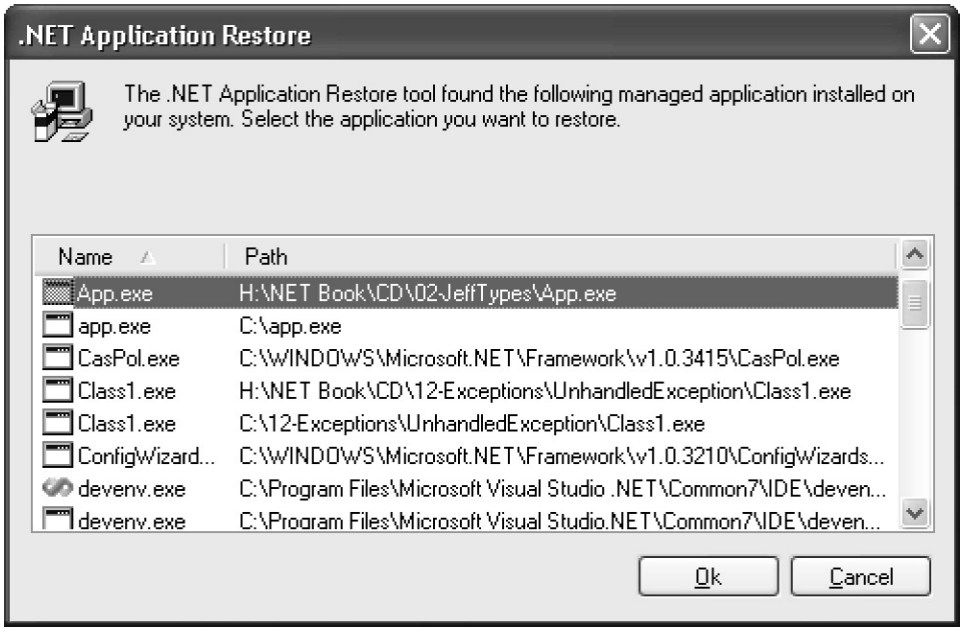



Figure 3–10 : .NET Application Configuration tool showing all applications that have had assembly load information recorded at one time or another

**Note** The .NET Framework Configuration tool is a Microsoft Management Console (MMC) snap–in and, therefore, isn't installed on Windows 98, Windows 98 Standard Edition, or Windows Me. However, on these operating systems, you can use the .NET Framework Wizards utility to do what I describe in this section. You can invoke this tool from the Start menu by clicking Program Files, then Administrative Tools, and then .NET Framework Wizards.

The dialog box in Figure 3–10 shows the applications about which the CLR has accumulated assembly load information. Basically, an entry appears here for every .ini file in the ApplicationHistory subdirectory. Once you select an application, the dialog box in Figure 3–11 appears.

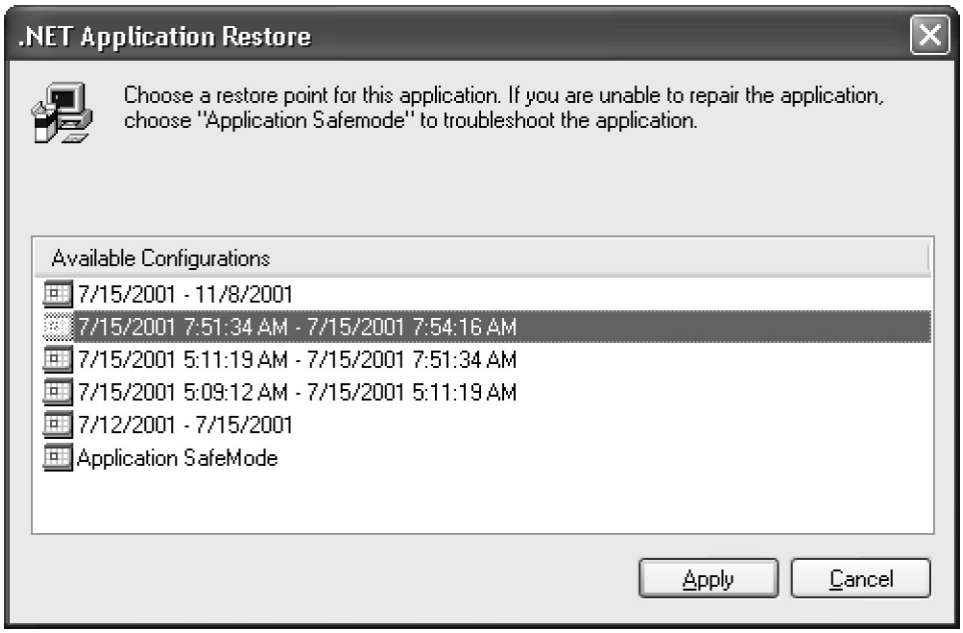


Figure 3–11 : .NET Application Configuration tool showing the dates when loaded assemblies differed

Each entry in this dialog box represents the set of assemblies that were loaded by the application. The user can select a date range during which the application was working correctly, and the tool will create or modify the application's XML configuration file so that the CLR will now load the last–known good set of assemblies for the application. The Application SafeMode entry ensures that the application loads with the same exact set of assemblies that it was built and tested with; it prevents the runtime from redirecting an assembly to a different version.

Changes to the application's XML configuration file can be identified by surrounding comment elements that contain ".NET Application Restore BeginBlock" and ".NET Application Restore EndBlock". The ".NET Application Restore RollBackBlock" contains the original XML configuration prior to restoring to a specific snapshot. Here's an example:

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>

            <!–.NET Application Restore BeginBlock #1 29437104.-387708080
                8/24/2001 6:48:25 PM–>

            <assemblyIdentity name="JeffTypes"
               publicKeyToken="32ab4ba45e0a69a1" culture="neutral" />

            <bindingRedirect
               oldVersion="1.0.0.0" newVersion="2.0.0.0" />

            <publisherPolicy apply="no"/>
            <!–.NET Application Restore EndBlock #1–>

            <!–.NET Application Restore RollBackBlock #1
                8/24 2001 6:48:25 PM<assemblyIdentity name="JeffTypes"
                publicKeyToken="32ab4ba45e0a69a1" culture=""/>–>

         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

# Part II: Working with Types and the Common Language Runtime

## Chapter List

*Chapter 4: Type Fundamentals*
*Chapter 5: Primitive, Reference, and Value Types*
*Chapter 6: Common Object Operations*

# Chapter 4: Type Fundamentals

In this chapter, I'm going to introduce the information that is fundamental to working with types and the common language runtime (CLR). In particular, I'll discuss the minimum set of behaviors that you can expect every type to have. I'll also describe type safety and the various ways you can cast objects from one type to another. Finally, I'll talk about namespaces and assemblies.

## All Types Are Derived from System.Object

The runtime requires that every object ultimately be derived from the **System.Object** type. This means that the following two type definitions (shown using C#) are identical:

```
// Implicitly derive from Object
class Employee {
   Ã
}
```

```
// Explicitly derive from Object
class Employee : System.Object {
   Ã
}
```

Because all object types are ultimately derived from **System.Object**, you are guaranteed that every object of every type has a minimum set of methods. Specifically, the **System.Object** class offers the public instance methods listed in Table 4–1.

Table 4–1 : Public Methods of System.Object

| Public Method | Description |
|---|---|
| **Equals** | Returns **true** if two objects have the same value. For more information about this method, see Chapter 6. |
| **GetHashCode** | Returns a hash code for this object's value. A type should override this method if its objects are to be used as a key in a hash table. The method should provide a good distribution for its objects. For more information about this method, see Chapter 6. |
| **ToString** | By default, returns the full name of the type (**this.Get–Type().FullName.ToString()**). However, it is common to override this method so that it returns a **String** object containing a string representation of the object's state. For example, the core types, such as **Boolean** and **Int32**, override this method to return a string representation of their values. It is also common to override this method for debugging purposes: you can call it and get a string showing the values of the object's fields. Note that **ToString** is expected to be aware of the **CultureInfo** associated with the calling thread. Chapter 12 discusses **ToString** in greater detail. |
| **GetType** | Returns an instance of a **Type**–derived object that identifies the type of this object. The returned **Type** object can be used with the Reflection classes to obtain metadata information about the type. Reflection is discussed in Chapter 20. The **GetType** method is nonvirtual, which prevents a class from overriding the method and lying about its type, violating type safety. |

In addition, types that derive from **System.Object** have access to the protected methods listed in Table 4–2.

Table 4–2 : Protected Methods of System.Object

| Protected Method | Description |
|---|---|
| **MemberwiseClone** | This nonvirtual method creates a new instance of the type and sets the new object's fields to be identical to **this** object's fields. A reference to the new instance is returned. For more information about this method, see Chapter 6. |
| **Finalize** | This virtual method is called when the garbage collector determines that the object is garbage but before the memory for the object is reclaimed. Types that require cleanup when collected should override this method. I'll talk about this important method in much more detail in Chapter 19. |

The CLR requires that all objects be created using the **new** operator (which emits the **newobj** IL instruction). The following line shows how to create an **Employee** object:

```
Employee e = new Employee("ConstructorParam1");
```

Here's what the **new** operator does:

1. It allocates memory for the object by allocating the number of bytes required for the specified type from the managed heap.
2. It initializes the object's overhead members. Every object instance has two additional members associated with the instance that the CLR uses to manage the object. The first member is the object's pointer to the type's method table, and the second member is a SyncBlockIndex.
3. The type's instance constructor is called, passing it any parameters (the string "**ConstructorParam1**", in the preceding example) specified in the call to **new**. Although most languages compile constructors so that they call the base type's constructor, the CLR doesn't require this call.

After **new** has performed all these operations, it returns a reference to the newly created object. In the preceding code example, this reference is saved in the variable **e**, which is of type **Employee**.

By the way, the **new** operator has no complementary **delete** operator; that is, there is no way to explicitly free the memory allocated for an object. The CLR imposes a garbage–collected environment (described in Chapter 19) that automatically detects when objects are no longer being used or accessed and frees the object's memory automatically.

## Casting Between Types

One of the most important features of the CLR is its type safety. At run time, the CLR always knows what type an object is. You can always discover an object's exact type by calling the **GetType** method. Because this method is nonvirtual, it is impossible for a type to spoof another type. For example, the **Employee** type can't override the **GetType** method and have it return a type of **SpaceShuttle**.

Developers frequently find it necessary to cast an object to various types. The CLR allows you to cast an object to its type or to any of its base types. Your programming language of choice decides how to expose casting operations to the developer. For example, C# doesn't require any special

syntax to cast an object to any of its base types because casts to base types are considered safe implicit conversions. However, C# does require that the developer explicitly cast an object to any of its derived types since such a cast could fail. The following code demonstrates casting to base and derived types:

```
// This type is implicitly derived from System.Object.
class Employee {
   Ã
}


class App {
   public static void Main() {
      // No cast needed since new returns an Employee object
      // and Object is a base type of Employee.
      Object o = new Employee();

      // Cast required since Employee is derived from Object.
      // Other languages (such as Visual Basic) might not require
      // this cast to compile.
      Employee e = (Employee) o;
   }
}
```

This example shows what is necessary for your compiler to compile your code. Now I'll explain what happens at run time. At run time, the CLR checks casting operations to ensure that casts are always to the object's actual type or any of its base types. For example, the following code will compile, but at run time, an **InvalidCastException** exception will be thrown:

```
class Manager : Employee {
   Ã
}


class App {
   public static void Main() {
      // Construct a Manager object and pass it to PromoteEmployee.
      // A Manager IS-A Object: PromoteEmployee runs OK.
      Manager m = new Manager();
      PromoteEmployee(m);

      // Construct a DateTime object and pass it to PromoteEmployee.
      // A DateTime is NOT derived from Employee: PromoteEmployee
      // throws a System.InvalidCastException exception.
      DateTime newYears = new DateTime(2001, 1, 1);
      PromoteEmployee(newYears);
   }


   public void PromoteEmployee(Object o) {
      // At this point, the compiler doesn't know exactly what
      // type of object o refers to. So the compiler allows the
      // code to compile. However, at run time, the CLR does know
      // what type o refers to (each time the cast is performed) and
      // it checks whether the object's type is Employee or any type
      // that is derived from Employee.
      Employee e = (Employee) o;
      Ã
   }
}
```

In the **Main** method, a **Manager** object is constructed and passed to **Promote–Employee**. This code compiles and executes because **Manager** is derived from **Object**, which is what **PromoteEmployee** expects. Once inside **PromoteEmployee**, the CLR confirms that **o** refers to an object that is either an **Employee** or a type that is derived from **Employee**. Because **Manager** is derived from **Employee**, the CLR performs the cast and allows **PromoteEmployee** to continue executing.

After **PromoteEmployee** returns, **Main** constructs a **DateTime** object and passes it to **PromoteEmployee**. Again, **DateTime** is derived from **Object**, so the compiler compiles the code that calls **PromoteEmployee**. However, inside **PromoteEmployee**, the CLR checks the cast and detects that **o** refers to a **DateTime** object and is therefore not an **Employee** or any type derived from **Employee**. At this point, the CLR can't allow the cast and throws a **System.InvalidCastException** exception.

If the CLR allowed the cast, there would be no type safety and the results would be unpredictable, including the possibility of application crashes and security breaches caused by the ability of types to easily spoof other types. Type spoofing is the cause of many security breaches and compromises an application's stability and robustness. Type safety is therefore an extremely important part of the .NET Framework.

By the way, the proper way to prototype the **PromoteEmployee** method would be to have it take an **Employee** instead of an **Object** as a parameter. I used **Object** so that I could demonstrate how the compilers and the CLR deal with casting.

## Casting with the C# is and as Operators

C# offers another way to cast using the **is** operator. The **is** operator checks whether an object is compatible with a given type, and the result of the evaluation is a **Boolean**: **true** or **false**. The **is** operator will never throw an exception. The following code demonstrates:

```
System.Object o = new System.Object();
System.Boolean b1 = (o is System.Object);  // b1 is true.
System.Boolean b2 = (o is Employee);       // b2 is false.
```

If the object reference is **null**, the **is** operator always returns **false** because there is no object available to check its type.

The **is** operator is typically used as follows:

```
if (o is Employee) {
   Employee e = (Employee) o;
   // Use e within the 'if' statement.
}
```

In this code, the CLR is actually checking the object's type twice: the **is** operator first checks to see if **o** is compatible with the **Employee** type. If it is, then inside the **if** statement, the CLR again verifies that **o** refers to an **Employee** when performing the cast. Because this programming paradigm is quite common, C# offers a way to simplify this code and improve its performance by providing an **as** operator:

```
Employee e = o as Employee;
if (e != null) {
   // Use e within the 'if' statement.
}
```

In this code, the CLR checks if **o** is compatible with the **Employee** type, and if it is, **as** returns a non−null pointer to the same object. If **o** is not compatible with the **Employee** type, then the **as** operator returns **null**. Notice that the **as** operator causes the CLR to verify an object's type just once. The **if** statement simply checks whether or not **e** is **null**—this check can be performed much more efficiently than verifying an object's type.

The **as** operator works just like casting except the **as** operator will never throw an exception. Instead, if the object can't be cast, the result is **null**. You'll want to check to see whether the resulting reference is **null**, or attempting to use the resulting reference will cause a **System.NullReferenceException** exception to be thrown. The following code demonstrates:

```
System.Object o = new System.Object();  // Creates a new Object object
Employee e = o as Employee;             // Casts o to an Employee
// The cast above fails: no exception is thrown, but e is set to null.

e.ToString();  // Accessing e throws a NullReferenceException.
```

To make sure you understand everything just presented, take the following quiz. Assume that these two class definitions exist:

```
class B {
   Int32 x;
}

class D : B {
   Int32 y;
}
```

Now examine the lines of C# code in Table 4−4. For each line, decide whether the line would compile and execute successfully (marked OK below), cause a compile−time error (CTE), or cause a run−time error (RTE).

Table 4−4 : Type−Safety Quiz

| Statement | OK | CTE | RTE |
|---|---|---|---|
| **System.Object o1 = new System.Object();** | ü | | |
| **System.Object o2 = new B();** | ü | | |
| **System.Object o3 = new D();** | ü | | |
| **System.Object o4 = o3;** | ü | | |
| **B b1 = new B();** | ü | | |
| **B b2 = new D();** | ü | | |
| **D d1 = new D();** | ü | | |
| **B b3 = new System.Object();** | | ü | |
| **D d3 = new System.Object();** | | ü | |
| **B b3 = d1;** | ü | | |

| | | | |
|---|---|---|---|
| **D d2 = b2;** | | ü | |
| **D d4 = (D) d1;** | ü | | |
| **D d5 = (D) b2;** | ü | | |
| **D d6 = (D) b1;** | | | ü |
| **B b4 = (B) o1;** | | | ü |
| **B b5 = (D) b2;** | ü | | |

## Namespaces and Assemblies

Namespaces allow for the logical grouping of related types, and developers typically use them to make it easier to locate a particular type. For example, the **System.Collections** namespace defines a bunch of collection types, and the **System.IO** namespace defines a bunch of types for performing I/O operations. Here's some code that constructs a **System.IO.FileStream** object and a **System.Collections.Queue** object:

```
class App {
   static void Main() {
      System.IO.FileStream fs = new System.IO.FileStream(...);
      System.Collections.Queue q = new System.Collections.Queue();
   }
}
```

As you can see, the code is pretty verbose; it would be nice if there were some shorthand way to refer to the **FileStream** and **Queue** types to reduce typing. Fortunately, many compilers do offer mechanisms to reduce programmer typing. The C# compiler provides this mechanism via the **using** directive, and Visual Basic provides it via the **Imports** statement. The following code is identical to the previous example:

```
// Include some namespaces in my C# application:
using System.IO;           // Try prepending "System.IO."
using System.Collections;  // Try prepending "System.Collections."

class App {
   static void Main() {
      FileStream fs = new FileStream(...);
      Queue q = new Queue();
   }
}
```

To the compiler, a namespace is simply an easy way of making a type's name longer and more likely to be unique by preceding the name with some symbols separated by dots. So the compiler interprets the reference to FileStream in this example to mean System.IO.FileStream. Similarly, the compiler interprets the reference to Queue to mean System.Collections.Queue.

Using the C# **using** directive and the **Imports** statement in Visual Basic is entirely optional; you're always welcome to type out the fully qualified name of a type if you prefer. The C# **using** directive instructs the compiler to "try" prepending different prefixes to a type name until a match is found.

**Important** The CLR doesn't know anything about namespaces. When you access a type, the CLR needs to know the full name of the type and which assembly contains the definition of the type so that the runtime can load the proper assembly, find the type, and manipulate it.

In the previous code example, the compiler needs to ensure that every type referenced exists and that my code is using that type in the correct way: calling methods that exist, passing the right number of arguments to these methods, ensuring that the arguments are the right type, using the method's return value correctly, and so on. If the compiler can't find a type with the specified name in the source files or in any referenced assemblies, it tries prepending "**System.IO.**". to the type name and checks if the generated name matches an existing type. If the compiler still can't find a match, it tries prepending "**System.Collections.**" to the type's name. The two **using** directives shown earlier allow me to simply type "**FileStream**" and "**Queue**" in my code—the compiler automatically expands the references to "**System.IO.FileStream**" and "**System.Collections.Queue**". I'm sure you can easily imagine how much typing this saves.

When checking for a type's definition, the compiler must be told which assemblies to examine. The complier will scan all the assemblies it knows about, looking for the type's definition. Once the compiler finds the proper assembly, the assembly information and the type information is emitted into the resulting managed module's metadata. To get the assembly information, you must pass the assembly that defines any referenced types to the compiler. The C# compiler, by default, automatically looks in the MSCorLib.dll assembly even if you don't explicitly tell it to. The MSCorLib.dll assembly contains the definitions of all the core .NET Framework Class Library (FCL) types, such as **Object**, **Int32**, **String**, and so on.

**Note** When Microsoft first started working on the .NET Framework, MSCorLib.dll was an acronym for Microsoft Common Object Runtime Library. Once ECMA started to standardize the CLR and parts of the FCL, MSCorLib.dll officially became the acronym for Multilanguage Standard Common Object Runtime Library. :

As you might imagine, there are some potential problems with the way that compilers treat namespaces: it's possible to have two (or more) types with the same name in different namespaces. Microsoft strongly recommends that you define unique names for types. However, in some cases, it's simply not possible. The runtime encourages the reuse of components. Your application might take advantage of a component that Microsoft created and another component that Wintellect created. These two companies might both offer a type called **Widget**—Microsoft's **Widget** does one thing, and Wintellect's **Widget** does something entirely different. In this scenario, you had no control over the naming of the types, so you can differentiate between the two widgets by using their fully qualified names when referencing them. To reference Microsoft's **Widget**, you would use **Microsoft.Widget**, and to reference Wintellect's **Widget**, you would use **Wintellect.Widget**.

In the following code, the reference to **Widget** is ambiguous and the C# compiler generates the following: "error CS0104: 'Widget' is an ambiguous reference":

```
using Microsoft;  // Try prepending "Microsoft".
using Wintellect; // Try prepending "Wintellect".

class MyApp {
   static void Main() {
      Widget w = new Widget();    // An ambiguous reference
   }
}
```

To remove the ambiguity, you must explicitly tell the compiler which **Wid–get** you want to create:

```
using Microsoft;  // Try prepending "Microsoft."
using Wintellect; // Try prepending "Wintellect."

class MyApp {
   static void Main() {
```

```
      Wintellect.Widget w = new Wintellect.Widget(); // Not ambiguous
   }
}
```

There's another form of the C# **using** directive that allows you to create an alias for a single type or namespace. This is handy if you have just a few types that you use from a namespace and don't want to pollute the global namespace with all of a namespace's types. The following code demonstrates another way to solve the ambiguity problem shown in the preceding code:

```
using Microsoft;  // Try prepending "Microsoft".
using Wintellect; // Try prepending "Wintellect".

// Define WintellectWidget symbol as an alias to Wintellect.Widget
using WintellectWidget = Wintellect.Widget;

class MyApp {
   static void Main() {
      WintellectWidget w = new WintellectWidget(); // No error now
   }
}
```

These methods of disambiguating a type are useful, but in some scenarios, you need to go further. Imagine that the Australian Boomerang Company (ABC) and the Alaskan Boat Corporation (ABC) are each creating a type, called **BuyProduct**, which they intend to ship in their respective assemblies. It's likely that both companies would create a namespace called **ABC** that contains a type called **BuyProduct**. Anyone who tries to develop an application that needs to buy both boomerangs and boats would be in for some trouble unless the programming language provides a way to programmatically distinguish between the assemblies, not just between the namespaces.

Unfortunately, the C# **using** directive only supports namespaces; it doesn't offer any way to specify an assembly. However, in the real world, this problem doesn't come up very often and is rarely an issue. If you're designing component types that you expect third parties to use, you should define these types in a namespace so that compilers can easily disambiguate types. In fact, to reduce the likelihood of conflict, you should use your full company name (not an acronym) to be your top–level namespace name. Referring to the .NET Framework SDK documentation, you can see that Microsoft uses a namespace of "Microsoft" for Microsoft–specific types. (See the **Microsoft.CSharp**, **Microsoft.VisualBasic**, and **Microsoft.Win32** namespaces as examples.)

Creating a namespace is simply a matter of writing a namespace declaration into your code as follows (in C#):

```
namespace CompanyName {    // CompanyName
   class A {               // CompanyName.A
      class B { ... }      // CompanyName.A.B
   }

   namespace X {           // CompanyName.X
      class C { ... }      // CompanyName.X.C
   }
}
```

Some compilers don't support namespaces at all, and other compilers are free to define what "namespace" means to a particular language. In C#, namespaces are implicitly public and you can't change this by using any access modifiers. However, C# does allow you to define types within a namespace that are internal (can't be used outside the assembly) or public (can be accessed by any assembly).

Be aware that a namespace and an assembly (the file that implements a type) aren't necessarily related. In particular, the various types belonging to a single namespace might be implemented in multiple assemblies. For example, the **System.Collections.ArrayList** type is implemented in the MSCorLib.dll assembly, and the **System.Collections.StringCollection** type is implemented in the System.dll assembly. In fact, there is no System.Collections.dll assembly.

A single assembly can house types in different namespaces. For example, the **System.Int32** and **System.Collections.ArrayList** types are both in the MSCorLib.dll assembly.

When you look up a type in the .NET Framework SDK documentation, the documentation will clearly indicate the namespace that the type belongs to and also what assembly the type is implemented in. In the Requirements section in Figure 4–1, you can see that the **ResXFileRef** type belongs in the **System.Resources** namespace but is implemented in the System.Windows.Forms.dll assembly. To compile code that references the **ResXFileRef** type, you'd add a **using System.Resources;** directive to your source code and you'd use the **/r:System.Windows.dll** compiler switch.

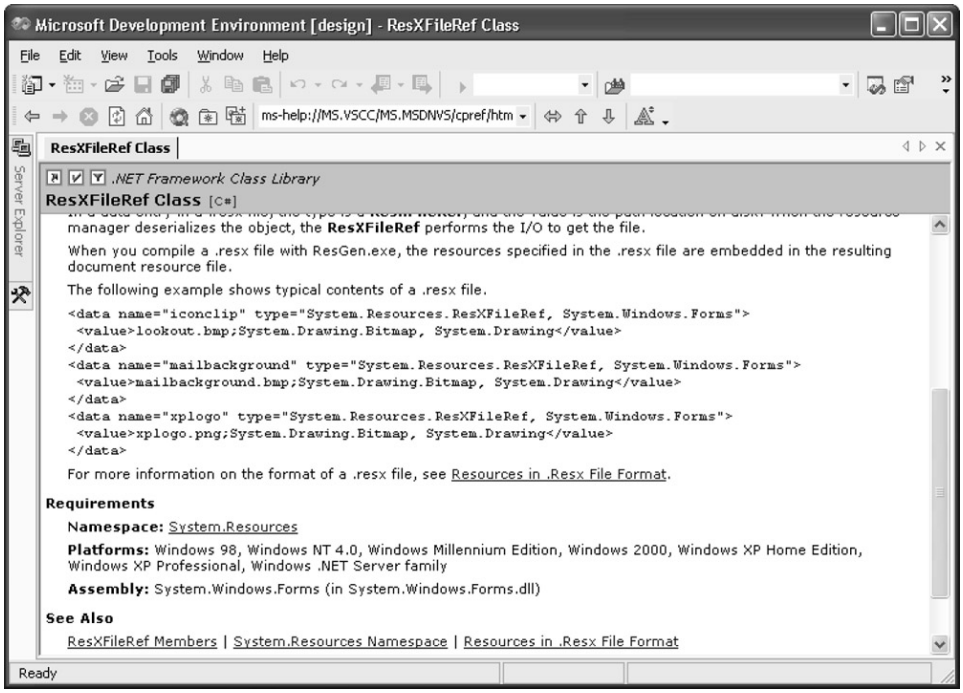


Figure 4–1 : Requirements section showing namespace and assembly information for a type

# Chapter 5: Primitive, Reference, and Value Types

In this chapter, I'll discuss the different kinds of types that you'll run into as a Microsoft .NET Framework developer. It is crucial that all developers be familiar with the different behaviors that these types exhibit. When I was first learning the .NET Framework, I didn't fully understand the difference between primitive, reference, and value types. This lack of clarity led me to unwittingly introduce subtle bugs and performance issues into my code. By explaining the differences between the types here, I'm hoping to save you some of the headaches I experienced while getting up to speed.

## Programming Language Primitive Types

Certain data types are so commonly used that many compilers allow code to manipulate them using simplified syntax. For example, you could allocate an integer using the following syntax:

```
System.Int32 a = new System.Int32();
```

But I'm sure you'd agree that declaring and initializing an integer using this syntax is rather cumbersome. Fortunately, many compilers (including C#) allow you to use syntax similar to the following instead:

```
int a = 0;
```

This syntax certainly makes the code more readable, and, of course, the intermediate language (IL) generated is identical no matter which syntax is used. Any data types the compiler directly supports are called *primitive types*. Primitive types map directly to types existing in the .NET Framework Class Library (FCL). For example, in C#, an **int** maps directly to the **System.Int32** type. Because of this, the following four lines of code all compile correctly and produce the exact same IL:

```
int          a = 0;                   // Most convenient syntax
System.Int32 a = 0;                   // Convenient syntax
int          a = new int();           // Inconvenient syntax
System.Int32 a = new System.Int32();  // Inconvenient syntax
```

Table 5–1 shows the FCL types that have corresponding primitives in C#. For the types that are compliant with the Common Language Specification (CLS), other languages will offer similar primitive types. However, languages aren't required to offer any support for the non–CLS–compliant types.

Table 5–1: FCL Types with Corresponding C# Primitives

| C# Primitive Type | FCL Type | CLS–Compliant | Description |
|---|---|---|---|
| **sbyte** | **System.SByte** | No | Signed 8–bit value |
| **byte** | **System.Byte** | Yes | Unsigned 8–bit value |
| **short** | **System.Int16** | Yes | Signed 16–bit value |
| **ushort** | **System.UInt16** | No | Unsigned 16–bit value |
| **int** | **System.Int32** | Yes | Signed 32–bit value |
| **uint** | **System.UInt32** | No | Unsigned 32–bit value |
| **long** | **System.Int64** | Yes | Signed 64–bit value |
| **ulong** | **System.UInt64** | No | Unsigned 64–bit value |

| | | | |
|---|---|---|---|
| **char** | **System.Char** | Yes | 16–bit Unicode character (**char** never represents an 8–bit value as it would in unmanaged C++.) |
| **float** | **System.Single** | Yes | IEEE 32–bit **float** |
| **double** | **System.Double** | Yes | IEEE 64–bit **float** |
| **bool** | **System.Boolean** | Yes | A **True**/**False** value |
| **decimal** | **System.Decimal** | Yes | A 128–bit high–precision floating–point value commonly used for financial calculations where rounding errors can't be tolerated. Of the 128 bits, 1 bit represents the sign of the value, 96 bits represent the value itself, and 8 bits represent the power of 10 to divide the 96–bit value by (can be anywhere from 0 to 28). The remaining bits are unused. |
| **object** | **System.Object** | Yes | Base type of all types |
| **string** | **System.String** | Yes | An array of characters |

The C# language specification states, "As a matter of style, use of the keyword is favored over use of the complete system type name." I disagree with the language specification; I prefer to use the FCL type names and completely avoid the primitive type names. In fact, I wish that compilers didn't even offer the primitive type names and forced developers to use the FCL type names instead. Here are my reasons:

- I've seen a number of developers confused, not knowing whether to use **string** or **String** in their code. Because, in C#, the **string** (a keyword) maps exactly to **System.String** (a FCL type), there is no difference and either can be used.
- In C#, **long** maps to **System.Int64**, but in a different programming language, **long** could map to an **Int16** or **Int32**. In fact, C++ with Managed Extensions does in fact treat **long** as an **Int64**. Someone reading source code in one language could easily misinterpret the code's intention if they were used to programming in a different programming language. In fact, most languages won't even treat **long** as a keyword and won't compile code that uses it.
- The FCL has many types with methods that have type names in the method. For example, the **BinaryReader** type offers methods such as **ReadBoolean**, **ReadInt32**, **ReadSingle**, and so on, and the **System.Convert** type offers methods such as **ToBoolean**, **ToInt32**, **ToSingle**, and so on. Although it's legal to write the following code, the line with **float** feels very unnatural to me, and it's not obvious that the line is correct:

```
BinaryReader br = new BinaryRead(...);
float  val = br.ReadSingle();   // OK, but feels unnatural
Single val = br.ReadSingle();   // OK and feels good
```

For all these reasons, I'll use the FCL type names throughout this book.

In many programming languages, you would expect the following code to compile and execute correctly:

```
Int32  i = 5;   // A 32-bit value
Int64  l = i;   // Implicit cast to a 64-bit value
```

However, based on the casting discussion presented in Chapter 4, you wouldn't expect this code to compile. After all, **System.Int32** and **System.Int64** are different types. Well, you'll be happy to know that the C# compiler does compile this code correctly and it runs as expected. Why?

The reason is that the C# compiler has intimate knowledge of primitive types and applies its own special rules when compiling the code. In other words, your compiler of choice recognizes common programming patterns and produces the necessary IL to make the written code work as expected. Specifically, compilers typically support patterns related to casting, literals, and operators, as shown in the following examples.

First, the compiler is able to perform implicit or explicit casts between primitive types like these:

```
Int32  i = 5;         // Implicit cast from Int32  to Int32
Int64  l = i;         // Implicit cast from Int32  to Int64
Single s = i;         // Implicit cast from Int32  to Single
Byte   b = (Byte) i;  // Explicit cast from Int32  to Byte
Int16  v = (Int16) s; // Explicit cast from Single to Int16
```

C# allows implicit casts if the conversion is "safe," that is, no loss of data is possible, such as converting an **Int32** to an **Int64**. But C# requires explicit casts if the conversion is potentially unsafe. For numeric types, "unsafe" means that you could lose precision or magnitude as a result of the conversion. For example, converting from **Int32** to **Byte** requires an explicit cast because precision might be lost from large **Int32** numbers; converting from **Single** to **Int64** requires a cast because **Single** can represent numbers of a larger magnitude than **Int64** can.

Be aware that different compilers can generate different code to handle these cast operations. For example, when casting a **Single** with a value of 6.8 to an **Int32**, some compilers could generate code to put a 6 in the **Int32**, and others could perform the cast by rounding the result up to 7. By the way, C# always truncates the result. For the exact rules that C# follows for casting primitive types, see the "Conversions" section in the C# language specification.

**Note** If you find yourself working with types that your compiler of choice doesn't support as primitive types, you might be able to use the static methods of the **System.Convert** type to help you cast between objects of different types. The **Convert** type knows how to convert between objects within the various core types in the FCL: **Boolean**, **Char**, **SByte**, **Byte**, **Int16**, **UInt16**, **Int32**, **UInt32**, **Int64**, **UInt64**, **Single**, **Double**, **Decimal**, **DateTime**, and **String**. The **Convert** type also offers a static **ChangeType** method that allows an object to be converted from one type to another arbitrary type as long as the type of the source object implements the **IConvertible** interface and specifically the **ToType** method.

In addition to casting, primitive types can be written as literals, as shown here:

```
Console.WriteLine(123.ToString() + 456.ToString());  // "123456"
```

Also, if you have an expression consisting of literals, the compiler is able to evaluate the expression at compile time, improving the application's performance.

```
Boolean found = false;    // Generated code sets found to 0
```

```
Int32 x = 100 + 20 + 3;   // Generated code sets x to 123
String s = "a " + "bc";   // Generated code sets s to "a bc"
```

Finally, the compiler automatically knows how to interpret operators (such as **+**, **–**, *, /, **%**, &, **^**, |, **==**, **!=**, >, <, **>=**, **<=**, <<, >>, ~, **!**, **++**, **—**, and so on) when used in code:

```
Int32 x = 100;
Int32 y = x + 23;
Boolean lessThanFifty = (y < 50);
```

## Checked and Unchecked Primitive Type Operations

Programmers are well aware that many arithmetic operations on primitives could result in an overflow:

```
Byte b = 100;
b = (Byte) (b + 200);         // b now contains 44.
```

**Important** The CLR performs arithmetic operations on 32–bit and 64–bit values only. So, **b** and **200** are first converted to 32–bit values and then added together. The result is a 32–bit value that must be cast to a **Byte** before the result can be stored back in the variable **b**. C# doesn't perform this cast for you implicitly, which is why the **Byte** cast on the second line of the preceding code is required.

In most programming scenarios, this silent overflow is undesirable and if not detected causes the application to behave in strange and unusual ways. In some rare programming scenarios (such as calculating a hash value or a checksum), however, this overflow is not only acceptable but is also desired.

Different languages handle overflows in different ways. C and C++ don't consider overflows to be an error and allow the value to wrap; the application continues running with its fingers crossed. Visual Basic, on the other hand, always considers overflows as errors and throws an exception when it detects an overflow.

The CLR offers IL instructions that allow the compiler to choose the desired behavior. The CLR has an instruction called **add** that adds two values together. The **add** instruction performs no overflow checking. The CLR also has an instruction called **add.ovf** that also adds two values together. However, **add.ovf** throws a **System.OverflowException** exception if an overflow occurs. In addition to these two IL instructions for the add operation, the CLR also has similar IL instructions for subtraction (**sub**/**sub.ovf**), multiplication (**mul**/**mul.ovf**), and data conversions (**conv**/**conv.ovf**).

C# allows the programmer to decide how overflows should be handled. By default, overflow checking is turned off. This means that the compiler generates IL code using the versions of the add, subtract, multiply, and conversion instructions that don't include overflow checking. As a result, the code runs fast—but developers must be assured that overflows won't occur or that their code is designed to anticipate these overflows.

One way to get the C# compiler to control overflows is to use the /**checked+** command–line switch. This switch tells the compiler to generate code using the overflow–checking versions of the add, subtract, multiply, and conversion IL instructions. The code executes more slowly because the CLR is checking these operations to see whether an overflow will occur. If an overflow does occur, the CLR throws an **OverflowException** exception. You should design your application's code to handle this exception and gracefully recover.

Rather than have overflow checking turned on or off globally, programmers are much more likely to want to decide case by case whether to have overflow checking. C# allows this flexibility by offering **checked** and **unchecked** operators. Here's an example. (Assume that the compiler is building unchecked code by default.)

```
Byte b = 100;
b = checked((Byte) (b + 200));    // OverflowException is thrown
```

In this example, **b** and **200** are first converted to 32–bit values and are added together; the result is 300. Then 300 is converted to a **Byte**: this generates the **OverflowException**. If the **Byte** were cast outside the **checked** operator, the exception wouldn't occur.

```
b = (Byte) checked(b + 200);      // b contains 44; no OverflowException
```

In addition to the **checked** and **unchecked** operators, C# also offers **checked** and **unchecked** statements. The statements cause all expressions within a block to be checked or unchecked:

```
checked {                         // Start of checked block
   Byte b = 100;
   b = (Byte) (b + 200);          // This expression is checked for overflow.
}                                 // End of checked block
```

In fact, if you use a **checked** statement, you can now use the **+=** operator, which simplifies the code a bit:

```
checked {          // Start of checked block
   Byte b = 100;
   b += 200;       // This expression is checked for overflow.
}                  // End of checked block
```

**Important** Because the **checked** operator and statement affect only which versions of the add, subtract, multiply, and data conversion IL instructions are produced, calling a method within a checked operator or statement has no impact on that method. The following code demonstrates:

```
checked {
   // Assume SomeMethod tries to load 400 into a Byte.
   SomeMethod(400);
   // SomeMethod might or might not throw an OverflowException.
   // It would if SomeMethod were compiled with checked instructions.
}
```

Here's the best way to go about using **checked** and **unchecked**:

- As you write your code, explicitly use **checked** around blocks where you want an exception thrown if an overflow occurs. In Chapter 18, I'll show you how to use exception handling to gracefully recover from exceptions.
- As you write your code, explicitly use **unchecked** around blocks where you don't want an exception thrown even if an overflow occurs. This would be code where you want overflow to occur silently.
- For any code that doesn't use **checked** or **unchecked**, the assumption is that you *do* want an exception to occur on overflow when you're developing and that you *don't* want checking to occur in released code.

Now, as you develop your application, turn on the compiler's /**checked+** switch for debug builds. Your application will run more slowly because the system will be checking for overflows on any code

that you didn't explicitly mark as **checked** or **unchecked**. If an exception occurs, you'll easily detect it and be able to fix the bug in your code. For the release build of your application, use the compiler's /**checked–** switch so that the code runs fast and exceptions won't be generated.

**Important** The **System.Decimal** type is a very special type. Although many programming languages (C# and Visual Basic included) consider **Decimal** a primitive type, the CLR does not. This means that the CLR doesn't have IL instructions that know how to manipulate a **Decimal** value. If you look up the **Decimal** type in the .NET Framework documentation, you'll see that it has public, static methods called **Add**, **Subtract**, **Multiply**, **Divide**, and so on. In addition, the **Decimal** type provides operator overload methods for **+**, **–**, *****, /, and so on.

When you compile code using **Decimal** values, the compiler generates code to call **Decimal**'s members to perform the actual operation. This means that manipulating **Decimal** values is slower than manipulating CLR primitive values. Also, because there are no IL instructions for manipulating **Decimal** values, the **checked** and **unchecked** operators, statements, and compiler command–line options have no effect. Operations on **Decimal** values always throw an **OverflowException** if the operation can't be performed safely.

## Reference Types and Values Types

The CLR supports two kinds of types: *reference types* and *value types*. Of the two, you'll run into reference types much more often. Reference types are always allocated from the managed heap, and the C# **new** operator returns the memory address of the object—the memory address refers to the object's bits. You need to bear in mind some performance considerations when you're working with reference types. First consider these facts:

- The memory must be allocated from the managed heap.
- Each object allocated on the heap has some additional overhead members associated with it that must be initialized.
- Allocating an object from the managed heap could force a garbage collection to occur.

If every type were a reference type, an application's performance would suffer greatly. Imagine how poor performance would be if every time you used an **Int32** value, a memory allocation occurred! To improve performance for simple, frequently used types, the CLR offers "lightweight" types called *value types*. Value type instances are usually allocated on a thread's stack (although they can also be embedded in a reference type object). The variable representing the instance doesn't contain a pointer to an instance; the variable contains the fields of the instance itself. Because the variable contains the instance's fields, a pointer doesn't have to be dereferenced to manipulate the instance. Value type instances don't come under the control of the garbage collector, thus reducing pressure in the managed heap and reducing the number of collections an application requires over its lifetime.

The .NET Framework Reference documentation clearly indicates which types are reference types and which are value types. When looking up a type in the documentation, any type called a *class* is a reference type. For example, the **System.Object** class, the **System.Exception** class, the **System.IO.FileStream** class, and the **System.Random** class are all reference types. On the other hand, the documentation refers to each value type as a *structure* or an *enumeration*. For example, the **System.Int32** structure, the **System.Boolean** structure, the **System.Decimal** structure, the **System.TimeSpan** structure, the **System.DayOfWeek** enumeration, the **System.IO.FileAttributes**

enumeration, and the **System.–Drawing.FontStyle** enumeration are all value types.

If you look more closely at the documentation, you'll notice that all the structures are immediately derived from the **System.ValueType** type. **System.ValueType** is itself immediately derived from the **System.Object** type. By definition, all value types must be derived from **ValueType**.

> **Note** All enumerations are derived from **System.Enum**, which is itself derived from **System.ValueType**. The CLR and all programming languages give enumerations special treatment. For more information about enumerated types, refer to Chapter 13.

Even though you can't choose a base type when defining your own value type, a value type can implement one or more interfaces if you choose. In addition, the CLR doesn't allow a value type to be used as a base type for any other reference type or value type. So, for example, it's not possible to define any new types using **Boolean**, **Char**, **Int32**, **Uint64**, **Single**, **Double**, **Decimal**, and so on as base types.

> **Important** For many developers (such as unmanaged C/C++ developers), reference types and value types will seem strange at first. In unmanaged C/C++, you declare a type and then the code that uses the type gets to decide if an instance of the type should be allocated on the thread's stack or in the application's heap. In managed code, the developer defining the type indicates where instances of the type are allocated; the developer using the type has no control over this.

The following code and Figure 5–1 demonstrate how reference types and value types differ:

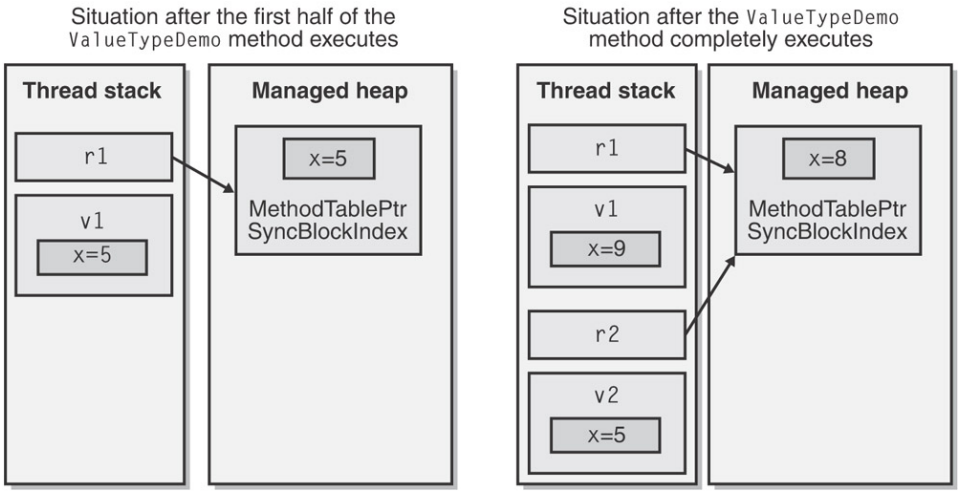



Figure 5–1 : Memory layout differences between reference and value types

```
// Reference type (because of Ôclass')
class  SomeRef { public Int32 x; }

// Value type (because of Ôstruct')
struct SomeVal { public Int32 x; }

static void ValueTypeDemo() {
   SomeRef r1 = new SomeRef();   // Allocated in heap
   SomeVal v1 = new SomeVal();   // Allocated on stack
   r1.x = 5;                     // Pointer dereference
   v1.x = 5;                     // Changed on stack
   Console.WriteLine(r1.x);      // Displays "5"
   Console.WriteLine(v1.x);      // Also displays "5"
```

```
   // The left side of Figure 5-1 reflects the situation
   // after the lines above have executed.

   SomeRef r2 = r1;              // Copies reference (pointer) only
   SomeVal v2 = v1;              // Allocate on stack & copies members
   r1.x = 8;                     // Changes r1.x and r2.x
   v1.x = 9;                     // Changes v1.x, not v2.x
   Console.WriteLine(r1.x);      // Displays "8"
   Console.WriteLine(r2.x);      // Displays "8"
   Console.WriteLine(v1.x);      // Displays "9"
   Console.WriteLine(v2.x);      // Displays "5"
   // The right side of Figure 5-1 reflects the situation
   // after ALL the lines above have executed.
}
```

In this code, the **SomeVal** type is declared using **struct** instead of the more common **class**. In C#, types declared using **struct** are value types, and types declared using **class** are reference types. As you can see, the behavior of reference types and value types differ quite a bit. As you use types in your code, you must be aware of whether the type is a reference type or a value type because it can greatly affect how you express your intentions in the code.

**Note** Other languages can have different syntax for describing value types versus reference types. For example, C++ with Managed Extensions uses the **__value** modifier.

In the preceding code, you saw this line:

```
SomeVal v1 = new SomeVal();   // Allocated on stack
```

The way this line is written makes it look like a **SomeVal** instance will be allocated on the managed heap. However, the C# compiler knows that **SomeVal** is a value type and produces code that allocates the **SomeVal** instance on the thread's stack. C# also ensures that all the fields in the value type instance are zeroed.

The preceding line could have been written like this instead:

```
SomeVal v1;   // Allocated on stack
```

This line also produces IL that allocates the instance on the thread's stack and zeroes the fields. The only difference is that C# "thinks" the instance initialized if you use the **new** operator. The following code will make this point clear:

```
// These two lines compile because C# thinks that
// v1's fields have been initialized to 0.
SomeVal v1 = new SomeVal();
Int32 a = v1.x;

// These two lines don't compile because C# doesn't think that
// v1's fields have been initialized to 0.
SomeVal v1;
Int32 a = v1.x;  // error CS0170: Use of possibly unassigned field Ôx'
```

When designing your own types, consider carefully whether to define your types as value types instead of reference types. In some situations, value types can give better performance. In particular, you should declare a type as a value type if *all* the following statements are true:

- The type acts like a primitive type.
- The type doesn't need to inherit from any other type.

- The type won't have any other types derived from it.
- Instances of the type aren't frequently passed as method parameters. By default, parameters are passed by value, which causes the fields in value type instances to be copied, frequently hurting performance.
- Instances of the type aren't frequently returned from methods. Again, a method that returns a value type causes the fields in the instance to be copied into memory allocated by the caller when the method returns, hurting performance.
- Instances of the type aren't frequently used in collections such as **ArrayList**, **Hashtable**, and so on. Classes that manage a set of generic objects require that value type instances be boxed. Boxing causes additional memory to be allocated, and additional memory copy operations hurt performance. (I'll explain boxing and unboxing in more detail in the next section.)

The main advantage of value types is that they're not allocated in the managed heap. Of course, value types have several limitations of their own when compared to reference types. Here are some of the ways in which value types and reference types differ:

- Value type objects have two representations: an *unboxed* form and a *boxed* form (discussed in the next section). Reference types are always in a boxed form.
- Value types are derived from **System.ValueType**. This type offers the same methods as defined by **System.Object**. However, **System.ValueType** overrides the **Equals** method so that it returns **true** if the values of the two object's fields match. In addition, **System.ValueType** overrides the **GetHashCode** method so that it produces a hash code value using an algorithm that takes into account the values in the object's instance fields. When defining your own value types, you should override and provide explicit implementations for the **Equals** and **GetHashCode** methods. I'll cover the **Equals** and **GetHashCode** methods in Chapter 6.
- Because you can't declare a new value type or a new reference type using a value type as a base class, you shouldn't introduce any new virtual methods into a value type. No methods can be abstract, and all methods are implicitly sealed (can't be overridden).
- Reference type variables contain the memory address of objects in the heap. By default, when a reference type variable is created, it is initialized to **null**, indicating that the reference type variable doesn't currently point to a valid object. Attempting to use a **null** reference type variable causes a **NullReferenceException** exception to be thrown. By contrast, value type variables always contain a value of the underlying type, and all members of the value type are initialized to 0. It's not possible to generate a **NullReferenceException** exception when accessing a value type.
- When you assign a value type variable to another value type variable, a field–by–field copy is made. When you assign a reference type variable to another reference type variable, only the memory address is copied.
- Because of the previous point, two or more reference type variables can refer to a single object in the heap, allowing operations on one variable to affect the object referenced by the other variable. On the other hand, value type variables each have their own copy of the "object's" data, and it's not possible for operations on one value type variable to affect another.
- Because unboxed value types aren't allocated on the heap, the storage allocated for them is freed as soon as the method that defines an instance of the type is no longer active. This means that a value type instance doesn't receive a notification (via a **Finalize** method) when its memory is reclaimed.

**Note** In fact, it would be quite odd to define a value type with a **Final–ize** method since the method would be called only on boxed instances. For this reason, many compilers

(including C# and Visual Basic) don't allow you to define **Finalize** methods on value types. Although the CLR allows a value type to define a **Finalize** method, the CLR won't call this method when a boxed instance of the value type is garbage collected.

How the CLR Controls the Layout of a Type's Fields

To improve performance, the CLR is capable of arranging the fields of a type any way it chooses. For example, the CLR might reorder fields in memory so that object references are grouped together and data fields are properly aligned and packed. However, when you define a type, you can tell the CLR whether it must keep the type's fields in the same order the developer specified them or whether it can reorder as it sees fit.

You tell the CLR what to do by applying the **System.Runtime.–InteropServices.StructLayoutAttribute** attribute on the class or structure you're defining. To this attribute's constructor, you can pass **LayoutKind.Auto** to have the CLR arrange the fields or **LayoutKind.Sequential** to have the CLR preserve your field layout. If you don't explicitly specify the **StructLayoutAttribute** on a type that you're defining, your compiler selects whatever layout it thinks best.

You should be aware that Microsoft's C# compiler selects **LayoutKind.Auto** for reference types (classes) and **LayoutKind.Sequential** for value types (structures). It is obvious that the C# compiler team feels that structures are commonly used when interoperating with unmanaged code, and for this to work, the fields must stay in the order defined by the programmer. However, if you're creating a value type that has nothing to do with interoperability with unmanaged code, you probably want to override the C# compiler's default. Here's an example:

```
using System;
using System.Runtime.InteropService;

// Let the CLR arrange the fields to improve performance for this
// value type.
[StructLayout(LayoutKind.Auto)]
struct Point {
   Int32 x, y;
}
```

## Boxing and Unboxing Value Types

Value types are lighter weight than reference types because they are not allocated in the managed heap, not garbage collected, and not referred to by pointers. However, in many cases, you must get a reference to an instance of a value type. For example, let's say that you wanted to create an **ArrayList** object (a type defined in the **System.Collections** namespace) to hold a set of **Point** structures. The code might look like this:

```
// Declare a value type.
struct Point {
   public Int32 x, y;
}

class App {
   static void Main() {
      ArrayList a = new ArrayList();
      Point p;            // Allocate a Point (not in the heap).
      for (Int32 i = 0; i < 10; i++) {
         p.x = p.y = i;   // Initialize the members in the value type.
```

```
            a.Add(p);         // Box the value type and add the
                              // reference to the array.
        }
        Ã
    }
}
```

With each iteration of the loop, a **Point** value type's fields are initialized. Then the **Point** is stored in the **ArrayList**. But let's think about this for a moment. What is actually being stored in the **ArrayList**? Is it the **Point** structure, the address of the **Point** structure, or something else entirely? To get the answer, you must look up **ArrayList**'s **Add** method and see what type its parameter is defined as. In this case, the **Add** method is prototyped as follows:

```
public virtual void Add(Object value);
```

From this, you can plainly see that **Add** takes an **Object** as a parameter, indicating that **Add** requires a reference (or pointer) to an object on the managed heap as a parameter. But in the preceding code, I'm passing **p** a **Point** value type. For this code to work, the **Point** value type must be converted into a true heap–managed object and a reference to this object must be obtained.

It's possible to convert a value type to a reference type using a mechanism called *boxing*. Internally, here's what happens when an instance of a value type is boxed:

1. Memory is allocated from the managed heap. The amount of memory allocated is the size the value type requires plus any additional overhead to consider this value type to be a true object. The additional overhead includes a method table pointer and a SyncBlockIndex.
2. The value type's fields are copied to the newly allocated heap memory.
3. The address of the object is returned. This address is now a reference to an object; the value type is now a reference type.

Some language compilers, like C#, automatically produce the IL code necessary to box a value type instance, but you still need to understand what's going on under the covers so that you're aware of code size and performance issues.

In the preceding code, the C# compiler detected that I was passing a value type to a method that requires a reference type, and it automatically emitted code to box the object. So at run time, the fields currently residing in the **Point** value type (**p**) are copied into the newly allocated **Point** object. The address of the boxed **Point** object (now a reference type) is returned and is then passed to the **Add** method. The **Point** object will remain in the heap until it is garbage collected. The **Point** value type variable (**p**) can be reused or freed since the **ArrayList** never knows anything about it. Note that the lifetime of the boxed value type extends beyond the lifetime of the unboxed value type.

Many languages designed for the CLR (for example, C# and Visual Basic) automatically emit the code necessary to box value types into reference types when necessary. However, some languages (such as C++ with Managed Extensions) require that the programmer write code to explicitly box value types when necessary.

Now that you know how boxing works, let's talk about unboxing. Let's say that in another piece of code you want to grab the first element out of the **ArrayList**:

```
Point p = (Point) a[0];
```

Here you're taking the reference (or pointer) contained in element 0 of the **ArrayList** and trying to put it into a **Point** value type, **p**. For this to work, all the fields contained in the boxed **Point** object

must be copied into the value type variable, **p**, which is on the thread's stack. The CLR accomplishes this copying in two steps. First the address of the **Point** fields in the boxed **Point** object is obtained. This process is called *unboxing*. Then the values of these fields are copied from the heap to the stack–base value type instance.

Unboxing is *not* the exact opposite of boxing. The unboxing operation is much less costly than boxing. Unboxing is really just the operation of obtaining a pointer to the raw value type (data fields) contained within an object. So, unlike boxing, unboxing doesn't involve the copying of any bytes in memory. However, an unboxing operation is typically followed by copying the fields, making these two operations the exact opposite of a boxing operation.

Obviously, boxing and unbox/copy operations hurt your application's performance in terms of both speed and memory, so you should be aware of when the compiler generates code to perform these operations automatically and try to write code that minimizes this code generation.

Internally, here's exactly what happens when a reference type is unboxed:

1. If the reference is **null**, a **NullReferenceException** is thrown.
2. If the reference doesn't refer to an object that is a boxed value of the desired value type, an **InvalidCastException** exception is thrown.
3. A pointer to the value type contained inside the object is returned. The value type that this pointer refers to doesn't know anything about the usual overhead associated with a true object: a method table pointer and a SyncBlockIndex. In effect, the pointer refers to the unboxed portion in the boxed object.

The second item means that the following code will *not* work as you might expect:

```
static void Main() {
   Int32  x = 5;
   Object o = x;         // Box x; o refers to the boxed object
   Int16  y = (Int16) o; // Throws an InvalidCastException
}
```

Logically, it makes sense to take the boxed **Int32** that **o** refers to and cast it to an **Int16**. However, when unboxing an object, the cast must be to the unboxed type—**Int32** in this case. Here's the correct way to write this code:

```
static void Main() {
   Int32  x = 5;
   Object o = x;                // Box x; o refers to the boxed object
   Int16  y = (Int16)(Int32) o; // Unbox to the correct type and cast
}
```

Even though, strictly speaking, an unboxing operation doesn't copy any fields, it is frequently followed immediately by a field copy, which does copy the fields from the heap to the stack. In fact, in C#, an unbox operation is always followed by a field copy. Let's take a look at some C# code demonstrating that unbox and copy operations always work together:

```
static void Main() {
   Point p;
   p.x = p.y = 1;
   Object o = p;   // Boxes p; o refers to the boxed object

   p = (Point) o;  // Unboxes o AND copies fields from object to stack
}
```

On the last line, the C# compiler emits an IL instruction to unbox **o** (get the address of the fields in the object) and another IL instruction to copy the fields from the heap to the stack–based variable **p**.

Now look at this code:

```
static void Main() {
   Point p;
   p.x = p.y = 1;
   Object o = p;   // Boxes p; o refers to the boxed object

   // Change Point's x field to 2
   p = (Point) o;  // Unboxes o AND copies fields from object to stack
   p.x = 2;
   o = p
}
```

The code at the bottom of this fragment just wants to change **Point**'s **x** field from 1 to 2. To do this, an unbox operation must be performed, followed by a field copy, followed by changing the field (on the stack), followed by a boxing operation (which creates a whole new object in the managed heap). Hopefully, you see the impact that boxing and unboxing/copying operations have on your application's performance.

Some languages, like C++ with Managed Extensions, allow you to unbox a boxed value type without copying the fields. Unboxing returns the address of the unboxed portion of a boxed object (ignoring the object's method table pointer and SyncBlockIndex overhead). You can now use this pointer to manipulate the unboxed instance's fields (which happen to be in a boxed object on the heap). For example, the previous code would be much more efficient if written in C++ with Managed Extensions because you could change the value of **Point**'s **x** field within the already boxed **Point** object. This would avoid both allocating a new object on the heap and copying all the fields twice!

**Important** If you're the least bit concerned about your application's performance, you must be aware of when the compiler produces the code that performs these operations. Some languages, such as C++ with Managed Extensions, require the programmer to explicitly write code to box and unbox value types. C++ with Managed Extensions offers the **__box** operator and **dynamic_cast** operator to do boxing and unboxing operations, respectively. This requirement makes the developer's job harder because he must write the necessary code; on the plus side, though, the developer will know exactly when boxing and unboxing operations are occurring. However, many languages (such as C# and Visual Basic) will automatically emit the IL code necessary to box and unbox value types. Although this built–in functionality makes the code look nicer and eases the burden on the developer, it also means that it's not obvious to the developer when boxing and unboxing operations are occurring.

Let's look at a few more examples that demonstrate boxing and unboxing:

```
public static void Main() {
   Int32  v = 5;            // Create an unboxed value type variable.
   Object o = v;            // o refers to a boxed Int32 containing 5.
   v = 123;                 // Changes the unboxed value to 123

   Console.WriteLine(v + ", " + (Int32) o);    // Displays "123, 5"
}
```

In this code, can you guess how many boxing operations occur? You might be surprised to discover that the answer is three! Let's analyze the code carefully to really understand what's going on. To

help you understand, I've included the IL code generated for the **Main** method shown in the preceding code. I've commented the code so that you can easily see the individual operations.

```
.method private hidebysig static void Main() cil managed
{
  .entrypoint
  // Code size       46 (0x2e)
  .maxstack  3
  .locals ([0] int32 v,
           [1] object o)

  // Load 5 into v.
  IL_0000:  ldc.i4.5
  IL_0001:  stloc.0

  // Box v, and store the reference pointer in o.
  IL_0002:  ldloc.0
  IL_0003:  box        [mscorlib]System.Int32
  IL_0008:  stloc.1

  // Load 123 into v.
  IL_0009:  ldc.i4.s   123
  IL_000b:  stloc.0

  // Box v, and leave the pointer on the stack for Concat.
  IL_000c:  ldloc.0
  IL_000d:  box        [mscorlib]System.Int32

  // Load the string on the stack for Concat.
  IL_0012:  ldstr      ", "

  // Unbox o: Get the pointer to the Int32's field on the stack.
  IL_0017:  ldloc.1
  IL_0018:  unbox      [mscorlib]System.Int32

  // Copy the bytes from the boxed Int32 to the stack.
  IL_001d:  ldind.i4

  // Box the Int32, and leave the pointer on the stack for Concat.
  IL_001e:  box        [mscorlib]System.Int32

  // Call Concat
  IL_0023:  call       string [mscorlib]System.String::Concat(object,
                                                              object,
                                                              object)

  // The string returned from Concat is passed to WriteLine.
  IL_0028:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_002d:  ret
} // End of method App::Main
```

First an **Int32** unboxed value type (**v**) is created and initialized to 5. Then an **Object** reference type (**o**) is created, and it wants to point to **v**. But because reference types must always point to objects in the heap, C# generated the proper IL code to box **v** and store the address of the boxed "copy" of **v** in **o**. Now the value 123 is placed into the unboxed value type **v;** this has no effect on the boxed **Int32** value, which keeps its value of 5.

Next is the call to the **WriteLine** method. **WriteLine** wants a **String** object passed to it, but there is no string object. Instead, these three items are available: an unboxed **Int32** value type (**v**), a **String** (which is a reference type), and a reference to a boxed **Int32** value type (**o**) that is being cast to an

unboxed **Int32**. These must somehow be combined to create a **String**.

To create a **String**, the C# compiler generates code that calls the **String** object's static **Concat** method. There are several overloaded versions of the **Concat** method; all of them perform identically—the only difference is in the number of parameters. Because a string is being created from the concatenation of three items, the compiler chooses the following version of the **Concat** method:

```
public static String Concat(Object arg0, Object arg1, Object arg2);
```

For the first parameter, **arg0**, **v** is passed. But **v** is an unboxed value parameter and **arg0** is an **Object**, so **v** must be boxed and the address to the boxed **v** is passed for **arg0**. For the **arg1** parameter, the "**,**" string is passed as a reference to a **String** object. Finally, for the **arg2** parameter, **o** (a reference to an **Object**) is cast to an **Int32**, creating a temporary instance of an **Int32** value type on the IL evaluation stack that receives a copy of the unboxed version of the value currently referred to by **o**. This temporary **Int32** value type must be boxed again with the memory address being passed for **Concat**'s **arg2** parameter.

The **Concat** method calls each of the specified object's **ToString** methods and concatenates each object's string representation. The **String** object returned from **Concat** is then passed to **WriteLine** to show the final result.

I should point out that the generated IL code is more efficient if the call to **WriteLine** is written as follows:

```
Console.WriteLine(v + ", " + o);    // Displays "123, 5"
```

This line is identical to the earlier version except that I've removed the **(Int32)** cast that preceded the variable **o**. This code is more efficient because **o** is already a reference type to an **Object** and its address can simply be passed to the **Concat** method. So, removing the cast saved three operations: an unbox, a field-by-field copy, and a box. You can easily see this savings by rebuilding the application and examining the generated IL code:

```
.method private hidebysig static void Main() cil managed
{
  .entrypoint
  // Code size       35 (0x23)
  .maxstack  3
  .locals ([0] int32 v,
           [1] object o)

  // Load 5 into v.
  IL_0000:  ldc.i4.5
  IL_0001:  stloc.0

  // Box v, and store the reference pointer in o.
  IL_0002:  ldloc.0
  IL_0003:  box        [mscorlib]System.Int32
  IL_0008:  stloc.1

  // Load 123 into v.
  IL_0009:  ldc.i4.s   123
  IL_000b:  stloc.0

  // Box v, and leave the pointer on the stack for Concat.
  IL_000c:  ldloc.0
  IL_000d:  box        [mscorlib]System.Int32
```

```
  // Load the string on the stack for Concat.
  IL_0012:  ldstr      ", "

  // Load the address of the boxed Int32 on the stack for Concat.
  IL_0017:  ldloc.1

  // Call Concat.
  IL_0018:  call       string [mscorlib]System.String::Concat(object,
                                                              object,
                                                              object)

  // The string returned from Concat is passed to WriteLine.
  IL_001d:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0022:  ret
} // end of method App::Main
```

A quick comparison of the IL for these two versions of the **Main** method shows that the version without the **(Int32)** cast is 11 bytes smaller than the version with the cast. The extra unbox/box steps in the first version are obviously generating more code. An even bigger concern, however, is that the extra boxing step allocates an additional object from the managed heap that must be garbage collected in the future. Certainly, both versions give identical results and the difference in speed isn't noticeable, but if you have extra, unnecessary boxing operations occurring in a loop, the performance and memory usage of your application would be seriously affected.

You can improve the previous code even more by calling **WriteLine** like this:

```
Console.WriteLine(v.ToString() + ", " + o);    // Displays "123, 5"
```

Now **ToString** is called on the unboxed value type **v** and a **String** is returned. String objects are already reference types and can simply be passed to the **Concat** method without requiring any boxing.

Let's look at yet another example that demonstrates boxing and unboxing:

```
public static void Main() {
   Int32 v = 5;               // Create an unboxed value type variable.
   Object o = v;              // o refers to the boxed version of v.

   v = 123;                   // Changes the unboxed value type to 123
   Console.WriteLine(v);      // Displays "123"

   v = (Int32) o;             // Unboxes o into v
   Console.WriteLine(v);      // Displays "5"
}
```

How many boxing operations do you count in this code? The answer is one. The reason that there is only one boxing operation is that the **System.Console** class defines a **WriteLine** method that accepts an **Int32** as a parameter:

```
public static void WriteLine(Int32 value);
```

In the two calls to **WriteLine** above, the variable **v**, an **Int32** unboxed value type, is passed by value. Now, it may be that **WriteLine** will box this **Int32** internally, but you have no control over that. The important thing is that you've done the best you could and have eliminated the boxing from your own code.

If you take a close look at the FCL, you'll notice many overloaded methods that differ based on their value type parameters. For example, the **System.Console** type offers several overloaded versions of the **WriteLine** method:

```
public static void WriteLine(Boolean);
public static void WriteLine(Char);
public static void WriteLine(Int32);
public static void WriteLine(UInt32);
public static void WriteLine(Int64);
public static void WriteLine(UInt64);
public static void WriteLine(Single);
public static void WriteLine(Double);
public static void WriteLine(Decimal);
```

You'll also find a similar set of overloaded methods for **System.Console**'s **Write** method, **System.IO.BinaryWriter**'s **Write** method, **System.IO.TextWriter**'s **Write** and **WriteLine** methods, **System.Runtime.Serialization.SerializationInfo**'s **AddValue** method, **System.Text.StringBuilder**'s **Append** and **Insert** methods, and so on. All these methods offer overloaded versions for the sole purpose of reducing the number of boxing operations for the common value types.

One last point about boxing: if you know that the code you're writing is going to cause the compiler to box a single value type repeatedly, your code will be smaller and faster if you manually box the value type. Here's an example:

```
using System;

class App {
   static void Main() {
      Int32 v = 5;   // Create an unboxed value type variable.

#if INEFFICIENT
      // When compiling the following line, v is boxed
      // three times, wasting time and memory.
      Console.WriteLine("{0}, {1}, {2}", v, v, v);
#else
      // The lines below have the same result, execute
      // much faster, and use less memory.
      Object o = v;  // Manually box v (just once).

      // No boxing occurs to compile the following line.
      Console.WriteLine("{0}, {1}, {2}", o, o, o);
#endif
   }
}
```

If this code is compiled with the **INEFFICIENT** symbol defined, the compiler will generate code to box **v** three times, causing three objects to be allocated from the heap! This is extremely wasteful since each object will have exactly the same contents: 5. If the code is compiled without the **INEFFICIENT** symbol defined, **v** is boxed just once, so only one object is allocated from the heap. Then, in the call to **Console.WriteLine**, the reference to the single boxed object is passed three times. This second version executes *much* faster and allocates less memory from the heap.

In these examples, it's fairly easy to recognize when an instance of a value type requires boxing. Basically, if you want a reference to an instance of a value type, the instance must be boxed. Usually this happens because you have a value type and you want to pass it to a method that requires a reference type. However, this situation isn't the only one in which you'll need to box an

instance of a value type.

Recall that unboxed value types are lighter–weight types than reference types for two reasons:

- They are not allocated on the managed heap.
- They don't have the additional overhead members that every object on the heap has: a method table pointer and a SyncBlockIndex.

Because unboxed value types don't have a SyncBlockIndex, you can't have multiple threads synchronize their access to the instance using the methods of the **System.Threading.Monitor** type. Because unboxed value types don't have a method table pointer, you can't call inherited implementations of virtual methods using an unboxed instance of the value type. In addition, casting an unboxed instance of a value type to one of the type's interfaces requires that the instance be boxed because interfaces are always reference types. (I'll talk about interfaces in Chapter 15.) The following code demonstrates:

```
using System;

struct Point : ICloneable {
   public Int32 x, y;

   // Override ToString method inherited from System.ValueType
   public override String ToString() {
      return String.Format("({0}, {1})", x, y);
   }

   // Implementation of ICloneable's Clone method
   public Object Clone() {
      return MemberwiseClone();
   }
}

class App {
   static void Main() {
      // Create an instance of the Point value type on the stack.
      Point p;

      // Initialize the instance's fields.
      p.x = 10;
      p.y = 20;

      // p does NOT get boxed to call ToString.
      Console.WriteLine(p.ToString());

      // p DOES get boxed to call GetType.
      Console.WriteLine(p.GetType());

      // p does NOT get boxed to call Clone.
      // Clone returns an object that is unboxed,
      // and its fields are copied into p2.
      Point p2 = (Point) p.Clone();

      // p2 DOES get boxed, and the reference is placed in c.
      ICloneable c = p2;

      // c does NOT get boxed because it is already boxed.
      // Clone returns a reference to an object that is saved in o.
      Object o = c.Clone();

      // o is unboxed, and fields are copied into p.
```

```
      p = (Point) o;
   }
}
```

This code demonstrates several scenarios related to boxing and unboxing:

- **Calling ToString** In the call to **ToString**, **p** doesn't have to be boxed. At first, you'd think that **p** would have to be boxed because **ToString** is a method that is inherited from the base type, **System.ValueType**. Normally, to call an inherited method, you'd need to have a pointer to the type's method table—and because **p** is an unboxed value type, there's no reference to **Point**'s method table. However, the C# compiler sees that **Point** overrides the **ToString** method, and it emits code that calls **ToString** directly. The compiler knows that polymorphism can't come into play here since **Point** is a value type and value types can't be used as the base type for any other type.
- **Calling GetType** In the call to **GetType**, **p** does have to be boxed. The reason is that the **Point** type doesn't implement **GetType;** it is inherited from **System.ValueType**. So to call **GetType**, you must have a pointer to **Point**'s method table, which can be obtained only by boxing **p**.
- **Calling Clone (first time)** In the first call to **Clone**, **p** doesn't have to be boxed because **Point** implements the **Clone** method and the compiler can just call it directly. Note that **Clone** returns an **Object**, which is a reference to a boxed **Point** object on the heap. This object must be unboxed and its fields copied to the unboxed value type **p2**.
- **Casting to ICloneable** When casting **p** to a variable that is of an interface type, **p** must be boxed because interfaces are reference types by definition. So, **p2** is boxed and the pointer to this boxed object is stored in the variable **c**.
- **Calling Clone (second time)** In the second call to **Clone**, no boxing occurs and the **Clone** method is called on the already boxed object residing in the heap. **Clone** creates a new object on the heap and returns a reference to this new object. This reference is saved in **o** (a reference type).
- **Casting to Point** When casting **o** to a **Point**, the object on the heap referred to by **o** is unboxed and its fields are copied from the heap to **p**, an instance of the **Point** type residing on the stack.

I realize that all this information about reference types, values types, and boxing might be overwhelming at first. However, a solid understanding of these concepts is critical to any .NET Framework developer's long–term success. Trust me: having a solid grasp of these concepts will allow you to build efficient applications faster and easier.

# Chapter 6: Common Object Operations

In this chapter, I'll describe how to properly implement the operations that all objects must exhibit. Specifically, I'll talk about object equality, identity, hash codes, and cloning.

## Object Equality and Identity

The **System.Object** type offers a virtual method, named **Equals**, whose purpose is to return **true** if two objects have the same "value". The .NET Framework Class Library (FCL) includes many methods, such as **System.Array**'s **IndexOf** method and **System.Collections.ArrayList**'s **Contains** method, that internally call **Equals**. Because **Equals** is defined by **Object** and because every type is ultimately derived from **Object**, every instance of every type offers the **Equals** method. For types that don't explicitly override **Equals**, the implementation provided by **Object** (or the nearest base class that overrides **Equals**) is inherited. The following code shows how **System.Object**'s **Equals** method is essentially implemented:

```
class Object {
   public virtual Boolean Equals(Object obj) {

      // If both references point to the same
      // object, they must be equal.
      if (this == obj) return(true);

      // Assume that the objects are not equal.
      return(false);
   }
   Ã
}
```

As you can see, this method takes the simplest approach possible: if the two references being compared point to the same object, **true** is returned; in any other case, **false** is returned. If you define your own types and you want to compare their fields for equality, **Object**'s default implementation won't be sufficient for you; you must override **Equals** and provide your own implementation.

When you implement your own **Equals** method, you must ensure that it adheres to the four properties of equality:

- **Equals** must be reflexive; that is, **x.Equals(x)** must return **true**.
- **Equals** must be symmetric; that is, **x.Equals(y)** must return the same value as **y.Equals(x)**.
- **Equals** must be transitive; that is, if **x.Equals(y)** returns **true** and **y.Equals(z)** returns **true**, then **x.Equals(z)** must also return **true**.
- **Equals** must be consistent. Provided that there are no changes in the two values being compared, **Equals** should consistently return **true** or **false**.

If your implementation of **Equals** fails to adhere to all these rules, your application will behave in strange and unpredictable ways.

Unfortunately, implementing your own version of **Equals** isn't as easy and straightforward as you might expect. You must do a number of operations correctly, and, depending on the type you're defining, the operations are slightly different. Fortunately, there are only three different ways to implement **Equals**. Let's look at each pattern individually.

## Implementing Equals for a Reference Type Whose Base Classes Don't Override Object's Equals

The following code shows how to implement **Equals** for a type that directly inherits **Object**'s **Equals** implementation:

```
// This is a reference type (because of 'class').
class MyRefType : BaseType {
   RefType refobj;   // This field is a reference type.
   ValType valobj;   // This field is a value type.

   public override Boolean Equals(Object obj) {
      // Because 'this' isn't null, if obj is null,
      // then the objects can't be equal.
      if (obj == null) return false;

      // If the objects are of different types, they can't be equal.
      if (this.GetType() != obj.GetType()) return false;

      // Cast obj to this type to access fields. NOTE: This cast can't
      // fail because you know that objects are of the same type.
      MyRefType other = (MyRefType) obj;

      // To compare reference fields, do this:
      if (!Object.Equals(refobj, other.refobj)) return false;

      // To compare value fields, do this:
      if (!valobj.Equals(other.valobj)) return false;

      return true;   // Objects are equal.
   }

   // Optional overloads of the == and != operators
   public static Boolean operator==(MyRefType o1, MyRefType o2) {
      if (o1 == null) return false;
      return o1.Equals(o2);
   }

   public static Boolean operator!=(MyRefType o1, MyRefType o2) {
      return !(o1 == o2);
   }
}
```

This version of **Equals** starts out by comparing **obj** against **null**. If the object being compared is not **null**, then the types of the two objects are compared. If the objects are of different types, then they can't be equal. If both objects are the same type, then you cast **obj** to **MyRefType**, which can't possibly throw an exception because you know that both objects are of the same type. Finally, the fields in both objects are compared, and **true** is returned if all fields are equal.

You must be very careful when comparing the individual fields. The preceding code shows two different ways to compare the fields based on what types of fields you're using.

- **Comparing reference type fields** To compare reference type fields, you should call **Object**'s static **Equals** method. **Object**'s static **Equals** method is just a little helper method that returns **true** if two reference objects are equal. Here's how **Object**'s static **Equals** method is implemented internally:

```
public static Boolean Equals(Object objA, Object objB) {
```

```
   // If objA and objB refer to the same object, return true.
   if (objA == objB) return true;

   // If objA or objB is null, they can't be equal, so return false.
   if ((objA == null) || (objB == null)) return false;

   // Ask objA if objB is equal to it, and return the result.
   return objA.Equals(objB);
}
```

- You use this method to compare reference type fields because it's legal for them to have a value of **null**. Certainly, calling **refobj.Equals(other.refobj)** will throw a **NullReferenceException** if **refobj** is **null**. **Object**'s static **Equals** helper method performs the proper checks against **null** for you.
- **Comparing value type fields** To compare value type fields, you should call the field type's **Equals** method to have it compare the two fields. You shouldn't call **Object**'s static **Equals** method because value types can never be **null** and calling the static **Equals** method would box both value type objects.

## Implementing Equals for a Reference Type When One or More of Its Base Classes Overrides Object's Equals

The following code shows how to implement **Equals** for a type that inherits an implementation of **Equals** other than the one **Object** provides:

```
// This is a reference type (because of 'class').
class MyRefType : BaseType {
   RefType refobj;   // This field is a reference type.
   ValType valobj;   // This field is a value type.

   public override Boolean Equals(Object obj) {
      // Let the base type compare its fields.
      if (!base.Equals(obj)) return false;

      // All the code from here down is identical to
      // that shown in the previous version.

      // Because 'this' isn't null, if obj is null,
      // then the objects can't be equal.
      // NOTE: This line can be deleted if you trust that
      // the base type implemented Equals correctly.
      if (obj == null) return false;

      // If the objects are of different types, they can't be equal.
      // NOTE: This line can be deleted if you trust that
      // the base type implemented Equals correctly.
      if (this.GetType() != obj.GetType()) return false;

      // Cast obj to this type to access fields. NOTE: This cast
      // can't fail because you know that objects are of the same type.
      MyRefType other = (MyRefType) obj;

      // To compare reference fields, do this:
      if (!Object.Equals(refobj, other.refobj)) return false;

      // To compare value fields, do this:
      if (!valobj.Equals(other.valobj)) return false;

      return true;   // Objects are equal.
```

```
   }

   // Optional overloads of the == and != operators
   public static Boolean operator==(MyRefType o1, MyRefType o2) {
      if (o1 == null) return false;
      return o1.Equals(o2);
   }

   public static Boolean operator!=(MyRefType o1, MyRefType o2) {
      return !(o1 == o2);
   }
}
```

This code is practically identical to the code shown in the previous section. The only difference is that this version allows its base type to compare its fields too. If the base type doesn't think the objects are equal, then they can't be equal.

It is very important that you do *not* call **base.Equals** if this would result in calling the **Equals** method provided by **System.Object**. The reason is that **Object**'s **Equals** method returns **true** only if the references point to the same object. If the references don't point to the same object, then **false** will be returned and your **Equals** method will always return **false**!

Certainly, if you're defining a type that is directly derived from **Object**, you should implement **Equals** as shown in the previous section. If you're defining a type that isn't directly derived from **Object**, you must first determine if that type (or any of its base types, except **Object**) provides an implementation of **Equals**. If any of the base types provide an implementation of **Equals**, then call **base.Equals** as shown in this section.

## Implementing Equals for a Value Type

As I mentioned in Chapter 5, all value types are derived from **System.ValueType**. **ValueType** overrides the implementation of **Equals** offered by **System.Object**. Internally, **System.ValueType**'s **Equals** method uses reflection (covered in Chapter 20) to get the type's instance fields and compares the fields of both objects to see if they have equal values. This process is very slow, but it's a reasonably good default implementation that all value types will inherit. However, it does mean that reference types inherit an implementation of **Equals** that is really identity and that value types inherit an implementation of **Equals** that is value equality.

For value types that don't explicitly override **Equals**, the implementation provided by **ValueType** is inherited. The following code shows how **System.–ValueType**'s **Equals** method is essentially implemented:

```
class ValueType {
   public override Boolean Equals(Object obj) {

      // Because 'this' isn't null, if obj is null,
      // then the objects can't be equal.
      if (obj == null) return false;

      // Get the type of 'this' object.
      Type thisType = this.GetType();

      // If 'this' and 'obj' are different types, they can't be equal.
      if (thisType != obj.GetType()) return false;

      // Get the set of public and private instance
      // fields associated with this type.
```

```
      FieldInfo[] fields = thisType.GetFields(BindingFlags.Public |
         BindingFlags.NonPublic | BindingFlags.Instance);

      // Compare each instance field for equality.
      for (Int32 i = 0; i < fields.Length; i++) {

         // Get the value of the field from both objects.
         Object thisValue = fields[i].GetValue(this);
         Object thatValue = fields[i].GetValue(obj);

         // If the values aren't equal, the objects aren't equal.
         if (!Object.Equals(thisValue, thatValue)) return false;
      }

      // All the field values are equal, and the objects are equal.
      return true;
   }
   Ã
}
```

Even though **ValueType** offers a pretty good implementation for **Equals** that would work for most value types that you define, you should still provide your own implementation of **Equals**. The reason is that your implementation will perform significantly faster and will be able to avoid extra boxing operations.

The following code shows how to implement **Equals** for a value type:

```
// This is a value type (because of 'struct').
struct MyValType {
   RefType refobj;   // This field is a reference type.
   ValType valobj;   // This field is a value type.

   public override Boolean Equals(Object obj) {
      // If obj is not your type, then the objects can't be equal.
      if (!(obj is MyValType)) return false;

      // Call the type-safe overload of Equals to do the work.
      return this.Equals((MyValType) obj);
   }

   // Implement a strongly typed version of Equals.
   public Boolean Equals(MyValType obj) {
      // To compare reference fields, do this:
      if (!Object.Equals(this.refobj, obj.refobj)) return false;

      // To compare value fields, do this:
      if (!this.valobj.Equals(obj.valobj)) return false;

      return true;   // Objects are equal.
   }

   // Optionally overload operator==
   public static Boolean operator==(MyValType v1, MyValType v2) {
      return (v1.Equals(v2));
   }

   // Optionally overload operator!=
   public static Boolean operator!=(MyValType v1, MyValType v2) {
      return !(v1 == v2);
   }
}
```

For value types, the type should define a strongly typed version of **Equals**. This version takes the defining type as a parameter, giving you type safety and avoiding extra boxing operations. You should also provide strongly typed operator overloads for the **==** and **!=** operators. The following code demonstrates how to test two value types for equality:

```
MyValType v1, v2;

// The following line calls the strongly typed version of
// Equals (no boxing occurs).
if (v1.Equals(v2)) { ... }

// The following line calls the version of
// Equals that takes an object (4 is boxed).
if (v1.Equals(4)) { ... }

// The following doesn't compile because operator==
// doesn't take a MyValType and an Int32.
if (v1 == 4) { ... }

// The following compiles, and no boxing occurs.
if (v1 == v2) { ... }
```

Inside the strongly typed **Equals** method, the code compares the fields in exactly the same way that you'd compare them for reference types. Keep in mind that the code doesn't do any casting, doesn't compare the two instances to see if they're the same type, and doesn't call the base type's **Equals** method. These operations aren't necessary because the method's parameter already ensures that the instances are of the same type. Also, because all value types are immediately derived from **System.ValueType**, you know that your base type has no fields of its own that need to be compared.

You'll notice in the **Equals** method that takes an **Object** that I used the **is** operator to check the type of **obj**. I used **is** instead of **GetType** because calling **GetType** on an instance of a value type requires that the instance be boxed. I demonstrated this in the "Boxing and Unboxing Value Types" section in Chapter 5.

## Summary of Implementing Equals and the ==/!= Operators

In this section, I summarize how to implement equality for your own types:

- **Compiler primitive types** Your compiler will provide implementations of the **==** and **!=** operators for types that it considers primitives. For example, the C# compiler knows how to compare **Object**, **Boolean**, **Char**, **Int16**, **Uint16**, **Int32**, **Uint32**, **Int64**, **Uint64**, **Single**, **Double**, **Decimal**, and so on for equality. In addition, these types provide implementations of **Equals**, so you can call this method as well as use operators.
- **Reference types** For reference types you define, override the **Equals** method and in the method do all the work necessary to compare object states and return. If your type doesn't inherit **Object**'s **Equals** method, call the base type's **Equals** method. If you want to, overload the **==** and **!=** operators and have them call the **Equals** method to do the actual work of comparing the fields.
- **Value types** For your value types, define a type–safe version of **Equals** that does all the work necessary to compare object states and return. Implement the type unsafe version of **Equals** by having it call the type–safe **Equals** internally. You also should provide overloads of the **==** and **!=** operators that call the type–safe **Equals** method internally.

### Identity

The purpose of a type's **Equals** method is to compare two instances of the type and return **true** if the instances have equivalent states or values. However, it's sometimes useful to see whether two references refer to the same, identical object. To do this, **System.Object** offers a static method called **ReferenceEquals**, which is implemented as follows:

```
class Object {
   public static Boolean ReferenceEquals(Object objA, Object objB) {
      return (objA == objB);
   }
}
```

As you can plainly see, **ReferenceEquals** simply uses the **==** operator to compare the two references. This works because of rules contained within the C# compiler. When the C# compiler sees that two references of type **Object** are being compared using the **==** operator, the compiler generates IL code that checks whether the two variables contain the same reference.

If you're writing C# code, you could use the **==** operator instead of calling **Object**'s **ReferenceEquals** method if you prefer. However, you must be very careful. The **==** operator is guaranteed to check identity only if the variables on both sides of the **==** operator are of the **System.Object** type. If a variable isn't of the **Object** type and if that variable's type has overloaded the **==** operator, the C# compiler will produce code to call the overloaded operator's method instead. So, for clarity and to ensure that your code always works as expected, don't use the **==** operator to check for identity; instead, you should use **Object**'s static **ReferenceEquals** method. Here's some code demonstrating how to use **ReferenceEquals**:

```
static void Main() {
   // Construct a reference type object.
   RefType r1 = new RefType();

   // Make another variable point to the reference object.
   RefType r2 = r1;

   // Do r1 and r2 point to the same object?
   Console.WriteLine(Object.ReferenceEquals(r1, r2));  // "True"

   // Construct another reference type object.
   r2 = new RefType();

   // Do r1 and r2 point to the same object?
   Console.WriteLine(Object.ReferenceEquals(r1, r2));  // "False"

   // Create an instance of a value type.
   Int32 x = 5;

   // Do x and x point to the same object?
   Console.WriteLine(Object.ReferenceEquals(x, x));    // "False"
   // "False" is displayed because x is boxed twice
   // into two different objects.
}
```

## Object Hash Codes

The designers of the FCL decided that it would be incredibly useful if any instance of any object could be placed into a hash table collection. To this end, **System.Object** provides a virtual

**GetHashCode** method so that an **Int32** hash code can be obtained for any and all objects.

If you define a type and override the **Equals** method, you should also override the **GetHashCode** method. In fact, Microsoft's C# compiler emits a warning if you define a type that overrides just one of these methods. For example, compiling the following type yields this warning: "warning CS0659: 'App' overrides Object.Equals(object o) but does not override Object.GetHashCode()."

```
class App {
   public override Boolean Equals(Object obj) { ... }
}
```

The reason why a type must define both **Equals** and **GetHashCode** is that the implementation of the **System.Collections.Hashtable** type requires that any two objects that are equal must have the same hash code value. So if you override **Equals**, you should override **GetHashCode** to ensure that the algorithm you use for calculating equality corresponds to the algorithm you use for calculating the object's hash code.

Basically, when you add a key/value pair to a **Hashtable** object, a hash code for the key object is obtained first. This hash code indicates what "bucket" the key/value pair should be stored in. When the **Hashtable** object needs to look up a key, it gets the hash code for the specified key object. This code identifies the "bucket" that is now searched looking for a stored key object that is equal to the specified key object. Using this algorithm of storing and looking up keys means that if you change a key object that is in a **Hashtable**, the **Hashtable** will no longer be able to find the object. If you intend to change a key object in a hash table, you should first remove the original object/value pair, next modify the key object, and then add the new key object/value pair back into the hash table.

Defining a **GetHashCode** method can be easy and straightforward. But, depending on your data types and the distribution of data, it can be tricky to come up with a hashing algorithm that returns a well–distributed range of values. Here's a simple example that will probably work just fine for **Point** objects:

```
class Point {
   Int32 x, y;
   public override Int32 GetHashCode() {
      return x ^ y;  // x XOR'd with y
   }
   Ã
}
```

When selecting an algorithm for calculating hash codes for instances of your type, try to follow these guidelines:

- Use an algorithm that gives a good random distribution for the best performance of the hash table.
- Your algorithm can also call the base type's **GetHashCode** method, including its return value in your own algorithm. However, you don't generally want to call **Object**'s or **ValueType**'s **GetHashCode** method because the implementation in either method doesn't lend itself to high–performance hashing algorithms.
- Your algorithm should use at least one instance field.
- Ideally, the fields you use in your algorithm should be immutable; that is, the fields should be initialized when the object is constructed and they should never again change during the object's lifetime.
- Your algorithm should execute as quickly as possible.

- Objects with the same value should return the same code. For example, two **String** objects with the same text should return the same hash code value.

**System.Object**'s implementation of the **GetHashCode** method doesn't know anything about its derived type and any fields that are in the type. For this reason, **Object**'s **GetHashCode** method returns a number that is guaranteed to uniquely identify the object within the AppDomain; this number is guaranteed not to change for the lifetime of the object. After the object is garbage collected, however, its unique number can be reused as the hash code for a new object.

**System.ValueType**'s implementation of **GetHashCode** uses reflection and returns the hash code of the first instance field defined in the type. This is a naive implementation that might be good for some value types, but I still recommend that you implement **GetHashCode** yourself. Even if your hash code algorithm returns the hash code for the first instance field, your implementation will be faster than **ValueType**'s implementation. Here's what **ValueType**'s implementation of **GetHashCode** looks like:

```
class ValueType {
   public override Int32 GetHashCode() {

      // Get this type's public/private instance fields.
      FieldInfo[] fields = this.GetType().GetFields(
         BindingFlags.Instance |
         BindingFlags.Public   | BindingFlags.NonPublic);

      if (fields.Length > 0) {
         // Return the hash code for the first non-null field.
         for (Int32 i = 0; i < fields.Length; i++) {
            Object obj = field[i].GetValue(this);
            if (obj != null) return obj.GetHashCode();
         }
      }

      // No non-null fields exist; return a unique value for the type.
      // NOTE: GetMethodTablePtrAsInt is an internal, undocumented method
      return GetMethodTablePtrAsInt(this);
   }
}
```

If you're implementing your own hash table collection for some reason or you're implementing any piece of code where you'll be calling **GetHashCode**, you should never persist hash code values. The reason is that hash code values are subject to change. For example, a future version of a type might use a different algorithm for calculating the object's hash code.

## Object Cloning

At times, you want to take an existing object and make a copy of it. For example, you might want to make a copy of an **Int32**, a **String**, an **ArrayList**, a **Delegate**, or some other object. For some types, however, cloning an object instance doesn't make sense. For example, it doesn't make sense to clone a **System.Threading.Thread** object since creating another **Thread** object and copying its fields doesn't create a new thread. Also, for some types, when an instance is constructed, the object is added to a linked list or some other data structure. Simple object cloning would corrupt the semantics of the type.

A class must decide whether or not it allows instances of itself to be cloned. If a class wants

instances of itself to be cloneable, the class should implement the **ICloneable** interface, which is defined as follows. (I'll talk about interfaces in depth in Chapter 15.)

```
public interface ICloneable {
   Object Clone();
}
```

This interface defines just one method, **Clone**. Your implementation of **Clone** is supposed to construct a new instance of the type and initialize the new object's state so that it is identical to the original object. The **ICloneable** interface doesn't explicitly state whether **Clone** should make a shallow copy of its fields or a deep copy. So you must decide for yourself what makes the most sense for your type and then clearly document what your **Clone** implementation does.

**Note** For those of you who are unfamiliar with the term, a *shallow copy* is when the values in an object's fields are copied but what the fields refer to is not copied. For example, if an object has a field that refers to a string and you make a shallow copy of the object, then you have two objects that refer to the same string. On the other hand, a *deep copy* is when you make a copy of what an object's fields refer to. So if you made a deep copy of an object that has a field that refers to a string, you'd be creating a new object and a new string—the new object would refer to the new string. The important thing to note about a deep copy is that the original and the new object share nothing; modifying one object has no effect on the other object.

Many developers implement **Clone** so that it makes a shallow copy. If you want a shallow copy made for your type, implement your type's **Clone** method by calling **System.Object**'s protected **MemberwiseClone** method, as demonstrated here:

```
class MyType : ICloneable {
   public Object Clone() {
      return MemberwiseClone();
   }
}
```

Internally, **Object**'s **MemberwiseClone** method allocates memory for a new object. The new object's type matches the type of the object referred to by the **this** reference. **MemberwiseClone** then iterates through all the instance fields for the type (and its base types) and copies the bits from the original object to the new object. Note that no constructor is called for the new object—its state will simply match that of the original object.

Alternatively, you can implement the **Clone** method entirely yourself, and you don't have to call **Object**'s **MemberwiseClone** method. Here's an example:

```
class MyType : ICloneable {
   ArrayList set;

   // Private constructor called by Clone
   private MyType(ArrayList set) {
      // Refer to a deep copy of the set passed.
      this.set = set.Clone();
   }

   public Object Clone() {
      // Construct a new MyType object, passing it the
      // set used by the original object.
      return new MyType(set);
   }
```

```
}
```

You might have realized that the discussion in this section has been geared toward reference types. I concentrated on reference types because instances of value types always support making shallow copies of themselves. After all, the system has to be able to copy a value type's bytes when boxing it. The following code demonstrates the cloning of value types:

```
static void Main() {
   Int32  x = 5;
   Int32  y = x;  // Copy the bytes from x to y.
   Object o = x;  // Boxing x copies the bytes from x to the heap.
   y = (Int32) o; // Unbox o, and copy bytes from the heap to y.
}
```

Of course, if you're defining a value type and you'd like your type to support deep cloning, then you should have the value type implement the **ICloneable** interface as shown earlier. (Don't call **MemberwiseClone**, but rather, allocate a new object and implement your deep copy semantics.)

# Part III: Designing Types

## Chapter List

*Chapter 7: Type Memebers and Their Accessibility*
*Chapter 8: Constants and Fields*
*Chapter 9: Methods*
*Chapter 10: properties*
*Chapter 11: Events*

# Chapter 7: Type Members and Their Accessibility

In Part II, I focused on types and what operations are guaranteed to exist on all instances of any type. I also explained the different types that you can create: reference types and value types. In this chapter and the subsequent ones in this part, I'll show how to design types using the different kinds of members that can be defined within a type. In Chapters 8 through 11, I'll discuss the various members in detail.

## Type Members

A type can define zero or more of the following members:

- **Constants (Chapter 8)** A *constant* is a symbol that identifies a never–changing data value. These symbols are typically used to make code more readable and maintainable. Constants are always associated with a type, not an instance of a type. In a sense, constants are always static.
- **Fields (Chapter 8)** A *field* represents a read–only or read/write data value. A field can be static, in which case the field is considered part of the type's state. A field can also be instance (nonstatic), in which case it's considered part of an object's state. I strongly encourage you to make fields private so that the state of the type or object can't be corrupted by code outside of the defining type.
- **Instance constructors (Chapter 9)** An *instance constructor* is a method used to initialize a new object's instance fields to a good initial state.
- **Type constructors (Chapter 9)** A *type constructor* is a method used to initialize a type's static fields to a good initial state.
- **Methods (Chapter 9)** A *method* is a function that performs operations that change or query the state of a type (static method) or an object (instance method). Methods typically read and write to the fields of the type or object.
- **Operator overloads (Chapter 9)** An *operator overload* is a method that defines how an object should be manipulated when certain operators are applied to the object. Because not all programming languages support operator overloading, operator overload methods are not part of the Common Language Specification (CLS).
- **Conversion operators (Chapter 9)** A *conversion operator* is a method that defines how to implicitly or explicitly cast or convert an object from one type to another type. As with operator overload methods, not all programming languages support conversion operators, so they're not part of the CLS.
- **Properties (Chapter 10)** A *property* is a method that allows a simple, fieldlike syntax for setting or querying part of the logical state of a type or object while ensuring that the state doesn't become corrupt.
- **Events (Chapter 11)** A *static event* is a mechanism that allows a type to send a notification to a listening type or a listening object. An *instance* (nonstatic) *event* is a mechanism that allows an object to send a notification to a listening type or a listening object. Events are usually fired in response to a state change occurring in the type or object offering the event. An *event* consists of two methods that allow types or objects ("listeners") to register and unregister interest in the "event." In addition to the two methods, events typically use a delegate field to maintain the set of registered listeners.
- **Types** A *type* can define other types nested within it. This approach is typically used to break a large, complex type down into smaller building blocks to simplify the implementation.

Again, the purpose of this chapter isn't to describe these various members in detail but to set the stage and explain what these various members all have in common.

Regardless of the programming language you're using, the corresponding compiler must process your source code and produce metadata for each kind of member in the preceding list and IL code for each method member. The format of the metadata is identical regardless of the source programming language you use, and this feature is what makes the CLR a *common language* runtime. The metadata is the common information that all languages produce and consume, enabling code in one programming language to seamlessly access code written in a completely different programming language.

This common metadata format is also used by the CLR, which determines how constants, fields, constructors, methods, properties, and events all behave at run time. Simply stated, metadata is the key to the whole Microsoft .NET Framework development platform; it enables the seamless integration of languages, types, and objects.

The following C# code shows a type definition that contains an example of all the possible members. The code shown here will compile (with warnings), but it isn't representative of a type that you'd normally create: most of the methods do nothing of any real value. Right now, I just want to show you how the compiler translates this type and its members into metadata. Once again, I'll discuss the individual members in the next few chapters.

```
using System;

class SomeType {                                                    1

   // Nested class
   class SomeNestedType { }                                         2

   // Constant, read-only, and static read/write field
   const    Int32 SomeConstant       = 1;                          3
   readonly Int32 SomeReadOnlyField  = 2;                          4
   static   Int32 SomeReadWriteField = 3;                          5

   // Type constructor
   static SomeType() { }                                            6

   // Instance constructors
   public SomeType() { }                                            7
   public SomeType(Int32 x) { }                                     8


   // Static and instance methods

   String InstanceMethod() { return null; }                         9
   static void Main() {}                                           10

   // Instance property
   Int32 SomeProp {                                                11
      get { return 0;  }                                           12
      set { }                                                      13
   }

   // Instance index property
   public Int32 this[String s] {                                   14
      get { return 0; }                                            15
      set { }                                                      16
   }

   // Instance event
   event EventHandler SomeEvent;                                   17
}
```

If you were to compile the type just defined and examine the metadata in ILDasm.exe, you'd see the output shown in Figure 7–1.

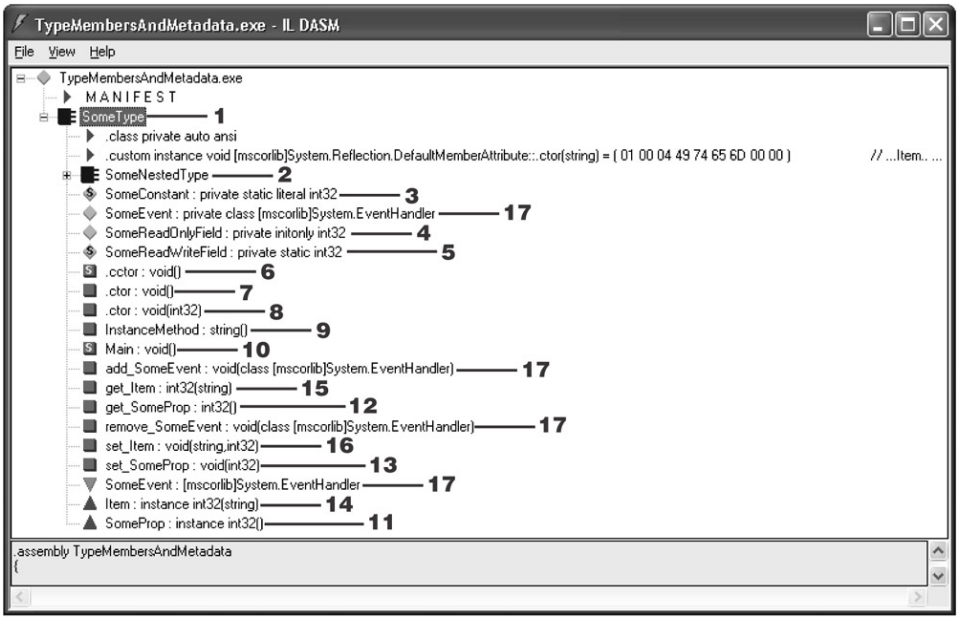



Figure 7–1: ILDasm.exe output showing metadata from preceding code

Notice that all the members defined in the source code cause the compiler to emit some metadata. In fact, some of the members (the event, 17) cause the compiler to generate additional members (a field and two methods) as well as additional metadata. I don't expect you to fully understand what you're seeing here now. But as you read the next few chapters, I encourage you to look back to this example to see how the member is defined and what effect that has on the metadata produced by the compiler.

## Accessibility Modifiers and Predefined Attributes

In this section, I'll summarize the accessibility modifiers and predefined attributes that can be placed on types, fields, and methods (including properties and events). The accessibility modifiers indicate which types and members can be legally referenced from code. The predefined attributes fine–tune this accessibility and allow you to change a member's semantics.

The CLR defines the set of possible accessibility modifiers, but each programming language chooses the syntax and term it wants to use to expose the modifier to developers. For example, the CLR uses the term *Assembly* to indicate that a member is accessible to any code within the same assembly. However, C# and Visual Basic call it **internal** and **Friend**, respectively.

Table 7–1 shows the accessibility modifiers that can be applied to a type, field, or method. *Private* is the most restrictive, and *Public* is the least restrictive.

Table 7–1: Accessibility Modifiers for Types, Fields, or Methods

| CLR Term | C# Term | Visual Basic Term | Description |
|---|---|---|---|
| Private | **private** | **Private** | Accessible only by methods in the defining |

| | | | |
|---|---|---|---|
| | | | type |
| Family | **protected** | **Protected** | Accessible only by methods in this type or one of its derived types without regard to assembly |
| Family and Assembly | (not supported) | (not supported) | Accessible only by methods in this type and by derived types in the defining assembly |
| Assembly | **internal** | **Friend** | Accessible only by methods in the defining assembly |
| Family or Assembly | **protected internal** | **Protected Friend** | Accessible only by methods in this type, any derived type, or any type defined in the defining assembly |
| Public | **public** | **Public** | Accessible to all methods in all assemblies |

When designing a type or a member, you can select only one accessibility modifier. So, for example, you can't mark a method as both *Assembly* and *Public*. Nested types (which are considered members) can be marked with any of the six accessibility modifiers. However, unnested types can be marked only with *Public* or *Assembly* accessibility because the other accessibility modifiers just don't make sense. If an unnested type isn't explicitly marked, C# and Visual Basic both default to *Assembly* (**internal**/**Friend**).

In addition to the accessibility modifiers, types and members can also be marked with some predefined attributes. The CLR defines the set of predefined attributes, but each programming language might choose different names for these attributes.

## Type Predefined Attributes

Table 7–2 shows the predefined attributes that can be applied to a type.

Table 7–2: Predefined Attributes for Types

| CLR Term | C# Term | Visual Basic Term | Description |
|---|---|---|---|
| Abstract | **abstract** | **MustInherit** | The type can't be instantiated. The type can be used as a base type for another type. If the derived type is not abstract, instances of it can be constructed. |
| Sealed | **sealed** | **NotInheritable** | The type can't be used as a base type. |

The CLR allows types to be marked as *Abstract* or as *Sealed* but not both. I think this restriction is unfortunate because a number of types don't allow instances of themselves to be created and can't be used as a base type.

For example, it doesn't make sense to construct an instance of the **Console** or **Math** type because these types contain only static methods. It also doesn't make sense to use either of these types as a base type for defining a new type. I think it would be nice to mark these types as *Abstract* (no instances can be created) and as *Sealed* (can't be used as a base type).

Because the CLR doesn't support this marking, if you are designing your own type that contains only static members, then you should mark the type as *Sealed* and define a private parameterless constructor that is never called. Defining the private constructor stops the C# compiler from automatically producing a public, parameterless constructor. Because code outside the type can't access a constructor, no instances of the type can be created.

## Field Predefined Attributes

Table 7–3 shows the predefined attributes that can be applied to a field.

Table 7–3: Predefined Attributes for Fields

| CLR Term | C# Term | Visual Basic Term | Description |
|---|---|---|---|
| Static | **static** | **Shared** | The field is part of the type's state as opposed to being part of an object's state. |
| InitOnly | **readonly** | **ReadOnly** | The field can be written to only by code contained in a constructor method. |

The CLR allows fields to be marked as *Static*, *InitOnly*, or both *Static* and *InitOnly*. Note that constants (discussed in Chapter 8) are always considered *Static* and can't be marked with *InitOnly*.

## Method Predefined Attributes

Table 7–4 shows the predefined attributes that can be applied to a method.

Table 7–4: Predefined Attributes for Methods

| CLR Term | C# Term | Visual Basic Term | Description |
|---|---|---|---|
| Static | **static** | **Shared** | Method is associated with the type itself, not an instance of the type. Static methods can't access instance fields or methods defined within the type because the static method isn't aware of any object. |
| Instance | (default) | (default) | Method is associated with an instance of the type, not the type itself. The method can access instance fields and methods as well as static fields and methods. |
| Virtual | **virtual** | **Overridable** | Most–derived method is called even if object is cast to a base type. Applies only to instance (nonstatic) methods. |
| Newslot | **new** | **Shadows** | Method should not override a virtual method defined by its base type; the method hides the inherited method. Applies only to virtual methods. |

| | | | |
|---|---|---|---|
| Override | **override** | **Overrides** | Explicitly indicates that the method is overriding a virtual method defined by its base type. Applies only to virtual methods. |
| Abstract | **abstract** | **MustOverride** | Indicates that a deriving type must implement a method with a signature matching this abstract method. A type with an abstract method is an abstract type. Applies only to virtual methods. |
| Final | **sealed** | **NotOverridable** | A derived type can't override this method. Applies only to virtual methods. |

In Chapter 9, I'll describe some of these attributes in detail. Any polymorphic instance method can be marked as either Abstract or Final but not both. Marking a virtual method as Final means that no more derived types can override the method—but this is an uncommon thing to want to do.

When compiling code, the language compiler is responsible for checking that the code is referencing types and members correctly. If the code references some type or member incorrectly, the language compiler has the responsibility of emitting the appropriate error message. In addition, the JIT compiler ensures that references to fields and methods are legal when compiling IL code into native CPU instructions. For example, if the verifier detects code that is improperly attempting to access a private field or method, the JIT compiler throws a **FieldAccessException** exception or a **MethodAccessException** exception, respectively. Verifying the IL code ensures that the accessibility modifiers and predefined attributes are properly honored at run time, even if a language compiler chose to ignore them and generated an otherwise–valid assembly.

# Chapter 8: Constants and Fields

In this chapter, I'll show you how to add data members to a type. Specifically, we'll look at constants and fields.

## Constants

A constant is a symbol that has a never–changing value. When defining a constant symbol, its value must be determinable at compile time. The compiler then saves the constant's value in the module's metadata. This means that you can define a constant only for types that your compiler considers as primitive types. Another point to remember is that constants are always considered part of a type, not part of an instance; this is fine since a constant's value never changes.

> **Note** In C#, the following types are primitives and can be used to define constants: **Boolean**, **Char**, **Byte**, **SByte**, **Decimal**, **Int16**, **UInt16**, **Int32**, **UInt32**, **Int64**, **UInt64**, **Single**, **Double**, and **String**.

When using a constant symbol, compilers look up the symbol in the metadata of the module that defines the constant, extract the constant's value, and embed the value in the emitted IL code. Because a constant's value is embedded directly in code, constants don't require any memory allocated for them at run time. In addition, you can't get the address of a constant and you can't pass a constant by reference. These constraints also mean that constants don't have a good cross–module versioning story, so you should use them only when you know that the value of a symbol will never change. (Defining **MaxInt32** as **32767** is a good example.) Let me demonstrate exactly what I mean. First take the following code and compile it into a DLL assembly:

```
using System;

public class Component {
   // NOTE: C# doesn't allow you to specify static for constants
   // because constants are always implicitly static.
   public const Int32 MaxEntriesInList = 50;
}
```

Then use the following code to build an application:

```
using System;

class App {
   static void Main() {
      Console.WriteLine("Max entries supported in list: "
         + Component.MaxEntriesInList);
   }
}
```

You'll notice that this application code references the **MaxEntriesInList** constant. When the compiler builds the application code, it sees that **MaxEntriesInList** is a constant literal with a value of **50** and embeds the **Int32** value of **50** right inside the application's IL code. In fact, after building the application code, the DLL assembly isn't even loaded at run time and can be deleted from the disk.

This example should make the versioning problem obvious to you. If the developer changes the **MaxEntriesInList** constant to **1000** and rebuilds the DLL assembly, the application code is not affected. For the application to pick up the new value, it would have to be recompiled as well. You

can't use constants if you need to have a value in one module picked up by another module at run time (instead of compile time). Instead, you can use read–only fields, which I'll discuss next.

## Fields

A field is a data member that holds an instance of a value type or a reference to a reference type. The CLR supports both type (static) and instance (nonstatic) fields. For type fields, the dynamic memory to hold the field is allocated when the type is loaded into an AppDomain (see Chapter 20), which typically happens the first time any method that references the type is JIT compiled. For instance fields, the dynamic memory to hold the field is allocated when an instance of the type is constructed.

Because fields are stored in dynamic memory, their value can be obtained at run time only. Fields also solve the versioning problem that exists with constants. In addition, a field can be of any data type, so you don't have to restrict yourself to your compiler's built–in primitive types (as you do for constants).

The CLR supports read–only fields and read/write fields. Most fields are read/write fields, meaning that the field's value might change multiple times as the code executes. However, read–only fields can be written to only within a constructor method (which is called only once, when an object is first created). Compilers and verification ensure that read–only fields are not written to by any method other than a constructor.

Let's take the example from the "Constants" section and fix the versioning problem by using a static read–only field. Here's the new version of the DLL assembly's code:

```
using System;

public class Component {
   // The static is required to associate the field with the type.
   public static readonly Int32 MaxEntriesInList = 50;
}
```

This is the only change you have to make; the application code doesn't have to change at all, although you must rebuild it to see the new behavior. Now, when the application's **Main** method runs, the CLR will load the DLL assembly (so this assembly is now required at run time) and grab the value of the **MaxEntriesInList** field out of the dynamic memory that was allocated for it. Of course, the value will be **50**.

Let's say that the developer of the DLL assembly changes the **50** to **1000** and rebuilds the assembly. When the application code is reexecuted, it will automatically pick up the new value: **1000**. In this case, the application code doesn't have to be rebuilt—it just works (although its performance is adversely affected). A caveat: this scenario assumes that the new version of the DLL assembly is not strongly named or that the versioning policy of the application is such that the CLR loads this new version.

The preceding example shows how to define a read–only static field that is associated with the type itself. You can also define read/write static fields and read–only and read/write instance fields, as shown here:

```
public class SomeType {
   // This is a static read-only field; its value is calculated and
   // stored in memory when this class is initialized at run time.
```

```
   public static readonly Random random = new Random();

   // This is a static read/write field.
   static Int32 numberOfWrites = 0;

   // This is an instance read-only field.
   public String readonly pathName = "Untitled";

   // This is an instance read/write field.
   public FileStream fs;

   public SomeType(String pathName) {
      // This line changes a read-only field.
      // This is OK because the code is in a constructor.
      this.pathName = pathName;
   }

   public String DoSomething() {
      // This line reads and writes to the static read/write field.
      numberOfWrites = numberOfWrites + 1;

      // This line reads the read-only instance field.
      return pathName;
   }
}
```

In this code, many of the fields are initialized inline. C# allows you to use this convenient inline initialization syntax to initialize a type's constants, read/write, and read–only fields. As you'll see in Chapter 9, C# treats initializing a field inline as shorthand syntax for initializing the field in a constructor.

# Chapter 9: Methods

In this chapter, I'll talk about the different kinds of methods a type can define and the various issues related to methods. Specifically, I'll show you how to define constructor methods (both instance and type), operator overload methods, and conversion operator methods (for implicit and explicit casting). In addition, I'll cover how to pass parameters by reference to a method and how to define methods that accept a variable number of parameters. Finally, I'll explain the virtual method versioning mechanism that exists to stop the potential for application instability when a base class's programming interface has changed.

## Instance Constructors

Constructors are methods that allow an instance of a type to be initialized to a good state. For code to be verifiable, the common language runtime (CLR) requires that every class (reference type) have at least one constructor defined within it. (This constructor can be private if you want to prevent code outside your class from creating any instances of the class.) When creating an instance of a reference type, memory is allocated for the instance, the object's overhead fields (method table pointer and SyncBlockIndex) are initialized, and the type's instance constructor is called to set the initial state of the object.

When constructing a reference type object, the memory allocated for the object is always zeroed out before the type's instance constructor is called. Any fields that the constructor doesn't explicitly overwrite are guaranteed to have a value of **0** or **null**.

By default, many compilers (including C#) define a public, parameterless constructor (often called a *default constructor*) for reference types when you don't explicitly define your own constructor. For example, the following type has a public, parameterless constructor, allowing any code that can access the type to construct an instance of the type.

```
class SomeType {
   // C# automatically defines a default public, parameterless constructor.
}
```

The preceding type is identical to the following type definition:

```
class SomeType {
   public SomeType() { }
}
```

A type can define several instance constructors. Each constructor must have a different signature, and each can have different accessibility. For verifiable code, a class's instance constructor must call its base class's constructor before accessing any of the inherited fields of the base class. Many compilers, including C#, generate the call to the base class's constructor automatically, so you typically don't have to worry or think about this at all. Ultimately, **System.Object**'s public, parameterless constructor gets called. This constructor does nothing—it simply returns.

In a few situations, an instance of a type can be created without an instance constructor being called. In particular, calling **Object**'s **MemberwiseClone** method allocates memory, initializes the object's overhead fields, and then copies the source object's bytes to the new object. Also, a constructor is usually not called when deserializing an object.

C# offers a simple syntax that allows the initialization of fields when a reference type object is

constructed:

```
class SomeType {
   Int32 x = 5;
}
```

When a **SomeType** object is constructed, its **x** field will be initialized to **5**. How does this happen? Well, if you examine the intermediate language (IL) for **SomeType**'s constructor method (also called **.ctor**), you'll see the code in Figure 9–1.

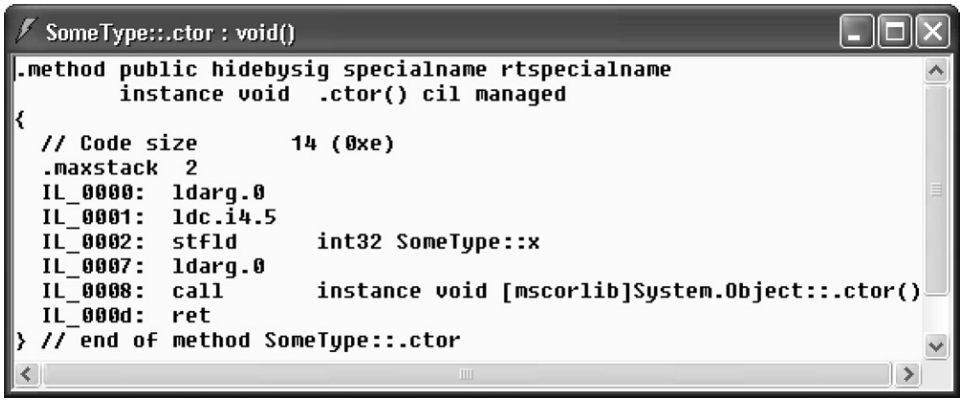



Figure 9–1 : The IL code for SomeType's constructor method

In Figure 9–1, you see that **SomeType**'s constructor contains code to store a **5** into **x** and then calls the base class's constructor. In other words, the C# compiler allows the convenient syntax that lets you initialize the instance fields inline and translates this to code in the constructor method to perform the initialization. This means that you should be aware of code explosion. Imagine the following class:

```
class SomeType {
   Int32  x = 5;
   String s = "Hi there";
   Double d = 3.14159;
   Byte   b;

   // Here are some constructors.
   public SomeType()          { ... }
   public SomeType(Int32 x)   { ... }
   public SomeType(String s) { ...; d = 10; }
}
```

When the compiler generates code for the three constructor methods, the beginning of each method includes the code to initialize **x**, **s**, and **d**. After this initialization code, the compiler appends to the method the code that appears in the constructor methods. For example, the code generated for the constructor that takes a **String** parameter includes the code to initialize **x**, **s**, and **d** and then overwrites **d** with the value **10**. Note that **b** is guaranteed to be initialized to **0** even though no code exists to explicitly initialize it.

Because there are three constructors in the preceding class, the compiler generates the code to initialize **x**, **s**, and **d** three times—once per constructor. If you have several initialized instance fields and a lot of overloaded constructor methods, you should consider defining the fields without the initialization, creating a single constructor that performs the common initialization, and having each constructor explicitly call the common initialization constructor. This approach will reduce the size of the generated code.

```
class SomeType {
```

```
   // No code here to explicitly initialize the fields
   Int32  x;
   String s;
   Double d;
   Byte   b;

   // This constructor must be called by all the other constructors.
   // This constructor contains the code to initialize the fields.
   public SomeType() {
      x = 5;
      s = "Hi There!";
      d = 3.14159;
   }

   // This constructor calls the default constructor first.
   public SomeType(Int32 x) : this() {
      this.x = x;
   }

   // This constructor calls the default constructor first.
   public SomeType(String s) : this() {
      this.s = s;
   }
}
```

Value type constructors work quite differently from reference type constructors. First, the CLR doesn't require value types to have any constructor methods defined within them. In fact, many compilers (including C#) don't give value types default parameterless constructors. The reason for the difference is that value types can be implicitly created. Examine the following code:

```
struct Point {
   public Int32 x, y;
}
class Rectangle {
   public Point topLeft, bottomRight;
}
```

To construct a **Rectangle**, the **new** operator must be used and a constructor must be specified. In this case, the constructor automatically generated by the C# compiler is called. When memory is allocated for the **Rectangle**, the memory includes the two instances of the **Point** value type. For performance reasons, the CLR doesn't attempt to call a constructor for each value type contained within the reference type. But as I mentioned earlier, the fields of the value types are initialized to **0**/**null**.

The CLR does allow you to define constructors on value types. The only way that these constructors will execute is if you write code to explicitly call one of them, as in **Rectangle**'s constructor shown here:

```
struct Point {
   public Int32 x, y;

   public Point(Int32 x, Int32 y) {
      this.x = x;
      this.y = y;
   }
}

class Rectangle {
   public Point topLeft, bottomRight;
```

```
   public Rectangle() {
      // In C#, new on a value type just lets the constructor
      // initialize the already allocated value type's memory.
      topLeft     = new Point(1, 2);
      bottomRight = new Point(100, 200);
   }
}
```

A value type's instance constructor is executed only when explicitly called. So if **Rectangle**'s constructor didn't initialize its **topLeft** and **bottomRight** fields using the **new** operator to call **Point**'s constructor, the **x** and **y** fields in both **Point** fields would be **0**.

In the **Point** value type defined earlier, no default parameterless constructor is defined. However, let's rewrite that code as follows:

```
struct Point {
   public Int32 x, y;

   public Point() {
      x = y = 5;
   }
}

class Rectangle {
   public Point topLeft, bottomRight;

   public Rectangle() {
   }
}
```

Now when a new **Rectangle** is constructed, what do you think the **x** and **y** fields in the two **Point** fields, **topLeft** and **bottomRight**, would be initialized to: **0** or **5**? (Hint: This is trick question.)

Many developers (especially those with a C++ background) would expect the C# compiler to emit code in **Rectangle**'s constructor that automatically calls **Point**'s default parameterless constructor for the **Rectangle**'s two fields. However, to improve the run–time performance of the application, the C# compiler doesn't automatically emit this code. In fact, many compilers will never emit code to call a value type's default constructor automatically, even if the value type offers a parameterless constructor. To have a value type's parameterless constructor execute, the developer must add explicit code to call a value type's constructor.

Based on the information in the preceding paragraph, you should expect the **x** and **y** fields in **Rectangle**'s two **Point** fields to be initialized to **0** in the code shown earlier because there are no explicit calls to **Point**'s constructor anywhere in the code.

However, I did say that my original question was a trick question. The "trick" part is that C# doesn't allow a value type to define a parameterless constructor. So the previous code won't actually compile. The C# compiler produces the following error when attempting to compile that code: "error CS0568: Structs cannot contain explicit parameterless constructors."

C# purposely disallows value types to define parameterless constructors to remove any confusion a developer might have about when that constructor gets called. If the constructor can't be defined, the compiler can never generate code to call it automatically. Without a parameterless constructor, a value type's fields are always initialized to **0**/**null**.

**Note** Strictly speaking, value type fields are guaranteed to be **0**/**null** when the value type is a field nested within a reference type. However, stack–based value type fields are not guaranteed to be **0**/**null**. For verifiability, any stack–based value type field must be written to prior to being read. If code could read a value type's field prior to writing to the field, a security breach is possible. C# and other compilers that produce verifiable code ensure that all stack–based value types have their fields zeroed out or at least written to before being read so that a verification exception won't be thrown at run time. For the most part, this means that you can assume that your value types have their fields initialized to **0** and you can completely ignore everything in this note.

Keep in mind that although C# doesn't allow value types with parameterless constructors, the CLR does. So if the unobvious behavior described earlier doesn't bother you, you can use another programming language (such as IL assembly language) to define your value type with a parameterless constructor.

Because C# doesn't allow value types with parameterless constructors, compiling the following type produces the following: "error CS0573: 'SomeType.x': cannot have instance field initializers in structs."

```
struct SomeValType {
   Int32 x = 5;
}
```

In addition, because verifiable code requires that every field of a value type be written to prior to any field being read, any constructors that you do have for a value type must initialize all the type's fields. The following type defines a constructor for the value type but fails to initialize all the fields:

```
struct SomeValType {
   Int32 x, y;

   // C# allows value types to have constructors that take parameters.
   public SomeValType(Int32 x) {
      this.x = x;
      // Notice that y is not initialized here.
   }
}
```

When compiling this type, the C# compiler produces the following: "error CS0171: Field 'SomeValType.y' must be fully assigned before control leaves the constructor." To fix the problem, assign a value (usually **0**) to **y** in the constructor.

**Important** In C#, your source code defines a constructor method via a method whose name matches the name of the type itself. When the C# compiler compiles the source code, it detects the constructor method and adds an entry to the module's method definition metadata table. In the table entry, the name of the constructor method is always **.ctor**. In Visual Basic source code, the developer defines a constructor by creating a method called **New**. This means that in Visual Basic, a developer can define a method (that is not a constructor method) whose name matches the name of the type. However, you should avoid this practice because the methods won't be callable from other programming languages directly. You can call the methods using reflection, though, as discussed in Chapter 20.

# Type Constructors

In addition to instance constructors, the CLR also supports type constructors (also known as *static constructors*, *class constructors*, or *type initializers*). A type constructor can be applied to interfaces (although C# doesn't allow this), reference types, and value types. Just as instance constructors are used to set the initial state of an instance of a type, type constructors are used to set the initial state of a type. By default, types don't have a type constructor defined within them. If a type has a type constructor, it can have no more than one. In addition, type constructors never have parameters. In C#, here's how to define a reference type and a value type that have type constructors:

```
class SomeRefType {
   static SomeRefType() {
      // This executes the first time a SomeRefType is accessed.
   }
}

struct SomeValType {
   // C# does allow value types to define parameterless type constructors.
   static SomeValType() {
      // This executes the first time a SomeValType is accessed.
   }
}
```

You'll notice that you define type constructors just as you would parameterless instance constructors except that you must mark them as **static**. Also, type constructors should always be private; C# makes them private for you automatically. In fact, if you explicitly mark a type constructor as private (or anything else) in your source code, the C# compiler issues the following error: "error CS0515: 'SomeValType.SomeValType()': access modifiers are not allowed on static constructors."

Type constructors should be private to prevent any developer–written code from calling them; the CLR is always capable of calling a type constructor. In addition, the CLR exercises freedom as to when it decides to call the type constructor. The CLR calls a type constructor at either of the following times:

- Just before the first instance of the type is created or just before the first access to a noninherited field or member of the class is accessed. This is called *precise* semantics because the CLR will call the type constructor at exactly the right time.
- Sometime before the first access of a noninherited static field. This is called *before–field–init* semantics because the CLR guarantees only that the static constructor will run sometime before the static field is accessed; it could run much earlier.

By default, compilers choose which of these semantics makes the most sense for the type you're defining and informs the CLR of this choice by setting the **beforefieldinit** metadata flag. Once executed, the type constructor will never be called again for the lifetime of that AppDomain (or process if the assembly is loaded in a domain–neutral fashion). Because the CLR is responsible for calling type constructors, you should always avoid writing any code that requires type constructors to be called in a specific order.

The CLR guarantees only that a type constructor has started execution—it can't guarantee that the type constructor has completed execution. This behavior was necessary to avoid deadlocks in the unusual case when two type constructors reference each other.

Finally, if a type constructor throws an unhandled exception, the CLR considers the type to be unusable. Attempting to access any fields or methods of the type will cause a **System.TypeInitializationException** to be thrown.

The code in a type constructor has access only to a type's static fields, and its usual purpose is to initialize those fields. As it does with instance fields, C# offers a simple syntax that allows you to initialize a type's static fields:

```
class SomeType {
   static Int32 x = 5;
}
```

When this code is built, the compiler automatically generates a type constructor for **SomeType**. It's as if the source code had originally been written as follows:

```
class SomeType {
   static Int32 x;
   static SomeType() { x = 5; }
}
```

Using ILDasm.exe, it's easy to verify what the compiler actually produced by examining the IL for the type constructor, shown in Figure 9–2. Type constructor methods are always called **.cctor** (for class constructor) in a method definition metadata table.

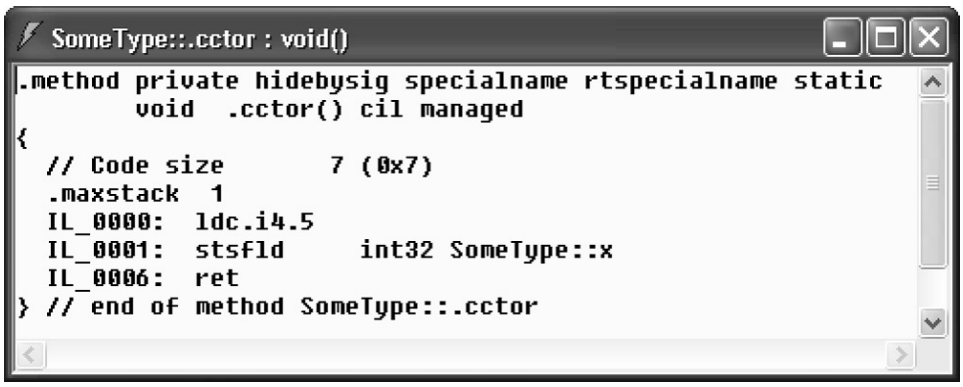



Figure 9–2 : The IL code for SomeType's type constructor method

In Figure 9–2, you see that the **.cctor** method is private and static. In addition, notice that the code in the method does in fact load a **5** into the static field **x**.

Type constructors shouldn't call a base type's type constructor. Such a call isn't necessary because none of a type's static fields are shared or inherited from its base type.

**Note** Some languages, like Java, expect that accessing a type causes its type constructor and all of its base type's type constructors to be called. In addition, interfaces implemented by the types must also have their type constructors called. The CLR doesn't offer this semantic. However, the CLR does offer compilers and developers the ability to provide this semantic via the **RunClassConstructor** method offered by the **System.Run–time.CompilerServices.RuntimeHelpers** type. Any language that requires this semantic would have its compiler emit code into a type's type constructor that calls this method for all base types and interfaces. When attempting to call a type constructor, the CLR knows if the type constructor has executed previously and, if it has, doesn't call it again.

Finally, assume that you have this code:

```
class SomeType {
   static Int32 x = 5;

   static SomeType() {
      x = 10;
   }
}
```

In this case, the C# compiler generates a single type constructor method. This constructor first initializes **x** to **5** and then initializes **x** to **10**. In other words, when the compiler generates IL code for the type constructor, it first emits the code required to initialize the static fields followed by the explicit code contained in your type constructor method. Emitting the code in this order is exactly the way it works for instance constructors too.

**Important** Occasionally, developers ask me if there's a way to get some code to execute when a type is unloaded. You should first know that types are unloaded only when the AppDomain shuts down. When the AppDomain shuts down, the object that identifies the type becomes unreachable and the garbage collector reclaims the type object's memory. This behavior leads many developers to believe that they could add a static **Finalize** method to the type, which will automatically get called when the type is unloaded. Unfortunately, the CLR doesn't support static **Finalize** methods. All is not lost, however. If you want some code to execute when an AppDomain shuts down, you can register a callback method with the **System.AppDomain** type's **DomainUnload** event.

## Operator Overload Methods

Some programming languages allow a type to define how operators should manipulate instances of the type. For example, a lot of types (such as **System.String**) overload the equality (**==**) and inequality (**!=**) operators. The CLR doesn't know anything about operator overloading because it doesn't even know what an operator is. Your programming language defines what each operator symbol means and what code should be generated when these special symbols appear.

For example, in C#, applying the **+** symbol to primitive numbers causes the compiler to generate code that adds the two numbers together. When the **+** symbol is applied to strings, the C# compiler generates code that concatenates the two strings together. For inequality, C# uses the **!=** symbol, while Visual Basic uses the <> symbol. Finally, the **^** symbol means exclusive OR (XOR) in C#, but it means exponent in Visual Basic.

Although the CLR doesn't know anything about operators, it does specify how languages should expose operator overloads so that they can be readily consumed by code written in a different programming language. Each programming language gets to decide for itself whether it will support operator overloads and, if it does, the syntax for expressing and using them. As far as the CLR is concerned, operator overloads are simply methods.

Your programming language of choice chooses whether or not to support operator overloading and what the syntax looks like. When you compile your source code, the compiler produces a method that identifies the behavior of the operator. For example, say that you define a class like this (in C#):

```
class Complex {
   public static Complex operator+(Complex c1, Complex c2) { ... }
}
```

The compiler emits a method definition for a method called **op_Addition;** the method definition entry also has the **specialname** flag set, indicating that this is a "special" method. When language compilers (including the C# compiler) see a **+** operator specified in source code, they look to see if one of the operand's types defines a **specialname** method called **op_Addition** whose parameters are compatible with the operand's types. If this method exists, the compiler emits code to call this method. If no such method exists, a compilation error occurs.

Table 9–1 shows the set of standard C# operator symbols and the corresponding recommended method name that compilers should emit and consume. I'll explain the table's third column in the next section.

Table 9–1: C# Operators and Their CLS–Compliant Method Names

| C# Operator Symbol | Special Method Name | Suggested CLS–Compliant Method Name |
|---|---|---|
| **+** | op_UnaryPlus | Plus |
| – | op_UnaryNegation | Negate |
| ~ | op_OnesComplement | OnesComplement |
| ++ | op_Increment | Increment |
| — | op_Decrement | Decrement |
| (none) | op_True | IsTrue { get; } |
| (none) | op_False | IsFalse { get; } |
| + | op_Addition | Add |
| += | op_AdditionAssignment | Add |
| – | op_Subtraction | Subtract |
| –= | op_SubtractionAssignment | Subtract |
| * | op_Multiply | Multiply |
| *= | op_MultiplicationAssignment | Multiply |
| / | op_Division | Divide |
| /= | op_DivisionAssignment | Divide |
| % | op_Modulus | Mod |
| %= | op_ModulusAssignment | Mod |
| ^ | op_ExclusiveOr | Xor |
| ^= | op_ExclusiveOrAssignment | Xor |
| & | op_BitwiseAnd | BitwiseAnd |
| &= | op_BitwiseAndAssignment | BitwiseAnd |
| \| | op_BitwiseOr | BitwiseOr |
| \|= | op_BitwiseOrAssignment | BitwiseOr |
| && | op_LogicalAnd | And |
| \|\| | op_LogicalOr | Or |
| ! | op_LogicalNot | Not |
| << | op_LeftShift | LeftShift |
| <<= | op_LeftShiftAssignment | LeftShift |
| >> | op_RightShift | RightShift |
| >>= | op_RightShiftAssignment | RightShift |

| | | |
|---|---|---|
| (none) | op_UnsignedRightShiftAssignment | RightShift |
| == | op_Equality | Equals |
| != | op_Inequality | Compare |
| < | op_LessThan | Compare |
| > | op_GreaterThan | Compare |
| <= | op_LessThanOrEqual | Compare |
| >= | op_GreaterThanOrEqual | Compare |
| **=** | op_Assign | Assign |

**Important** If you examine the core .NET Framework Class Library (FCL) types (**Int32**, **Int64**, **UInt32**, and so on), you'll see that they don't define any operator overload methods. The reason they don't is that the CLR offers IL instructions to directly manipulate instances of these types. If the types were to offer methods and if compilers were to emit code to call these methods, a run–time performance cost would be associated with the method call. Plus, the method would ultimately have to execute some IL instructions to perform the expected operation anyway. This is the reason why the core FCL types don't define any operator overload methods. Here's what this means to you: if the programming language you're using doesn't support one of the core FCL types, you won't be able to perform any operations on instances of that type. This certainly applies to Visual Basic and its lack of support for the unsigned integer types.

## Operators and Programming Language Interoperability

Operator overloading can be a very useful tool, allowing developers to express their thoughts with succinct code. However, not all programming languages support operator overloading. (For example, Visual Basic and Java don't.) So when a Visual Basic developer applies the **+** operator to a type that Visual Basic doesn't consider to be a primitive, the compiler generates an error and won't compile the code. So here's the problem that needs to be solved: How can a developer using a language that doesn't support operator overloading call the operator methods defined by a type that was written in a language that supports operator overloading?

Visual Basic doesn't offer special syntax that allows a type to define an overload for the **+** operator. In addition, Visual Basic doesn't know how to translate code using a **+** symbol to call the **op_Addition** method. However, Visual Basic (like all languages) does support the ability to call a type's methods. So in Visual Basic, you can call an **op_Addition** method that was generated by a type built with the C# compiler.

Given that information, you'd likely believe that you could also define a type in Visual Basic that offers an **op_Addition** method that would be callable by C# code using the **+** operator. However, you'd be wrong. When the C# compiler detects the **+** operator, it looks for an **op_Addition** method that has the **specialname** metadata flag associated with it so that the compiler knows for sure that the **op_Addition** method is intended to be an operator overload method. Because the **op_Addition** method produced by Visual Basic won't have the **specialname** flag associated with it, the C# compiler will produce a compilation error. Of course, code in any language can explicitly call a method that just happens to be named **op_Addition;** but the compilers won't translate a usage of the **+** symbol to call this method.

The following source code files summarize this discussion. The first one is a Visual Basic type

(defined in a library) that offers an **op_Addition** method. Again, the code isn't implemented correctly, but it compiles and demonstrates what I've been talking about.

```
Imports System

Public Class VBType

   ' Define an op_Addition method that adds two VBType objects together.
   ' This is NOT a true overload of the + operator because the Visual Basic
   ' compiler won't associate the specialname metadata flag with this method.
   Public Shared Function op_Addition(a as VBType, b as VBType) As VBType
      return Nothing
   End Function
End Class
```

The second one is a C# application that adds two instances of **VBType**:

```
using System;

public class CSharpApp {
   public static void Main() {

      // Construct an instance of VBType.
      VBType vb = new VBType();

      // When not commented out, the following line produces a
      // compiler error because VBType's op_Addition method is missing
      // the specialname metadata flag.
      // vb = vb + vb;

      // The following line compiles and runs; it just doesn't look nice.
      vb = VBType.op_Addition(vb, vb);
   }
}
```

As you can see from the preceding code, the C# code can't use the **+** symbol to add two **VBType** objects together. However, it can add the objects together by explicitly calling **VBType**'s **op_Addition** method.

Now let's reverse the example and build a Visual Basic application that uses a C# type. Here's a C# type (defined in a library) that offers an overload of the **+** operator:

```
using System;

public class CSharpType {

   // Overload the + operator
   public static CSharpType operator+(CSharpType a, CSharpType b) {
      return null;
   }
}
```

And here's a Visual Basic application that adds two instances of the **CSharpType**:

```
Imports System

Public Class VBApp
   Public Shared Sub Main()

      ' Construct an instance of the CSharpType.
```

```
      Dim cs as new CSharpType()

      ' When uncommented, the following line produces a
      ' compiler error because Visual Basic doesn't know how to translate
      ' the + symbol to call CSharpType's op_Addition method.
      ' cs = cs + cs

      ' The following line compiles and runs; it just doesn't look nice.
      cs = CSharpType.op_Addition(cs, cs)
   End Sub
End Class
```

Here the Visual Basic code can't use the **+** symbol to add two **CSharpType** objects together because Visual Basic doesn't know to translate the **+** symbol to call the **op_Addition** method. However, the Visual Basic code can add the objects together by explicitly calling **CSharpType**'s **op_Addition** method (even though this method has the **specialname** metadata flag associated with it).

---

Jeff's Opinion About Microsoft's Operator Method Name Rules

I'm sure that all these rules about when you can and can't call an operator overload method seem very confusing and overly complicated. If compilers that supported operator overloading just didn't emit the **specialname** metadata flag, the rules would be a lot simpler and programmers would have an easier time working with types that offer operator overload methods. Languages that support operator overloading would support the operator symbol syntax, and all languages would support calling the various **op_** methods explicitly. I can't come up with any reason why Microsoft made this so difficult, and I hope they'll loosen these rules with future versions of their compilers.

For a type that defines operator overload methods, Microsoft recommends that the type also define friendlier public instance methods that call the operator overload methods internally. For example, a public friendly named method, called **Add**, should be defined by a type that overloads the **op_Addition** or **op_AdditionAssignment** method. The third column in Table 9–1 lists the recommended friendly name for each operator. So the **Complex** type shown earlier should be defined like this:

```
class Complex {
   public static Complex operator+(Complex c1, Complex c2) { ... }
   public Complex Add(Complex c) { return(this + c); }
}
```

Certainly, code written in any programming language can call any of the friendly operator methods, such as **Add**. Microsoft's guideline that types offer these friendly method names complicates the story even more. I feel that this additional complication is unnecessary and that calling these friendly named methods would cause an additional performance hit unless the JIT compiler is able to inline the code in the friendly named method. Inlining the code would cause the JIT compiler to optimize the code, removing the additional method call and boosting run–time performance.

---

## Conversion Operator Methods

Occasionally, you need to convert an object from one type to an object of a different type. For example, I'm sure you've had to convert a **Byte** to an **Int32** at some point in your life. When the source type and the target type are a compiler's primitive types, the compiler knows how to emit the

necessary code to convert the object.

However, if neither type is one of the compiler's primitive types, the compiler won't know how to perform the conversion. For example, imagine that the FCL included a **Rational** data type. It might be convenient to convert an **Int32** object or a **Single** object to a **Rational** object. Moreover, it also might be nice to convert a **Rational** object to an **Int32** or a **Single** object.

To make these conversions, the **Rational** type should define public constructors that take a single parameter: an instance of the type that you're converting from. You should also define public instance **To*Xxx*** methods that take no parameters (just like the very popular **ToString** method). Each method will convert an instance of the defining type to the ***Xxx*** type. Here's how to correctly define conversion constructors and methods for a **Rational** type:

```
class Rational {
   // Constructs a Rational from an Int32
   public Rational(Int32 numerator) { ... }

   // Constructs a Rational from a Single
   public Rational(Single value) { ... }

   // Converts a Rational to an Int32
   public Int32 ToInt32() { ... }

   // Converts a Rational to a Single
   public Single ToSingle() { ... }
}
```

By invoking these constructors and methods, a developer using any programming language can convert an **Int32** or a **Single** object to a **Rational** object and convert a **Rational** object to an **Int32** or a **Single** object. The ability to do these conversions can be quite handy, and when designing a type, you should seriously consider what conversion constructors and methods make sense for your type.

In the previous section, I discussed how some programming languages offer operator overloading. Well, some programming languages (like C#) also offer conversion operator overloading. *Conversion operators* are methods that convert an object from one type to another type. You define a conversion operator method using special syntax. The following code adds four conversion operator methods to the **Rational** type:

```
class Rational {
   // Constructs a Rational from an Int32
   public Rational(Int32 numerator) { ... }

   // Constructs a Rational from a Single
   public Rational(Single value) { ... }

   // Converts a Rational to an Int32
   public Int32 ToInt32() { ... }

   // Converts a Rational to a Single
   public Single ToSingle() { ... }


   // Implicitly constructs and returns a Rational from an Int32
   public static implicit operator Rational(Int32 numerator) {
      return new Rational(numerator);
   }
```

```
   // Implicitly constructs and returns a Rational from a Single
   public static implicit operator Rational(Single value) {
      return new Rational(value);
   }

   // Explicitly returns an Int32 from a Rational
   public static explicit operator Int32(Rational r) {
      return r.ToInt32();
   }

   // Explicitly returns a Single from a Rational
   public static explicit operator Single(Rational r) {
      return r.ToSingle();
   }
}
```

Like operator overload methods, conversion operator methods must also be **public** and **static**. For conversion operator methods, you must also indicate whether a compiler can emit code to call a conversion operator method implicitly or whether the source code must explicitly indicate when the compiler is to emit code to call a conversion operator method. In C#, you use the **implicit** keyword to indicate to the compiler that an explicit cast doesn't have to appear in the source code in order to emit code that calls the method. The **explicit** keyword allows the compiler to call the method only when an explicit cast exists in the source code.

After the **implicit** or **explicit** keyword, you tell the compiler that the method is a conversion operator by specifying the **operator** keyword. After the **operator** keyword, you specify the type that an object is being cast to; in the parentheses, you specify the type that an object is being cast from.

Defining the conversion operators in the preceding **Rational** type allows you to write code like this (in C#):

```
class App {
   static void Main() {
      Rational r1 = 5;         // Implicit cast from Int32 to Rational
      Rational r2 = 2.5;       // Implicit cast from Single to Rational

      Int32  x = (Int32)  r1;  // Explicit cast from Rational to Int32
      Single s = (Single) r1;  // Explicit cast from Rational to Single
   }
}
```

Under the covers, the C# compiler detects the casts (type conversions) in the code and internally generates IL code that calls the conversion operator methods defined by the **Rational** type. But what are the names of these methods? Well, compiling the **Rational** type and examining its metadata shows that the compiler produces one method for each conversion operator defined. For the **Rational** type, the metadata for the four conversion operator methods looks like this:

```
public static Rational op_Implicit(Int32 numerator)
public static Rational op_Implicit(Single value)
public static Int32    op_Explicit(Rational r)
public static Single   op_Explicit(Rational r)
```

As you can see, methods that convert an object from one type to another are always named **op_Implicit** or **op_Explicit**. You should define an implicit conversion operator only when precision or magnitude isn't lost during a conversion, such as when converting an **Int32** to a **Rational**. However, you should define an explicit conversion operator if precision or magnitude is lost during the conversion, as when converting a **Rational** object to an **Int32**.

**Important** The two **op_Explicit** methods take the same parameter, a **Rational**. However, the methods differ by their return value, an **Int32** and a **Single**. This is an example of two methods that differ only by their return type. The CLR fully supports the ability for a type to define multiple methods that differ only by return type. However, very few languages expose this ability. As you're probably aware, C++, C#, Visual Basic, and Java are all examples of languages that don't support the definition of multiple methods that differ only by their return type. A few languages (such as IL assembly language) allow the developer to explicitly select which of these methods to call. Of course, IL assembly language programmers shouldn't take advantage of this ability because the methods they define can't be callable from other programming languages. Even though C# doesn't expose this ability to the C# programmer, the compiler does take advantage of this ability internally when a type defines conversion operator methods.

C# has full support for conversion operators. When it detects code where you're using an object of one type and an object of a different type is expected, the compiler searches for an implicit conversion operator method capable of performing the conversion and generates code to call that method. If an implicit conversion operator method exists, the compiler emits a call to it in the resulting IL code.

If the compiler sees source code that is explicitly casting an object from one type to another type, the compiler searches for an implicit or explicit conversion operator method. If one exists, the compiler emits the call to the method. If the compiler can't find an appropriate conversion operator method, it issues an error and doesn't compile the code.

To really understand operator overload methods and conversion operator methods, I strongly encourage you to use the **System.Decimal** type as a role model. **Decimal** defines several constructors that allow you to convert objects from various types to a **Decimal**. It also offers several **To*Xxx*** methods that let you convert a **Decimal** object to another type. Finally, the type defines several conversion operators and operator overload methods as well.

## Passing Parameters by Reference to a Method

By default, the CLR assumes that all method parameters are passed by value. When reference type objects are passed, the reference (or pointer) to the object is passed (by value) to the method. This means that the method can modify the object and the caller will see the change. For value type instances, a copy of the instance is passed to the method. This means that the method gets its own private copy of the value type and the instance in the caller isn't affected.

**Important** In a method, you must know whether each parameter passed is a reference type or a value type because the code you write to manipulate the parameter could be markedly different.

The CLR allows you to pass parameters by reference instead of by value. In C#, you do this by using the **out** and **ref** keywords. Both keywords tell the C# compiler to emit metadata indicating that this designated parameter is passed by reference, and the compiler uses this to generate code to pass the address of the parameter rather than the parameter itself.

The difference between the two keywords has to do with which method is responsible for initializing the object being referred to. If a method's parameter is marked with **out**, the caller isn't expected to have initialized the object prior to calling the method. The called method can't read from the value, and the called method must write to the value before returning. If a method's parameter is marked

with **ref**, the caller must initialize the parameter's value prior to calling the method. The called method can read from the value and/or write to the value.

Reference and value types behave very differently with **out** and **ref**. Let's look at using **out** and **ref** with value types first:

```
class App {
   static void Main() {
      Int32 x;
      SetVal(out x);         // x doesn't have to be initialized.
      Console.WriteLine(x);  // Displays "10"
   }

   static void SetVal(out Int32 v) {
      v = 10;  // This method must initialize v.
   }
}
```

In this code, **x** is declared on the thread's stack. The address of **x** is then passed to **SetVal**. **SetVal**'s **v** is a pointer to an **Int32** value type. Inside **SetVal**, the **Int32** that **v** points to is changed to **10**. When **SetVal** returns, **x** has a value of **10** and "10" is displayed on the console. Using **out** with value types is efficient because it prevents instances of the value type's fields from being copied when making method calls.

Now let's look at an example that uses **ref** instead of **out**:

```
class App {
   static void Main() {
      Int32 x = 5;
      AddVal(ref x);        // x must be initialized.
      Console.WriteLine(x); // Displays "15"
   }

   static void AddVal(ref Int32 v) {
      v += 10;  // This method can use the initialize value in v.
   }
}
```

In this code, **x** is declared on the thread's stack and is initialized to **5**. The address of **x** is then passed to **AddVal**. **AddVal**'s **v** is a pointer to an **Int32** value type. Inside **AddVal**, the **Int32** that **v** points to is required to have a value already. So, **AddVal** can use the initial value in any expression it desires. **AddVal** can also change the value and the new value will be "returned" back to the caller. In this example, **AddVal** adds **10** to the initial value. When **AddVal** returns, **Main**'s **x** will contain "15", which is what gets displayed in the console.

To summarize, from an IL or a CLR perspective, **out** and **ref** do exactly the same thing: they both cause a pointer to the instance to be passed. The difference is that the compiler helps ensure that your code is correct. The following code that attempts to pass an uninitialized value to a method expecting a **ref** parameter produces a compilation error:

```
class App {
   static void Main() {
      Int32 x;              // x is not initialized.

      // The following line fails to compile, producing
      // error CS0165: Use of unassigned local variable 'x'.
      AddVal(ref x);
```

```
      Console.WriteLine(x); // Displays "15"
   }

   static void AddVal(ref Int32 v) {
      v += 10;  // This method can use the initialized value in v.
   }
}
```

**Important** I'm frequently asked why C# requires that a call to a method must specify **out** or **ref**. After all, the compiler knows whether the method being called requires **out** or **ref** and should be able to compile the code correctly. It turns out that the compiler can indeed do the right thing automatically. However, the designers of the C# language felt that the caller should explicitly state its intention. This way, at the call site, it's obvious that the method being called is expected to change the value of the variable being passed.

In addition, the CLR allows you to overload methods based on their use of **out** and **ref** parameters. For example, in C#, the following code is legal and compiles just fine:

```
class Point {
   static void Add(Point p) { ... }
   static void Add(ref Point p) { ... }
}
```

It's not legal to overload methods that differ only by **out** and **ref** because the JIT compiled code for the methods would be identical. So I couldn't define the following method in the preceding **Point** type:

```
   static void Add(out Point p) { ... }
```

Using **out** and **ref** with value types gives you the same behavior that you already get when passing reference types by value. With value types, **out** and **ref** allow a method to manipulate a single value type instance. The caller must allocate the memory for the instance, and the callee manipulates that memory. With reference types, the caller allocates memory for a pointer to a reference object and the callee manipulates this pointer. Because of this behavior, using **out** and **ref** with reference types is useful only when the method is going to "return" a reference to an object that it knows about. The following code demonstrates:

```
class App {
   static public void Main() {
       FileStream fs;

       // Open the first file to be processed.
       StartProcessingFiles(out fs);

       // Continue while there are more files to process.
       for (; fs != null; ContinueProcessingFiles(ref fs)) {

          // Process a file.
          fs.Read(...);
       }
   }

   static void StartProcessingFiles(out FileStream fs) {
      fs = new FileStream(...);
   }

   static void ContinueProcessingFiles(ref FileStream fs) {
      fs.Close();  // Close the last file worked on.
```

```
      // Open the next file, or if no more files, return null.
      if (noMoreFilesToProcess) fs = null;
      else fs = new FileStream (...);
   }
}
```

As you can see, the big difference with this code is that the methods that have **out** or **ref** reference type parameters are constructing an object and the pointer to the new object is returned to the caller. You'll also notice that the **ContinueProcessingFiles** method can manipulate the object being passed into it before returning a new object. This is possible because the parameter is marked with the **ref** keyword.

You can simplify the preceding code a bit, as shown here:

```
class App {
   static public void Main() {
   FileStream fs = null;   // Initialized to null (required)

   // Open the first file to be processed.
   ProcessFiles(ref fs);

   // Continue while there are more files to process.
   for (; fs != null; ProcessFiles(ref fs)) {

      // Process a file.
      fs.Read(...);
   }
}

   void ProcessingFiles(ref FileStream fs) {
      // Close the previous file if one was open.
      if (fs != null) fs.Close();  // Close the last file worked on.

      // Open the next file, or if no more files, return null.
      if (noMoreFilesToProcess) fs = null;
      else fs = new FileStream (...);
   }
}
```

Here's another example that demonstrates how to use the **ref** keyword to implement a method that swaps two reference types:

```
static public void Swap(ref Object a, ref Object b) {
   Object t = b;
   b = a;
   a = t;
}
```

To swap references to two **String** objects, you'd probably think that you could write code like this:

```
static public void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";
   Ã
   Swap(ref s1, ref s2);
   Console.WriteLine(s1);  // Displays "Richter"
   Console.WriteLine(s2);  // Displays "Jeffrey"
}
```

However, this code won't compile. The problem is that variables passed by reference to a method must be the same type. In other words, **Swap** expects two references to an **Object** type, not two references to a **String** type. To swap the two **String** references, you must do this:

```
static public void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";
   Ã
   // Variables that are passed by reference
   // must match what the method expects.
   Object o1 = s1, o2 = s2;
   Swap(ref o1, ref o2);

   // Now cast the objects back to strings.
   s1 = (String) o1;
   s2 = (String) o2;

   Console.WriteLine(s1);  // Displays "Richter"
   Console.WriteLine(s2);  // Displays "Jeffrey"
}
```

This version of **SomeMethod** does compile and execute as expected. The reason why the parameters passed must match the parameters expected by the method is to ensure that type safety is preserved. The following code, which thankfully won't compile, shows how type safety could be compromised.

```
class SomeType {
   public Int32 val;
}

class App {
   static void Main() {
      SomeType st;

      // The following line generates error CS1503: Argument '1':
      // cannot convert from 'ref SomeType' to 'ref object'.
      GetAnObject(out st);

      Console.WriteLine(st.val);
  }

   void GetAnObject(out Object o) {
      o = new String('X', 100);
   }
}
```

In this code, **Main** clearly expects **GetAnObject** to return a **SomeType** object. However, because **GetAnObject**'s signature indicates a reference to an **Object**, **GetAnObject** is free to initialize **o** to an object of any type. In this example, when **GetAnObject** returned to **Main**, **st** would refer to a **String**, which is clearly not a **SomeType** object, and the call to **Console.WriteLine** would certainly fail. Fortunately, the C# compiler won't compile the preceding code because **st** is a reference to **SomeType** but **GetAnObject** requires a reference to an **Object**.

## Passing a Variable Number of Parameters to a Method

It's sometimes convenient for the developer to define a method that can accept a variable number of parameters. For example, the **System.String** type offers methods allowing an arbitrary number

of strings to be concatenated together and methods allowing the caller to specify a set of strings that are to be formatted together.

To declare a method that accepts a variable number of arguments, you declare the method as follows:

```
static Int32 Add(params Int32[] values) {
   // NOTE: it is possible to pass this
   // array to other methods if you want to.

   Int32 sum = 0;
   for (Int32 x = 0; x < values.Length; x++)
      sum += values[x];
   return sum;
}
```

Everything in this method should look very familiar to you except for the **params** keyword that is applied to the last parameter of the method signature. Ignoring the **params** keyword for the moment, it's obvious that this method accepts an array of **Int32** values and iterates over the array, adding up all the values. The resulting sum is returned to the caller.

Obviously, code can call this method as follows:

```
static void Main() {
   // Displays "15"
   Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 } ));
}
```

It's clear that the array can easily be initialized with an arbitrary number of elements and then passed off to **Add** for processing. Although the preceding code would compile and work correctly, it is a little ugly. As developers, we would certainly prefer to have written the call to **Add** as follows:

```
static void Main() {
   // Displays "15"
   Console.WriteLine(Add(1, 2, 3, 4, 5));
}
```

You'll be happy to know that we can do this because of the **params** keyword. The **params** keyword tells the compiler to apply an instance of the **System.Param–ArrayAttribute** custom attribute (which I'll discuss in Chapter 16) to the parameter. Because the **params** keyword is just shorthand for this attribute, the **Add** method's prototype could have been defined like this:

```
static Int32 Add([ParamArray] Int32[] values) {
   Ã
}
```

When the C# compiler detects a call to a method, the compiler checks all the methods with the specified name, where no parameter has the **ParamArray** attribute applied. If a method exists that can accept the call, the compiler generates the code necessary to call the method. However, if the compiler can't find a match, it looks for methods that have a **ParamArray** attribute to see whether the call can be satisfied. If the compiler finds a match, it emits code that constructs an array and populates its elements before emitting the code that calls the selected method.

In the previous example, no **Add** method is defined that takes five **Int32**–compatible arguments; however, the compiler sees that the source code has a call to **Add** that is being passed a list of **Int32** values and that there is an **Add** method whose array–of–**Int32** parameter is marked with the

**ParamArray** attribute. So the compiler considers this a match and generates code that coerces the parameters into an **Int32** array and then calls the **Add** method. The end result is that you can write the code easily passing a bunch of parameters to **Add**, but the compiler generates code as though you'd written the first version that explicitly constructs and initializes the array.

Only the last parameter to a method can be marked with the **params** keyword (**ParamArrayAttribute**). This parameter must also identify a single–dimension array of any type. It's legal to pass **null** or a reference to an array of **0** entries as the last parameter to the method. The following call to **Add** compiles fine, runs fine, and produces a resulting sum of **0** (as expected):

```
static void Main() {
   // Displays "0"
   Console.WriteLine(Add());
}
```

So far, all the examples have shown how to write a method that takes an arbitrary number of **Int32** parameters. How would you write a method that takes an arbitrary number of parameters where the parameters could be any type? The answer is very simple: just modify the method's prototype so that it takes an **Object[]** instead of an **Int32[]**. Here's a method that displays the **Type** of every object passed to it:

```
class App {
   static void Main() {
      DisplayTypes(new Object(), new Random(), "Jeff", 5);
   }

   static void DisplayTypes(params Object[] objects) {
      foreach (Object o in objects)
         Console.WriteLine(o.GetType());
   }
}
```

Running this code yields the following output:

```
System.Object
System.Random
System.String
System.Int32
```

## How Virtual Methods Are Called

Methods represent code that performs some operation on the type (static methods) or an instance of the type (nonstatic methods). All methods have a name, a signature, and a return value. A type can have multiple methods with the same name as long as each method has a different set of parameters or a different return value. So it's possible to define two methods with the same name and same parameters as long as the methods have a different return type. However, except for IL assembly language, I'm not aware of any language that takes advantage of this "feature"; most languages require that methods differ by parameters and ignore a method's return type when determining uniqueness.

By examining metadata, the CLR can determine whether a nonstatic method is a virtual or nonvirtual method. However, the CLR doesn't use this information when calling a method. Instead, the CLR offers two IL instructions for calling a method: **call** and **callvirt**. The **call** IL instruction calls a method based on the type of the reference, and the **callvirt** IL instruction calls a method based on

the type of the object referred to. When compiling your source code, the compiler knows whether or not you're calling a virtual method and emits the proper **call** or **callvirt** IL instruction. This means that it's possible to call a virtual method nonvirtually, an approach that is commonly used when your code calls a virtual method defined in your type's base class, as shown here:

```
class SomeClass {
   // ToString is a virtual method defined in the base class: Object.
   public override String ToString() {

      // Compiler uses the 'call' IL instruction to call
      // Object's ToString method nonvirtually.

      // If the compiler were to use 'callvirt' instead of 'call', this
      // method would call itself recursively until the stack overflowed.
      return base.ToString();
   }
}
```

Compilers also typically emit the **call** IL instruction when calling a virtual method using a reference to a sealed type. Emitting **call** instead of **callvirt** improves performance because the CLR doesn't have to check the actual type of the object being referenced. In addition, for value types (which are always sealed), using **call** prevents the boxing of the value type, which reduces memory and CPU usage.

Regardless of whether **call** or **callvirt** is used to call an instance method, all instance methods always receive a hidden **this** pointer as the method's first parameter. The **this** pointer refers to the object being operated on.

## Virtual Method Versioning

Back in the old days, a single company was responsible for all the code that made up an application. Today, many different companies often contribute parts to help construct another company's application. For example, lots of applications today use components created by other companies—in fact, the COM(+) and .NET technologies encourage this practice. When applications consist of many parts created and distributed by different companies, many versioning issues come up.

I talked about some of these versioning issues in Chapter 3 when I explained strongly named assemblies and discussed how an administrator can ensure that an application binds to the assemblies that it was built and tested with. However, other versioning issues cause source code compatibility problems. For example, you must be very careful when adding or modifying members of a type if that type is used as a base type. Let's look at some examples.

CompanyA has designed the following type, **Phone**:

```
namespace CompanyA {
   class Phone {
      public void Dial() {
         Console.WriteLine("Phone.Dial");
         // Do work to dial the phone here.
      }
   }
}
```

Now imagine that CompanyB defines another type, **BetterPhone**, which uses CompanyA's **Phone** type as its base:

```
namespace CompanyB {
   class BetterPhone : CompanyA.Phone {
      public void Dial() {
         Console.WriteLine("BetterPhone.Dial");
         EstablishConnection();
         base.Dial();
      }

      protected virtual void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

When CompanyB attempts to compile its code, the C# compiler issues the following warning: "warning CS0108: The keyword new is required on 'BetterPhone.Dial()' because it hides inherited member 'Phone.Dial()'." This warning is notifying the developer that **BetterPhone** is defining a **Dial** method, which will hide the **Dial** method defined in **Phone**. This new method could change the semantic meaning of **Dial** (as defined by CompanyA when it originally created the **Dial** method).

It's a very nice feature of the compiler to warn you of this potential semantic mismatch. The compiler also tells you how to remove the warning by adding the **new** keyword before the definition of **Dial** in the **BetterPhone** class. Here's the fixed **BetterPhone** class:

```
namespace CompanyB {
   class BetterPhone : CompanyA.Phone {

      // This Dial method has nothing to do with Phone's Dial method.
      new public void Dial() {
         Console.WriteLine("BetterPhone.Dial");
         EstablishConnection();
         base.Dial();
      }

      protected virtual void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

At this point, CompanyB can use **BetterPhone** in its application. Here's some sample code that CompanyB might write:

```
class App {
   static void Main() {
      CompanyB.BetterPhone phone = new CompanyB.BetterPhone();
      phone.Dial();
   }
}
```

When this code runs, the following output is displayed:

```
BetterPhone.Dial
BetterPhone.EstablishConnection
```

```
Phone.Dial
```

This output shows that CompanyB is getting the behavior it desires. The call to **Dial** is calling the new **Dial** method defined by **BetterPhone**, which calls the virtual **EstablishConnection** method and then calls the **Phone** base type's **Dial** method.

Now let's imagine that several companies have decided to use CompanyA's **Phone** type. Let's further imagine that these other companies have decided that the ability to establish a connection in the **Dial** method is a really useful feature. This feedback is given to CompanyA, who now goes and revises its **Phone** class:

```
namespace CompanyA {
   class Phone {
      public void Dial() {
         Console.WriteLine("Phone.Dial");
         EstablishConnection();
         // Do work to dial the phone here.
      }

      protected virtual void EstablishConnection() {
         Console.WriteLine("Phone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

Now when CompanyB compiles its **BetterPhone** type (derived from this new version of CompanyA's **Phone**), the compiler issues this warning: "warning CS0114: 'BetterPhone.EstablishConnection()' hides inherited member 'Phone.EstablishConnection()'. To make the current member override that implementation, add the override keyword. Otherwise, add the new keyword."

The compiler is alerting you to the fact that both **Phone** and **BetterPhone** offer an **EstablishConnection** method and that the semantics of both might not be identical: simply recompiling **BetterPhone** can no longer give the same behavior as it did when using the first version of the **Phone** type.

If CompanyB decides that the **EstablishConnection** methods are not semantically identical in both types, then CompanyB can tell the compiler that the **Dial** and **EstablishConnection** methods defined in **BetterPhone** are the "correct" methods to use and that they have no relationship with the **Dial** and **Establish–Connection** methods defined in the **Phone** base type. CompanyB informs the compiler by keeping **new** on the **Dial** method and by adding **new** to the **Establish–Connection** method:

```
namespace CompanyB {
   class BetterPhone : CompanyA.Phone {

      // Keep 'new' to mark this method as having no
      // relationship to the base type's Dial method.
      new public void Dial() {
         Console.WriteLine("BetterPhone.Dial");
         EstablishConnection();
         base.Dial();
      }

      // Add 'new' to mark this method as having no
      // relationship to the base type's EstablishConnection method.
```

```
      new protected virtual void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

In this code, the **new** keyword tells the compiler to emit metadata, making it clear to the CLR that **BetterPhone**'s **Dial** and **EstablishConnection** methods are intended to be treated as new functions that are introduced by the **BetterPhone** type. The CLR will know that there is no relationship between **Phone**'s and **BetterPhone**'s methods.

**Note** Without the **new** keyword, the developer of **BetterPhone** couldn't use the method names **Dial** and **EstablishConnection**. This would most likely cause a ripple effect of changes throughout the entire source code base, breaking source and binary compatibility. This type of pervasive change is usually undesirable, especially in any moderate to large project. However, if changing the method name causes only moderate updates in the source code, you should change the name of the methods so that the two different meanings of **Dial** and **EstablishConnection** don't confuse other developers.

When the same application code (in the **Main** method) executes, the output is as follows:

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
Phone.EstablishConnection
```

This output shows that **Main**'s call to **Dial** calls the new **Dial** method defined by **BetterPhone**. **Dial** then calls the virtual **EstablishConnection** method that is also defined by **BetterPhone**. When **BetterPhone**'s **EstablishConnection** method returns, **Phone**'s **Dial** method is called. **Phone**'s **Dial** method calls **Establish–Connection**, but because **BetterPhone**'s **EstablishConnection** is marked with **new**, **BetterPhone**'s **EstablishConnection** method isn't considered an override of **Phone**'s virtual **EstablishConnection** method. As a result, **Phone**'s **Dial** method calls **Phone**'s **EstablishConnection** method—this is the desired behavior.

Alternatively, CompanyB could have gotten the new version of CompanyA's **Phone** type and decided that **Phone**'s semantics of **Dial** and **EstablishConnection** are exactly what it's been looking for. In this case, CompanyB would modify its **BetterPhone** type by removing its **Dial** method entirely. In addition, because CompanyB now wants to tell the compiler that **BetterPhone**'s **EstablishConnection** method is related to **Phone**'s **EstablishConnection** method, the **new** keyword must be removed. Simply removing the **new** keyword isn't enough, though, because now the compiler can't tell exactly what the intention is of **BetterPhone**'s **EstablishConnection** method. To express his intent exactly, the CompanyB developer must also change **BetterPhone**'s **EstablishConnection** method from **virtual** to **override**. The following code shows the new version of **BetterPhone**:

```
namespace CompanyB {
   class BetterPhone : CompanyA.Phone {

      // Delete the Dial method (inherit Dial from base).

      // Remove 'new' and change 'virtual' to 'override' to
      // mark this method as having a relationship to the base
      // type's EstablishConnection method.
      protected override void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
```

```
            // Do work to establish the connection.
        }
    }
}
```

Now when the same application code (in the **Main** method) executes, the output is as follows:

```
Phone.Dial
BetterPhone.EstablishConnection
```

This output shows that **Main**'s call to **Dial** calls the **Dial** method defined by **Phone** and inherited by **BetterPhone**. Then when **Phone**'s **Dial** method calls the virtual **EstablishConnection** method, **BetterPhone**'s **EstablishConnection** method is called because it overrides the virtual **EstablishConnection** method defined by **Phone**.

# Chapter 10: Properties

In this chapter, I'll talk about properties. Properties allow source code to call a method using a simplified syntax. The common language runtime (CLR) offers two kinds of properties: parameterless properties, which are called *properties*, and parameterful properties, which are called different names by different programming languages. For example, C# calls parameterful properties *indexers*, and Visual Basic calls them *default properties*.

## Parameterless Properties

Many types define state information that can be retrieved or altered. Frequently, this state information is implemented as field members of the type. For example, here's a type definition that contains two fields:

```
public class Employee {
   public String Name;  // The employee's name
   public Int32  Age;   // The employee's age
}
```

If you were to create an instance of this type, you could easily get or set any of this state information with code similar to the following:

```
Employee e = new Employee();
e.Name = "Jeffrey Richter"; // Set the employee's Name.
e.Age  = 35;                // Set the employee's Age.

Console.WriteLine(e.Name);    // Displays "Jeffrey Richter"
```

Querying and setting an object's state information in the way I just demonstrated is very common. However, I would argue that the preceding code should never be implemented as it's shown. One of the covenants of object-oriented design and programming is *data encapsulation*. Data encapsulation means that your type's fields should never be publicly exposed because it's too easy to write code that improperly uses the fields, corrupting the object's state. For example, a developer could easily corrupt an **Employee** object with code like this:

```
e.Age = -5; // How could someone be Ð5 years old?
```

There are additional reasons for encapsulating access to a type's data field. For example, you might want access to a field to execute some side effect, cache some value, or lazily create some internal object. You might also want access to the field to be thread safe. Or perhaps the field is a logical field whose value isn't represented by bytes in memory but whose value is instead calculated using some algorithm.

For any of these reasons, when designing a type, I strongly suggest that all your fields be private or at least protected—never public. Then, to allow a user of your type to get or set state information, you expose methods for that specific purpose. Methods that wrap access to a field are typically called *accessor methods*. These accessor methods can optionally perform sanity checking and ensure that the object's state is never corrupted. For example, I'd rewrite the previous class as follows:

```
public class Employee {
   private String Name;    // Field is now private
   private Int32  Age;     // Field is now private
```

```
   public String GetName() {
      return(Name);
   }

   public void SetName(String value) {
      Name = value;
   }

   public Int32 GetAge() {
      return(Age);
   }

   public void SetAge(Int32 value) {
      if (value < 0)
         throw new ArgumentOutOfRangeException(
            "Age must be greater than or equal to 0");
      Age = value;
   }
}
```

Although this is a simple example, you should still be able to see the enormous benefit you get from encapsulating the data fields. You should also be able to see how easy it is to make read–only or write–only properties: just don't implement one of the accessor methods.

Encapsulating the data as shown earlier has two disadvantages. First, you have to write more code because you now have to implement additional methods. Second, users of the type must now call methods rather than simply refer to a single field name.

```
e.SetAge(35);    // Updates the age
e.SetAge(-5);    // Throws ArgumentOutOfRangeException
```

Personally, I think these disadvantages are quite minor. Nevertheless, the CLR offers a mechanism, properties, that alleviates the first disadvantage a little and removes the second disadvantage entirely.

The class shown here uses properties and is functionally identical to the class shown earlier:

```
public class Employee {
   private String _Name; // Prepending '_' avoids name conflict.
   private Int32  _Age;  // Prepending '_' avoids name conflict.

   public String Name {
      get { return(_Name); }
      set { _Name = value; } // The 'value' keyword always identifies
                             // the new value.
   }

   public Int32 Age {
      get { return(_Age); }
      set {
         if (value < 0)    // The 'value' keyword always identifies the
                           // new value.
            throw new ArgumentOutOfRangeException(
               "Age must be greater than or equal to 0");
         _Age = value;
      }
   }
}
```

As you can see, properties complicate the definition of the type slightly, but the fact that they allow you to write your code as follows more than compensates for the extra work:

```
e.Age = 35;    // Updates the age
e.Age = -5;    // Throws ArgumentOutOfRangeException
```

You can think of properties as *smart fields*: fields with additional logic behind them. The CLR supports static, instance, and virtual properties. In addition, properties can be marked with any accessibility modifier (discussed in Chapter 7) and defined within an interface (discussed in Chapter 15).

Each property has a name and a type (which can't be **void**). It isn't possible to overload properties (that is, have two properties with the same name if their types are different). When you define a property, you typically specify both a **get** and a **set** method. However, you can leave out the **set** method to define a read–only property or leave out the **get** method to define a write–only property.

It's also quite common for the property's **get**/**set** methods to manipulate a private field defined within the type. This field is commonly referred to as the *backing field*. The **get** and **set** methods don't have to access a backing field, however. For example, the **System.Threading.Thread** type offers a **Priority** property that communicates directly with the operating system; the **Thread** object doesn't maintain a field for a thread's priority. Another example of properties without backing fields are those read–only properties calculated at run time—for example, the length of a zero–terminated array or the area of a rectangle when you have its height and width.

When you define a property, the compiler emits up to three things into the resulting managed module:

- A method representing the property's **get** accessor method. This is emitted only if you define a **get** accessor for the property.
- A method representing the property's **set** accessor method. This is emitted only if you define a **set** accessor for the property.
- A property definition in the managed module's metadata. This is always emitted.

Refer back to the **Employee** type shown earlier. As the compiler compiles this type, it comes across the **Name** and **Age** properties. Because both properties have **get** and **set** accessor methods, the compiler emits four method definitions into the **Employee** type. It's as though the original source were written as follows:

```
public class Employee {
   private String _Name; // Prepending '_' avoids name conflict.
   private Int32  _Age;  // Prepending '_' avoids name conflict.

   public String get_Name(){
      return _Name;
   }
   public void   set_Name(String value) {
      _Name = value; // The 'value' always identifies the new value.
   }

   public Int32 get_Age() {
      return _Age;
   }

   public void  set_Age(Int32 value) {
      if (value < 0)    // The 'value' always identifies the new value.
         throw new ArgumentOutOfRangeException(
```

```
            "Age must be greater than or equal to 0");
         _Age = value;
      }
   }
}
```

The compiler automatically generates names for these methods by prepending **get_** or **set_** to the property name specified by the developer.

C# has built–in support for properties. When the C# compiler sees code that's trying to get or set a property, the compiler actually emits a call to one of these methods. If you're using a programming language that doesn't directly support properties, you can still access properties by calling the desired accessor method. The effect is exactly the same; it's just that the source code doesn't look as pretty.

In addition to emitting the accessor methods, compilers also emit a property definition entry into the managed module's metadata for each property defined in the source code. This entry contains some flags and the type of the property, and it refers to the **get** and **set** accessor methods. This information exists simply to draw an association between the abstract concept of a "property" and its accessor methods. Compilers and other tools can use this metadata, which can be obtained by using the **System.Reflection.PropertyInfo** class. The CLR doesn't use this metadata information, though, and requires only the accessor methods at run time.

You should use properties only for operations that execute quickly because the syntax to access a property is identical to the syntax for accessing a field. Code with this syntax traditionally doesn't take long to execute; use methods for operations that require more execution time. For example, calculating the area of a rectangle is fast, so it would make sense to use a read–only property. But calculating the number of elements in a linked–list collection might be quite slow, so you might want to use a method instead of a read–only property.

For simple **get** and **set** accessor methods, the JIT compiler *inlines* the code so that there's no run–time performance hit as a result of using properties rather than fields. Inlining is when the code for a method (or accessor method, in this case) is compiled directly in the method that is making the call. This removes the overhead associated with making a call at run time at the expense of making the compiled method's code bigger. Because property accessor methods typically contain very little code, inlining them can make code smaller and can make it execute faster.

## Parameterful Properties

In the previous section, the **get** accessor methods for the properties accepted no parameters. For this reason, I called these properties *parameterless properties*. These properties are easy to understand because they have the feel of accessing a field. In addition to these fieldlike properties, the CLR also supports what I call *parameterful properties*, whose **get** access methods accept one or more parameters. Different programming languages expose parameterful properties in different ways. Also, languages use different terms to refer to parameterful properties: C# calls them *indexers*, Visual Basic calls them *default properties*, and C++ with Managed Extensions calls them *index properties*. In this section, I'll focus on how C# exposes its indexers using parameterful properties.

In C#, parameterful properties (indexers) are exposed using an arraylike syntax. In other words, you can think of an indexer as a way for the C# developer to overload the **[]** operator. Here's an example of a **BitArray** type that allows arraylike syntax to index into the set of bits maintained by an

instance of the type:

```
public class BitArray {
   // Private array of bytes that hold the bits
   private Byte[] byteArray;
   private Int32 numBits;

   // Constructor that allocates the byte array and sets all bits to 0
   public BitArray(Int32 numBits) {
      // Validate arguments first.
      if (numBits <= 0)
         throw new ArgumentOutOfRangeException("numBits must be > 0");

      // Save the number of bits.
      this.numBits = numBits;

      // Allocate the bytes for the bit array.
      byteArray = new Byte[(numBits + 7) / 8];
   }

   // This is the indexer.
   public Boolean this[Int32 bitPos] {

      // This is the index property's get accessor method.
      get {
         // Validate arguments first
         if ((bitPos < 0) || (bitPos >= numBits))
            throw new IndexOutOfRangeException();

         // Return the state of the indexed bit.
         return (byteArray[bitPos / 8] & (1 << (bitPos % 8))) != 0);
      }

      // This is the index property's set accessor method.
      set {
         if ((bitPos < 0) || (bitPos >= numBits))
            throw new IndexOutOfRangeException();

         if (value) {
            // Turn the indexed bit on.
            byteArray[bitPos / 8] = (Byte)
               (byteArray[bitPos / 8] | (1 << (bitPos % 8)));
         } else {
            // Turn the indexed bit off.
            byteArray[bitPos / 8] = (Byte)
               (byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
         }
      }
   }
}
```

Using the **BitArray** type's indexer is incredibly simple:

```
// Allocate a BitArray that can hold 14 bits.
BitArray ba = new BitArray(14);

// Turn all the even-numbered bits on by calling the set accessor.
for (Int32 x = 0; x < 14; x++) {
   ba[x] = (x % 2 == 0);
}

// Show the state of all the bits by calling the get accessor.
```

```
foreach (Int32 x = 0; x < 14; x++) {
   Console.WriteLine("Bit " + x + " is " + (ba[x] ? "On" : "Off"));
}
```

In the **BitArray** example, the indexer takes one **Int32** parameter, **bitPos**. All indexers must have at least one parameter, but they can have more. These parameters (as well as the return type) can be of any type.

It's quite common to create an indexer that takes an **Object** as a parameter to look up values in an associative array. In fact, the **System.Collections.–Hashtable** type offers an indexer that takes a key (of type **Object**) and returns the value associated with the key (also of type **Object**). Unlike parameterless properties, a type can offer multiple, overloaded indexers as long as their signatures differ.

Like a parameterless property's **set** accessor method, an indexer's **set** accessor method also contains a hidden parameter, called **value** in C#. This parameter indicates the new value desired for the "indexed element."

The CLR doesn't differentiate parameterless properties and parameterful properties; to the CLR, each is simply a pair of methods defined within a type. As mentioned earlier, different programming languages require different syntax to create and use parameterful properties. The fact that C# requires **this[...]** as the syntax for expressing an indexer was purely a choice made by the C# team. What this choice means is that C# allows indexers to be defined only on instances of objects. C# doesn't offer syntax allowing a developer to define a static indexer property, although the CLR does support static parameterful properties.

Because the CLR treats parameterful properties just as it does parameterless properties, the compiler emits the same three items into the resulting managed module:

- A method representing the parameterful property's **get** accessor method. This is emitted only if you define a **get** accessor for the property.
- A method representing the parameterful property's **set** accessor method. This is emitted only if you define a **set** accessor for the property.
- A property definition in the managed module's metadata, which is always emitted. There's no special parameterful property metadata definition table because, to the CLR, parameterful properties are just properties.

For the **BitArray** type shown earlier, the compiler compiles the indexer as though the original source code were written as follows:

```
public class BitArray {
   Ã
   // This is the indexer's get accessor method.
   public Boolean get_Item(Int32 bitPos) { ... }

   // This is the indexer's set accessor method.
   public void    set_Item(Int32 bitPos, Boolean value)  { ... }
}
```

The compiler automatically generates names for these methods by prepending **get_** or **set_** to **Item**. Because the C# syntax for an indexer doesn't allow the developer to specify a name, the C# compiler team had to choose a name to use for the accessor methods and they chose **Item**. A single type can define multiple indexers as long as the indexers all take different parameter sets.

When examining the .NET Framework Reference documentation, you can tell if a type offers an indexer by looking for a property named **Item**. For example, the **System.Collections.SortedList** type offers a public instance property named **Item;** this property is **SortedList**'s indexer.

When you program in C#, you never see the name of **Item**, so you don't normally care that the compiler has chosen this name for you. However, if you're designing an indexer for a type that other programming languages will be accessing, you might want to change the name given to your indexer's **get** and **set** accessor methods. C# allows you to rename these methods by applying the **System.Runtime.CompilerServices.IndexerName** custom attribute to the indexer. The following code demonstrates:

```
public class BitArray {
   Ã
   [System.Runtime.CompilerServices.IndexerName("Bit")]
   public Boolean this[Int32 bitPos] {
      Ã
   }
}
```

Now the compiler will emit methods called **get_Bit** and **set_Bit** instead of **get_Item** and **set_Item**.

Here's some Visual Basic code that demonstrates how to access this C# indexer:

```
' Construct an instance of the SomeType type.
Dim ba as new BitArray(10)
   Ã
' Visual Basic uses () instead of [] to specify array elements.
Console.WriteLine(ba(2))       ' Displays True or False

' Visual Basic also allows you to access the indexer by its name.
Console.WriteLine(cs.Bit(2))   ' Displays same as previous line
```

In some programming languages, the **IndexerName** attribute allows you to define multiple indexers with the same signature because each can have a different name. However, C# won't allow you to do this because its syntax doesn't refer to the indexer by name; the compiler wouldn't know which indexer you were referring to. Attempting to compile the following C# source code causes the compiler to generate the following: "error CS0111: Class 'SomeType' already defines a member called 'this' with the same parameter types."

```
using System;
using System.Runtime.CompilerServices;

public class SomeType {

   // Define a get_Item accessor method.
   public Int32 this[Boolean b] {
      get { return 0; }
   }

   // Define a get_Jeff accessor method.
   [IndexerName("Jeff")]
   public String this[Boolean b] {
      get { return ""; }
   }
}
```

You can clearly see that C# thinks of indexers as a way to overload the **[]** operator, and this operator can't be used to disambiguate parameterful properties with different method names.

By the way, the **System.String** type is an example of a type that changed the name of its indexer. The name of **String**'s indexer is **Chars** instead of **Item**. This property allows you to get the individual characters within a string. For programming languages that don't use **[]** operator syntax to access this property, **Chars** was decided to be a more meaningful name.

---

Selecting the Primary Parameterful Property

C#'s limitations with respect to indexers begs the following questions: What if a type is defined in a programming language that does allow the developer to define several parameterful properties? How can this type be consumed from C#? The answer is that a type must select one of the parameterful property method names to be the default property by applying an instance of **System.Reflection.DefaultMemberAttribute** to the class itself. For the record, **DefaultMemberAttribute** can be applied to a class, a structure, or an interface. In C#, when you compile a type that defines a parameterful property, the C# compiler automatically applies an instance of **DefaultMemberAttribute** to the defining type. This attribute's constructor specifies the name that is to be used for the type's default parameterful property.

So, in C#, if you define a type that has a parameterful property and you don't specify the **IndexerName** attribute, the defining type will have a **DefaultMember** attribute indicating **Item**. If you apply the **IndexerName** attribute to a parameterful property, the defining type will have a **DefaultMember** attribute indicating the string name specified in the **IndexerName** attribute. Remember, C# won't compile the code if it contains parameterful properties with different names.

For a language that supports several parameterful properties, one of the property method names must be selected and identified by the type's **DefaultMember** attribute. This is the only parameterful property that C# will be able to access.

---

When the C# compiler sees code that is trying to get or set an indexer, the compiler actually emits a call to one of these methods. Some programming languages might not support parameterful properties. To access a parameterful property from one of these languages, you must call the desired accessor method explicitly.

To the CLR, there's no difference between paramterless properties and parameterful properties, so you use the same **System.Reflection.PropertyInfo** class to find the association between a parameterful property and its accessor methods. The JIT compiler is also free to inline an accessor method's code into the calling method's code.

# Chapter 11: Events

## Overview

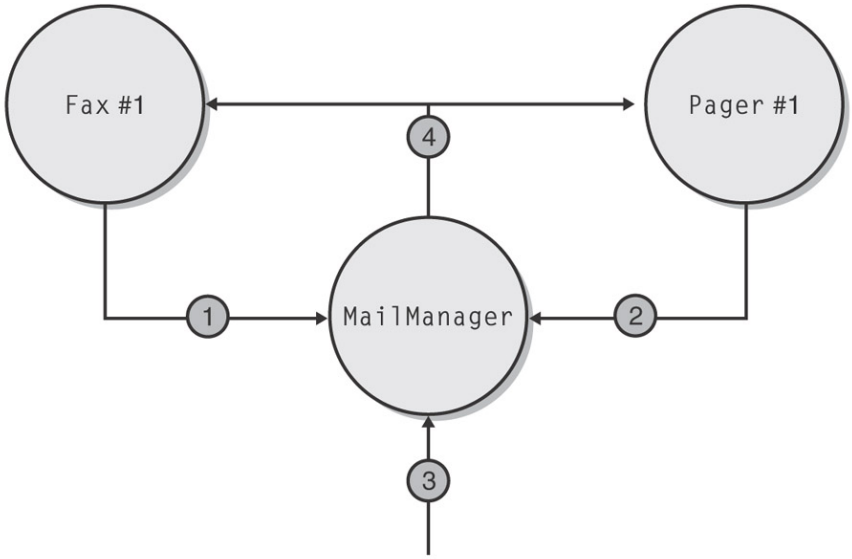
In this chapter, I'll talk about the last kind of member that a type can define: events. A type that defines an event member allows the type (or instances of the type) to notify other objects that something special has happened. For example, the **Button** class defines an event called **Click**. When a **Button** object is clicked, sev−eral objects in the application might want to receive a notification and perform some action. Events are type members that allow for this interaction. Specifically, defining an event member means that a type is offering three capabilities:

- The capability for objects to register their interest in the event
- The capability for objects to unregister their interest in the event
- The capability for the object defining the event to maintain the set of registered objects and to notify these objects when something special happens

The CLR's eventing model is based on *delegates*. A delegate is a type−safe way to invoke a callback method. In this chapter, I'll be using delegates, but I won't fully explain the ins and outs of them until Chapter 17.

To help you fully understand the way events work within the CLR, I'll start by defining a scenario where events are useful. Suppose you want to design an e−mail application. When an e−mail message arrives, the user might like the message to be forwarded to a fax machine or a pager. In architecting this application, let's say that you'll first design a type, called **MailManager**, that receives the incoming e−mail messages. **MailManager** will expose an event, called **MailMsg**. Other types (such as **Fax** and **Pager**) might register interest in this event. When **MailManager** receives a new e−mail message, it will fire the event, distributing the message to each of the registered objects. Each object can process the message any way it desires.

When the application initializes, let's instantiate just one instance of **MailManager**—the application can then instantiate any number of **Fax** and **Pager** types. Figure 11−1 shows how the application initializes and what happens when a new e−mail message arrives.




1. A `Fax` object registers interest with the `MailManager`'s event.
2. A `Pager` object registers interest with the `MailManager`'s event.
3. A new mail message arrives at `MailManager`.
4. The `MailManager` object fires the notification off to all the registered objects, which process the mail message as desired.

Figure 11–1 : Architecting an application to use events

Here's how the application illustrated in Figure 11–1 works. The application initializes by constructing an instance of **MailManager**. **MailManager** offers a **MailMsg** event. When the **Fax** and **Pager** objects are constructed, they register themselves with **MailManager**'s **MailMsg** event so that **MailManager** knows to notify the **Fax** and **Pager** objects when new e–mail messages arrive. Now, when **MailManager** receives a new e–mail message (sometime in the future), it will fire the **MailMsg** event, giving all the registered objects an opportunity to process the new message any way they want.

## Designing a Type That Exposes an Event

Let's look at the type definition for **MailManager** to really understand all the steps you must take to properly define an event member using Microsoft's recommended design pattern:

```
class MailManager {
   // The MailMsgEventArgs type is defined within the MailManager type.
   public class MailMsgEventArgs : EventArgs {

      // 1. Type defining information passed to receivers of the event
      public MailMsgEventArgs(
         String from, String to, String subject, String body) {

         this.from    = from;
         this.to      = to;
         this.subject = subject;
         this.body    = body;
      }

      public readonly String from, to, subject, body;
```

```
   }

   // 2. Delegate type defining the prototype of the callback method
   //    that receivers must implement
   public delegate void MailMsgEventHandler(
      Object sender, MailMsgEventArgs args);

   // 3. The event itself
   public event MailMsgEventHandler MailMsg;

   // 4. Protected, virtual method responsible for notifying registered
   //    objects of the event
   protected virtual void OnMailMsg(MailMsgEventArgs e) {

      // Has any objects registered interest with the event?
      if (MailMsg != null) {

         // Yes—notify all the objects in the delegate linked list.
         MailMsg(this, e);
      }
   }

   // 5. Method that translates the input into the desired event.
   //    This method is called when a new e-mail message arrives.
   public void SimulateArrivingMsg(String from, String to,
      String subject, String body) {

      // Construct an object to hold the information you want
      // to pass to the receivers of the notification.
      MailMsgEventArgs e =
         new MailMsgEventArgs(from, to, subject, body);

      // Call the virtual method notifying the object that the event
      // occurred. If no derived type overrides this method, the object
      // will notify all the registered listeners.
      OnMailMsg(e);
   }
}
```

All the work related to implementing this architecture is the responsibility of the developer who is designing the **MailManager** type. The developer must define the following items:

1. **Define a type that will hold any additional information that should be sent to receivers of the event notification** By convention, types that hold event information are derived from **System.EventArgs**, and the name of the type should end with **EventArgs**. In this example, the **MailMsgEventArgs** type has fields identifying who sent the message (from), who is receiving the message (to), the subject of the message (subject), and the message text itself (body).

   **Note** The **EventArgs** type is defined in the .NET Framework Class Library (FCL) and looks like this:

   ```
   [Serializable]
   public class EventArgs {
      public static readonly EventArgs Empty = new EventArgs();
      public EventArgs() {  }
   }
   ```

   As you can see, this type is nothing to write home about. It simply serves as a base type from which other types can derive. Many events don't have any additional information to

pass on. For example, when a **Button** notifies its registered receivers that the button has been clicked, just invoking the callback method is enough information. When you're defining an event that doesn't have any additional data to pass on, just use **EventArgs.Empty** rather than constructing a new **EventArgs** object.

2. **Define a delegate type specifying the prototype of the method that will be called when the event fires** By convention, the name of the delegate should end with **EventHandler**. Also by convention, the prototype should have a **void** return value and take two parameters (although some event handlers in the FCL, such as **System.ResolveEventHandler**, violate this convention). The first parameter is an **Object** that refers to the object sending the notification, and the second parameter is an **EventArgs**–derived type containing any additional information that receivers of the notification require.

   If you're defining an event that has no additional information that you want to pass to receivers of the event, you don't have to define a new delegate; you can use the FCL's **System.EventHandler** delegate and pass **EventArgs.Empty** for the second parameter. The prototype of **EventHandler** is as follows:

   ```
   public delegate void EventHandler(Object sender, EventArgs e);
   ```

3. **Define an event** In this example, **MailMsg** is the name of the event. This event is of the **MailMsgEventHandler** type, meaning that all receivers of the event notification must supply a callback method whose prototype matches that of the **MailMsgEventHandler** delegate.
4. **Define a protected, virtual method responsible for notifying registered objects of the event** The **OnMailMsg** method is called when a new e–mail message arrives. This method receives an initialized **MailMsgEventArgs** object containing additional information about the event. This method should first check to see whether any objects have registered interest in the event, and if they have, fire the event.

   A type that uses **MailManager** as a base type is free to override the **OnMailMsg** method. This capability gives the derived type control over the firing of the event. The derived type can handle the new e–mail message in any way it sees fit. Usually, a derived type calls the base type's **OnMailMsg** method so that the registered object receives the notification. However, the derived type might decide not to have the event forwarded on.
5. **Define a method that translates the input into the desired event** Your type must have some method that takes some input and translates it into the firing of an event. In this example, the **SimulateArrivingMsg** method is called to indicate that a new e–mail message has arrived into **MailManager**. **SimulateArrivingMsg** accepts information about the message and constructs a new **MailMsgEventArgs** object, passing the message information to its constructor. **MailManager**'s own virtual **OnMailMsg** method is then called to formally notify the **MailManager** object of the new e–mail message. Normally, this causes the event to be fired, notifying all the registered objects. (As mentioned before, a type using **MailManager** as a base type can override this behavior.)

Let's now take a closer look at what it really means to define the **MailMsg** event. When the compiler examines the source code, the compiler comes across the line that defines the event:

```
public event MailMsgEventHandler MailMsg;
```

The C# compiler translates this single line of source code into the following three constructs:

```
// 1. A PRIVATE delegate field that is initialized to null
private MailMsgEventHandler MailMsg = null;

// 2. A PUBLIC add_* method
//    Allows objects to register interest in the event.
```

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void add_MailMsg(MailMsgEventHandler handler) {
   MailMsg = (MailMsgEventHandler)
      Delegate.Combine(MailMsg, handler);
}

// 3. A PUBLIC remove_* method
//    Allows objects to unregister interest in the event.
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void remove_MailMsg(MailMsgEventHandler handler) {
   MailMsg = (MailMsgEventHandler)
      Delegate.Remove(MailMsg, handler);
}
```

The first construct is simply a field of the appropriate delegate type. This field is a reference to the head of a linked list of delegates that want to be notified of this event. This field is initialized to **null**, meaning that no listeners have registered interest in the event. When an object registers interest in the event, this field refers to an instance of the **MailMsgEventHandler** delegate. Each **MailMsgEventHandler** delegate instance has a pointer to yet another **MailMsgEventHandler** delegate or to **null** to mark the end of the linked list. When a listener registers interest in an event, the listener is simply adding an instance of the delegate type to the linked list. Obviously, unregistering means removing the delegate from the linked list.

You'll notice that the delegate field, **MailMsg** in this example, is always private even though the original line of source code defines the event as public. The reason for making the delegate field private is to prevent code outside the defining type from manipulating it improperly. If the field were public, any code could assign to the field, wiping out all the delegates that have registered interest in the event.

The second construct the C# compiler generates is a method that allows other objects to register their interest in the event. The C# compiler automatically names this function by prepending **add_** to the event's name (**MailMsg**). The C# compiler automatically generates the code that is inside this method. The code always calls **System.Delegate**'s static **Combine** method, which adds the instance of a delegate to the linked list of delegates and returns the new head of the linked list.

The third and final construct the C# compiler generates is a method that allows an object to unregister its interest in the event. Again, the C# compiler automatically names this function by prepending **remove_** to the event's name (**MailMsg**). The code inside this method always calls **Delegate**'s static **Remove** method, which removes the instance of a delegate from the linked list of delegates and returns the new head of the linked list.

You should also notice that both the **add** and **remove** methods have a **MethodImplAttribute** attribute applied to them. More specifically, these methods are marked as synchronized, making them thread safe: multiple listeners can register or unregister themselves with the event at the same time without corrupting the linked list.

In this example, the **add** and **remove** methods are public. The reason they are public is that the original line of source code declared the event to be public. If the event had been declared protected, the **add** and **remove** methods generated by the compiler would also have been declared protected. So, when you define an event in a type, the accessibility of the event determines what code can register and unregister interest in the event, but only the type itself can ever access the delegate field directly.

In addition to emitting the preceding three constructs, compilers also emit an event definition entry into the managed module's metadata. This entry contains some flags and the underlying delegate

type, and it refers to the **add** and **remove** accessor methods. This information exists simply to draw an association between the abstract concept of an "event" and its accessor methods. Compilers and other tools can use this metadata, and certainly, this information can be obtained by using the **System.Reflection.EventInfo** class. However, the CLR itself doesn't use this metadata information and requires only the accessor methods at run time.

## Designing a Type That Listens for an Event

The hard work is definitely behind you at this point. In this section, I'll show you how to define a type that uses an event provided by another type. Let's start off by examining the code for the **Fax** type:

```
class Fax {
   // Pass the MailManager object to the constructor.
   public Fax(MailManager mm) {

      // Construct an instance of the MailMsgEventHandler
      // delegate that refers to the FaxMsg callback method.
      // Register the callback with MailManager's MailMsg event.
      mm.MailMsg += new MailManager.MailMsgEventHandler(FaxMsg);
   }

   // This is the method that MailManager will call to
   // notify the Fax object that a new e-mail message has arrived.
   private void FaxMsg(
      Object sender, MailManager.MailMsgEventArgs e) {

      // The 'sender' identifies the MailManager in case
      // you want to communicate back to it.

      // The 'e' identifies the additional event information
      // that the MailManager wants to provide.

      // Normally, the code here would fax the e-mail message. This test
      // implementation displays the information on the console.
      Console.WriteLine("Faxing mail message:");
      Console.WriteLine(
         "   To: {0}\n   From: {1}\n   Subject: {2}\n   Body: {3}\n",
         e.from, e.to, e.subject, e.body);
   }

   public void Unregister(MailManager mm) {

      // Construct an instance of the MailMsgEventHandler
      // delegate that refers to the FaxMsg callback method.
      MailManager.MailMsgEventHandler callback =
         new MailManager.MailMsgEventHandler(FaxMsg);

      // Unregister with MailManager's MailMsg event.
      mm.MailMsg -= callback;
   }
}
```

When the e-mail application initializes, it would first construct a **MailManager** object and save the reference to this object in a variable. Then the application would construct a **Fax** object, passing the reference to the **MailManager** object as a parameter. In the **Fax** constructor, a new **MailManager.MailMsgEventHandler** delegate object is constructed. This new delegate object is a wrapper around the **Fax** type's **FaxMsg** method. You'll notice that the **FaxMsg** method returns **void** and takes the same two parameters as defined by the **MailManager**'s **MailMsgEventHandler**

delegate—this is required for the code to compile.

After constructing the delegate, the **Fax** object registers its interest in **MailManager**'s **MailMsg** event using C#'s **+=** operator:

```
mm.MailMsg += new MailManager.MailMsgEventHandler(FaxMsg);
```

Because the C# compiler has built–in support for events, the compiler translates the use of the **+=** operator into the following line of code to add the object's interest in the event:

```
mm.add_MailMsg(new MailManager.MailMsgEventHandler(FaxMsg));
```

Even if you're using a programming language that doesn't directly support events, you can still register a delegate with the event by calling the **add** accessor method explicitly. The effect is identical; it's just that the source code doesn't look as pretty. It's the **add** method that registers the delegate with the event by adding it to the event's linked list of delegates.

When **MailManager** fires the event, the **Fax** object's **FaxMsg** method gets called. The method is passed a reference to the **MailManager** object. Most of the time, this parameter is ignored, but it can be used if the **Fax** object wants to access fields or methods of the **MailManager** object in response to the event notification. The second parameter is a reference to a **MailMsgEventArgs** object. This object contains any additional information that **MailManager** thought would be useful to event receivers.

From the **MailMsgEventArgs** object, the **FaxMsg** method has easy access to the message's sender, the message's recipient, the message's subject, and the message's text. In a real **Fax** object, this information would be faxed to somewhere. In this example, the information is simply displayed in the console window.

When an object is no longer interested in receiving event notifications, it should unregister its interest. For example, the **Fax** object would unregister its interest in the **MailMsg** event if the user no longer wanted her e–mail forwarded to a fax. As long as an object stays registered with another object's event, the object can't be garbage collected. If your type implements **IDisposable**'s **Dispose** method, disposing of the object should cause it to unregister interest in all events. (See Chapter 19 for more information about **IDisposable**.)

Code that demonstrates how to unregister for an event is shown in **Fax**'s **Unregister** method. This method is practically identical to the code shown in the **Fax** type's constructor. The only difference is that this code uses **–=** instead of **+=**. When the C# compiler sees code using the **–=** operator to unregister a delegate with an event, the compiler emits a call to the event's **remove** method:

```
mm.remove_MailMsg(callback};
```

As with the **+=** operator, even if you're using a programming language that doesn't directly support events, you can still unregister a delegate with the event by calling the **remove** accessor method explicitly. The **remove** method unregisters the delegate from the event by scanning the linked list for a delegate that wraps the same method as the callback passed in. If a match is found, the existing delegate is removed from the event's linked list of delegates. If a match isn't found, no error occurs and the linked list is unaltered.

By the way, C# requires that your code use the **+=** and **–=** operators to add and remove delegates from the linked list. If you try to call the **add** or **remove** method explicitly, the C# compiler produces a "cannot explicitly call operator or accessor" error.

The **MailManager** sample application (which can be downloaded from *http://www.wintellect.com*) shows all the source code for the **MailManager** type, the **Fax** type, and the **Pager** type. You'll notice that the **Pager** type is implemented quite similarly to the **Fax** type.

## Explicitly Controlling Event Registration

Sometimes the **add** and **remove** methods the compiler generates are not ideal. For example, if you're adding and removing delegates frequently and you know that your application is single threaded, the overhead of synchronizing access to the object that owns the delegate can really hurt your application's performance.

A more common reason for implementing **add** and **remove** is if your type defines many events. I'll cover this scenario specifically in the next section, "Designing a Type That Defines Lots of Events."

Fortunately, C# and many other compilers allow the developer to create explicit implementations of the **add** and **remove** accessor methods. The following example shows how you could modify **MailManager**. This new version provides explicit implementation of the **add** and **remove** methods.

```
class MailManager {
   // 1. Type-defining information passed to receivers of the event
   public class MailMsgEventArgs : EventArgs { ... }

   // 2. Delegate type defining the prototype of the callback method
   //    that receivers must implement
   public delegate void MailMsgEventHandler(
      Object sender, MailMsgEventArgs args);

   // 3a. Explicitly define a private delegate linked list field.
   private MailMsgEventHandler mailMsgEventHandlerDelegate;

   // 3b. Explicitly define an event and its accessor methods.
   public event MailMsgEventHandler MailMsg {

      // Add handler (passed as 'value') to the delegate linked list.
      add {
         mailMsgEventHandlerDelegate = (MailMsgEventHandler)
            Delegate.Combine(mailMsgEventHandlerDelegate, value);
      }

      // Remove handler (passed as 'value') from the delegate linked list.
      remove {
         mailMsgEventHandlerDelegate = (MailMsgEventHandler)
            Delegate.Remove(mailMsgEventHandlerDelegate, value);
      }
   }

   // 4. Protected, virtual method responsible for
   //    notifying registered objects of the event
   protected virtual void OnMailMsg(MailMsgEventArgs e) {

      // Have any objects registered interest with the event?
      if (mailMsgEventHandlerDelegate != null) {

         // Yes—notify all the objects in the delegate linked list.
         mailMsgEventHandlerDelegate(this, e);
      }
   }
```

```
   // 5. Method that translates the input into the desired event.
   //    This method is called when a new e-mail message arrives.
   public void SimulateArrivingMsg(String from, String to,
      String subject, String body) { ... }
}
```

In this new version of **MailManager**, steps 1, 2, and 5 have required no modification. They are exactly the same as shown in the earlier example. However, step 3 has been broken up into two steps (3a and 3b), and step 4 has also had a slight modification. Let's discuss these changes.

**3a Explicitly define a private field that is a reference to a delegate linked list** In the original event syntax, the C# compiler automatically defined a private field for you. Using this new event syntax, the developer must not only explicitly supply implementations for the **add** and **remove** accessor methods but also explicitly supply the field.

**3b Explicitly define an event and its accessor methods** The field defined in step 3a is just a field (a reference to a **MailMsgEventHandler** delegate). There's nothing that makes this field an event. The new event syntax is actually what defines an event within the type. The code in the **add** and **remove** blocks provides the implementation for the accessor methods. Notice that each method accepts a hidden parameter, called **value**, which is of the **MailMsgEventHandler** type. Inside, these methods implement the code necessary to add or remove a **MailMsgEventHandler** delegate from the linked list. Unlike properties, which can have a **get** accessor method, a **set** accessor method, or both, events must always have both **add** and **remove** accessor methods.

The explicit implementations shown in the preceding example have exactly the same behavior as the methods that the C# compiler would have provided except that the **MethodImplAttribute** is omitted. In fact, you can still use the **+=** and **–=** operators in your source code and the compiler will know to emit calls to the explicit methods. The accessor methods are no longer thread safe, but their performance is better. Of course, you should remove the **MethodImplAttribute** only if you know for sure that just one thread at a time will be accessing the event's delegate.

**4 Define a protected, virtual method responsible for notifying registered objects of the event** Semantically, the **OnMailMsg** method is identical to the previous version. The only difference is that the name of the event has been replaced with the name of the delegate field.

## Designing a Type That Defines Lots of Events

In the previous section, I described a scenario in which you might want to explicitly provide **add** and **remove** accessor methods for an event. However, when you're explicitly implementing the accessor methods yourself, you can get a bit more creative with their implementation. Let's see how explicitly implementing these methods can reduce the memory usage of an application.

The **System.Windows.Forms.Control** type defines about 60 events. If the **Control** type implemented the events by allowing the compiler to implicitly generate the **add** and **remove** accessor methods and delegate fields, then every **Control** object would have 60 delegate fields in it just for the events! Because objects never register interest in most of these events, an enormous amount of memory is wasted for each object created from a **Control**–derived type. By the way, the

**System.Web.UI.Control** type also uses the following technique to reduce the memory wasted by unused events.

By explicitly implementing the event's **add** and **remove** methods creatively, you can greatly reduce the amount of space wasted by each object. In this section, I'll show you how to define a type that exposes many events efficiently.

The basic idea is that each object will maintain a collection (usually a hash table) of event/delegate pairs. When a new object is constructed, this collection is empty. When interest in an event is registered, the type of event is looked up in the collection. If the event is there, the new delegate is combined with the linked list of delegates for this event. If the event isn't identified in the collection, the event is added with the delegate.

When the event needs to fire, the event is looked up in the collection. If the collection doesn't have an entry for the event, then nothing has registered interest in the event and no delegates are called back. If the event is in the collection, then the delegate linked list associated with the event is invoked.

The following code demonstrates the recommended design pattern that should be used to design a type that defines lots of events:

```
class TypeWithLotsOfEvents {

   // 1. Define a protected instance field that references a collection.
   //    The collection manages a set of Event/Delegate pairs.
   //    NOTE: The EventHandlerSet type is not part of the FCL.
   //    It is my own type, and I describe it later in this chapter.
   protected EventHandlerSet eventSet =
     EventHandlerSet.Synchronized(new EventHandlerSet());


   // 2.  Define the members necessary for the Foo event.
   // 2a. Construct a static, read-only object to identify this event.
   //     Each object has its own hash code for looking up this
   //     event's delegate linked list in the object's collection.
   protected static readonly Object fooEventKey = new Object();

   // 2b. Define the EventArgs-derived type for this event.
   public class FooEventArgs : EventArgs {}

   // 2c. Define the delegate prototype for this event.
   public delegate void FooEventHandler(Object sender, FooEventArgs e);

   // 2d. Define the event's accessor methods that add/remove the
   //     delegate from the collection.
   public event FooEventHandler Foo {
      add    { eventSet.AddHandler(fooEventKey, value); }
      remove { eventSet.RemoveHandler(fooEventKey, value); }
   }

   // 2e. Define the protected, virtual On method for this event.
   protected virtual void OnFoo(FooEventArgs e) {
      eventSet.Fire(fooEventKey, this, e);
   }

   // 2f. Define the method that translates input to this event.
   public void SimulateFoo() {
      OnFoo(new FooEventArgs());
   }
```

```
   // 3.  Define the members necessary for the Bar event.
   // 3a. Construct a static, read-only object to identify this event.
   //     Each object has its own hash code for looking up this
   //     event's delegate linked list in the object's collection.
   protected static readonly Object barEventKey = new Object();

   // 3b. Define the EventArgs-derived type for this event.
   public class BarEventArgs : EventArgs {}

   // 3c. Define the delegate prototype for this event.
   public delegate void BarEventHandler(Object sender, BarEventArgs e);

   // 3d. Define the event's accessor methods that add/remove the
   //     delegate from the collection.
   public event BarEventHandler Bar {
      add    { eventSet.AddHandler(barEventKey, value); }
      remove { eventSet.RemoveHandler(barEventKey, value); }
   }

   // 3e. Define the protected, virtual On method for this event.
   protected virtual void OnBar(BarEventArgs e) {
      eventSet.Fire(barEventKey, this, e);
   }

   // 3f. Define the method that translates input to this event.
   public void SimulateBar() {
      OnBar(new BarEventArgs());
   }
}
```

Implementing this design pattern is the responsibility of the developer who is designing the type that defines the events; the developer using the type has no idea how the events are implemented internally. Now I'll walk you through the steps in the preceding code:

1. **Define a protected instance field that references a collection** Each instance of this type will maintain its own set of registered event listeners using this collection. The collection is usually implemented via a hash table to provide for fast lookup of an event. Each entry in the collection has a key (usually of type **System.Object**) that uniquely identifies the event type and a value that identifies the head of a delegate linked list representing the methods that should be called back when the event fires. This field should be protected so that derived types can use the collection for any additional events they want to define. In the preceding code, I'm using an **EventHandlerSet** collection. This is a collection type that I've defined. Internally, the **EventHandlerSet** type uses a **System.Collections.Hashtable**. The **EventHandlerSet** type has some additional members that make it well suited to working with events and delegates. I'll explain the **EventHandlerSet** type later in this chapter.
2. **Define the members necessary for an event the type wants to expose** For each event that you want the type to expose, you must define a set of members. These members are described in steps 2a through 2f. You'll notice that no instance data fields are used—they are what would have caused the wasted memory.

   **a. Construct a static, read-only object to identify this event** There must be some way to uniquely identify each event in the collection. Because I'm using a hash table, I just need a unique key, which can be generated from an object's hash code. All instances of the type can share this unique key, so I obtain the key by constructing an **Object** and saving the reference to it in a static, read-only field. This technique conserves memory only if you expect users of the type to construct several instances of it. If users construct only a single

instance of the type, this technique will actually use more memory; in this case, you shouldn't use it. Although in my example, I've defined the static field to be protected so that derived types can use it, you can just as easily define this field as private instead.

**b. Define the EventArgs–derived type that will hold any additional information that should be sent to receivers of the event notification** I discussed this type and its usage earlier in this chapter. If the event has no special information to pass on, you can simply use the **System.EventArgs** type defined in the FCL instead of defining a type of your own. In my example, I've defined the type for demonstration purposes and therefore didn't define any fields within the type.

**c. Define a delegate type specifying the prototype of the method that will be called when the event fires** I discussed this delegate type and its usage earlier in this chapter. In my example, because the **FooEventArgs** type doesn't have any fields in it, I don't need to define a special delegate type. If your event doesn't have any special information that it needs to send to the listeners, you can also avoid defining your own delegate type and just use the **System.EventHandler** delegate type defined in the FCL.

**d. Explicitly define an event and its accessor methods** This is the interesting part of this technique. In this step, you explicitly define your own **add** and **remove** accessor methods for each event you want your type to expose. In my example, the **add** and **remove** methods call the **EventHandlerSet**'s **AddHandler** and **RemoveHandler** methods, respectively. **EventHandlerSet**'s **AddHandler** method scans the collection to see whether a key of **fooEventKey** already exists. If the key doesn't exist, it is added to the hash table and the key's value indicates the delegate that should be called. If the key already exists, the value (the new delegate) is combined to the already existing delegate's linked list. **RemoveHandler** does the opposite operation. (Again, I'll talk about the **EventHandlerSet** type later in this chapter.) In my example, all access to the **EventHandlerSet** collection is thread safe, so I don't specify the **MethodImplAttribute** attribute on the event's **add** and **remove** accessor methods.

**e. Define a protected, virtual method responsible for notifying registered objects of the event** The **OnFoo** method is called whenever the "Foo event" occurs to the object. In my example, this method calls the **EventHandlerSet**'s **Fire** method. This method takes the object whose hash code identifies the key to look up in the collection. The **Fire** method then scans the collection for the key; if it finds the key, it invokes the associated delegate linked list, passing **this** (the object firing the event) and an **EventArgs**–derived object identifying any additional information to the listeners. You'll see exactly how **EventHandlerSet**'s **Fire** method is implemented later.

**f. Define a method that translates the input into the desired event** I talked about this method and its usage earlier in the chapter. In my example, the **SimulateFoo** method is called to simulate that a "Foo event" has occurred. All the method does is fire the event, passing it any additional information about the event.

3. **Define the members necessary for an event the type wants to expose** This step is just a repeat of steps 2a through 2f for each additional event you want your type to expose. In my example, my type exposes two events, **Foo** and **Bar**. Steps 3a through 3f are for the **Bar** event.

## Designing the EventHandlerSet Type

The code shown earlier makes extensive use of an **EventHandlerSet** type. As I mentioned, this type isn't in the FCL; I defined it. Here's the source code for this type:

```
public class EventHandlerSet : IDisposable {

   // The private hash table collection used to maintain
   // the event key/delegate value pairs
   private Hashtable events = new Hashtable();


   // An index property that gets/sets the delegate associated
   // with the passed event object's hash code key
   public virtual Delegate this[Object eventKey] {
      // Returns null if object is not in set
      get { return (Delegate) events[eventKey]; }
      set { events[eventKey] = value; }
   }


   // Adds/Combines a delegate for the indicated event hash code key
   public virtual void AddHandler(Object eventKey, Delegate handler) {
      events[eventKey] =
         Delegate.Combine((Delegate) events[eventKey], handler);
   }

   // Removes a delegate for the indicated event hash code key
   public virtual void RemoveHandler(
      Object eventKey, Delegate handler) {

      events[eventKey] =
         Delegate.Remove((Delegate) events[eventKey], handler);
   }


   // Fires the delegate for the indicated event hash code key
   public virtual void Fire(Object eventKey,
      Object sender, EventArgs e) {

      Delegate d = (Delegate) events[eventKey];
      if (d != null) d.DynamicInvoke(new Object[] { sender, e });
   }


   // Disposing this object allows the memory used by the hash table
   // to be reclaimed at the next garbage collection, preventing
   // generation promotion.
   public void Dispose() { events = null; }


   // This static method returns a thread-safe wrapper around the
   // specified EventHandlerSet object.
   public static EventHandlerSet Synchronized(
      EventHandlerSet eventHandlerSet) {

      if (eventHandlerSet == null)
         throw new ArgumentNullException("eventHandlerSet");
      return new SynchronizedEventHandlerSet(eventHandlerSet);
   }
```

```
   // This class provides a thread-safe wrapper over a simple EventHandlerSet
   private class SynchronizedEventHandlerSet : EventHandlerSet {

      // The reference to the thread-unsafe object
      private EventHandlerSet eventHandlerSet;

      // Construct a thread-safe wrapper over the thread-unsafe object.
      public SynchronizedEventHandlerSet(
         EventHandlerSet eventHandlerSet) {
         this.eventHandlerSet = eventHandlerSet;
         Dispose();  // Let the base type's hash table object be freed.
      }

      // Thread-safe indexer property
      public override Delegate this[Object eventKey] {
         [MethodImpl(MethodImplOptions.Synchronized)]
         get { return eventHandlerSet[eventKey]; }

         [MethodImpl(MethodImplOptions.Synchronized)]
         set { eventHandlerSet[eventKey] = value; }
      }

      // Thread-safe AddHandler method
      [MethodImpl(MethodImplOptions.Synchronized)]
      public override void AddHandler(
         Object eventKey, Delegate handler) {
         eventHandlerSet.AddHandler(eventKey, handler);
      }


      // Thread-safe RemoveHandler method
      [MethodImpl(MethodImplOptions.Synchronized)]
      public override void RemoveHandler(
         Object eventKey, Delegate handler) {
         eventHandlerSet.RemoveHandler(eventKey, handler);
      }


      // Thread-safe Fire method
      [MethodImpl(MethodImplOptions.Synchronized)]
      public override void Fire(
         Object eventKey, Object sender, EventArgs e) {
         eventHandlerSet.Fire(eventKey, sender, e);
      }
   }
}
```

The implementation of the **EventHandlerSet** type is pretty straightforward; it's mostly just a bunch of manipulations on a hash table. However, there is one thing worth mentioning about the **Fire** method. This method is responsible for firing all the delegates in the chain, and it determines which delegate object to use from the **events** hash table. Because the hash table can contain several different delegate types, it is impossible to construct a type–safe call to the delegate at compile time. So, I call the **System.Delegate** type's **DynamicInvoke** method, passing it the callback method's parameters as an array of objects. Internally, **DynamicInvoke** will check the type safety of the parameters with the callback method being called and call the method. If there is a type mismatch, then **DynamicInvoke** will throw an exception.

**Note** The FCL defines a type, **System.ComponentModel.Event–HandlerList**, that does essentially the same thing as my **EventHandlerSet** type. The

**System.Windows.Forms.Control** and **System.Web.UI.Control** types use the **EventHandlerList** type internally to maintain their sparse set of events. You're certainly welcome to use the FCL's **EventHandlerList** type if you'd like. The difference between the **EventHandlerList** type and my **EventHandlerSet** type is that **EventHandlerList** uses a linked list instead of a hash table. This means that accessing elements managed by the **EventHandlerList** is slower than using my **EventHandlerSet**. In addition, the **EventHandlerList** doesn't offer any thread-safe way to access the collection; you would have to implement your own thread-safe wrapper around the **EventHandlerList** collection if you need this.

# Part IV: Essential Types

## Chapter List

*Chapter 12: Working with Text*
*Chapter 13: Enumerated Types and Bit Flags*
*Chapter 14: Arrays*
*Chapter 15: Interfaces*
*Chapter 16: Custom Attributes*
*Chapter 17: Delegates*

# Chapter 12: Working with Text

In this chapter, I'll explain the mechanics of working with individual characters and strings in the Microsoft .NET Framework. I'll start by talking about the **System.Char** structure and the various ways that you can manipulate a character. Then I'll go over the more useful **System.String** class, which allows you to work with immutable strings. (Once created, immutable strings can't be modified in any way.) After examining strings, I'll show you how to perform various operations efficiently to build a string dynamically via the **System.Text.StringBuilder** class. With the string basics out of the way, I'll then describe how to format objects into strings and how to efficiently persist or transmit strings using various encodings.

## Characters

In the .NET Framework, characters are always represented in 16-bit Unicode code values, easing the development of global applications. A character is represented with an instance of the **System.Char** structure (a value type). The **System.Char** type is pretty simple. It offers two public read-only constant fields: **MinValue**, defined as 0x0000, and **MaxValue**, defined as 0xFFFF.

Given an instance of a **Char**, you can call the static **GetUnicodeCategory** method, which returns a value of the **System.Globalization.UnicodeCategory** enumerated type. This value indicates whether the character is a control character, a currency symbol, a lowercase letter, an uppercase letter, a punctuation character, a math symbol, and so on (as defined by the Unicode 3.0 standard). To ease developing, the **Char** type also offers several static methods, such as **IsDigit**, **IsLetter**, **IsWhiteSpace**, **IsUpper**, **IsLower**, **IsPunctuation**, **IsLetterOrDigit**, **IsControl**, **IsNumber**, **IsSeparator**, **IsSurrogate**, and **IsSymbol**. All these methods call **GetUnicodeCategory** internally and simply return **true** or **false**. Note that all these methods take either a single character for a parameter or a **String** and the index of a character within the **String** as parameters.

In addition, you can convert a single character to its lowercase or uppercase equivalent by calling the static **ToLower** or **ToUpper** method. The call to one of these methods converts the character using the culture information associated with the calling thread (which the methods obtain internally by querying **System.Threading.Thread**'s static **CurrentCulture** property); or you can specify a particular culture by passing an instance of a **System.Globalization.CultureInfo** class to these methods. **ToLower** and **ToUpper** require culture information because letter casing is a culture-dependent operation. For example, Turkish considers the uppercase of U+0069 (LATIN SMALL LETTER I) to be U+0130 (LATIN CAPITAL LETTER I WITH DOT ABOVE), whereas other cultures consider the result to be U+0049 (LATIN CAPITAL LETTER I).

In addition to these static methods, the **Char** type also offers a few instance methods of its own. The **Equals** method returns **true** if two **Char** instances represent the same 16-bit Unicode code point. The **CompareTo** method (defined by the **IComparable** interface) returns a comparison of two code points; this comparison is not culture-sensitive. Chapter 15 explains how the **IComparable** interface and its **CompareTo** method work. The **ToString** method returns a **String** consisting of a single character. The opposite of **ToString** is **Parse**, which takes a single-character **String** and returns the character.

The last method, **GetNumericValue**, returns the numeric equivalent of a character. The following code demonstrates:

```
using System;

class App {
```

```
   static void Main() {
      Double d;

      // '\u0033' is the "digit 3"
      d = Char.GetNumericValue('\u0033'); // '3' would work too
      Console.WriteLine(d.ToString());    // Displays "3"

      // '\u00bc' is the "vulgar fraction one quarter ('ù')"
      d = Char.GetNumericValue('\u00bc');
      Console.WriteLine(d.ToString());    // Displays "0.25"

      // 'A' is the "Latin capital letter A"
      d = Char.GetNumericValue('A');
      Console.WriteLine(d.ToString());    // Displays "-1"
   }
}
```

Finally, three techniques allow you to convert between various numeric types to **Char** instances and vice versa. The techniques are listed here in order of preference:

- **Casting** The easiest way to convert a **Char** to a numeric value such as an **Int32** is simply by casting. Of the three techniques, this is the most efficient because the compiler emits intermediate language (IL) instructions to do the conversion and no methods have to be called. In addition, some languages (such as C#) allow you to indicate whether the conversion should be done using checked or unchecked code (discussed in Chapter 5). This lets you decide whether you'd like a **System.OverflowException** thrown when converting a value causes a loss of data. The only downside of this technique is that your compiler must treat the desired numeric type as a primitive. So, in Visual Basic, for example, you can't use this technique to convert a **Char** to a **UInt16** (or vice versa) because Visual Basic doesn't consider **UInt16** a primitive type.
- **Use the Convert type** The **System.Convert** type offers several static methods that know how to convert a **Char** to a numeric type and vice versa. All these methods perform the conversion as a checked operation, causing an **OverflowException** to be thrown should the conversion result in the loss of data.
- **Use the IConvertible interface** The **Char** type and all the numeric types in the .NET Framework Class Library (FCL) implement the **IConvertible** interface. This interface defines methods such as **ToUInt16** and **ToChar**. This technique performs least effectively of the three because calling an interface method on a value type requires that the instance be boxed—**Char** and all the numeric types are value types. **IConvertible**'s methods throw exceptions if the type can't be converted (such as converting a **Char** to a **Boolean**) or if the conversion results in a loss of data. Note that many types (including the FCL's **Char** and numeric types) implement **IConvertible**'s methods as explicit interface member implementations (described in Chapter 15). This means that you explicitly have to cast the instance to an **IConvertible** before you can call any of the interface's methods.

The following code demonstrates how to use these three techniques.

```
using System;

class App {
   static void Main() {
      Char  c;
      Int32 n;

      // Convert number <-> character using C# casting
      c = (Char) 65;
      Console.WriteLine(c);                  // Displays "A"
```

```
      n = (Int32) c;
      Console.WriteLine(n);                  // Displays "65"

      c = unchecked((Char) (65536 + 65));
      Console.WriteLine(c);                  // Displays "A"



      // Convert number <-> character using Convert
      c = Convert.ToChar(65);
      Console.WriteLine(c);                  // Displays "A"

      n = Convert.ToInt32(c);
      Console.WriteLine(n);                  // Displays "65"


      // This demonstrates Convert's range checking
      try {
         c = Convert.ToChar(70000);          // Too big for 16 bits
         Console.WriteLine(c);               // Doesn't execute
      }
      catch (OverflowException) {
         Console.WriteLine("Can't convert 70000 to a Char.");
      }


      // Convert number <-> character using IConvertible
      c = ((IConvertible) 65).ToChar(null);
      Console.WriteLine(c);                  // Displays "A"

      n = ((IConvertible) c).ToInt32(null);
      Console.WriteLine(n);                  // Displays "65"
   }
}
```

# The System.String Type

Certainly, one of the most used types in any application is **System.String**. A **String** represents an immutable ordered set of characters. The **String** type is derived immediately from **Object**, making it a reference type. (No string ever lives on a thread's stack.) The **String** type also implements several interfaces (**IComparable**, **ICloneable**, **IConvertible**, and **IEnumerable**).

## Constructing Strings

Many programming languages (including C#) consider **String** to be a primitive type—that is, the compiler lets you express literal strings directly in their source code. The compiler places these literal strings in the module's metadata, and they are accessed at run time using a mechanism called *string interning* (which I'll talk about later in this chapter).

In C#, you can't use the **new** operator to construct a **String** object:

```
using System;

class App {
   static void Main() {
      String s = new String("Hi there.");  // <- Error
   }
```

```
}
```

Instead, you must use the following special and simplified syntax:

```
using System;

class App {
   static void Main() {
      String s = "Hi there.";
   }
}
```

If you were to compile this code and examine its IL (using ILDasm.exe), you'd see the following:

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       7 (0x7)
  .maxstack  1
  .locals (string V_0)
  IL_0000:  ldstr      "Hi there."
  IL_0005:  stloc.0
  IL_0006:  ret
} // end of method App::Main
```

The IL instruction that constructs a new instance of an object is **newobj**. However, no **newobj** instruction appears in the IL code example. Instead, you see the special **ldstr** (load string) IL instruction that constructs a **String** object using a literal string obtained from metadata. This shows you that the CLR does, in fact, have a special way of constructing **String** objects.

In rare cases, you might need a **String** object that isn't constructed from a literal string contained in metadata. To accomplish this, you would use C#'s **new** operator and call one of the constructors provided by the **String** type. The constructors that take **Char*** or **SByte*** parameters were designed to be callable from code written using C++ with Managed Extensions. These constructors create a **String** object, initializing the string from an array of **Char** instances or signed bytes. The other constructors don't have any pointer parameters and can be called from any managed programming language.

C# offers some special syntax that helps you enter literal strings into the source code. For special characters, such as new lines, carriage returns, and backspaces, C# uses the escape mechanism familiar to C/C++ developers:

```
// String containing carriage-return and newline characters
String s = "Hi\r\nthere.";
```

**Important** Although the preceding example hard-codes carriage-return and newline characters into the string, I don't recommend this practice. Instead, the **System.Environment** type defines a read-only **NewLine** property that returns a string consisting of these characters when your application is running on Windows. However, the **NewLine** property is platform sensitive, and it returns the appropriate string required to obtain a newline by the underlying platform. So, for example, if the common language runtime (CLR) is ported to a UNIX system, the **NewLine** property would return a string consisting of just a single character "**n**". Here's the proper way to define the previous string so that it works correctly on any platform:

```
String s = "Hi" + Environment.NewLine + "there.";
```

You can concatenate several strings to form a single string using C#'s **+** operator as follows:

```
// Three literal strings concatenated to form a single literal string
String s = "Hi" + " " + "there.";
```

In this code, because all the strings are literal strings, the compiler concatenates them at compile time and ends up placing just one string, "Hi there.", in the module's metadata. Using the **+** operator on nonliteral strings performs the concatenation at run time. To concatenate several strings together at run time, don't use the **+** operator because it creates multiple string objects in the garbage collected heap. Instead, use the **System.Text.StringBuilder** type (which I'll explain later in this chapter).

Finally, C# also offers a special way to declare a string in which all characters between quotes are considered part of the string. These special declarations are called *verbatim strings* and are typically used when specifying the path of a file or directory or when working with regular expressions. Here's an example:

```
// Specifying the pathname of an application
String file = "C:\\Windows\\System32\\Notepad.exe";

// Specifying the pathname of an application using a verbatim string
String file = @"C:\Windows\System32\Notepad.exe";
```

These two code lines produce identical results. However, the @ symbol before the string on the second line tells the compiler that the string is a verbatim string. In effect, this tells the compiler to treat backslash characters as backslash characters instead of escape characters, making the path much more readable.

Now that you've seen how to construct a string, let's talk about some of the operations you can perform on **String** objects.

## Strings Are Immutable

The most important thing to know about a **String** object is that it is immutable; that is, once created, a string can never get longer, get shorter, or have any of its characters changed. Having immutable strings offers several benefits. First, it allows you to perform operations on a string without actually changing the string:

```
if (s.ToLower().SubString(10, 20).EndsWith("exe")) {
   Ã
}
```

Here, **ToLower** returns a new string; it doesn't modify the characters of the string **s**. **SubString** operates on the string returned by **ToLower** and also returns a new string, which is then examined by **EndsWith**. The two temporary strings created by **ToLower** and **SubString** are not referenced long by the application code, and the garbage collector will reclaim their memory at the next collection. These short–lived objects are very quick to collect, and it's usually not worth investing time in your algorithm in an attempt to improve performance.

Having immutable strings also means that there are no thread synchronization issues when manipulating a string. In addition, it's possible for two **String** references to refer to the same string object instead of different string objects if the strings are identical. This can reduce the number of strings in the system, conserving memory usage, and it is what string interning (discussed later in the chapter) is all about.

For performance reasons, the **String** type is tightly integrated with the CLR. Specifically, the CLR knows the exact layout of the fields defined within the **String** type, and the CLR accesses these fields directly. This performance and direct access come at a small development cost: the **String** class is sealed. If you were able to define your own type, using **String** as a base type, you could add your own fields, which would break the assumptions the CLR makes. In addition, you could break some assumptions that the CLR has made about **String** objects being immutable.

## Comparing Strings

Comparing is probably the most common operation performed on strings. Fortunately, the **String** type offers several static and instance methods that allow you to compare strings in very useful ways. Table 12–1 summarizes these methods.

Table 12–1: Methods for Comparing Strings

| Member | Member Type | Description |
|---|---|---|
| **Compare** | Static method | Returns how two strings should be sorted with respect to each other. Unlike the **CompareTo** method, this method gives you control over the culture used (via a **CultureInfo** object) and case sensitivity. If you need more advanced string comparison functionality, see the discussion of the **CompareInfo** class later in this section. |
| **CompareTo** | Instance method | Returns how two strings should be sorted with respect to each other. Note that this method always uses the **CultureInfo** object associated with the calling thread. |
| **StartsWith**/**EndsWith** | Instance method | Returns **true** if the string starts/ends with the specified string. The comparison is always case sensitive and uses the **CultureInfo** object associated with the calling thread. To perform the same operations with case insensitivity, use **CompareInfo**'s **IsPrefix** and **IsSuffix** methods. |
| **CompareOrdinal** | Static method | Returns **true** if two strings have the same set of characters. Unlike the other compare methods, this method simply compares the code values of the characters in the string; it doesn't do a culturally aware comparison, and the comparison is always case sensitive. This method is faster than the other compare methods, but it shouldn't be used for sorting strings that are destined to be shown to a user since the resulting set of strings might not be ordered the way a user would expect. For example, **CompareOrdinal** will sort "a" after "A" since the code value of "a" is a larger value than the code value for "A". |

| | | |
|---|---|---|
| **Equals** | Static and instance methods | Returns **true** if two strings have the same set of characters. Internally, **Equals** calls **String**'s static **CompareOrdinal** method. The static **Equals** method first checks whether the two references refer to the same object; if they do, it returns **true** instead of comparing the string's individual characters. Comparing references first improves performance greatly when using the interned string mechanism, which I'll describe shortly. |
| **GetHashCode** | Instance method | Returns a hash code for the string. |

In addition to these methods, the **String** type provides overloads of the **==** and **!=** operators. Internally, these operator methods are implemented by calling **String**'s static **Equals** method.

There are two reasons to compare two strings with each other. The first is to determine whether the two strings represent essentially the same string. The second is to sort the strings, usually for presentation to a user. To determine whether two strings are equal, you can use the **CompareOrdinal** method. This method is fast because it compares only the code values in the strings. However, some strings are logically equal even though they might not have the same code values.

To determine whether these two strings are equal, you should call the **Compare** method instead. Internally, this method uses culture-specific sorting tables (as defined by the Unicode 3.1 standard) that are part of the .NET Framework itself. Because these tables are included in the .NET Framework, all versions of the .NET Framework (regardless of underlying operating system platform) will compare and sort the strings in the same way. When comparing two strings for equality, you can have the **Compare** method perform a case-sensitive or case-insensitive comparison depending on your application's needs.

**Important** When comparing strings or converting strings to uppercase or lowercase, you should usually use **InvariantCulture**. For example, use **InvariantCulture** when comparing and converting pathnames and filenames, registry keys and values, reflection strings, XML tags and XML attribute names, and any other programmatic strings. In fact, you should use a noninvariant culture only when you're comparing and converting strings that are to be displayed to the user in a linguistically correct manner. And, of course, the culture you use should match the user's chosen language and optionally the user's country.

The following code demonstrates the difference between calling **Compare-Ordinal** and **Compare**:

```
using System;
using System.Globalization;

class App {
   static void Main() {
      String s1 = "Strass", s2 = "Stra§";
      Int32 x;

      // CompareOrdinal returns nonzero: strings have different values.
      x = String.CompareOrdinal(s1, s2);
      Console.WriteLine("CompareOrdinal: '{0}' {2} '{1}'", s1, s2,
         (x == 0) ? "equals" : "does not equal");
```

```
      // Compare returns zero: strings have the same value.
      x = new CultureInfo("de-DE").CompareInfo.Compare(s1, s2);
      Console.WriteLine("Compare: '{0}' {2} '{1}'", s1, s2,
         (x == 0) ? "equals" : "does not equal");
   }
}
```

Building and running this code produces the following output:

```
CompareOrdinal: 'Strass' does not equal 'Stra§'
Compare: 'Strass' equals 'Stra§'
```

You'd also use the **Compare** method to sort strings. However, you should always have **Compare** perform case-sensitive comparisons when comparing strings for sorting. The reason is that two strings that differ only by case are considered equal, and as you sort the strings in a list, the order of the "equal" strings could be different each time you run the application; this could confuse the user.

Internally, **String**'s **Compare** method obtains the **CurrentCulture** associated with the calling thread and then reads the **CurrentCulture**'s **CompareInfo** property. This property returns a reference to a **System.Globalization.CompareInfo** object. Because a **CompareInfo** object encapsulates a culture's character comparison tables, there's only one **CompareInfo** object per culture.

**String**'s **Compare** method simply determines the culture's **CompareInfo** object and then calls the object's **Compare** method. **String**'s **Compare** method allows you to specify a case-sensitive or case-insensitive comparison. However, **CompareInfo**'s **Compare** method exposes richer control over the comparison, which is required for some applications. These applications will have to obtain a reference to the desired culture's **CompareInfo** object and call its **Compare** method directly.

Specifically, some overloads of **CompareInfo**'s **Compare** method take a **CompareOptions** enumerated type, which defines the following symbols: **IgnoreCase**, **IgnoreKanaType**, **IgnoreNonSpace**, **IgnoreSymbols**, **IgnoreWidth**, **None**, **Ordinal**, and **StringSort**. For a complete description of these symbols, consult the .NET Framework documentation.

The following code demonstrates how culture plays an important part when sorting strings:

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Globalization;
using System.Threading;

class App {
   static void Main() {
      StringBuilder sb = new StringBuilder();
      String[] sign = new String[] { "<", "=", ">" };
      Int32 x;

      // The following code demonstrates how strings compare
      // differently for different cultures.
      String s1 = "cotÞ", s2 = "c™te";

      // Sorting strings for French in France
      x = new CultureInfo("fr-FR").CompareInfo.Compare(s1, s2);
      sb.AppendFormat("fr-FR Compare: {0} {2} {1}",
         s1, s2, sign[x + 1]);
      sb.Append(Environment.NewLine);
```

```
        // Sorting strings for Japanese in Japan
        x = new CultureInfo("ja-JP").CompareInfo.Compare(s1, s2);
        sb.AppendFormat("ja-JP Compare: {0} {2} {1}",
           s1, s2, sign[x + 1]);
        sb.Append(Environment.NewLine);

        // Sorting strings for the thread's culture
        x = Thread.CurrentThread.CurrentCulture.CompareInfo.
           Compare(s1, s2);
        sb.AppendFormat("{0} Compare: {1} {3} {2}",
           Thread.CurrentThread.CurrentCulture.Name,
           s1, s2, sign[x + 1]);
        sb.Append(Environment.NewLine);
        sb.Append(Environment.NewLine);

        // The following code demonstrates how to use
        // CompareInfo.Compare's advanced options with two Japanese
        // strings. These strings represent the word "shinkansen"
        // (the name for the Japanese high-speed train) in both
        // hiragana and katakana.
        s1 = "!?#?$?";  // ("\u3057\u3093\u304B\u3093\u305b\u3093")
        s2 = "%=>=<=";  // ("\uff7c\uff9d\uff76\uff9d\uff7e\uff9d")

        // Here's the result of a default comparison.
        x = String.Compare(s1, s2, true, new CultureInfo("ja-JP"));
        sb.AppendFormat("Simple ja-JP Compare: {0} {2} {1}",
           s1, s2, sign[x + 1]);
        sb.Append(Environment.NewLine);

        // Here's the result of a comparison that ignores Kana type.
        CompareInfo ci = CompareInfo.GetCompareInfo("ja-JP");
        x = ci.Compare(s1, s2, CompareOptions.IgnoreKanaType);
        sb.AppendFormat("Advanced ja-JP Compare: {0} {2} {1}",
           s1, s2, sign[x + 1]);

        MessageBox.Show(sb.ToString(), "StringSorting Results");
    }
}
```

Building and running this code produces the output shown in Figure 12–1.




Figure 12–1: StringSorting results

To see Japanese characters in the source code and in the message box, Windows must have the East Asian Language files installed (which use approximately 230 MB of disk space). To install these files, open the Regional And Language Options dialog box (shown in Figure 12–2) in Control Panel, select the Languages tab, check the Install Files For East Asian Languages check box, and press OK. This causes Windows to install the East Asian language fonts and Input Method Editor (IME) files.




Figure 12–2: Installing East Asian Language files using the Regional And Language Options Control Panel dialog box

Also, the source code file can't be saved in ANSI; I used UTF–8, which the Visual Studio .NET editor and Microsoft's C# compiler handle just fine.

In addition to **Compare**, the **CompareInfo** class offers **IndexOf**, **IsLastIndexOf**, **IsPrefix**, and **IsSuffix** methods. Because all these methods offer overloads that take a **CompareOptions** enumeration value as a parameter, they give you more control than the corresponding methods defined by the **String** class.

## String Interning

As I said in the preceding section, comparing strings is a common operation for many applications—it's also a task that can hurt performance significantly. The reason for the performance hit is that string comparisons require that each character in the string be checked, one by one, until two characters are determined to be different. To compare a string to see whether it contains the value "Hello", a loop must compare two characters five times. In addition, if you have several instances of the string "Hello" in memory, you're wasting memory because strings are immutable. You'll use memory much more efficiently if there is just one "Hello" string in memory and all references to the "Hello" string point to a single object.

If your application frequently compares strings for equality or if you expect to have many string objects with the same value, you can enhance performance substantially if you take advantage of the string interning mechanism in the CLR. To understand how the string interning mechanism works, examine the following code:

```
String s = "Hello";
Console.WriteLine(Object.ReferenceEquals("Hello", s));
```

Do you think this code displays "True" or "False"? Many people expect "False". After all, there are two "Hello" string objects and **ReferenceEquals** returns **true** only if the two references passed to it point to the same object. Build and run this code, however, and you'll see that "True" is displayed. Let's see why.

When the CLR initializes, it creates an internal hash table in which the keys are strings and the values are references to string objects in the managed heap. Initially, the table is empty (of course). When the JIT compiler compiles this method, it looks up each literal string in the hash table. The compiler looks for the first "Hello" string and because it doesn't find one, constructs a new **String** object in the managed heap (that refers to this string) and adds the "Hello" **String** and a reference to the object into the hash table. Then the JIT compiler looks up the second "Hello" string in the hash table. It finds this string, so nothing happens. Because there are no more literal strings in the code, the code can now execute.

When the code executes, it needs a reference to the "Hello" string. The CLR looks up "Hello" in the hash table, finds it, and returns a reference to the previously created **String** object. This reference is saved in the variable **s**. For the second line of code, the CLR again looks up "Hello" in the hash table and finds it. The reference to the same **String** object is passed along with **s** to **Object**'s static **ReferenceEquals** method, which returns **true**.

All literal strings embedded in the source code are always added to the internal hash table when a method referencing the strings is JIT compiled. But what about strings that are dynamically constructed at run time? What do you expect the following code to display?

```
String s1 = "Hello";
String s2 = "Hel";
String s3 = s2 + "lo";
Console.WriteLine(Object.ReferenceEquals(s1, s3));
Console.WriteLine(s1.Equals(s3));
```

In this code, the string referred to by **s2** ("Hel"), and the literal string, "lo", are concatenated. The result is a newly constructed string object, referred to by **s3**, that resides on the managed heap.

This dynamically created string does contain "Hello", but the string isn't added to the internal hash

table. Therefore, **ReferenceEquals** returns **false** because the two references point to different string objects. However, the call to **Equals** produces a result of **true** because the strings do, in fact, represent the same set of characters. Obviously, **ReferenceEquals** performs much better than **Equals**, and an application's performance is greatly improved if all the string comparisons simply compare references instead of characters. Plus, an application requires fewer objects in the heap if there's a way to collapse dynamic strings with the same set of characters down to single objects in the heap.

Fortunately, the **String** type offers two static methods that allow you to do this:

```
public static String Intern(String str);
public static String IsInterned(String str);
```

The first method, **Intern**, takes a **String** and looks it up in the internal hash table. If the string exists, a reference to the already existing **String** object is returned. If the application no longer holds a reference to the original **String** object, the garbage collector is able to free the memory of that string. The preceding code can now be rewritten using **Intern** as follows:

```
String s1 = "Hello";
String s2 = "Hel";
String s3 = s2 + "lo";
s3 = String.Intern(s3);
Console.WriteLine(Object.ReferenceEquals(s1, s3));
Console.WriteLine(s1.Equals(s3));
```

Now **ReferenceEquals** returns a value of **true** and the comparison is much faster. In addition, the **String** object that **s3** originally referred to is now free to be garbage collected. This code actually executes slower than the previous version because of the work that **String**'s **Intern** method must perform. You should intern strings only if you intend to compare a string multiple times in your application. Otherwise, you'll hurt performance instead of improve it.

Note that the garbage collector can't free the strings the internal hash table refers to because the hash table holds the reference to those **String** objects. **String** objects referred to by the internal hash table can't be freed until there are no AppDomains in the process that refer to the string object. Also note that string interning occurs on a per–process basis, meaning that a single string object can be accessed from multiple AppDomains, conserving memory usage. The capability of multiple AppDomains to access a single string also improves performance since strings never have to be marshaled across AppDomains within a single process; just the reference is marshaled.

As I mentioned earlier, the **String** type also offers a static **IsInterned** method. Like the **Intern** method, the **IsInterned** method takes a **String** and looks it up in the internal hash table. If the string is in the hash table, **IsInterned** returns a reference to the interned string object. If the string isn't in the hash table, however, **IsInterned** returns **null;** it doesn't add the string to the hash table.

The C# compiler uses the **IsInterned** method to allow **switch**/**case** statements to work efficiently on strings. For example, you can write the following C# code:

```
using System;

class App {
   static void Main() {
      Lookup("Jeff", "Richter");
      Lookup("Fred", "Flintstone");
   }
```

```
   static void Lookup(String firstName, String lastName) {
      switch (firstName + " " + lastName) {
         case "Jeff Richter":
            Console.WriteLine("Jeff");
            break;

         default:
            Console.WriteLine("Unknown");
            break;
      }
   }
}
```

I compiled this code and used ILDasm.exe to examine the IL, which follows. I've inserted comments to fully explain what's going on.

```
.method private hidebysig static void  Lookup(string firstName,
                                              string lastName) cil managed
{
  // Code size       53 (0x35)
  .maxstack  3
  .locals (object V_0)

  // Concatenate firstName, " ", and lastName into a new String.
  IL_0000:  ldarg.0
  IL_0001:  ldstr      " "
  IL_0006:  ldarg.1
  IL_0007:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)

  // Duplicate the reference to the concatenated string.
  IL_000c:  dup

  // Store a reference to the string in a temporary stack variable.
  IL_000d:  stloc.0

  // If Concat returns null, branch to IL_002a.
  IL_000e:  brfalse.s  IL_002a

  // See if the concatenated string is in the internal hash table.
  IL_0010:  ldloc.0
  IL_0011:  call string [mscorlib]System.String::IsInterned(string)

  // Overwrite the temporary variable with a reference to the interned
  // string. Note  that null indicates the string wasn't in the hash table.
  IL_0016:  stloc.0

  // Compare the reference of the interned 'switch' string with a
  // reference to the interned "Jeff Richter" string.
  IL_0017:  ldloc.0
  IL_0018:  ldstr      "Jeff Richter"

  // If references refer to different String objects, branch to IL_002a.
  IL_001d:  bne.un.s   IL_002a

  // The references do match; display "Jeff" to the console and return.
  IL_001f:  ldstr      "Jeff"
  IL_0024:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0029:  ret

  // Display "Unknown" to the console and return.
```

```
  IL_002a:  ldstr      "Unknown"
  IL_002f:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0034:  ret
} // end of method App::Lookup
```

The important thing to notice in this code is that the IL code calls **IsInterned**, passing the string specified in the **switch** statement. If **IsInterned** returns **null**, the string can't match any of the **case** strings, causing the **default** code to execute: "Unknown" is displayed to the user. However, if **IsInterned** sees that the **switch** string does exist in the internal hash table, it returns a reference to the hash table's **String** object. The address of the interned string is then compared with the addresses of the interned literal strings specified by each **case** statement. Comparing the addresses is much faster than comparing all the characters in each string, and the code determines very quickly which **case** statement to execute.

## String Pooling

When compiling source code, your compiler must process each literal string and emit the string into the managed module's metadata. If the same literal string appears several times in your source code, then emitting all these strings into the metadata will bloat the size of the resulting file.

To remove this bloat, many compilers (include the C# compiler) write the literal string into the module's metadata only once. All code that references the string will be modified to refer to the one string in the metadata. This ability of a compiler to merge multiple occurrences of a single string into a single instance can reduce the size of a module substantially. This process is nothing new—C/C++ compilers have been doing it for years. (Microsoft's C/C++ compiler calls this *string pooling*.) Even so, string pooling is another way to improve the performance of strings and just one more piece of knowledge you should have in your repertoire.

## Examining a String's Characters

Although comparing strings is useful for sorting them or for detecting equality, sometimes you just need to examine the characters within a string. The **String** type offers several methods to help you do this. Table 12–2 summarizes these methods.

Table 12–2: Methods for Examining String Characters

| Member | Member Type | Description |
|---|---|---|
| **Length** | Instance read–only property | Returns the number of characters in the string |
| **Chars** | Instance read–only indexer property | Returns the character at the specified index within the string |
| **GetEnumerator** | Instance method | Returns an **IEnumerator** that can be used to iterate over all the characters in the string |
| **ToCharArray** | Instance method | Returns a **Char[ ]** that contains a portion of the string |
| **IndexOf**/**LastIndexOf** | Instance methods | Returns the index of the first/last character/string matching a specified value |
| **IndexOfAny**/**LastIndexOfAny** | Instance methods | Returns the index of the first/last character matching an array of |

| | | |
|---|---|---|
| | | specified characters |

In reality, a **System.Char** represents a single 16–bit Unicode code value that doesn't necessarily equate to an abstract Unicode character. For example, some abstract Unicode characters are a combination of two code values. When combined, the U+0625 (Arabic letter Alef with Hamza below) and U+0650 (Arabic Kasra) characters form a single abstract character.

In addition, some Unicode abstract characters require more than a 16–bit value to represent them. These characters are represented using two 16–bit code values. The first code value is called the *high surrogate,* and the second code value is called the *low surrogate*. High surrogates have a value between U+D800 and U+DBFF, and low surrogates have a value between U+DC00 and U+DFFF. The use of surrogates allows Unicode to express more than a million different characters.

Surrogate characters are rarely used in the United States and Europe but are frequently used in East Asia. To properly work with abstract Unicode characters, you should use the **System.Globalization.StringInfo** type. You can call this type's static **GetTextElementEnumerator** method to acquire a **System.Globalization.TextElementEnumerator** object that allows you to enumerate through all the abstract Unicode characters contained in the string.

Alternatively, you could call **StringInfo**'s static **ParseCombiningCharacters** method to obtain an array of **Int32** values. The length of the array indicates how many abstract Unicode characters are contained in the string. Each element of the array identifies an index into the string where the first code value for the abstract Unicode character can be found.

The following code demonstrates how to properly use the **StringInfo**'s **GetTextElementEnumerator** and **ParseCombiningCharacters** methods to manipulate a string's abstract Unicode characters:

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Globalization;

class App {
   static void Main() {
      // The following string contains combining characters.
      String s = "a\u0304\u0308bc\u0327";
      EnumTextElements(s);
      EnumTextElementIndexes(s);
   }

   static void EnumTextElements(String s) {
      StringBuilder sb = new StringBuilder();

      TextElementEnumerator charEnum =
         StringInfo.GetTextElementEnumerator(s);
      while (charEnum.MoveNext()) {
         sb.AppendFormat(
            "Character at index {0} is '{1}'{2}",
            charEnum.ElementIndex, charEnum.GetTextElement(),
            Environment.NewLine);
      }
      MessageBox.Show(sb.ToString(),
         "Result of GetTextElementEnumerator");
   }

   static void EnumTextElementIndexes(String s) {
```

```
      StringBuilder sb = new StringBuilder();

      Int32[] textElemIndex = StringInfo.ParseCombiningCharacters(s);
      for (Int32 i = 0; i < textElemIndex.Length; i++) {
         sb.AppendFormat(
            "Character {0} starts at index {1}{2}",
            i, textElemIndex[i], Environment.NewLine);
      }
      MessageBox.Show(sb.ToString(),
         "Result of ParseCombiningCharacters");
   }
}
```

Building and running this code produces the message boxes shown in Figures 12–3 and 12–4.

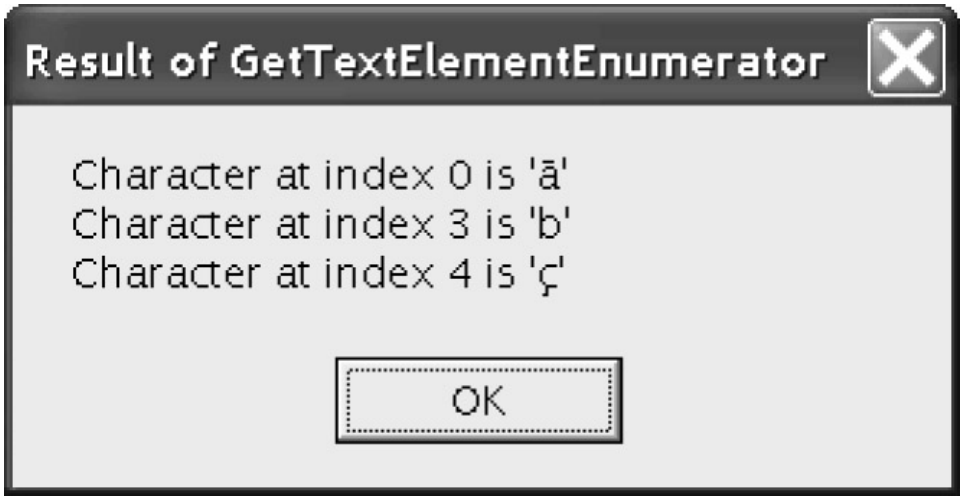



Figure 12–3: Result of GetTextElementEnumerator

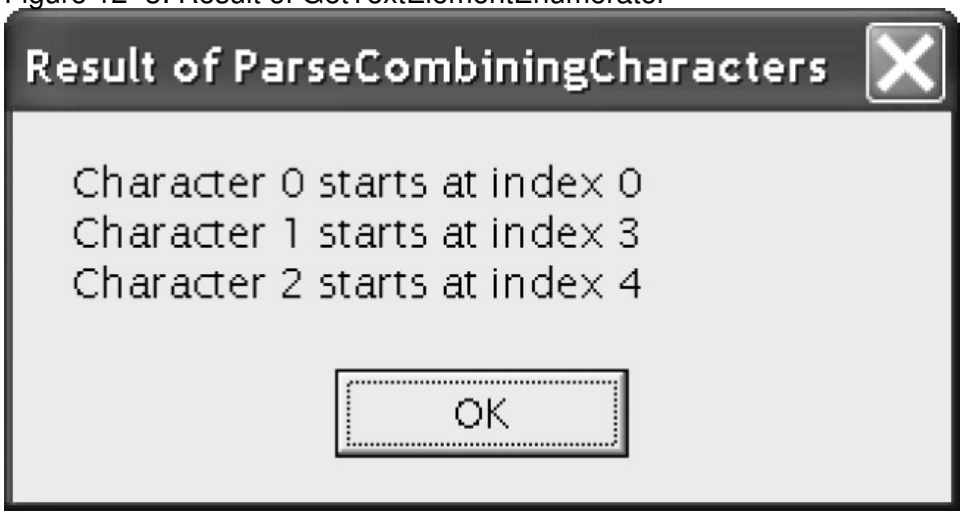



Figure 12–4: Result of ParseCombiningCharacters

**Note** To see the message box text in Figure 12–3 correctly, I had to open the Windows Display Properties dialog box and change the font used by message box text to Lucida Sans Unicode because this font contains glyphs for these combining characters. This is also why I don't have the code display the results to the console.

In this example, I'm calling **StringInfo**'s static **GetTextElementEnumerator** method. I pass a **String** to this method and it returns a **TextElementEnumerator** object. I can now use this enumerator object as I would any other enumerator object. The **TextElementEnumerator** object also offers a read–only **ElementIndex** property that returns the index of the code value in the original string where the character begins and a **GetTextElement** method that returns a string

consisting of all the code values necessary to make up the character.

In addition, **StringInfo** provides a static **ParseCombiningCharacters** method that parses a string and returns an array of **Int32**s. Each element in the returned array is an index of a code–point unit that is the start of an abstract character. The .NET Framework SDK documentation shows an example of how to call this method. Note that the **StringInfo** class defines a public constructor, but this is a bug; there's never a reason to construct an instance of a **StringInfo**.

## Other String Operations

The **String** type also offers methods that allow you to copy a string or parts of it. Table 12–3 summarizes these methods.

Table 12–3: Methods for Copying Strings

| Member | Type Method | Description |
|---|---|---|
| **Clone** | Instance | Returns a reference to the same object (**this**). This is OK because **String** objects are immutable. This method implements **String**'s **ICloneable** interface. |
| **Copy** | Static | Returns a new string that is a duplicate of the specified string. This method is rarely used and exists to help applications that treat strings as tokens. Normally, strings with the same set of characters are interned to a single string. This method creates a new string object so that the references (pointers) are different even though the strings contain the same characters. |
| **CopyTo** | Instance | Copies a portion of the string's characters to an array of characters. |
| **SubString** | Instance | Returns a new string representing a portion of the original string. |
| **ToString** | Instance | Returns a reference to the same object (**this**). |

In addition to these methods, **String** offers many static and instance methods that manipulate a string, such as **Insert**, **Remove**, **PadLeft**, **Replace**, **Split**, **Join**, **ToLower**, **ToUpper**, **Trim**, **Concat**, **Format**, and so on. Again, the important thing to remember about all these methods is that they return new string objects; because strings are immutable, once they're created, they can't be modified in any way.

# Dynamically Constructing a String Efficiently

Because the **String** type represents an immutable string, the FCL provides another type, **System.Text.StringBuilder**, which allows you to perform dynamic operations with strings and characters to create a **String**. Think of **StringBuilder** as a fancy constructor to create a **String** that can be used with the rest of the framework. In general, you should design methods that take **String** parameters, not **StringBuilder** parameters, unless you define a method that returns a string dynamically constructed by the method itself.

Internally, a **StringBuilder** object has a field that refers to an array of **Char** structures. **StringBuilder**'s members allow you to manipulate this character array, effectively shrinking the string or changing the characters in the string. If you grow the string past the allocated array of characters, the **StringBuilder** automatically allocates a new, larger array, copies the characters, and starts using the new array. The previous array is garbage collected.

When finished using the **StringBuilder** object to construct your string, "convert" the **StringBuilder**'s character array into a **String** simply by calling the **StringBuilder**'s **ToString** method. Internally, this method just returns a reference to the string field maintained inside the **StringBuilder**. This makes the **StringBuilder**'s **ToString** method very fast because the array of characters isn't copied.

The **String** returned from **StringBuilder**'s **ToString** method must be immutable. So if you ever call a method that attempts to modify the string field maintained by the **StringBuilder**, the **StringBuilder**'s methods know that **ToString** was called on it and they internally create and use a new character array, allowing you to perform manipulations without affecting the string returned by the previous call to **ToString**.

## Constructing a StringBuilder Object

Unlike with the **String** class, the CLR has no special knowledge of the **StringBuilder** class. In addition, most languages (including C#) don't consider the **StringBuilder** class to be a primitive type. You construct a **StringBuilder** object as you would any other nonprimitive type:

```
StringBuilder sb = new StringBuilder(...);
```

The **StringBuilder** type offers many constructors. The job of each constructor is to allocate and initialize the three internal fields maintained by each **StringBuilder** object:

- **Maximum capacity** An **Int32** field that specifies the maximum number of characters that can be placed in the string. The default is **Int32.MaxValue** (2 billion). It's unusual to change this value. However, you might specify a smaller maximum capacity to ensure that you never create a string over a certain length. Once constructed, a **StringBuilder**'s maximum capacity field can't be changed.
- **Capacity** An **Int32** field indicating the size of the character array field being maintained by the **StringBuilder**. The default is 16. If you have some idea how many characters you'll place in the **StringBuilder**, you should use this number to set the capacity when constructing the **StringBuilder** object.
- When appending characters to the character array, the **StringBuilder** detects whether the array is trying to grow beyond the array's capacity. If it is, the **StringBuilder** automatically doubles the capacity field, allocates a new array (the size of the new capacity), and copies the characters from the original array into the new array. The original array will be garbage collected in the future. Dynamically growing the array hurts performance; avoid this by setting a good initial capacity.

- **Character array** An array of **Char** structures that maintains the set of characters in the "string." The number of characters is always less than or equal to the capacity and maximum capacity fields. You can use the **StringBuilder**'s **Length** property to obtain the number of characters used in the array. The **Length** is always less than or equal to the **StringBuilder**'s capacity field. When constructing a **StringBuilder**, you can pass a **String** to initialize the character array. If you don't specify a string, the array initially contains no characters—that is, the **Length** property returns **0**.

## StringBuilder's Members

Unlike a **String**, a **StringBuilder** represents a mutable string. This means that most of **StringBuilder**'s members change the contents in the array of characters and don't cause new objects to be allocated in the managed heap. A **String−Builder** allocates a new object on only two occasions:

- You dynamically build a string that is longer than the capacity you've set.
- You attempt to modify the array after **StringBuilder**'s **ToString** method has been called.

You should also be aware that in the interest of faster performance, **StringBuilder**'s methods are not thread safe. This is usually fine since it's unusual for multiple threads to access a single **StringBuilder** object. If your application requires thread−safe manipulation of a **StringBuilder** object, you must explicitly add the thread synchronization code.

Table 12−4 summarizes **StringBuilder**'s members.

Table 12−4: StringBuilder's Members

| Member | Member Type | Description |
|---|---|---|
| **MaxCapacity** | Read−only property | Returns the largest number of characters that can be placed in the string. |
| **Capacity** | Read/write property | Gets or sets the size of the character array. Trying to set the capacity smaller than the string's length throws an **ArgumentOutOf−RangeException** exception. |
| **EnsureCapacity** | Method | Guarantees that the character array is at least the size specified to this method. If the value passed is larger than the **StringBuilder**'s current capacity, the current capacity gets bigger. If the current capacity is already bigger than the value passed to this method, no change occurs. |
| **Length** | Read−only property | Returns the number of characters in the "string." This will likely be smaller than the character array's current capacity. |
| **ToString** | Method | The parameterless version of this method returns a **String** representing the **StringBuilder**'s character array field. This method is efficient because it doesn't create a new **String** object. Any |

| | | |
|---|---|---|
| | | attempt to modify the **StringBuilder**'s array causes the **StringBuilder** to allocate and use a new array (initializing it from the old array). The version of **ToString** that takes **startIndex** and **length** parameters creates a new **String** object representing the desired portion of the **StringBuilder**'s string. |
| **Chars** | Read/write | Gets or sets the character at the specified ndexer index into the character array. In C#, this is an roperty indexer (parameterful property) that you access using array syntax (**[ ]**). |
| **AppendInsert** | Method | Appends or inserts a single object into the character array, growing the array if necessary. The object is converted to a string using the general format and the culture associated with the calling thread. |
| **AppendFormat** | Method | Appends the specified objects into the character array, growing the array if necessary. The objects are converted to strings using the formatting and culture information provided by the caller. **AppendFormat** is one of the most common methods used with **StringBuilder** objects. |
| **Replace** | Method | Replaces one character with another or one string with another from within the character array. |
| **Remove** | Method | Removes a range of characters from the character array. |
| **Equals** | Method | Returns **true** only if both **StringBuilder** objects have the same maximum capacity, capacity, and characters in the array. |

One important thing to note about **StringBuilder**'s methods is that most of them return a reference to the same **StringBuilder** object. This allows a convenient syntax to chain several operations together:

```
StringBuilder sb = new StringBuilder();
String s = sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").
   Replace(' ', '-').Remove(4, 3).ToString();
Console.WriteLine(s);  // "Jeff-Richter"
```

You'll notice that the **String** and **StringBuilder** classes don't have full method parity; that is, **String** has **ToLower**, **ToUpper**, **EndsWiths**, **PadLeft**, **Trim**, and so on. The **StringBuilder** class doesn't offer any of these methods. On the other hand, the **StringBuilder** class offers a richer **Replace** method that allows you to replace characters or strings in a portion of the string (not the whole string). It's unfortunate that there isn't complete parity between these two classes because now you must convert between **String** and **StringBuilder** to accomplish certain tasks. For example, to build up a string, convert all characters to uppercase, and then insert a string requires code like this:

```
// Construct a StringBuilder to do string manipulations.
StringBuilder sb = new StringBuilder();

// Perform some string manipulations using the StringBuilder.
sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").Replace(" ", "-");

// Convert the StringBuilder to a String in
// order to uppercase all the characters.
String s = sb.ToString().ToUpper();

// Clear the StringBuilder (allocates a new Char array).
sb.Length = 0;

// Load the uppercase String into the StringBuilder,
// and do more manipulations.
sb.Append(s).Insert(8, "Marc-");

// Convert the StringBuilder back to a String.
s = sb.ToString();

// Display the String to the user.
Console.WriteLine(s);  // "JEFFREY-Marc-RICHTER"
```

It's inconvenient to have to write this code just because **StringBuilder** doesn't offer all the operations that **String** does. In the future, I hope that Microsoft will add more string operation methods to **StringBuilder** to make it a more complete class.

## Obtaining a String Representation for an Object

You frequently need to obtain a string representation for an object. Usually this is necessary when you want to display a numeric type (such as **Byte**, **Int32**, **Single**, and so on) or a **DateTime** object to the user. Because the .NET Framework is an object–oriented platform, every type is responsible for providing code that converts an instance's "value" to a string equivalent. When designing how types should accomplish this, the designers of the FCL decided to devise a pattern that would be used consistently throughout. In this section, I'll describe this pattern.

You can obtain a string representation for any object by calling the **ToString** method. A public, parameterless **ToString** method is defined by **System.Object** and is therefore callable using an instance of any type. Semantically, **ToString** returns a string representing the object's current value, and this string should be formatted for the calling thread's current culture; that is, the string representation of a number should use the proper decimal separator, digit grouping symbol, and so on associated with the culture assigned to the calling thread.

**System.Object**'s implementation of **ToString** simply returns the full name of the object's type. This value isn't particularly useful but is a reasonable default for the many types that can't offer a sensible string. For example, what should a string representation of a **FileStream** or a **Hashtable** object look like?

All types that want to offer a reasonable way to obtain a string representing the current value of the object should override the **ToString** method. For all base types built into the FCL (**Byte**, **Int32**, **UInt64**, **Double**, and so on), Microsoft has already overridden these types' **ToString** method and implemented it to return a culturally aware string.

## Specific Formats and Cultures

The parameterless **ToString** method has two problems. First, the caller has no control over the formatting of the string. For example, an application might want to format a number into a currency string or a decimal string, or a percent string or a hexadecimal string. Second, the caller can't choose to format a string using a specific culture. This second problem is more troublesome for server–side application code than for client–side code. On rare occasions, an application needs to format a string using a culture different from the culture associated with the calling thread. To have more control over string formatting, you need a version of the **ToString** method that allows you to specify specific formatting and culture information.

Types that want to offer the caller a choice in formatting and culture implement the **System.IFormattable** interface:

```
public interface IFormattable {
   String ToString(String format, IFormatProvider formatProvider);
}
```

In the FCL, all the base types (**Byte**, **SByte**, **Int16**/**UInt16**, **Int32**/**UInt32**, **Int64**/**UInt64**, **Single**, **Double**, **Decimal**, and **DateTime**) implement this interface. In addition, some other types, like **GUID**, also implement it. Finally, every enumerated type automatically implements the **IFormattable** interface to obtain a meaningful string symbol from the numeric value stored in an instance of an enumerated type.

**IFormattable**'s **ToString** method takes two parameters. The first, **format**, is a string that tells the method how the object should be formatted. **ToString**'s second parameter, **formatProvider**, is an instance of a type that implements the **System.IFormatProvider** interface. This type supplies specific culture information to the **ToString** method. I'll discuss how shortly.

The type implementing the **IFormattable** interface's **ToString** method determines which format strings it's going to recognize. If you pass a format string that the type doesn't recognize, the type should throw a **System.FormatException** exception.

Many of the types that Microsoft has defined in the FCL recognize several formats. For example, the **DateTime** type supports "d" for short date, "D" for long date, "g" for general, "M" for month/day, "s" for sortable, "T" for time, "u" for universal time in ISO 8601 format, "U" for universal time in long date format, "Y" for year/month, and more. All enumerated types support "G" for general, "F" for flags, "D" decimal, and "X" for hexadecimal. I'll cover formatting enumerated types in more detail in Chapter 13.

Also, all the built–in numeric types support "C" for currency, "D" for decimal, "E" for scientific (exponential), "F" for fixed–point, "G" for general, "N" for number, "P" for percent, "R" for round–trip, and "X" for hexadecimal. In fact, the numeric types also support picture format strings just in case the simple format strings don't offer you exactly what you're looking for. Picture format strings contain special characters that tell the type's **ToString** method exactly how many digits to show, exactly where to place a decimal separator, exactly how many digits to place after the decimal separator, and so on. For complete information about format strings, see "Formatting Strings" in the .NET Framework SDK.

Calling **ToString** passing **null** for the format string is identical to calling **ToString** and passing "G" for the format string. In other words, objects format themselves using the "General format" by default. When implementing a type, choose a format that you think will be the most commonly used format; this format is the "General format." By the way, the **ToString** method that takes no

parameters assumes that the caller wants the general format.

So now that format strings are out of the way, let's turn to culture information. By default, strings are formatted using the culture information associated with the calling thread. The parameterless **ToString** method certainly does this, and so does **IFormattable**'s **ToString** if you pass **null** for the **formatProvider** parameter.

Culture–sensitive information applies when you're formatting numbers (including currency, integers, floating point, and percentages), dates, and times. A type that represents a GUID has a **ToString** method that just returns a string representing the GUID's value. There's no need to consider the thread's current culture when generating the GUID's string.

When formatting a number, the **ToString** method sees what you've passed for the **formatProvider** parameter. If **null** is passed, then **ToString** determines the culture associated with the calling thread by reading the **System.Threading.–Thread.CurrentThread.CurrentCulture** property. This property returns an instance of the **System.Globalization.CultureInfo** type.

Using this object, **ToString** reads its **NumberFormat** or **DateTimeFormat** property, depending on whether a number or date/time is being formatted. These properties return an instance of **System.Globalization.NumberFormatInfo** or **System.Globalization.DateTimeFormatInfo**, respectively. The **NumberFormatInfo** type defines a bunch of properties, such as **CurrencyDecimalSeparator**, **CurrencySymbol**, **NegativeSign**, **NumberGroupSeparator**, and **PercentSymbol**. Likewise, the **DateTimeFormatInfo** type defines an assortment of properties, such as **Calendar**, **DateSeparator**, **DayNames**, **LongDatePattern**, **ShortTimePattern**, and **TimeSeparator**. **ToString** reads these properties when constructing and formatting a string.

When calling **IFormattable**'s **ToString** method, instead of passing **null**, you can pass a reference to an object whose type implements the **IFormatProvider** interface:

```
public interface IFormatProvider {
   Object GetFormat(Type formatType);
}
```

Here's the basic idea behind the **IFormatProvider** interface: when a type implements this interface, it is saying that an instance of the type knows how to provide culture–specific formatting information and that the culture information associated with the calling thread should be ignored.

The **System.Globalization.CultureInfo** type is one of the very few types defined in the FCL that implements the **IFormatProvider** interface. If you want to format a string for, say, Vietnam, you'd construct a **CultureInfo** object and pass that object in as **ToString**'s **formatProvider** parameter. The following code obtains a string representation of a **Decimal** numeric value formatted as currency appropriate for Vietnam:

```
Decimal price = 123.54M;
String s = price.ToString("C", new CultureInfo("vi-VN"));
System.Windows.Forms.MessageBox.Show(s);
```

If you build and run this code, the message box shown in Figure 12–5 appears.



Figure 12–5: Numeric value formatted correctly to represent Vietnamese currency

Internally, **Decimal**'s **ToString** method sees that the **formatProvider** argument is not **null** and calls the object's **GetFormat** method as follows:

```
NumberFormatInfo nfi = (NumberFormatInfo)
   formatProvider.GetFormat(typeof(NumberFormatInfo));
```

This is how **ToString** asks the (**CultureInfo**) object for the appropriate number formatting information. Number types (like **Decimal**) ask for only number formatting information. But other types (like **DateTime**) could call **GetFormat** like this:

```
DateTimeFormatInfo dtfi = (DateTimeFormatInfo)
   formatProvider.GetFormat(typeof(DateTimeFormatInfo));
```

Actually, since **GetFormat**'s parameter can identify any type, the method is flexible enough to allow any type of format information to be requested. The types in version 1 of the .NET Framework call **GetFormat** asking only for number or date/time information; in the future, other kinds of formatting information could be requested.

By the way, if you want to obtain a string for an object that isn't formatted for any particular culture, you should call **System.Globalization.CultureInfo**'s static **InvariantCulture** property and pass the object returned as **ToString**'s **formatProvider** parameter:

```
Decimal price = 123.54M;
String s = price.ToString("C", CultureInfo.InvariantCulture);
System.Windows.Forms.MessageBox.Show(s);
```

If you build and run this code, the message box shown in Figure 12–6 appears. Notice the first character in the resulting string: Û. This is the international sign for currency (U+00A4).



Figure 12–6: Numeric value formatted to represent a culture–neutral currency

Normally, you wouldn't display a string formatted using the invariant culture to a user. Typically, you'd just save this string in a data file so that it could be parsed later. Basically, the invariant culture allows you to pass understandable strings from culture to culture.

In the FCL, just three types implement the **IFormatProvider** interface. The first is **CultureInfo**, which I've already explained. The other two are **NumberFormatInfo** and **DateTimeFormatInfo**. When **GetFormat** is called on a **NumberFormatInfo** object, the method checks whether the type being requested is a **NumberFormatInfo**. If it is, **this** is returned; if it's not, **null** is returned. Similarly, calling **GetFormat** on a **DateTimeFormatInfo** object returns **this** if a **DateTime–FormatInfo** is requested and **null** if it's not. These two types implement this interface simply as a programming convenience.

When trying to obtain a string representation of an object, the caller commonly specifies a format and uses the culture associated with the calling thread. For this reason, you often call **ToString**, passing a string for the format parameter and **null** for the **formatProvider** parameter. To make calling **ToString** easier for you, many types offer several overloads of the **ToString** method. For example, the **Decimal** type offers four different **ToString** methods:

```
// This version calls ToString(null, null).
// Meaning: General format, thread's culture information
String ToString();

// This version is where the actual implementation of ToString goes.
// This version implements IFormattable's ToString method.
// Meaning: Caller-specified format and culture information
String ToString(String format, IFormatProvider formatProvider);

// This version simply calls ToString(format, null).
// Meaning: Caller-specified format, thread's culture information
String ToString(String format);

// This version simply calls ToString(null, formatProvider).
// Meaning: General format, caller-specified culture information
String ToString(IFormatProvider formatProvider);
```

## Formatting Multiple Objects into a Single String

So far, I've explained how an individual type formats its own objects. At times, however, you want to construct strings that consist of many formatted objects. For example, the following string has a

date, a person's name, and an age:

```
String s = String.Format("On {0}, {1} is {2} years old.",
   DateTime.Now, "Wallace", 35);
Console.WriteLine(s);
```

If you build and run this code on January 23, 2002, at 4:37 P.M., you'll see the following line of output:

```
On 1/23/2002 4:37:37 PM, Wallace is 35 years old.
```

**String**'s static **Format** method takes a format string that identifies replaceable parameters using numbers in braces. The format string used in this example tells the **Format** method to replace "{0}" with the first parameter after the format string (the current date/time), replace "{1}" with the second parameter after the format string ("Wallace"), and replace "{2}" with the third parameter after the format string (35).

Internally, the **Format** method calls each object's **ToString** method to obtain a string representation for the object. Then the returned strings are all appended and the complete, final string is returned. This is all fine and good, but it means that all the objects are formatted using their general format and the calling thread's culture information.

You can have more control when formatting an object if you specify format information within braces. For example, the following code is identical to the previous example except that I've added formatting information to replaceable parameters 0 and 2:

```
String s = String.Format("On {0:D}, {1} is {2:E} years old.",
   DateTime.Now, "Wallace", 35);
Console.WriteLine(s);
```

If you build and run this code on January 23, 2002, at 4:37 P.M., you'll see the following line of output:

```
On Wednesday, January 23, 2002, Wallace is 3.500000E+001 years old.
```

When the **Format** method parses the format string, it sees that replaceable parameter 0 should have its **IFormattable** interface's **ToString** method passing "D" and **null** for its two parameters. Likewise, **Format** calls replaceable parameter 2's **IFormattable ToString** method, passing "E" and **null**. If the type doesn't implement the **IFormattable** interface, then **Format** calls its parameterless **ToString** method and the general format is appended into the resulting string.

The **String** class offers several overloads of the static **Format** method. One version takes an object that implements the **IFormatProvider** interface so that you can format all the replaceable parameters using caller–specified culture information. Obviously, **Format** calls each object's **ToString** method, passing it whatever **IFormatProvider** object was passed to **Format**.

If you're using **StringBuilder** instead of **String** to construct a string, you can call **StringBuilder**'s **AppendFormat** method. This method works exactly like **String**'s **Format** method except that it formats a string and appends to the **StringBuilder**'s character array. Like **String**'s **Format**, **AppendString** takes a format string, and there's a version that takes an **IFormatProvider**.

**System.Console** offers **Write** and **WriteLine** methods that also take format strings and replaceable parameters. However, there are no overloads of **Console**'s **Write** and **WriteLine** methods that allow you to pass an **IFormatProvider**. If you want to format a string for a specific culture, you have

to call **String**'s **Format** method, first passing the desired **IFormatProvider** object and then passing the resulting string to **Console**'s **Write** or **WriteLine** method. This shouldn't be a big deal since, as I said earlier, it's rare for client–side code to format a string using a culture other than the one associated with the calling thread.

## Providing Your Own Custom Formatter

By now it should be clear that the formatting capabilities in the .NET Framework were designed to offer you a great deal of flexibility and control. However, we're not quite done. It's possible for you to define a method that **StringBuilder**'s **AppendFormat** method will call whenever any object is being formatted into a string. In other words, instead of calling **ToString** for each object, **AppendFormat** can call a function that you define, allowing you to format any or all of the objects any way you want. What I'm about to describe works only when calling **StringBuilder**'s **AppendFormat** method. **String**'s **Format** method doesn't support this mechanism.

Let me explain this mechanism by way of an example. Let's say that you're formatting HTML text that a user will view in an Internet browser. You want all **Int32** values to display in bold. To accomplish this, every time an **Int32** value is formatted into a **String**, you want to surround the string with HTML bold tags: <**B**> and </**B**>**.** The following code demonstrates how easy it is to do this:

```
using System;
using System.Text;
using System.Globalization;
using System.Threading;


class BoldInt32s : IFormatProvider, ICustomFormatter {
   public Object GetFormat(Type formatType) {
      if (formatType == typeof(ICustomFormatter)) return this;
      return Thread.CurrentThread.CurrentCulture.GetFormat(formatType);
   }

   public String Format(String format, Object arg,
      IFormatProvider formatProvider) {

      String s;
      IFormattable formattable = arg as IFormattable;

      if (formattable == null) s = arg.ToString();
      else s = formattable.ToString(format, formatProvider);

      if (arg.GetType() == typeof(Int32))
         return "<B>" + s + "</B>";
      return s;
   }
}


class App {
   static void Main() {

      StringBuilder sb = new StringBuilder();
      sb.AppendFormat(new BoldInt32s(), "{0} {1} {2:M}",
         "Jeff", 123, DateTime.Now);
      Console.WriteLine(sb);
   }
}
```

When you compile and run this code, it displays the following output:

```
Jeff <B>123</B> January 23
```

In **Main**, I'm constructing an empty **StringBuilder** and then appending a formatted string into it. When I call **AppendFormat**, the first parameter is an instance of the **BoldInt32s** class. This class implements the **IFormatProvider** interface that I discussed earlier. In addition, this class implements the **ICustomFormatter** interface:

```
public interface ICustomFormatter {
   String Format(String format, Object arg,
      IFormatProvider formatProvider);
}
```

This interface's **Format** method is called whenever **StringBuilder**'s **AppendFormat** needs to obtain a string for an object. You can do some pretty clever things inside this method that give you a great deal of control over string formatting. Let's look inside the **AppendFormat** method to see exactly how it works. The following pseudocode shows how **AppendFormat** does its stuff:

```
public StringBuilder AppendFormat(IFormatProvider formatProvider,
   String format, params Object[] args) {

   // If an IFormatProvider was passed, find out
   // whether it offers an ICustomFormatter object.
   ICustomFormatter cf = null;
   if (formatProvider != null)
      cf = (ICustomFormatter)
         formatProvider.GetFormat(typeof(ICustomFormatter));

   // Keep appending literal characters (not shown in this pseudocode)
   // and replaceable parameters to the StringBuilder's character array.
   while (MoreReplaceableArgumentsToAppend) {
      String argStr;   // Formatted argument string to append

      // If a custom formatter is available, let it format the argument.
      if (cf != null)
         argStr = cf.Format(argFormat, argObj, formatProvider);

      // If there is no custom formatter or if it didn't format
      // the argument, try something else.
      if (argStr == null) {

         // Does the argument's type support rich formatting?
         IFormattable formattable = arg as IFormattable;
         if (formattable != null) {
            // Yes; pass the format string and provider to
            // the type's IFormattable ToString method.
            argStr = formattable.ToString(argFormat, formatProvider);
         else {
            // No; get the general format using
            // the thread's culture information.
            if (arg != null) argStr = arg.ToString();
            else s = String.Empty;
         }
      }

      // Append argStr's characters to the character array field member.
      ...
   }
   return this;
```

```
}
```

When **Main** calls **AppendFormat**, **AppendFormat** calls my format provider's **GetFormat** method, passing it the **ICustomFormatter** type. The **GetFormat** method defined in my **BoldInt32s** type sees that the **ICustomFormatter** is being requested and returns a reference to its own object. If any other type is requested of **GetFormat**, I call the **GetFormat** method using the **CultureInfo** object associated with the calling thread.

Whenever **AppendFormat** needs to format a replaceable parameter, it calls **ICustomFormatter**'s **Format** method. In my example, this calls the **Format** method defined by my **BoldInt32s** type. In my **Format** method, I check whether the object being formatted supports rich formatting via the **IFormattable** interface. If the object doesn't, then I call the simple, parameterless **ToString** method to format the object. If the object doesn't support **IFormattable**, then I call the rich **ToString** method, passing it the format string and the format provider.

Now that I have the formatted string, I check whether the object is an **Int32** type, and if it is, I wrap the formatting string in <**B**> and </**B**> HTML tags and return the new string. If the object is not an **Int32**, then I simply return the formatted string without any further processing.

## Parsing a String to Obtain an Object

In the preceding section, I explained how to take an object and obtain a string representation of that object. In this section, I'll talk about the opposite: how to take a string and obtain an object representation of it. Obtaining an object from a string isn't a very common operation, but it does occasionally come in handy. Microsoft felt it necessary to formalize a mechanism by which strings can be parsed into objects.

Any type that can parse a string offers a public, static method called **Parse**. This method takes a **String** and returns an instance of the type; in a way, **Parse** acts like a constructor. In the FCL, a **Parse** method exists on all the numeric types as well as for **DateTime**, **TimeSpan**, and a few other types (like the SQL data types).

Let's look at how to parse a string into a number type. All the numeric types (**Byte**, **SByte**, **Int16**/**UInt16**, **Int32**/**UInt32**, **Int64**/**UInt64**, **Single**, **Double**, and **Decimal**) offer at least one **Parse** method. Here I'll show you just the **Parse** method defined by the **Int32** type. (The **Parse** methods for the other numeric types are identical.)

```
public static Int32 Parse(String s, NumberStyles style,
   IFormatProvider provider);
```

Just from looking at the prototype, you should be able to guess exactly how this method works. The **String** parameter, **s**, identifies a string representation of a number you want parsed into an **Int32** object. The **System.Globalization.**–
**NumberStyles** parameter, **style**, is a set of bit flags that identify characters that **Parse** should expect to find in the string. And the **IFormatProvider** parameter, **provider**, identifies an object that the **Parse** method can use to obtain culture–specific information as discussed earlier in this chapter.

For example, the following code causes **Parse** to throw a **System.Format**–
**Exception** exception because the string being parsed contains a leading space:

```
Int32 x = Int32.Parse(" 123", NumberStyles.None, null);
```

To allow **Parse** to skip over the leading space, change the **style** parameter as follows:

```
Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);
```

Table 12–5 shows the bit symbols that the **NumberStyles** type defines.

Table 12–5: Bit Symbols Defined by the NumberStyles Type

| Symbol | Value | Description |
|---|---|---|
| **None** | 0x00000000 | None of the special characters represented by any of the bits in the remaining rows in this table are allowed in the string. |
| **AllowLeadingWhite** | 0x00000001 | The string can contain leading/trailing white–space characters (identified by the following Unicode code points: 0x0009, 0x000A, 0x000B, 0x000C, 0x000D, and 0x0020). |
| **AllowTrailingWhite** | 0x00000002 | |
| **AllowLeadingSign** | 0x00000004 | The string can contain a valid leading/trailing sign character. **NumberFormatInfo**'s **PositiveSign** and **NegativeSign** properties determine valid leading–sign characters. |
| **AllowTrailingSign** | 0x00000008 | |
| **AllowParentheses** | 0x00000010 | The string can contain parentheses. |
| **AllowDecimalPoint** | 0x00000020 | The string can contain a valid decimal–separator character. **NumberFormatInfo**'s **NumberDecimalSeparator** and **CurrencyDecimalSeparator** properties determine valid decimal–separator characters. |
| **AllowThousands** | 0x00000040 | The string can contain a valid grouping–separator character. **NumberFormatInfo**'s **NumberGroupSeparator** and **CurrencyGroupSeparator** properties determine valid grouping–separator characters. **NumberFormatInfo**'s **NumberGroupSizes** and **CurrencyGroupSizes** properties determine the number of digits in the group. |
| **AllowExponent** | 0x00000080 | The string can contain a number expressed in exponent format:<br>{e\|E} [{+\|–}] *n*<br>where *n* is a number. |
| **AllowCurrencySymbol** | 0x00000100 | The string can contain a valid currency symbol, which **NumberFormatInfo**'s **CurrencySymbol** property determines. |
| **AllowHexSpecifier** | 0x00000200 | The string can contain hex digits (0Ð9, AÐF), and the string is considered to be a hex value. |

In addition to the bit symbols in Table 12–5, the **NumberStyles** enumerated type also defines some symbols that represent common combinations of the individual bits. Table 12–6 shows these.

Table 12–6: Symbols for NumberStyles's Bit Combinations

| Symbol | Bit Set |
|---|---|
| Integer | AllowLeadingWhite \| AllowTrailingWhite \| AllowLeadingSign |
| Number | AllowLeadingWhite \| AllowTrailingWhite \| AllowLeadingSign \| AllowTrailingSign \| AllowDecimalPoint \| AllowThousands |
| Float | AllowLeadingWhite \| AllowTrailingWhite \| AllowLeadingSign \| AllowDecimalPoint \| AllowExponent |
| Currency | AllowLeadingWhite \| AllowTrailingWhite \| AllowLeadingSign \| AllowTrailingSign \| AllowParentheses \| AllowDecimalPoint \| AllowThousands \| AllowCurrencySymbol |
| HexNumber | AllowLeadingWhite \| AllowTrailingWhite \| AllowHexSpecifier |
| Any | AllowLeadingWhite \| AllowTrailingWhite \| AllowLeadingSign \| AllowTrailingSign \| AllowParentheses \| AllowDecimalPoint \| AllowThousands \| AllowCurrencySymbol \| AllowExponent |

Here's a code fragment showing how to parse a hexadecimal number:

```
Int32 x = Int32.Parse("1A", NumberStyles.HexNumber, null);
Console.WriteLine(x);  // Displays "26"
```

This **Parse** method accepts three parameters. For convenience, many types offer additional overloads of **Parse** so that you don't have to pass as many arguments. For example, **Int32** offers four overloads of the **Parse** method:

```
// Passes NumberStyles.Integer for style
// and null for provider parameters.
public static Int32 Parse(String s);

// Passes null for the provider parameter.
public static Int32 Parse(String s, NumberStyles style);

// Passes NumberStyles.Integer for the style parameter.
public static Int32 Parse(String s, IFormatProvider provider) {
```

```
// This is the method I've been talking about in this section.
public static int Parse(String s, NumberStyles style,
   IFormatProvider provider);
```

The **DateTime** type also offers a **Parse** method:

```
public static DateTime Parse(String s,
   IFormatProvider provider, DateTimeStyles styles);
```

This method works just like the **Parse** method defined on the number types except that **DateTime**'s **Parse** method takes a set of bit flags defined by the **DateTimeStyles** enumerated type instead of the **NumberStyles** enumerated types. Table 12–7 shows the bit symbols that the **DateTimeStyles** type defines.

Table 12–7: Bit Symbols Defined by the DateTimeStyles Type

| Symbol | Value | Description |
|---|---|---|
| **None** | 0x00000000 | None of the special characters represented by the bits in the remaining rows in this table are allowed in the string. |
| **AllowLeadingWhite** | 0x00000001 | The string can contain leading/trailing/inner white–space characters(identified by the following Unicode code points: 0x0009, 0x000A, 0x000B, 0x000C, 0x000D, and 0x0020). |
| **AllowTrailingWhite** | 0x00000002 | |
| **AllowInnerWhite** | 0x00000004 | |
| **NoCurrentDateDefault** | 0x00000008 | When parsing a string that contains only a time (no date), set the date to January 1, 0001 instead of the current date. |
| **AdjustToUniversal** | 0x00000010 | When parsing a string that contains a time–zone specifier ("GMT", "Z", "+xxxx", " xxxx"), adjust the parsed time base to Greenwich Mean Time. |

In addition to these bit symbols, the **DateTimeStyles** enumerated type also defines an **AllowWhiteSpaces** symbol that represents all the **White** symbols OR'd together (**AllowLeadingWhite** | **AllowInnerWhite** | **AllowTrailingWhite**).

For convenience, the **DateTime** type offers three overloads of the **Parse** method:

```
// null for formatProvider and DateTimeStyles.None
public static DateTime Parse(String s);
```

```
// DateTimeStyles.None
public static DateTime Parse(String s, IFormatProvider provider);

// This is the method I've been talking about in this section.
public static DateTime Parse(String s,
   IFormatProvider provider, DateTimeStyles styles);
```

Parsing dates and times is complex. Many developers have found **DateTime**'s **Parse** method too forgiving in that it sometimes parses strings that don't contain dates or times. For this reason, the **DateTime** type also offers a **ParseExact** method that accepts a picture format string that indicates exactly how the date/time string is formatted and how it should be parsed. For more information about picture format strings, see the **DateTimeFormatInfo** class in the .NET Framework SDK.

## Encodings: Converting Between Characters and Bytes

In Win32, programmers all too frequently have to write code to convert Unicode characters and strings to Multi–Byte Character Set (MBCS) characters and strings. I've certainly written my share of this code, and it's very tedious to write and error prone to use. In the CLR, all characters are represented as 16–bit Unicode code values and all strings are composed of 16–bit Unicode code values. This makes working with characters and strings easy at run time.

At times, however, you want to save strings to a file or transmit them over a network. If the strings consist mostly of characters readable by English–speaking people, then saving or transmitting a set of 16–bit values isn't very efficient because half of the bytes written would contain zeros. Instead, it would be more efficient to *encode* the 16–bit values into a compressed array of bytes and then *decode* the array of bytes back into an array of 16–bit values.

Encodings also allow a managed application to interact with strings created by non–Unicode systems. For example, if you want to produce a file readable by an application running on a Japanese version of Windows 95, you have to save the Unicode text using the Shift–JIS (code page 932) encoding. Likewise, you'd use a Shift–JIS encoding to read a text file produced on a Japanese Windows 95 system into the CLR.

Encoding is typically done when you want to send a string to a file or network stream using the **System.IO.BinaryWriter** or **System.IO.StreamWriter** type. Decoding is typically done when you want to read a string from a file or network stream using the **System.IO.BinaryReader** or **System.IO.StreamReader** type. If you don't explicitly select an encoding, all these types default to using UTF–8. (UTF stands for Unicode Transformation Format.) However, at times, you might want to encode or decode a string.

Fortunately, the FCL offers some types to make character encoding and decoding easy. The two most frequently used encodings are UTF–16 and UTF–8.

- UTF–16 encodes each 16–bit character as 2 bytes. It doesn't affect the characters at all, and no compression occurs—its performance is excellent. UTF–16 encoding is also referred to as *Unicode encoding*. Also note that UTF–16 can be used to convert from little endian to big endian and vice versa.
- UTF–8 encodes some characters as 1 byte, some characters as 2 bytes, some characters as 3 bytes, and some characters as 4 bytes. Characters with a value below 0x0080 are compressed to 1 byte, which works very well for characters used in the United States. Characters between 0x0080 and 0x07FF are converted to 2 bytes, which works well for

European and Middle Eastern languages. Characters of 0x0800 and above are converted to 3 bytes, which works well for East Asian languages. Finally, surrogate character pairs are written out as 4 bytes. UTF−8 is an extremely popular encoding, but it's less useful than UTF−16 if you encode many characters with values of 0x0800 or above.

Although the UTF−16 and UTF−8 encodings are by far the most common, the FCL also supports some encodings that are used less frequently:

- UTF−7 encoding is typically used with older systems that work with characters that can be expressed using 7−bit values. You should avoid this encoding because it usually ends up expanding the data rather than compressing it. The Unicode Consortium has deprecated this encoding starting with the Unicode 3.0 standard.
- ASCII encodes the 16−bit characters into ASCII characters; that is, any 16−bit character with a value less than 0x0080 is converted to a single byte. Any character with a value greater than 0x007F can't be converted, and the character's value is lost. For strings consisting of characters in the ASCII range (0x00 to 0x7F), this encoding compresses the data in half and is very fast (because the high byte is just chopped off). This encoding isn't good if you have characters outside the ASCII range because the character's values are lost.

Finally, the FCL also allows you to encode 16−bit characters to an arbitrary code page. Like the ASCII encoding, encoding to a code page is dangerous because any character whose value can't be expressed in the specified code page is lost. You should always use UTF−16 or UTF−8 encoding unless you must work with some legacy files or applications that already use one of the other encodings.

When you need to encode or decode a set of characters, you should obtain an instance of a class derived from **System.Text.Encoding**. **Encoding** is an abstract base class that offers several static properties, each of which returns an instance of an **Encoding**−derived class. (Each encoding class is basically a wrapper around the **WideCharToMultiByte** and **MultiByteToWideChar** Win32 functions that you might be familiar with.)

Here's an example that encodes and decodes characters using UTF−8:

```
using System;
using System.Text;

class App {
   static void Main() {
      // This is the string I'm going to encode.
      String s = "Hi there.";

      // Obtain an Encoding-derived object that knows how
      // to encode/decode using UTF-8.
      Encoding encodingUTF8 = System.Text.Encoding.UTF8;

      // Encode a string into an array of bytes.
      Byte[] encodedBytes = encodingUTF8.GetBytes(s);

      // Show the encoded byte values.
      Console.WriteLine("Encoded bytes: " +
         BitConverter.ToString(encodedBytes));

      // Decode the byte array back to a string.
      String decodedString = encodingUTF8.GetString(encodedBytes);

      // Show the decoded string.
      Console.WriteLine("Decoded string: " + decodedString);
```

```
    }
}
```

This code yields the following output:

```
Encoded bytes: 48-69-20-74-68-65-72-65-2E
Decoded string: Hi there.
```

In addition to the **UTF8** static property, the **Encoding** class also offers the following static properties: **Unicode**, **BigEndianUnicode**, **UTF7**, **ASCII**, and **Default**. The **Default** property returns an object that knows how to encode/decode using the user's code page as specified using the Regional and Language Options Control Panel Applet in Windows. (See the **GetACP** Win32 function for more information.) However, using the **Default** property is discouraged.

In addition to these properties, **Encoding** also offers a static **GetEncoding** method that allows you to specify a code page (by integer or by string) and returns an object that can encode/decode using the specified code page. You can call **GetEncoding** passing "Shift–JIS" or 932, for example.

When you first request an encoding object, the **Encoding** class's property or **GetEncoding** method constructs a single object for the requested encoding and returns this object. If an already requested encoding object is requested in the future, the encoding class simply returns the object it previously constructed; it doesn't construct a new object for each request. This efficiency reduces the number of objects in the system and puts less pressure in the garbage–collected heap.

Instead of calling one of **Encoding**'s static properties or its **GetEncoding** method, you could also construct an instance of one of the following classes: **System.Text.UnicodeEncoding**, **System.Text.UTF8Encoding**, **System.Text.–UTF7Encoding**, or **System.Text.ASCIIEncoding**. However, keep in mind that constructing any of these classes creates new objects in the managed heap, which hurts performance.

Three of these classes, **UnicodeEncoding**, **UTF8Encoding**, and **UTF7Encoding**, offer multiple constructors allowing you more control over the encoding and byte order marks (BOMs). You might want to explicitly construct instances of these encoding types when working with a **BinaryWriter** or a **StreamWriter**. The **ASCIIEncoding** class has only a single constructor and therefore doesn't offer any more control over the encoding. If you need an **ASCIIEncoding** object, always obtain it by querying **Encoding**'s **ASCII** property; never construct an instance of the **ASCIIEncoding** class yourself.

Once you have an **Encoding**–derived object, you can convert an array of characters to an array of bytes by calling the **GetBytes** method. (Several overloads of this method exist.) To convert an array of bytes to an array of characters, call the **GetChars** method or the more useful **GetString** method. (Several overloads exist for both these methods.) The preceding code demonstrated calls to the **GetBytes** and **GetString** methods.

Although not that useful, all **Encoding**–derived types offer a **GetByteCount** method that obtains the number of bytes necessary to encode a set of characters without actually encoding. You could use this method to allocate an array of bytes. There's also a **GetCharCount** method that returns the number of characters that would be decoded without actually decoding. These methods are useful if you're trying to save memory and reuse an array.

The **GetByteCount**/**GetCharCount** methods aren't that fast because they must analyze the array of characters/bytes in order to return an accurate result. If you prefer speed to an exact result, you can call the **GetMaxByteCount** or **GetMaxChar–Count** method instead. Both methods take an

integer specifying the number of characters or number of bytes and return a worst–case value.

Each **Encoding**–derived object offers a set of public read–only properties that you can query to obtain detailed information about the encoding. Table 12–8 briefly describes these properties.

Table 12–8: Properties of Encoding–Derived Classes

| Property | Type | Description |
|---|---|---|
| **EncodingName** | **String** | Returns the encoding's human–readable name. |
| **CodePage** | **Int32** | Returns the encoding's code page. |
| **WindowsCodePage** | **Int32** | Returns the encoding's closest Windows code page. |
| **WebName** | **String** | Returns the IANA–registered name. (IANA stands for Internet Assigned Numbers Authority.) For more information, go to *http://www.iana.org*. |
| **HeaderName** | **String** | Returns mail agent header tag. |
| **BodyName** | **String** | Returns mail agent body tag. |
| **IsBrowserDisplay** | **Boolean** | Returns **true** if browser clients can use encoding for display purposes. |
| **IsBrowserSave** | **Boolean** | Returns **true** if browser clients can use encoding for saving purposes. |
| **IsMailNewsDisplay** | **Boolean** | Returns **true** if mail and news clients can use encoding for display purposes. |
| **IsMailNewsSave** | **Boolean** | Returns **true** if mail and news clients can use encoding for saving purposes. |

To illustrate the properties and their meanings, I wrote the following program that displays these properties for several different encodings:

```
using System;
using System.Text;

class App {
   static void Main() {
      Show(Encoding.Unicode);
      Show(Encoding.BigEndianUnicode);
      Show(Encoding.UTF8);
      Show(Encoding.UTF7);
      Show(Encoding.ASCII);
      Show(Encoding.Default);
      Show(Encoding.GetEncoding(0));   // Same as Default
      Console.WriteLine();
      Console.WriteLine("Below are some specific code pages:");
      Show(Encoding.GetEncoding(437));
      Show(Encoding.GetEncoding(28595));
      Show(Encoding.GetEncoding(57008));
      Show(Encoding.GetEncoding(54936));
      Show(Encoding.GetEncoding(874));
```

```
   }


   static void Show(Encoding e) {
      Console.WriteLine(
         "{1}{0}" +
         "\tCodePage={2}, WindowsCodePage={3}{0}" +
         "\tWebName={4}, HeaderName={5}, BodyName={6}{0}" +
         "\tIsBrowserDisplay={7}, IsBrowserSave={8}{0}" +
         "\tIsMailNewsDisplay={9}, IsMailNewsSave={10}{0}",
         Environment.NewLine,
         e.EncodingName,
         e.CodePage, e.WindowsCodePage,
         e.WebName, e.HeaderName, e.BodyName,
         e.IsBrowserDisplay, e.IsBrowserSave,
         e.IsMailNewsDisplay, e.IsMailNewsSave);
   }
}
```

Running this program yields the following output:

```
Unicode
    CodePage=1200, WindowsCodePage=1200
    WebName=utf-16, HeaderName=utf-16, BodyName=utf-16
    IsBrowserDisplay=False, IsBrowserSave=True
    IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode (Big-Endian)
    CodePage=1201, WindowsCodePage=1200
    WebName=unicodeFFFE, HeaderName=unicodeFFFE, BodyName=unicodeFFFE
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode (UTF-8)
    CodePage=65001, WindowsCodePage=1200
    WebName=utf-8, HeaderName=utf-8, BodyName=utf-8
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True

Unicode (UTF-7)
    CodePage=65000, WindowsCodePage=1200
    WebName=utf-7, HeaderName=utf-7, BodyName=utf-7
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=True, IsMailNewsSave=True

US-ASCII
    CodePage=20127, WindowsCodePage=1252
    WebName=us-ascii, HeaderName=us-ascii, BodyName=us-ascii
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=True, IsMailNewsSave=True

Western European (Windows)
    CodePage=1252, WindowsCodePage=1252
    WebName=Windows-1252, HeaderName=Windows-1252, BodyName=iso-8859-1
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True

Western European (Windows)
    CodePage=1252, WindowsCodePage=1252
    WebName=Windows-1252, HeaderName=Windows-1252, BodyName=iso-8859-1
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True
```

```
Below are some specific code pages:
OEM United States
    CodePage=437, WindowsCodePage=1252
    WebName=IBM437, HeaderName=IBM437, BodyName=IBM437
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=False, IsMailNewsSave=False

Cyrillic (ISO)
    CodePage=28595, WindowsCodePage=1251
    WebName=iso-8859-5, HeaderName=iso-8859-5, BodyName=iso-8859-5
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True

ISCII Kannada
    CodePage=57008, WindowsCodePage=57008
    WebName=x-iscii-ka, HeaderName=x-iscii-ka, BodyName=x-iscii-ka
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=False, IsMailNewsSave=False

Chinese Simplified (GB18030)
    CodePage=54936, WindowsCodePage=936
    WebName=GB18030, HeaderName=GB18030, BodyName=GB18030
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True

Thai (Windows)
    CodePage=874, WindowsCodePage=874
    WebName=windows-874, HeaderName=windows-874, BodyName=windows-874
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True
```

Table 12–9 completes the discussion of the methods offered by all **Encoding**–derived classes.

Table 12–9: Methods of the Encoding–Derived Classes

| Method | Description |
|---|---|
| **GetPreamble** | Returns an array of bytes indicating what should be written to a stream before writing any encoded bytes. Frequently, these bytes are referred to as the byte order mark (BOM) byte. When you start reading from a stream, the BOM bytes automatically help detect what encoding was used when the stream was written so that the correct decoder can be used. For most **Encoding**–derived classes, this method returns an array of 0 bytes—that is, no preamble bytes. A **UTF8Encoding** object can be explicitly constructed so that this method returns a 3–byte array of 0xEF, 0xBB, 0xBF. A **UnicodeEncoding** object can be explicitly constructed so that this method returns a 2–byte array of 0xFE, 0xFF for big endian encoding or a 2–byte array of 0xFF, 0xFE for little endian encoding. |
| **Convert** | Converts an array of bytes specified in a source encoding to an array of bytes specified by a destination encoding. Internally, this static method calls the source encoding object's **GetChars** method and passes the result to the destination encoding object's **GetBytes** method. The resulting byte array is returned to the caller. |
| **Equals** | |

| | |
|---|---|
| | Returns **true** if two **Encoding**–derived objects represent the same code page and preamble setting. |
| **GetHashCode** | Returns the encoding object's code page. |

## Encoding/Decoding Streams of Characters and Bytes

Imagine that you're reading a UTF–16 encoded string via a **System.Net.Sockets.NetworkStream** object. The bytes will very likely stream in as chunks of data. In other words, you might first read 5 bytes from the stream, followed by 7 bytes. In UTF–16, each character consists of 2 bytes. So calling **Encoding**'s **GetString** method passing the first array of 5 bytes will return a string consisting of just two characters. If you later call **GetString** passing in the next 7 bytes that come in from the stream, **GetString** will return a string consisting of three characters and all the code points will have the wrong values!

This data corruption problem occurs because none of the **Encoding**–derived classes maintains any state in between calls to their methods. If you'll be encoding or decoding characters/bytes in chunks, you must do some additional work so that state is maintained between calls, preventing any loss of data.

To decode chunks of bytes, you should obtain a reference to an **Encoding**–derived object (as described in the previous section) and call its **GetDecoder** method. This method returns a reference to a newly constructed object whose type is derived from the **System.Text.Decoder** class. Like the **Encoding** class, the **Decoder** class is an abstract base class. If you look in the .NET Framework SDK documentation, you won't find any classes that represent concrete implementations of the **Decoder** class. However, the FCL does define a bunch of **Decoder**–derived classes, such as **UTF8Decoder**. These classes are all internal to the FCL, but the **GetDecoder** method can construct instances of these classes and return them to your application code.

All **Decoder**–derived classes offer two methods: **GetChars** and **GetCharCount**. Obviously, these methods are used for decoding and work similarly to **Encoding**'s **GetChars** and **GetCharCount** methods, discussed earlier. When you call one of these methods, it decodes the byte array as much as possible. If the byte array doesn't contain enough bytes to complete a character, the leftover bytes are saved inside the decoder object. The next time you call one of these methods, the decoder object uses the leftover bytes plus the new byte array passed to it—this ensures that the chunks of data are decoded properly. **Decoder** objects are very useful when reading bytes from a stream.

An **Encoding**–derived type can be used for stateless encoding and decoding. However, a **Decoder**–derived type can be used only for decoding. If you want to encode strings in chunks, call **GetEncoder** instead of calling the **Encoding** object's **GetDecoder** method. **GetEncoder** returns a newly constructed object whose type is derived from the **System.Text.Encoder** class, which is also an abstract base class. Again, the .NET Framework SDK documentation doesn't contain any classes representing concrete implementations of the **Encoder** class. However, the FCL does define some **Encoder**–derived classes, such as **UTF8Encoder**. As with the **Decoder**–derived classes, these classes are all internal to the FCL, but the **GetEncoder** method can construct instances of these classes and return them to your application code.

All **Encoder**–derived classes offer two methods: **GetBytes** and **GetByteCount**. On each call, the **Encoder**–derived object maintains any leftover state information so that you can encode data in chunks.

## Base-64 String Encoding and Decoding

Today, the UTF-16 and UTF-8 encodings are becoming quite popular. Also gaining in popularity is the ability to encode a sequence of bytes to a base-64 string. The FCL does offer methods to do base-64 encoding and decoding, and you might expect that this would be accomplished via an **Encoding**-derived type. However, for some reason, base-64 encoding and decoding is done using some static methods offered by the **System.Convert** type.

To encode a base-64 string as an array of bytes, you call **Convert**'s static **FromBase64String** or **FromBase64CharArray** method. Likewise, to decode an array of bytes as a base-64 string, you call **Convert**'s static **ToBase64String** or **ToBase64CharArray** method. The following code demonstrates how to use some of these methods:

```
using System;

class App {
   static void Main() {
      // Get a set of 10 randomly generated bytes.
      Byte[] bytes = new Byte[10];
      new Random().NextBytes(bytes);

      // Display the bytes.
      Console.WriteLine(BitConverter.ToString(bytes));

      // Decode the bytes into a base-64 string, and show the string.
      String s = Convert.ToBase64String(bytes);
      Console.WriteLine(s);

      // Encode the base-64 string back to bytes, and show the bytes.
      bytes = Convert.FromBase64String(s);
      Console.WriteLine(BitConverter.ToString(bytes));
   }
}
```

Compiling this code and running the executable produces the following output. (Your output might vary from mine because of the randomly generated bytes.)

```
34-24-99-7A-66-BF-D1-5F-41-1C
NCSZema/0V9BHA==
34-24-99-7A-66-BF-D1-5F-41-1C
```

# Chapter 13: Enumerated Types and Bit Flags

In this chapter, I'll discuss enumerated types and bit flags. Since Windows has used these constructs for so many years, I'm sure that many of you are already familiar with how to use enumerated types and bit flags. However, the common language runtime (CLR) and the .NET Framework Class Library (FCL) work together to make enumerated types and bit flags real object–oriented types that offer cool new features that I suspect most developers aren't familiar with. It's amazing to me how these new features, which are the focus of this chapter, make developing application code so much easier.

## Enumerated Types

An enumerated type is a type that defines a set of symbolic names and value pairs. For example, the **Color** type shown here defines a set of symbols, with each symbol identifying a single color:

```
enum Color {
   Red,          // Assigned a value of 0
   Green,        // Assigned a value of 1
   Blue,         // Assigned a value of 2
   Orange        // Assigned a value of 3
}
```

Of course, programmers can always write a program using 0 to represent Red, 1 to represent Green, and so on. However, programmers shouldn't hard–code numbers into their code and should use an enumerated type instead, for at least two reasons:

- Enumerated types make the program much easier to write, read, and maintain. With enumerated types, the symbolic name is used throughout the code and the programmer doesn't have to mentally map that Red is 0 or that 0 means Red. Also, should a symbol's numeric value change, the code can simply be recompiled without requiring any changes to the source code. In addition, documentation tools and other utilities, such as a debugger, can show meaningful symbolic names to the programmer.
- Enumerated types are strongly typed. For example, the compiler will report an error if I attempt to pass **Color.Orange** as a value to a method requiring a **Fruit** enumerated type as a parameter.

In the CLR, enumerated types are more than just symbols that the compiler cares about. Enumerated types are treated as first–class citizens in the type system, which allows for very powerful operations that simply can't be done with enumerated types in other environments (such as in unmanaged C++, for example).

Every enumerated type inherits directly from **System.Enum**, which inherits from **System.ValueType**, which in turn inherits from **System.Object**. So, enumerated types are value types (described in Chapter 5) and can be represented in unboxed and boxed forms. However, unlike other value types, an enumerated type can't define any methods, properties, or events.

When an enumerated type is compiled, the C# compiler turns each symbol into a constant field of the type. For example, the compiler treats the **Color** enumeration shown earlier as if you had written code similar to the following:

```
struct Color : System.Enum {
   public const Color Red    = (Color) 0;
   public const Color Green  = (Color) 1;
```

```
   public const Color Blue   = (Color) 2;
   public const Color Orange = (Color) 3;
}
```

The C# compiler won't actually compile this code, but this example does give you an idea of what's happening internally. Basically, an enumerated type is just a structure that has a bunch of constant fields defined in it. The fields are emitted to the module's metadata and can be accessed via reflection. This means that you can get all the symbols and their values associated with an enumerated type at run time. It also means that you can convert a string symbol into its equivalent numeric value.

**Important** Symbols defined by an enumerated type are constant values. This means that compilers convert code that references an enumerated type's symbol to a numeric value at compile time. Once this occurs, no reference to the enumerated type exists in metadata and the assembly that defines the enumerated type doesn't have to be available at run time. If you have code that references the enumerated type—rather than just having references to symbols defined by the type—the assembly that defines the enumerated type will be required at run time. Some versioning issues arise because enumerated type symbols are constants instead of read–only values. I explained these issues in Chapter 8.

To simplify these operations and make it so that you don't have to be familiar with reflection, the **System.Enum** base type offers several static methods and instance methods that expose the special operations that can be performed on an instance of an enumerated type. I'll discuss some of these operations next.

For example, the **Enum** type has a static method called **GetUnderlyingType**:

```
static Type GetUnderlyingType(Type enumType);
```

This method returns the core type used to hold an enumerated type's value. Every enumerated type has an underlying type, which can be **byte**, **sbyte**, **short**, **ushort**, **int** (the most common and what C# chooses by default), **uint**, **long**, or **ulong**. Of course, these C# primitive types correspond to FCL types. However, the C# compiler requires that you specify a primitive type here; using a base class type (like **Int32**) generates an error. The following code shows how to declare an enumerated type with an underlying type of **byte** (**System.Byte**) by using C#:

```
enum Color : byte {
   Red,
   Green,
   Blue,
   Orange
}
```

With the **Color** enumerated type defined in this way, the following code shows what **GetUnderlyingType** will return:

```
// The following line displays "System.Byte".
Console.WriteLine(Enum.GetUnderlyingType(typeof(Color)));
```

Given an instance of an enumerated type, it's possible to map that value to one of four string representations by using the **System.Enum.ToString** method:

```
Color c = Color.Blue;
```

```
Console.WriteLine(c.ToString());    // "Blue" (General format)
Console.WriteLine(c.ToString("G")); // "Blue" (General format)
Console.WriteLine(c.ToString("D")); // "2"    (Decimal format)
Console.WriteLine(c.ToString("X")); // "2"    (heX format)
```

Internally, the **ToString** method calls **System.Enum**'s static **Format** method:

```
public static String Format(Type enumType,
   Object value, String format);
```

Generally, I prefer to call the **ToString** method because it requires less code and it's easier to call. But using **Format** has one advantage over **ToString**. **For−mat** lets you pass a numeric value for the value parameter; you don't have to have an instance of the enumerated type. For example, the following code will display "Blue":

```
// The following line displays "Blue".
Console.WriteLine(Enum.Format(typeof(Color), 2, "G"))
```

> **Note** It's possible to declare an enumerated type that has multiple symbols all with the same numeric value. When converting a numeric value to a symbol, **Enum**'s methods return one of the symbols. However, there's no guarantee as to which symbol name is returned. Also, if no symbol is defined for the numeric value you're looking up, a string containing the numeric value is returned.

It's also possible to create an array that contains one element for each symbolic name in an enumerated type. You use **System.Enum**'s static **GetValues** method:

```
static Array GetValues(Type enumType);
```

Using this method along with the **ToString** method, you can display all of an enumerated type's symbolic and numeric values, like so:

```
Color[] colors = (Color[]) Enum.GetValues(typeof(Color));
Console.WriteLine("Number of symbols defined: " + colors.Length);
Console.WriteLine("Value\tSymbol\n−−\t−−−");
foreach (Color c in colors) {
   // Display each symbol in Decimal and General format.
   Console.WriteLine("{0,5:D}\t{1:G}", c, c);
}
```

The previous code produces the following output:

```
Number of symbols defined: 4
Value   Symbol
 −−−−   −−−
    0   Red
    1   Green
    2   Blue
    3   Orange
```

This discussion shows some of the cool operations that can be performed on enumerated types. I suspect the **ToString** method with the general format will be used quite frequently to show symbolic names in a program's user interface, as long as the strings don't need to be localized (since enumerated types offer no support for localization). In addition to the **GetValues** method, the **Enum** type also offers the following two static methods that return an enumerated type's symbols:

```
// Returns a String representation for the numeric value
public static String   GetName(Type enumType, Object value);
```

```
// Returns an array of Strings: one per symbol defined in the enum
public static String[] GetNames(Type enumType);
```

I've discussed a lot of methods that you can use to look up an enumerated type's symbol. But you also need a method that can look up a symbol's equivalent value, an operation that could be used to convert a symbol that a user enters into a text box, for example. Converting a symbol to an instance of an enumerated type is easily accomplished using **Enum**'s static **Parse** method:

```
public static Object Parse(Type enumType,
   String value, Boolean ignoreCase);
```

Here's some code demonstrating how to use this method:

```
// Because Orange is defined as 3, 'c' is initialized to 3.
Color c = (Color) Enum.Parse(typeof(Color), "orange", true);

// Because Brown isn't defined, an ArgumentException is thrown.
Color c = (Color) Enum.Parse(typeof(Color), "Brown", false);

// Creates an instance of the Color enum with a value of 1
Color c = (Color) Enum.Parse(typeof(Color), "1", false);

// Creates an instance of the Color enum with a value of 23
Color c = (Color) Enum.Parse(typeof(Color), "23", false);
```

for an enumerated type:

```
// Displays "True" because Color defines Green as 1
Console.WriteLine(Enum.IsDefined(typeof(Color), 1));

// Displays "True" because Color defines Red as 0
Console.WriteLine(Enum.IsDefined(typeof(Color), "Red"));

// Displays "False" because a case-sensitive check is performed
Console.WriteLine(Enum.IsDefined(typeof(Color), "red"));

// Displays "False" because Color doesn't have a symbol of value 10
Console.WriteLine(Enum.IsDefined(typeof(Color), 10));
```

The **IsDefined** method is frequently used to do parameter validation. Here's an example:

```
public void SetColor(Color c) {
   if (!Enum.IsDefined(typeof(Color), c)) {
      throw(new ArgumentOutOfRangeException("c", "Not a valid Color"));
   }
   Ã
}
```

The parameter validation is useful because someone could call **SetColor** like this:

```
SetColor((Color) 547);
```

Because no symbol has a corresponding value of 547, the **SetColor** method will throw an **ArgumentOutOfRangeException** exception, indicating which parameter was invalid and why.

Finally, the **System.Enum** type offers a set of static **ToObject** methods that convert an instance of a **Byte**, **SByte**, **Int16**, **UInt16**, **Int32**, **UInt32**, **Int64**, or **UInt64** to an instance of an enumerated type.

Enumerated types are always used in conjunction with some other type. Typically, they're used for the type's method parameters, properties, and fields. A common question that arises is whether to define the enumerated type nested within the type that requires it or whether to define the enumerated type at the same level as the type that requires it. If you examine the FCL, you'll see that usually an enumerated type is defined at the same level as the class that requires it. The reason is simply to make the developer's life a little easier by reducing the amount of typing required. So, you should define your enumerated type at the same level unless you're concerned about name conflicts.

## Bit Flags

Programmers frequently work with sets of bit flags. When you call the **System.IO.File** type's **GetAttributes** method, it returns an instance of a **FileAttributes** type. A **FileAttributes** type is an instance of an **Int32**–based enumerated type, where each bit reflects a single attribute of the file. The **FileAttributes** type is defined in the FCL as follows:

```
[Flags, Serializable]
public enum FileAttributes {
   ReadOnly          = 0x0001,
   Hidden            = 0x0002,
   System            = 0x0004,
   Directory         = 0x0010,
   Archive           = 0x0020,
   Device            - 0x0040,
   Normal            = 0x0080,
   Temporary         = 0x0100,
   SparseFile        = 0x0200,
   ReparsePoint      = 0x0400,
   Compressed        = 0x0800,
   Offline           = 0x1000,
   NotContentIndexed = 0x2000,
   Encrypted         = 0x4000
}
```

To determine whether a file is hidden, you would execute code like this:

```
String file = @"C:\Boot.ini";
FileAttributes attributes = File.GetAttributes(file);
Console.WriteLine("Is {0} hidden? {1}", file,
   (attributes & FileAttributes.Hidden) != 0);
```

And here's code demonstrating how to change a file's attributes to read–only and hidden:

```
File.SetAttributes(@"C:\Boot.ini",
   FileAttributes.ReadOnly | FileAttributes.Hidden);
```

As the **FileAttributes** type shows, it's common to use enumerated types to express the set of bit flags that can be combined. However, although enumerated types and bit flags are similar, they don't have exactly the same semantics. For example, enumerated types represent single numeric values, and bit flags represent a set of flags, some of which are on and some of which are off.

When defining an enumerated type that is to be used to identify bit flags, you should, of course, explicitly assign to each of the symbols numeric values that map to individual bits. It's also highly recommended that you apply the **System.FlagsAttribute** custom attribute type to the enumerated type, as shown here:

```
[Flags]     // The C# compiler allows either "Flags" or "FlagsAttribute".
enum Actions {
   Read   = 0x0001,
   Write  = 0x0002,
   Delete = 0x0004,
   Query  = 0x0008,
   Sync   = 0x0010
}
```

Because **Actions** is an enumerated type, you can use all the methods described in the previous section when working with bit flag enumerated types. However, it would be nice if some of those functions behaved a little differently. For example, let's say you had the following code:

```
Actions actions = Actions.Read | Actions.Write; // 0x0003
Console.WriteLine(actions.ToString());          // "Read, Write"
```

When **ToString** is called, it attempts to translate the numeric value into its symbolic equivalent. The numeric value is 0x0003, which has no symbolic equivalent. However, the **ToString** method detects the existence of the **[Flags]** attribute on the **Actions** type, and **ToString** now treats the numeric value not as a single value but as a set of bit flags. Because the 0x0001 and 0x0002 bits are set, **ToString** generates the following string: "Read, Write". If you remove the **[Flags]** attribute from the **Actions** type, **ToString** would produce the following string: "3".

I discussed the **ToString** method in the previous section, and I showed that it offered three ways to format the output: "G" (general), "D" (decimal), and "X" (hex). When you're formatting an instance of an enumerated type by using the general format, the type is first checked to see whether the **[Flags]** attribute is applied to it. If this attribute is not applied, a symbol matching the numeric value is looked up and returned. If the **[Flags]** attribute is applied, a symbol matching each 1 bit is looked up and concatenated to a string; each symbol is separated by a comma.

If you prefer, you could define the **Actions** type without the **[Flags]** attribute and still get the correct string by using the "F" format:

```
// [Flags]     // Commented out now
enum Actions {
   Read   = 0x0001,
   Write  = 0x0002,
   Delete = 0x0004,
   Query  = 0x0008,
   Sync   = 0x0010
}

Actions actions = Actions.Read | Actions.Write; // 0x0003
Console.WriteLine(actions.ToString("F"));       // "Read, Write"
```

As I mentioned in the previous section, the **ToString** method actually calls **System.Enum**'s static **Format** method internally. This means that you can use the "F" format when calling the static **Format** method. If the numeric value contains 1 bit with no matching symbols, the returned string will contain just a decimal number and none of the symbols will appear in the string.

Note that the symbols you define in your enumerated type don't have to be pure powers of 2. For example, the **Actions** type could define a symbol called **All** with a value of 0x001F. If an instance of the **Actions** type has a value of 0x001F, formatting the instance will produce a string that contains "All". The other symbol strings won't appear.

So far, I've discussed how to convert numeric values into a string of flags. It's also possible to convert a string of comma-delimited symbols into a numeric value by calling **Enum**'s static **Parse** method. Here's some code demonstrating how to use this method:

```
// Because Query is defined as 8, 'a' is initialized to 8.
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query", true);

// Because Query and Read are defined, 'a' is initialized to 9.
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query,Read", false);

// Creates an instance of the Actions enum with a value of 28
Actions a = (Actions) Enum.Parse(typeof(Actions), "28", false);
Console.WriteLine(a.ToString());  // "Delete, Query, Sync"

// Creates an instance of the Actions enum with a value of 333
Actions a = (Actions) Enum.Parse(typeof(Actions), "333", false);
Console.WriteLine(a.ToString());  // "333"
```

Again, when **Parse** is called, it checks whether the **[Flags]** custom attribute has been applied to the enumerated type. If the attribute exists, **Parse** splits the string into individual symbols, looks up each symbol, and bitwise-ORs the corresponding numeric value into the resulting instance of the enumerated type. See Chapter 16 for more information about custom attributes.

The **[Flags]** attribute affects how **ToString**, **Format**, and **Parse** behave. Compilers are also encouraged to look for this attribute and ensure that the enumerated type is being manipulated as a set of bit flags. For example, a compiler could allow only bit operations on the bit flag enumerated type and disallow other arithmetic operations, such as multiplication and division. The C# compiler ignores the **[Flags]** attribute completely; anything you can do with an enumerated type you can do with a bit flag enumerated type.

When using a Visual Studio .NET form designer, you can use a property window to make various settings at design time. If some of these settings are enumerated types, the form designer checks whether the **[Flags]** attribute is applied to the type and displays the possible values accordingly.

# Chapter 14: Arrays

## Overview

Arrays are mechanisms that allow you to treat several items as a single collection. The Microsoft .NET common language runtime (CLR) supports single–dimension arrays, multidimension arrays, and jagged arrays (that is, arrays of arrays). All array types are implicitly derived from **System.Array**, which itself is derived from **System.Object**. This means that arrays are always reference types that are allocated on the managed heap and that your application's variable contains a reference to the array and not the array itself. The following code makes this clearer:

```
Int32[] myIntegers;          // Declares a reference to an array
myIntegers = new Int32[100]; // Creates an array of 100 Int32s
```

On the first line, **myIntegers** is a variable that's capable of pointing to a single–dimension array of **Int32**s. Initially, **myIntegers** will be set to **null** because I haven't allocated an array. The second line of code allocates an array of 100 **Int32** values; all the **Int32**s are initialized to 0. Even though **Int32**s are value types, the memory block large enough to hold these values is allocated from the managed heap. The memory block contains 100 unboxed **Int32** values. The address of this memory block is returned and saved in the variable **myIntegers**.

You can also create arrays of reference types:

```
Control[] myControls;         // Declares a reference to an array
myControls = new Control[50]; // Creates an array of 50 Control references
```

On the first line, **myControls** is a variable capable of pointing to a single–dimension array of **Control** references. Initially, **myControls** will be set to **null** because I haven't allocated an array. The second line allocates an array of 50 **Control** *references;* all of these references are initialized to **null**. Because **Control** is a reference type, creating the array creates only references; the actual objects aren't created at this time. The address of this memory block is returned and saved in the variable **myControls**.

Figure 14–1 shows how arrays of value types and arrays of reference types look in the managed heap.

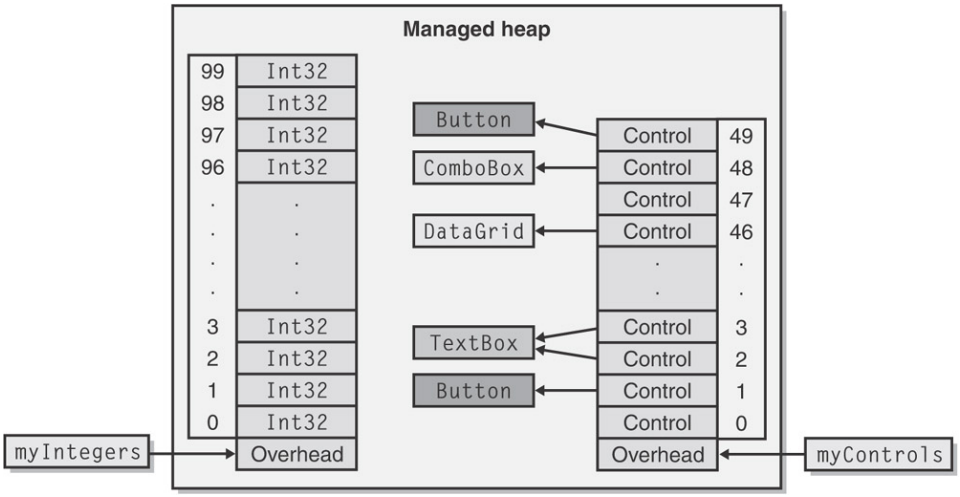



Figure 14–1 : Arrays of value and reference types in the managed heap
In the figure, the **Controls** array shows the result after the following lines have executed:

```
myControls[1] = new Button();
myControls[2] = new TextBox();
myControls[3] = myControls[2];  // Two elements refer to the same object.
myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

Common Language Specification (CLS) compliance requires that all arrays be zero–based. This allows a method written in C# to create an array and pass the array's reference to code written in another language, such as Microsoft Visual Basic. In addition, because zero–based arrays are, by far, the most common arrays, Microsoft has spent a lot of time optimizing their performance. However, the CLR does support non–zero–based arrays even though their use is discouraged. For those of you who don't care about performance or cross–language portability, I'll demonstrate how to create and use non–zero–based arrays later in this section.

Notice in Figure 14–1 that each array has some additional overhead information associated with it. This information contains the rank of the array (number of dimensions), the lower bounds for each dimension of the array (almost always 0), and the length of each dimension. The overhead also contains the type of each element in the array. Shortly, I'll mention the methods that allow you to query this overhead information.

So far, I've shown examples demonstrating how to create single–dimension arrays. When possible, you should stick with single–dimension, zero–based arrays, sometimes referred to as *SZ arrays*, or *vectors*. Vectors give the best performance because you can use specific IL instructions—such as **newarr**, **ldelem**, **ldelema**, **ldlen**, and **stelem**—to manipulate them. However, if you prefer to work with multidimension arrays, you can. Here are some examples of multidimension arrays:

```
// Create a two-dimension array of Doubles.
Double[,] myDoubles = new Double[10, 20];

// Create a three-dimension array of String references.
String[,,] myStrings = new String[5, 3, 10];
```

The CLR also supports jagged arrays. Zero–based, single–dimension jagged arrays have the same performance as normal vectors. However, accessing the elements of a jagged array means that two or more array accesses must occur. Note that jagged arrays are not CLS–compliant—because the CLS doesn't allow a **System.Array** object to be an element of an array—and can't be passed between code written in different programming languages. Fortunately, C# supports jagged arrays. Here are some examples of how to create an array of polygons, where each polygon consists of an array of **Point** instances:

```
// Create a single-dimension array of Point arrays.
Point[][] myPolygons = new Point[3][];

// myPolygons[0] refers to an array of 10 Point instances.
myPolygons[0] = new Point[10];

// myPolygons[1] refers to an array of 20 Point instances.
myPolygons[1] = new Point[20];

// myPolygons[2] refers to an array of 30 Point instances.
myPolygons[2] = new Point[30];

// Display the Points in the first polygon.
for (Int32 x = 0, l = myPolygons[0].Length; x < l; x++)
   Console.WriteLine(myPolygons[0][x]);
```

**Note** The CLR verifies that an index into an array is valid. In other words, you can't create an array with 100 elements in it (numbered 0 through 99) and then try to access the element at index 100 or –5. Doing so will cause a **System.IndexOutOfRangeException** exception to be thrown. Allowing access to memory outside the range of an array would be a breach of type safety and a potential security hole, and the CLR doesn't allow verifiable code to do this. Usually, the performance associated with index checking is insubstantial because the JIT (just–in–time) compiler normally checks array bounds once before a loop executes instead of at each loop iteration. However, if you're still concerned about the performance hit of the CLR's index checks, you can use unsafe code in C# to access the array. The "Fast Array Access" section later in this chapter demonstrates how to do this.

## All Arrays Are Implicitly Derived from System.Array

The **System.Array** type offers several static and instance members. Because all arrays are implicitly derived from **System.Array**, these members can be used to manipulate arrays of value types or reference types. Also note that **Array** implements several interfaces: **ICloneable**, **IEnumerable**, **ICollection**, and **IList**. These interfaces allow arrays to be conveniently used in many different scenarios.

Table 14–1 summarizes the methods offered by **System.Array** and the interfaces that it implements.

Table 14–1: Members of System.Array

| Member | Member Type | Description |
|---|---|---|
| **Rank** | Read–only instance property | Returns the number of dimensions in the array. |
| **GetLength** | Instance method | Returns the number of elements in the specified dimension of the array. |
| **Length** | Read–only instance property | Returns the total number of elements in the array. |
| **GetLowerBound** | Instance method | Returns the lower bound of the specified dimension. This is almost always 0. |
| **GetUpperBound** | Instance method | Returns the upper bound of the specified dimension. This is almost always the number of elements in the dimension minus 1. |
| **IsReadOnly** | Read–only instance property | Indicates whether the array is read–only. For arrays, this is always **false**. |
| **IsSynchronized** | Read–only instance property | Indicates whether the array access is thread–safe. For arrays, this is always **false**. |
| **SyncRoot** | Read–only instance property | Retrieves an object that can be used to synchronize access to the array. For arrays, this is always a reference to the array itself. |
| **IsFixedSize** | Read–only instance property | Indicates whether the array is a fixed size. For arrays, this is always **true**. |
| **GetValue** | Instance method | Returns a reference to the element located at the specified position in the |

| | | |
|---|---|---|
| | | array. If the array contains value types, the return value refers to a boxed copy of the element. This rarely used method is required only when you don't know at design time the number of dimensions in an array. |
| **SetValue** | Instance method | Sets the element located at the specified position in the array. This rarely used method is required only when you don't know at design time the number of dimensions in an array. |
| **GetEnumerator** | Instance method | Returns an **IEnumerator** for the array. This allows using C#'s **foreach** statement (or an equivalent in another language). For multidimension arrays, the enumerator iterates through all the elements, with the right–most dimension changing the fastest. |
| **Sort** | Static method | Sorts the elements in one array, in two arrays, or in a section of an array. The array element type must imple–ment the **IComparer** interface or must pass an object whose type implements the **IComparer** interface. |
| **BinarySearch** | Static method | Searches the specified array for the specified element using a binary search algorithm. This method assumes that the array's elements are sorted. The array element type must implement the **IComparer** interface. You usually use the **Sort** method before calling **BinarySearch**. |
| **IndexOf** | Static method | Returns the index of the first occurrence of a value in a one–dimension array or in a portion of it. |
| **LastIndexOf** | Static method | Returns the index of the last occurrence of a value in a one–dimension array or in a portion of it. |
| **Reverse** | Static method | Reverses the order of the elements in the specified one–dimension array or in a portion of it. |
| **Clone** | Instance method | Creates a new array that's a shallow copy of the source array. |
| **CopyTo** | Instance method | Copies elements from one array to another array. |
| **Copy** | Static method | Copies a section of one array to another array, performing any appropriate casting required. |
| **Clear** | Static method | Sets a range of elements in the array to 0 or to a **null** object reference. |

| | | |
|---|---|---|
| **CreateInstance** | Static method | Creates an instance of an array. This rarely used method allows you to dynamically (at run time) define arrays of any type, rank, and bounds. |
| **Initialize** | Instance method | Calls the default constructor for each element in an array of value types. This method does nothing if the elements in the array are reference types. C# doesn't allow you to define default constructors for value types, so this method has no use for arrays of C# structures. This method is primarily for compiler vendors. |

## Casting Arrays

For arrays with reference type elements, the CLR allows you to implicitly cast the source array's element type to a target type. For the cast to succeed, both array types must have the same number of dimensions, and an implicit or explicit conversion from the source element type to the target element type must exist. The CLR doesn't allow the casting of arrays with value type elements to any other type. (However, by using the **Array.Copy** method, you can create a new array that has the desired effect.) The following code demonstrates how array casting works:

```
// Create a two-dimension FileStream array.
FileStream[,] fs2dim = new FileStream[5, 10];

// Implicit cast to a two-dimension Object array
Object[,] o2dim = fs2dim;

// Can't cast from 2-dimension array to one-dimension array
// Compiler error CS0030: Cannot convert type 'object[*,*]' to
// 'System.IO.Stream[]'
Stream[] s1dim = (Stream[]) o2dim;

// Explicit cast to two-dimension Stream array
Stream[,] s2dim = (Stream[,]) o2dim;

// Explicit cast to two-dimension Type array
// Compiles but throws InvalidCastException at run time
Type[,] t2dim = (Type[,]) o2dim;

// Create a one-dimension Int32 array (value types).
Int32[] i1dim = new Int32[5];

// Can't cast from array of value types to anything else
// Compiler error CS0030: Cannot convert type 'int[]' to 'object[]'
Object[] o1dim = (Object[]) i1dim;

// Array.Copy creates a new array, coercing each element in the source
// array to the desired type in the destination array.
// The following code creates an array of references to boxed Int32s.
Object[] o1dim = new Object[i1dim.Length];
Array.Copy(i1dim, o1dim, i1dim.Length);
```

The **Array.Copy** method is not just a method that copies elements from one array to another fast. The **Copy** method is also capable of converting each array element as it is copied if conversion is

required. The **Copy** method is capable of performing the following conversions.

- Boxing value type elements to reference type elements, such as when copying an **Int32[]** to an **Object[]**
- Unboxing reference type elements to value type elements, such as when copying an **Object[]** to an **Int32[]**
- Widening CLR primitive value types, such as when copying an **Int32[]** to a **Double[]**

Here's another example showing the usefulness of **Copy**:

```
// Define a value type that implements an interface.
struct MyValueType : ICloneable {
   Ã
}


class App {
   static void Main() {
      // Create an array of 100 value types.
      MyValueType[] src = new MyValueType[100];

      // Create an array of ICloneable references.
      ICloneable[] dest = new ICloneable[src.Length];

      // Initialize an array of ICloneable elements to refer to boxed
      // versions of elements in the source array.
      Array.Copy(src, dest, src.Length);
   }
}
```

As you might imagine, the .NET Framework Class Library (FCL) takes advantage of **Array**'s **Copy** method quite frequently.

## Passing and Returning Arrays

Arrays are always passed by reference to a method. Because the CLR doesn't support the notion of constant parameters, the method is able to modify the elements in the array. If you don't want to allow this, you must make a copy of the array and pass the copy into the method. Note that the **Array.Copy** method performs a shallow copy and, therefore, if the array's elements are reference types, the new array refers to the already existing objects.

To obtain a deep copy, you might want to clone the individual elements, but this requires that each object's type implements the **ICloneable** interface. Alternatively, you could serialize each object to a **System.IO.MemoryStream** and then immediately deserialize the memory stream to construct a new object. Depending on the object's types, the performance of these operations can be prohibitive, and not all types are serializable either.

Similarly, some methods return a reference to an array. If the method constructs and initializes the array, returning a reference to the array is fine. But if the method wants to return a reference to an internal array maintained by a field, you must decide whether you want the method's caller to have direct access to this array. If you do, just return the array's reference. But most often you won't want the method's caller to have such access, so the method should construct a new array and call **Array.Copy**, returning a reference to the new array. Again, you might want to clone each of the objects before returning the array reference.

If you define a method that is to return a reference to an array and if that array has no elements in it, your method can return either **null** or a reference to an array with zero elements in it. When you're implementing this kind of method, Microsoft strongly recommends that you implement the method by having it return a zero–length array because doing so simplifies the code that a developer calling the method must write. For example, this easy–to–understand code runs correctly even if there are no appointments to iterate over:

```
// This code is easier to write and understand.
Appointment[] appointments = GetAppointmentsForToday();
for (Int32 a = 0, n = appointments.Length; a < n; a++) {
   Ã
}
```

The following code also runs correctly if there are no appointments to iterate over. However, this code is slightly more difficult to write and understand:

```
// This code is harder to write and understand.
Appointment[] appointments = GetAppointmentsForToday();
if (appointments != null) {
   for (Int32 a = 0, n = appointments.Length; a < n; a++) {
      Ã
   }
}
```

If you design your methods so that they return arrays with zero elements instead of **null**, callers of your methods will have an easier time working with them. By the way, you should do the same for fields. If your type has a field that's a reference to an array, you should always have the field refer to an array even if the array has no elements in it. Allowing the field to be **null** will just make your type harder to use.

## Creating Arrays That Have a Nonzero Lower Bound

Earlier I mentioned that it's possible to create and work with arrays that have nonzero lower bounds. You can dynamically create your own arrays by calling **Array**'s static **CreateInstance** method. Several overloads of this method exist, but they all allow you to specify the type of the elements in the array, the number of dimensions in the array, the lower bounds of each dimension, and the number of elements in each dimension. **CreateInstance** allocates memory for the array, saves the parameter information in the overhead portion of the array's memory block, and returns a reference to the array. You can cast the reference returned from **CreateInstance** to a variable so that it's easier for you to access the elements in the array.

Here's some code that demonstrates how to dynamically create a two–dimension array of **System.Decimal** values. The first dimension represents calendar years and goes from 1995 to 2004 inclusive. The second dimension represents quarters and goes from 1 to 4 inclusive.

```
// I want a two-dimension array [1995..2004][1..4].
Int32[] lowerBounds = { 1995, 1 };
Int32[] lengths     = {   10, 4 };
Decimal[,] quarterlyRevenue = (Decimal[,])
   Array.CreateInstance(typeof(Decimal), lengths, lowerBounds);
```

The following code iterates over all the elements in the dynamic array. I could have hard–coded the array's bounds into the code, which would have given better performance, but I decided to use some of **System.Array**'s **GetLowerBound** and **GetUpperBound** methods for demonstration

purposes.

```
Int32 firstYear = quarterlyRevenue.GetLowerBound(0);
Int32 lastYear  = quarterlyRevenue.GetUpperBound(0);
Console.WriteLine("{0,4}  {1,9}  {2,9}  {3,9}  {4,9}",
   "Year", "Q1", "Q2", "Q3", "Q4");

for (Int32 year = firstYear; year <= lastYear; year++) {
   Console.Write(year + "  ");

   for (Int32 quarter = quarterlyRevenue.GetLowerBound(1);
      quarter <= quarterlyRevenue.GetUpperBound(1); quarter++) {

      Console.Write("{0,9:C}  ", quarterlyRevenue[year, quarter]);
   }
   Console.WriteLine();
}
```

## Fast Array Access

Each time an element of an array is accessed, the CLR ensures that the index is within the array's bounds. This prevents you from accessing memory that is outside of the array, which would potentially corrupt other objects. If an invalid index is used to access an array element, the CLR throws a **System.IndexOut–OfRangeException** exception.

As you might expect, the CLR's index checking comes at a performance cost. If you have confidence in your code and if you don't mind resorting to nonverifiable (unsafe) code, you can access an array without having the CLR perform its index checking. The following C# code demonstrates this approach:

```
using System;

class App {
   unsafe static void Main() {

      // Construct an array consisting of five Int32 elements.
      Int32[] arr = new Int32[] { 1, 2, 3, 4, 5 };

      // Obtain a pointer to the array's 0th element.
      fixed (Int32* element = &arr[0]) {

         // Iterate through each element in the array.
         // NOTE: The following code has a bug!
         for (Int32 x = 0, n = arr.Length; x <= n; x++) {
            Console.WriteLine(element[x]);
         }
      }
   }
}
```

To compile this code, enter the following at the command line (assuming the source is contained in a file named UnsafeArrayAccess.cs):

```
csc.exe /unsafe UnsafeArrayAccess.cs
```

After you build this small application, running it produces the following results:

```
1
2
3
4
5
0
```

Although you'd expect only five values to appear, six values actually appear because of a bug in the source code. In the **for** loop, the test expression should be **x** < **n**, not **x** <= **n**. You must be very careful when using unsafe code!

By the way, if you use ILDasm.exe to examine the intermediate language (IL) for **Main**, you'll see the following, which I've commented:

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       58 (0x3a)
  .maxstack  3
  .locals ([0] int32[] arr,
           [1] int32& pinned element,
           [2] int32 x,
           [3] int32 n)

  // Construct an array of five Int32 elements.
  IL_0000:  ldc.i4.5
  IL_0001:  newarr     [mscorlib]System.Int32

  // Initialize the array's elements with values stored in metadata.
  IL_0006:  dup
  IL_0007:  ldtoken    field valuetype
    '<PrivateImplementationDetails>`/'$$struct0x6000001-1'
    '<PrivateImplementationDetails>`::'$$method0x6000001-1'
  IL_000c:  call
    void [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::
      InitializeArray(class [mscorlib]System.Array,
      valuetype [mscorlib]System.RuntimeFieldHandle)

  // Save the reference to the array in the arr variable.
  IL_0011:  stloc.0

  // Get the address of arr's 0th element and save it in element.
  IL_0012:  ldloc.0
  IL_0013:  ldc.i4.0
  IL_0014:  ldelema    [mscorlib]System.Int32
  IL_0019:  stloc.1

  // Initialize x to 0.
  IL_001a:  ldc.i4.0
  IL_001b:  stloc.2

  // Initialize n to the length of arr.
  IL_001c:  ldloc.0
  IL_001d:  ldlen
  IL_001e:  conv.i4
  IL_001f:  stloc.3

  // Branch to the for loop's test.
  IL_0020:  br.s       IL_0032

  // Calculate element + (4 * x) — 4 is the number of bytes in an Int32.
  IL_0022:  ldloc.1
```

```
  IL_0023:  conv.i
  IL_0024:  ldc.i4.4
  IL_0025:  ldloc.2
  IL_0026:  mul
  IL_0027:  add

  // Pass the value at this address to Console.WriteLine.
  IL_0028:  ldind.i4
  IL_0029:  call       void [mscorlib]System.Console::WriteLine(int32)

  // Add 1 to x.
  IL_002e:  ldloc.2
  IL_002f:  ldc.i4.1
  IL_0030:  add
  IL_0031:  stloc.2

  // for loop test: loop again if x <= n.
  IL_0032:  ldloc.2
  IL_0033:  ldloc.3
  IL_0034:  ble.s      IL_0022

  // End of loop: Put null in element (for safety).
  IL_0036:  ldc.i4.0
  IL_0037:  conv.u
  IL_0038:  stloc.1

  // Return from Main.
  IL_0039:  ret
} // end of method App::Main
```

For comparison, here's a version that doesn't use unsafe code:

```
using System;

class App {
   static void Main() {

      Int32[] arr = new Int32[] { 1, 2, 3, 4, 5 };

      for (Int32 x = 0, n = arr.Length; x <= n; x++) {
         Console.WriteLine(arr[x]);
      }
   }
}
```

If you build this and use ILDasm.exe to examine the IL code, you'll see the following:

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       43 (0x2b)
  .maxstack  3
  .locals init ([0] int32[] arr,
           [1] int32 x,
           [2] int32 n)

  // Construct an array of five Int32 elements.
  IL_0000:  ldc.i4.5
  IL_0001:  newarr     [mscorlib]System.Int32

  // Initialize the array's elements with values stored in metadata.
  IL_0006:  dup
```

```
  IL_0007:  ldtoken    field valuetype
    '<PrivateImplementationDetails>'/'$$struct0x6000001-1'
    '<PrivateImplementationDetails>'::'$$method0x6000001-1'
  IL_000c:  call
    void [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::
      InitializeArray(class [mscorlib]System.Array,
      valuetype [mscorlib]System.RuntimeFieldHandle)
  // Save the reference to the array in the arr variable.
  IL_0011:  stloc.0

  // Initialize x to 0
  IL_0012:  ldc.i4.0
  IL_0013:  stloc.1

  // Initialize n to the length of arr.
  IL_0014:  ldloc.0
  IL_0015:  ldlen
  IL_0016:  conv.i4
  IL_0017:  stloc.2

  // Branch to the for loop's test.
  IL_0018:  br.s       IL_0026

  // Pass the element in arr[x] to Console.WriteLine.
  IL_001a:  ldloc.0
  IL_001b:  ldloc.1
  IL_001c:  ldelem.i4
  IL_001d:  call       void [mscorlib]System.Console::WriteLine(int32)

  // Add 1 to x.
  IL_0022:  ldloc.1
  IL_0023:  ldc.i4.1
  IL_0024:  add
  IL_0025:  stloc.1

  // for loop test: loop again if x < n.
  IL_0026:  ldloc.1
  IL_0027:  ldloc.2
  IL_0028:  ble.s      IL_001a

  // Return from Main.
  IL_002a:  ret
} // End of method App::Main
```

It's true that there's less IL code in the type–safe version. However, it's the type–safe version's **ldelem** instruction that causes the CLR to do index checking. The unsafe version uses the **ldind.i4** instruction, which simply obtains a 4–byte value from a memory address. Note that this unsafe array manipulation technique is usable with arrays whose elements are **SByte**, **Byte**, **Int16**, **UInt16**, **Int32**, **UInt32**, **Int64**, **UInt64**, **Char**, **Single**, **Double**, **Decimal**, **Boolean**, an enumerated type, or a value type structure whose fields are any of the aforementioned types.

## Redimensioning an Array

**Array**'s static **CreateInstance** method allows you to dynamically construct an array when you don't know at compile time the types of elements that the array is to maintain. The method is also useful when you don't know at compile time how many dimensions the array is to have and the bounds of those dimensions. In the section "Creating Arrays That Have a Nonzero Lower Bound," I demonstrated how to dynamically construct an array by using arbitrary bounds. You can also use

the **CreateInstance** method to redimension an arbitrary array, like so:

```
using System;

class App {

   static void Main() {
      // Construct an array of three elements.
      Int32[] arr = new Int32[] { 1 , 2, 3 };

      // Display all the elements in the array.
      foreach (Int32 x in arr)
         Console.Write(x + " ");
      Console.WriteLine();

      // Redimension the array so that it contains five elements.
      arr = (Int32[]) Redim(arr, 5);

      // Display all the elements in the array.
      foreach (Int32 x in arr)
         Console.Write(x + " ");
      Console.WriteLine();

      // Redimension the array so that it now contains two elements.
      arr = (Int32[]) Redim(arr, 2);

      // Display all the elements in the array.
      foreach (Int32 x in arr)
         Console.Write(x + " ");
   }


   public static Array Redim(Array origArray, Int32 desiredSize) {

      // Determine the types of each element.
      Type t = origArray.GetType().GetElementType();

      // Construct a new array with the desired number of elements.
      // The array's type must match the original array's type.
      Array newArray = Array.CreateInstance(t, desiredSize);

      // Copy the elements from the original array into the new array.
      Array.Copy(origArray, 0,
         newArray, 0, Math.Min(origArray.Length, desiredSize));

      // Return the new array.
      return newArray;
   }
}
```

If you build and run this application, you'll see the following output:

```
1 2 3
1 2 3 0 0
1 2
```

# Chapter 15: Interfaces

In this chapter, I'll explain how interfaces identify functionality that can be tacked onto a type. I'll then show you how a type can implement an interface to offer this well–defined functionality, allowing the type to be used in various scenarios easily. Finally, I'll demonstrate useful techniques for avoiding problems when you use interfaces—namely, duplication of member names and compromised compile–time type safety.

## Interfaces and Inheritance

When programming, it's useful to think of an object as being of multiple types because the type of an object describes its capabilities and behavior. For example, you could design a **SortedList** type that maintains a set of objects in a sorted order. You could add any **System.Object**–derived type into the **SortedList** as long as the type supported the ability to compare itself to another type.

In a sense, the **SortedList** would like to accept only types that are derived from a hypothetical **System.Comparable** type. But many existing types aren't derived from **System.Comparable**. You can't add objects of these types to a **Sorted–List**, and a **SortedList** type becomes much less useful as a result.

Ideally, you'd want to take an existing **System.Object**–derived type and sometimes treat it as though it were a **System.Comparable**–derived type. The ability to treat an object as being of multiple types is frequently referred to as *multiple inheritance*. The common language runtime (CLR) supports single implementation inheritance and multiple interface inheritance.

The CLR allows a type to inherit from only one other type, which has **System.Object** as its root base type. This type of inheritance is called *implementation inheritance* because the derived type inherits all the behavior and capabilities of its base type: the derived type behaves exactly like the base type. Once the base type is inherited, however, the derived type can override the base type's behavior. This overriding of the base type's behavior (implementation) makes the new, derived type unique.

*Interface inheritance* means that a type inherits the method signatures of its interfaces but not their implementations. When a type inherits an interface, it is promising to provide its own implementations for the methods; if the type doesn't implement the interface's methods, the type is considered abstract and it won't be possible to construct an instance of the type.

Interfaces don't derive from any **System.Object**–derived type. An interface is simply an abstract type that consists of a set of virtual methods, each with its own name, parameters, and return type. Interface methods can't contain any implementation; hence, interface types are incomplete (abstract). Note that interfaces can also define events, parameterless properties, and parameterful properties (indexers in C#) because all of these are just syntax shorthands that map to methods anyway. The CLR also allows an interface to contain static methods, static fields, constants, and static constructors. However, a CLS–compliant interface must not have any of these static members because some programming languages aren't able to define or access them. In fact, C# prevents an interface from defining any static members. In addition, the CLR doesn't allow an interface to contain any instance fields or instance constructors.

Here are the definitions of four interfaces that are defined in the .NET Framework Class Library (FCL):

```
public interface System.IComparable {
```

```
   Int32 CompareTo(Object object);
}


public interface System.Collections.IEnumerable {
   IEnumerator GetEnumerator();
}


public interface System.Collections.IEnumerator {
   Boolean MoveNext();
   void Reset();

   Object Current { get; }  // Read-only property
}


public interface System.Collections.ICollection : IEnumerable {
   void CopyTo(Array array, Int32 index);

   Int32   Count          { get; } // Read-only property
   Boolean IsSynchronized { get; } // Read-only property
   Object  SyncRoot       { get; } // Read-only property
}
```

By convention, the name of an interface type is prefixed with an uppercase **I**. An interface definition can be marked with modifiers—such as **public**, **protected**, **internal**, and **private**—in the same way that a class or structure can be marked. Of course, **public** is used more than 99 percent of the time. These modifiers control the visibility of the interface definition and indicate which referents can see it.

**Important** Nonstatic members of an interface are always considered public and virtual. This can't be changed. However, in C#, if you implement an interface method in a type and omit the **virtual** keyword, the method is considered virtual *and* sealed—a type derived from the implementing type can't override the method.

Although an interface can't inherit another type's implementation, it can "inherit" the contract of other interfaces (as in the case of **ICollection** having **IEnumerable** as a base). In fact, an interface can include the contract of multiple interfaces. When a type "inherits" an interface, that type must implement all the methods defined by the interface and all the methods defined by any of the interface's "inherited" contracts as well. For example, any type that implements the **ICollection** interface must provide implementations for the **CopyTo**, **Count**, **IsSynchronized**, and **SyncRoot** members. In addition, the type must also provide an implementation for **IEnumerable**'s **GetEnumerator** method.

The **System.ICloneable** interface is defined (in MSCorLib.dll) as follows:

```
public interface ICloneable {
   Object Clone();
}
```

The following code shows how to define a type that implements this interface and also shows code that clones an object:

```
using System;

// Point is derived from System.Object and implements ICloneable.
public sealed class Point : ICloneable {
```

```
   public Int32 x, y;

   public Point(Int32 x, Int32 y) {
      this.x = x;
      this.y = y;
   }

   public override Boolean Equals(Object o) {
      Point other = o as Point;
      if (other == null) return false;
      return (x == this.x) && (y == this.y);
   }

   // This is the implementation for ICloneable's Clone.
   public Object Clone() { return MemberwiseClone(); }
}

class App {
   static public void Main() {
      Point p1 = new Point(1, 2);

      // Create another Point with the same values.
      Point p2 = (Point) p1.Clone();

      // p1 and p2 refer to two different objects: False is displayed.
      Console.WriteLine(Object.ReferenceEquals(p1, p2));

      // p1 and p2 have the same value: True is displayed.
      Console.WriteLine(p1.Equals(p2));
   }
}
```

As mentioned earlier, a type must inherit one type (even if it's **System.Object**, as in the preceding example). In addition, a type can implement zero or more interfaces. The FCL's **System.String** type, for example, inherits **System.Object**'s implementation and implements the **IComparable**, **ICloneable**, **IConvertible**,
and **IEnumerable** interfaces. This means that the **String** type isn't required to implement the methods its **Object** base type offers. If the **String** type chooses not to implement **Object**'s methods explicitly, then it simply inherits **Object**'s methods. However, the **String** type must implement the methods declared in all the interfaces; if it didn't, it would be an incomplete (abstract) type.

At the beginning of this section, I said that interfaces let you treat a single object as though it were of different types. Any type that implements interfaces allows its objects to be treated as any of its interfaces. See the following code for an example:

```
// Create a String object.
String s = "Jeffrey";
// Using s, I can call any method defined in
// String, Object, IComparable, ICloneable, IConvertible, IEnumerable.

// Make an IComparable variable that references s.
IComparable comparable = s;
// Using comparable, I can call any method declared by IComparable only.

// Make an ICloneable variable that references s.
ICloneable cloneable = s;
// Using cloneable, I can call any method declared by ICloneable only.

// Make an IEnumerable variable that references s.
IEnumerable enumerable = (IEnumerable) cloneable;
```

```
// You can cast a variable from one interface to another as long as
// the object's type implements both interfaces.
```

In this code, it doesn't matter which variable I use; I'm always affecting the **String** object identified by **s**. However, the variable's type indicates the legal actions that I can perform on that string object.

Let's return to the **SortedList** discussion. It's now possible to place a **String** object into the **SortedList** because it implements the **IComparable** interface. The only minor issue is that the **String** type is required to implement the **CompareTo** method (in **IComparable**), which needs to compare two **Object**s and return a value indicating which **Object** should come first.

**IComparable**'s **CompareTo** method takes an **Object** parameter, not a **String** parameter. Any type implementing the **CompareTo** method must define the method so that its signature matches that of the interface's method. In the method, the code can do any casting necessary to perform the expected behavior. I'll address this "shortcoming" of interfaces in the upcoming section "Explicit Interface Member Implementations."

A **SortedList** type can successfully manage objects of any type (**String**, **DateTime**, **Int32**—whatever) as long as the types implement the **IComparable** interface.

**Important** Like a reference type, a value type can implement zero or more interfaces. However, when you cast an instance of a value type to an interface type, the value type instance must be boxed. This is because interfaces are always considered reference types, and the methods they define are always **virtual**. Recall that unboxed value types don't have a pointer to the type's method table. Boxing the value type allows the CLR to look up the type's method table so that the virtual method can be called.

I often hear the question, "Should I design a base type or an interface?" The answer isn't always clear–cut. Here are some guidelines that might help you:

- **IS–A vs. CAN–DO relationship** A type can inherit only one implementation. If the derived type can't claim an IS–A relationship with the base type, then don't use a base type; use an interface. Interfaces imply a CAN–DO relationship. If the CAN–DO functionality feels like it belongs with various object types, use an interface.
- **Ease of use** It's easier for you, as a developer, to define a new type derived from a base type than to create an interface. The base type can provide a lot of functionality, so the derived type probably needs only relatively small modifications to its behavior. If you supply an interface, the new type must implement all the members.
- **Consistent implementation** No matter how well an interface contract is documented, it's very unlikely that everyone will implement the contract 100 percent correctly. In fact, COM suffers from this very problem, which is why some COM objects work correctly only with Microsoft Word or with Microsoft Internet Explorer. By providing a base type with a good default implementation, you start off using a type that works and is well tested; you can then modify whatever parts need changing.
- **Versioning** If you add a method to the base type, the derived type inherits the new member's default implementation for free. In fact, the user's source code doesn't even have to be recompiled. Adding a new member to an interface forces the user of the interface to modify the source code before recompiling.

In the FCL, the classes related to streaming data use an implementation inheritance design. The **System.IO.Stream** class is an abstract base class. It provides a bunch of methods, such as **Read** and **Write**. Other classes—**System.IO.–FileStream**, **System.IO.MemoryStream**, and **System.Net.Sockets.NetworkStream**—are derived from **Stream**. Microsoft chose an IS–A

relationship between each of these three classes and the **Stream** class because it made implementing the concrete classes easier. For example, the derived classes need to implement only synchronous I/O operations; they inherit the ability to perform asynchronous I/O operations from the **Stream** base class.

Admittedly, choosing to use inheritance for the stream classes isn't entirely clear–cut; the **Stream** base class actually provides very little implementation. However, if you consider the Windows Forms control classes, where **Button**, **CheckBox**, **ListBox**, and all the other controls are derived from **System.Windows.Forms.Control**, it's easy to imagine all the code that **Control** implements, which the various control classes simply inherit to function correctly.

By contrast, Microsoft designed the FCL collections to be interface based. The **System.Collections** namespace defines several collection–related interfaces: **IEnumerable**, **ICollection**, **IList**, and **IDictionary**. Then Microsoft provided a number of concrete classes, such as **ArrayList**, **Hashtable**, **Queue**, **SortedList**, and so on, that implement combinations of these interfaces. Here the designers chose a CAN–DO relationship between the classes and the interfaces because the implementations of these various collection classes are radically different from one another. In other words, there isn't a lot of sharable code between an **Array–List**, a **Hashtable**, and a **Queue**.

The operations that all these collection classes offer are, nevertheless, pretty consistent. For example, they all maintain a set of elements that can be enumerated, and they all allow adding and removing of elements. This consistency is the reason that making the collection classes interface based makes a lot of sense. If you have a reference to an object whose type implements the **IList** interface, you can write code to add elements, remove elements, and search for an element without having to know exactly what type of collection you're working with. This is a very powerful mechanism.

## Designing an Application That Supports Plug–In Components

When you're building extensible applications, interfaces should be the centerpiece. Suppose, for example, that you're writing an application and you want others to be able to create types that your application can load and use seamlessly. Here's the way to design this application.

- Create an assembly that defines an interface whose methods are used as the communication mechanism between the application and the plug–in components. When defining the parameters and return values for the interface methods, try to use other interfaces or types defined in MSCorLib.dll. If you want to pass and return your own data types, define them in this assembly too. Once you settle on your interface definitions, give this assembly a strong name (discussed in Chapter 3) and then package and deploy it to your partners and users. Consider this assembly immutable—that is, don't change its contents.

**Note** You can use types defined in MSCorLib.dll because the CLR always loads the version of MSCorLib.dll that matches the version of the CLR itself. Also, only a single version of MSCorLib.dll is ever loaded into a process. In other words, different versions of MSCorLib.dll never load side by side (as described in Chapter 3). As a result, you won't have any type version mismatches and your application will require less memory.

- Create a separate assembly containing your application's types. This assembly will, obviously, reference the interface and types defined in the first assembly. Feel free to modify

the code in this assembly to your heart's content. Because the plug-in developers won't reference this assembly, you can put out a new version of it every hour if you want to and not affect any of the plug-in developers.

- The plug-in developers will, of course, define their own types in their own assembly. Their assembly will also reference the types in your interface assembly. The plug-in developers are also able to put out a new version of their assembly as often as they'd like, and the application will be able to consume the plug-in types without any problem whatsoever.

This small section contains some very important information. When using types across assemblies, you need to be concerned with assembly versioning issues. Take your time, and isolate the types that you use for communication across assembly boundaries into their own assembly. Avoid mutating or changing these type definitions—and don't change the version number of the assembly.

Also, avoid defining a type whose base type is defined in another assembly, even though the .NET Framework Design Guidelines encourage this; it's the wrong thing to do, because assemblies will be tied to specific versions of other assemblies. As a consequence of the CLR's side-by-side support, you'll end up pulling several different versions of an assembly into a single AppDomain. This can cause significant memory usage and severely hurt performance. It might also prevent—or at least make difficult—communication between the assemblies.

In addition, most compilers don't let you reference multiple versions of a specific assembly when building a managed module. This limitation can make it difficult for code to take advantage of new features (offered by a newer version of an assembly) and still communicate (via required types in an older version of an assembly).

## Changing Fields in a Boxed Value Type Using Interfaces

Let's have some fun and see how well you understand value types, boxing, and unboxing. Examine the following code, and see whether you can figure out what it displays on the console:

```
using System;

// Point is a value type.
struct Point {
   public Int32 x, y;

   public void Change(Int32 x, Int32 y) {
      this.x = x; this.y = y;
   }

   public override String ToString() {
      return String.Format("({0}, {1})", x, y);
   }
}

class App {
   static void Main() {
      Point p = new Point();

      p.x = p.y = 1;
      Console.WriteLine(p);  // Displays (1, 1)

      p.Change(2, 2);
      Console.WriteLine(p);  // Displays (2, 2)
```

```
      Object o = p;
      Console.WriteLine(o);  // Displays (2, 2)

      ((Point) o).Change(3, 3); // Changes temporary Point on stack!
      Console.WriteLine(o);  // Displays (2, 2)
   }
}
```

Very simply, **Main** creates an instance of a **Point** value type on its stack and then changes its **x** and **y** fields to **1**. The first call to **WriteLine** calls **ToString** on the unboxed **Point**, and "(1, 1)" is displayed, as expected. Then, **p** is used to call the **Change** method, which changes the values of **p**'s **x** and **y** fields on the stack to **2**. The second call to **WriteLine** displays "(2, 2)", as expected.

Now, **p** is boxed, and **o** refers to the boxed **Point** object. The third call to **WriteLine** again shows "(2, 2)", which is also expected. Finally, I want to call the **Change** method to update the fields in the boxed **Point** object. However, **Object** (the type of the variable **o**) doesn't know anything about the **Change** method, so I must first cast **o** to a **Point**. Casting **o** to a **Point** unboxes **o** and copies the fields in the boxed **Point** to a temporary **Point** on the thread's stack. The **x** and **y** fields of this temporary point are changed to 3 and 3, but the boxed **Point** isn't affected by this call to **Change**. When **WriteLine** is called the second time, "(2, 2)" is displayed again. Many developers do *not* expect this.

Some languages, such as C++ with Managed Extensions, let you change the fields in a boxed value type, but C# does not. However, you can fool C# into allowing this by using an interface. The following code is a modified version of the previous code:

```
using System;

// Interface defining a Change method
interface IChangeBoxedPoint {
   void Change(Int32 x, Int32 y);
}


// Make the value type implement the interface.
struct Point : IChangeBoxedPoint {
   public Int32 x, y;

   public void Change(Int32 x, Int32 y) {
      this.x = x; this.y = y;
   }

   public override String ToString() {
      return String.Format("({0}, {1})", x, y);
   }
}


class App {
   static void Main() {
      Point p = new Point();

      p.x = p.y = 1;
      Console.WriteLine(p);  // Displays (1, 1)

      p.Change(2, 2);
      Console.WriteLine(p);  // Displays (2, 2)

      Object o = p;
```

```
      Console.WriteLine(o);  // Displays (2, 2)

      ((Point) o).Change(3, 3); // Changes temporary Point on stack!
      Console.WriteLine(o);  // Displays (2, 2)

      // Boxes p, changes the boxed object and discards it
      ((IChangeBoxedPoint) p).Change(4, 4);
      Console.WriteLine(p);  // Displays (2, 2)

      // Changes the boxed object and shows it
      ((IChangeBoxedPoint) o).Change(5, 5);
      Console.WriteLine(o);  // Displays (5, 5)
   }
}
```

This code is almost identical to the previous version. The main difference is that the **Change** method is defined by the **IChangeBoxedPoint** interface, and the **Point** type now implements this interface. Inside **Main**, the first four calls to **WriteLine** are the same and produce the same results I had before (as expected). However, I've added two more examples at the end of **Main**.

In the first example, the unboxed **Point**, **p**, is cast to an **IChangeBoxedPoint**. This cast causes the value in **p** to be boxed. **Change** is called on the boxed value, which does change its **x** and **y** fields to 4 and 4, but after **Change** returns, the boxed object is immediately ready to be garbage collected. So the fifth call to **WriteLine** displays "(2, 2)"—many developers won't expect this result.

In the last example, the boxed **Point** referred to by **o** is cast to an **IChangeBoxedPoint**. No boxing is necessary here because **o** is already a boxed **Point**. Then **Change** is called, which *does* change the boxed **Point**'s **x** and **y** fields. The interface method **Change** has allowed me to change the fields in a boxed **Point** object! Now, when **WriteLine** is called, it displays "(5, 5)", as expected.

The purpose of this whole example is to demonstrate how an interface method is able to modify the fields of a boxed value type. In C#, this isn't possible without using an interface method.

**Important** A number of developers reviewed the chapters of this book. After reading through some of my code samples (such as the preceding one), these reviewers would tell me that they've sworn off value types. I must say that these little value type nuances have cost me days of debugging time, which is why I spend time pointing them out in this book. I hope you'll remember some of these nuances and that you'll be prepared for them if and when they strike you and your code. Certainly, you shouldn't be scared of value types. They are useful types, and they have their place. After all, a program needs a little **Int32** love now and then. Just keep in mind that value types and reference types have very different behaviors, depending on how they're used. In fact, you should take the preceding code and declare the **Point** as a **class** instead of a **struct** to appreciate the results.

## Implementing Multiple Interfaces That Have the Same Method

Defining a type that implements an interface is usually easy and straightforward. You simply implement methods in the type that match the methods and signatures defined by the interface. You do have to remember to make the methods public, but there's nothing special about the code in the implementation of an interface method. And calling an interface–defined method is easy; you can just call the method as if it were any other method defined by the type.

Occasionally, you might find yourself defining a type that implements multiple interfaces that define methods with the same name and signature. For example, imagine that there are two interfaces defined as follows:

```
public interface IWindow {
   Object GetMenu();
}

public interface IRestaurant {
   Object GetMenu();
}
```

Let's say that you want to define a type that implements both these interfaces. You'd have to implement the type's method as follows:

```
// This type is derived from System.Object and
// implements the IWindow and IRestaurant interfaces.
public class GiuseppePizzaria : IWindow, IRestaurant {

   // This is the implementation for IWindow's GetMenu method.
   Object IWindow.GetMenu() { ... }

   // This is the implementation for IRestaurant's GetMenu method.
   Object IRestaurant.GetMenu() { ... }

   // This is a GetMenu method that has nothing to do with an interface.
   public Object GetMenu() { ... }
}
```

Because this type must implement multiple and separate **GetMenu** methods, you need to tell the C# compiler which **GetMenu** method contains the implementation for a particular interface. In C#, you qualify the method name by preceding it with the name of the interface itself. So, in the previous example, the **IWindow.GetMenu** method signals to the compiler that this method contains the implementation for **IWindow**'s **GetMenu** method. Similarly, the **IRestaurant.GetMenu** method signals that this method contains the implementation for **IRestaurant**'s **GetMenu** method. The unqualified **GetMenu** method simply identifies a method that the type defines; this method isn't related to any interface.

Note that the interface methods are not declared as **public**. The reason they're not is that these methods lead a double life: sometimes they're public, and sometimes they're private. The following code will make this distinction clear to you:

```
static void SomeMethod() {
   // Construct an instance of the type.
   GuiseppePizzaria gp = new GuiseppePizzaria();

   Object menu;

   // Call the type's public GetMenu method.
   // Using a reference to a GuiseppePizzaria, the
   // interface methods are private and can't be called.
   menu = gp.GetMenu());

   // Call IWindow's GetMenu method.
   // Using a reference to an IWindow, IWindow's GetMenu
   // method is the ONLY method that can be called.
   menu = ((IWindow) gp).GetMenu();

   // Call IRestaurant's GetMenu method.
```

```
   // Using a reference to an IRestaurant, IRestaurant's GetMenu
   // method is the ONLY method that can be called.
   menu = ((IRestaurant) gp).GetMenu();
}
```

When you define an interface method in a type using an interface-qualified name, the method is considered private and can't be called using a variable that is a reference to the type itself. However, when you cast the reference to a specific interface, the only methods that are callable are the methods defined by that interface. So, when the **gp** variable is cast to an **IWindow**, the only method callable is **IWindow**'s **GetMenu** method. Likewise for the **IRestaurant** interface.

As I said, a type very rarely implements multiple interfaces that define the same method, so you rarely need to do what I've done in the preceding code. However, this technique of qualifying a method with an interface name is interesting and has other uses. In fact, I'll focus on this technique again in the next section.

## Explicit Interface Member Implementations

Interfaces are great because they define a standard way for types to communicate with each other. However, this flexibility comes at the cost of compile-time type safety because most interface methods accept parameters of type **System.Object** or return a value whose type is **System.Object**. Look at the very common **IComparable** interface:

```
public interface IComparable {
   Int32 CompareTo(Object other);
}
```

This interface defines one method that accepts a parameter of type **System.Object**. If I define my own type that implements this interface, the type definition might look like this:

```
struct SomeValueType : IComparable {
   private Int32 x;
   public SomeValueType(Int32 x) { this.x = x; }
   public Int32 CompareTo(Object other) {
      return(x - ((SomeValueType) other).x);
   }
}
```

Using **SomeValueType**, I can now write the following code:

```
static void Main() {
   SomeValueType v = new SomeValueType(0);
   Object o = new Object();
   Int32 n = v.CompareTo(o);
}
```

In this code, I'm comparing apples and oranges. OK, I'm really comparing **Some-ValueType**s and **Object**s, but you see that this code doesn't make a lot of sense because **Object** doesn't have an **x** field. The compiler can't detect the flaw in the logic, however, and it produces code—no warnings or errors are generated. But at run time, when **CompareTo** is called, an **InvalidCastException** is thrown when **other** is cast to **SomeValueType**.

As always, developers prefer compile-time errors to run-time errors, and the CLR offers a feature called *explicit interface member implementations* that makes compile-time type checking possible.

Let's modify **SomeValueType** to use an explicit interface member implementation:

```
struct SomeValueType : IComparable {
   private Int32 x;
   public SomeValueType(Int32 x) { this.x = x; }

   public Int32 CompareTo(SomeValueType other) {
      return(x - other.x);
   }

   // NOTE: No public/private used on the next line
   Int32 IComparable.CompareTo(Object other) {
      return CompareTo((SomeValueType) other);
   }
}
```

Notice several things about this new version. First, it now has two **CompareTo** methods. The first **CompareTo** method no longer takes an **Object** as a parameter; it now takes a **SomeValueType** instead. Because this parameter has changed, the code that casts **other** to **SomeValueType** is no longer necessary and has been removed. The resulting code is easier to maintain and is also type–safe at compile time. The following code demonstrates:

```
SomeValueType v1 = new SomeValueType(1);
SomeValueType v2 = new SomeValueType(2);
Int32 n;

n = v1.CompareTo(new Object()); // Compile-time error
n = v1.CompareTo(v2);           // Calls CompareTo
```

Changing the first **CompareTo** method so that it's type–safe means that **Some–ValueType** no longer adheres to the contract placed on it by implementing the **IComparable** interface. So **SomeValueType** must implement a **CompareTo** method that satisfies the **IComparable** contract. This is the job of the second **IComparable.–CompareTo** method, which is an explicit interface member implementation.

This **IComparable.CompareTo** method returns an **Int32** and accepts a **System.Object**, just like the method the **IComparable** interface defines. However, you need to keep in mind three characteristics of this **CompareTo** method. First, the name of the method is qualified with the name of the interface: **IComparable.CompareTo**. This is very important. Basically, the interface name tells the CLR that **IComparable.CompareTo** should be called only when using a reference to an **IComparable** object. To understand this constraint clearly, examine the following code:

```
SomeValueType v1 = new SomeValueType(1);
SomeValueType v2 = new SomeValueType(2);
Int32 n;

n = v1.CompareTo(v2);           // Calls CompareTo
n = v2.CompareTo(v1);           // Calls CompareTo

// NOTE: Getting an IComparable reference forces v1 to be boxed.
IComparable comparable = (IComparable) v1;

// NOTE: This calls IComparable.CompareTo, which takes an Object as
// a parameter. This forces v2 to be boxed.
n = comparable.CompareTo(v2);
```

Second, **IComparable.CompareTo** is implemented by calling the other **CompareTo** method after casting **other** to a **SomeValueType**. This means that the existing code can be leveraged.

Finally, notice that the **IComparable.CompareTo** method isn't prefixed with a **public** or **private** access modifier. In fact, the C# compiler produces one of the following errors if a **public** or **private** access modifier is added:

```
error CS0106: The modifier 'public' is not valid for this item
error CS0106: The modifier 'private' is not valid for this item
```

The fact that **IComparable.CompareTo** can't have a **public** or **private** access modifier tells you a lot about explicit interface method implementations. When you're working with an instance of a **SomeValueType**, the **IComparable.CompareTo** method is private; it's not possible to call this method. Using **v1** or **v2** from the preceding code, the only **CompareTo** method that is accessible is the type–safe version that takes a **SomeValueType** parameter.

When you have a reference to an **IComparable** object, however, the **IComparable.CompareTo** method is public. In fact, it's the only method accessible because an **IComparable** reference variable can be used to call methods from the **IComparable** interface only. So the **IComparable.CompareTo** explicit interface member implementation is sometimes private and sometime public—that's why the compiler doesn't allow you to explicitly provide access modifiers for it.

The purpose of explicit interface member implementations is to provide for more type safety during application development. Although my example uses a value type, this mechanism can also be applied to reference types to improve their type safety. In addition to improving type safety, another beneficial side effect occurs when an explicit interface member implementation is provided on a value type, which **SomeValueType** happens to be.

Because **IComparable**'s **CompareTo** method takes an **Object** as a parameter, you can pass a value type for this parameter, but it must be boxed. As you know, boxing allocates memory from the heap, copies fields, and hurts performance, so avoid it whenever possible. Fortunately, explicit interface member implementations allow you to get rid of unwanted boxing operations. To understand how, examine the following code:

```
public static void Main() {
   SomeValueType v1 = new SomeValueType(1);
   SomeValueType v2 = new SomeValueType(2);
   Int32 n;

   n = v1.CompareTo(v2);                  // No boxing
   n = ((IComparable) v1).CompareTo(v2)); // Boxes v1 and v2
}
```

In this code, the first call to **CompareTo** calls the fully type–safe **CompareTo** method. Because this method isn't part of an interface and because it takes a **SomeValueType** as a parameter, it can simply be called on the unboxed instance **v1**. In addition, **v2** doesn't have to be boxed and can be passed directly to the method.

On the second call to **CompareTo**, I first cast **v1** to an **IComparable**. This boxes **v1** and returns an **IComparable** reference, which I then use to call **IComparable.CompareTo**. This version of **CompareTo** is the explicit interface method implementation, which takes a **System.Object** as a parameter. So **v2** must be boxed before **IComparable.CompareTo** can be called. Two boxing operations have occurred!

Explicit interface method implementations are frequently used when implementing interfaces such as **ICloneable**, **IComparable**, **ICollection**, **IList**, and **IDictionary**. They let you create type–safe

versions of these interface's methods, and they enable you to reduce boxing operations for value types.

---

Be Careful with Explicit Interface Method Implementations

When examining the methods for a type in the .NET Framework reference documentation, explicit interface method implementations do not appear. Their absence has confused many developers. For example, if you look up the **System.Int32** type in the reference documentation, you'll see that **Int32** implements the **IConvertible** interface. However, if you look at the "Int32 Members" help page, you won't see any of **IConvertible**'s methods (**ToBoolean**, **ToByte**, **ToChar**, **ToSingle**, and so on) listed. Why?

When the .NET Framework was in beta, Microsoft noticed that the **Int32** type offered around 20 methods and they were afraid that developers would quickly become overwhelmed when looking at the documentation for a "simple **Int32** type." So Microsoft decided not to show explicit interface method implementations in the documentation to keep the documentation less cluttered. Then, of course, they had to change some of the type's methods to explicit interface method implementations so that these methods wouldn't appear. For the **Int32** type, they chose to make the methods that implement the **IConvertible** interface explicit interface method implementations. Because **IConvertible** defines 15 methods, the documentation for the **Int32** type now shows only 5 methods. This certainly makes the help look less cluttered and makes the **Int32** type look much more like a "simple type."

However, I've run into many developers who have just been confused by this decision. They see that the **Int32** type implements **IConvertible**, but the documentation doesn't reflect that these methods exist. And to make matters even worse, you can't call an **IConvertible** method on an **Int32** directly. For example, the following method won't compile:

```
static void Main() {
   Int32 x = 5;
   Single s = x.ToSingle(null);
}
```

When compiling this method, the C# compiler produces the following: "error CS0117: 'int' does not contain a definition for 'ToSingle'." This error message confuses the developer even more because it's clearly stating that the **Int32** type doesn't define a **ToSingle** method when, in fact, it does.

To call **ToSingle** on an **Int32**, you must first cast the **Int32** to an **IConvertible,** as shown in the following method:

```
static void Main() {
   Int32 x = 5;
   Single s = ((IConvertible) x).ToSingle(null);
}
```

Requiring this cast isn't obvious at all, and many developers won't figure this out on their own. But an even more troublesome problem exists: casting the **Int32** value type to an **IConvertible** also boxes the value type, wasting memory and hurting performance.

Was all this really necessary just to reduce documentation clutter? I don't think so. In my opinion, Microsoft should have left the documentation alone—it would have been accurate, complete, and less confusing, and the resulting code would be obvious and easier to write, and it would execute efficiently. Remember, my example here talks about the **Int32** type and how it implements **IConvertible** methods. But Microsoft chose to "hide documentation" for many types that implement

various interfaces.

This discussion clearly shows you that explicit interface implementation methods should be used with great care. When many developers first learn about explicit interface implementation methods, they think they're cool and they start using them whenever possible. Don't do this! Explicit interface implementation methods are useful in some circumstances but you should avoid them whenever possible since they make using a type much less obvious.

---

# Chapter 16: Custom Attributes

In this chapter, I'll discuss one of the most innovative features that the Microsoft .NET Framework has to offer: *custom attributes*. Custom attributes allow anyone (not just Microsoft) to define information that can be applied to almost any metadata table entry. This extensible metadata information can be queried at run time to dynamically alter the way code executes. As you use the various .NET Framework technologies (Windows Forms, Web Forms, XML Web services, and so on), you'll see that they all take advantage of custom attributes, allowing developers to express their intentions in code very easily. A solid understanding of custom attributes is necessary for any .NET Framework developer.

## Using Custom Attributes

Attributes, such as **public**, **private**, **static**, and so on, can be applied to types and members. I think we'd all agree on the usefulness of applying attributes, but wouldn't it be even more useful if we could define our own attributes? For example, what if I could define a type and somehow indicate that the type can be remoted via serialization? Or maybe I could apply an attribute to a method to indicate that certain security permissions must be granted before the method can execute.

Of course, creating and applying user–defined attributes to types and methods would be great and convenient, but it would require the compiler to be aware of these attributes so that it would emit the attribute information into the resulting metadata. Because compiler vendors usually prefer not to release the source code for their compiler, Microsoft came up with another way to allow user–defined attributes. This mechanism, called *custom attributes*, is an incredibly powerful mechanism that's useful at both application design time and run time. Anyone can define and use custom attributes, and all compilers that target the common language runtime (CLR) must be designed to recognize custom attributes and emit them into the resulting metadata.

The first thing you should realize about custom attributes is that they're just a way to associate additional information with a target. The compiler emits this additional information into the managed module's metadata. Most attributes have no meaning for the compiler; the compiler simply detects the attributes in the source code and emits the corresponding metadata.

The .NET Framework Class Library (FCL) ships with many predefined attributes. For example, the **System.FlagsAttribute** attribute causes an enumerated type to act like a set of bit flags, the **System.SerializableAttribute** attribute allows a type's fields to be serialized and deserialized (typically used for remoting of a method's arguments and return value), several security–related attributes enable a method to ensure that it has a required privilege granted before attempting some particular kind of access, lots of interoperability–related attributes serve to allow managed code to call unmanaged code, and so on.

Following is some C# code with many attributes applied to it. It's not important to understand what this code does. I just want you to see what attributes look like.

```
[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Auto)]
class OSVERSIONINFO {
   public OSVERSIONINFO() {
      OSVersionInfoSize = (UInt32) Marshal.SizeOf(this);
   }

   public UInt32 OSVersionInfoSize = 0;
   public UInt32 MajorVersion      = 0;
   public UInt32 MinorVersion      = 0;
```

```
   public UInt32 BuildNumber       = 0;
   public Uint32 PlatformId        = 0;

   [MarshalAs(UnmanagedType.ByValTStr, SizeConst=128)]
   public String CSDVersion        = null;
}

class MyClass {
   [DllImport("Kernel32", CharSet=CharSet.Auto, SetLastError=true)]
   public static extern Boolean GetVersionEx(
      [In, Out] OSVERSIONINFO ver);
}
```

In C#, you apply a custom attribute to a target by placing the attribute in square brackets immediately before the target. So, in this case, the **StructLayout** attribute is applied to the **OSVERSIONINFO** class, the **MarshalAs** attribute is applied to the **CSDVersion** field, the **DllImport** attribute is applied to the **GetVersionEx** method, and the **In** and **Out** attributes are applied to **GetVersionEx**'s **ver** parameter. Every programming language defines the syntax that a developer must use in order to apply a custom attribute to a target. Microsoft Visual Basic, for example, requires angle brackets (<, >) instead of square brackets.

The CLR allows attributes to be applied to just about anything that can be represented in a file's metadata. Most commonly, attributes are applied to entries in the following definition tables: TypeDef (class, struct, enum, interfaces, and delegates), MethodDef (including constructors), ParamDef, FieldDef, PropertyDef, EventDef, AssemblyDef, and ModuleDef. Although it's rare, attributes can also be applied to entries in reference tables, such as AssemblyRef, ModuleRef, TypeRef, and MemberRef. Finally, custom attributes can be applied to other pieces of metadata—such as security permissions, exported types, and resources.

Although the CLR allows custom attributes to be applied to any of these entities, most programming languages allow attributes to be applied only to entries in the various definition metadata tables. This is certainly true of Microsoft's C# compiler. Specifically, C# allows you to apply an attribute only to source code that defines any of the following targets: assembly, module, type, field, method, method parameter, method return value, property, and event.

When you're applying an attribute, C# allows you to specify a prefix specifically indicating what target the attribute applies to. The following code shows all the possible prefixes. In many cases, if you leave out the prefix, the compiler can still determine what target an attribute applies to, as shown in the previous example. However, explicitly specifying a prefix removes all ambiguity.

```
using System;

[assembly: MyAttribute(1)]    // Applied to the assembly
[module:   MyAttribute(2)]    // Applied to the module

[type:     MyAttribute(3)]    // Applied to the type
class SomeType {

   [property: MyAttribute(4)] // Applied to the property
   public String SomeProp { get { return null; } }

   [event: MyAttribute(5)]    // Applied to the event
   public event EventHandler SomeEvent;

   [field: MyAttribute(6)]    // Applied to the field
   public Int32 SomeField = 0;

   [return: MyAttribute(7)]   // Applied to the return value
```

```
   [method: MyAttribute(8)]   // Applied to the method
   public Int32 SomeMethod(
      [param: MyAttribute(9)] // Applied to the parameter
      Int32 SomeParam) { return SomeParam; }
}
```

Now that you know how to apply a custom attribute, let's find out what an attribute really is. A custom attribute is simply an instance of a type. For Common Language Specification (CLS) compliance, custom attribute types must be derived, directly or indirectly, from **System.Attribute**. C# allows only CLS–compliant attributes. By examining the .NET Framework SDK documentation, you'll see that the following types (from the earlier example) are defined: **StructLayoutAttribute**, **MarshalAsAttribute**, **DllImportAttribute**, **InAttribute**, and **OutAttribute**. All these types happen to be defined in the **System.Runtime.InteropService** namespace, but attribute types can be defined in any namespace. Upon further examination, you'll notice that all these types are derived from **System.Attribute**, as all CLS–compliant attribute types must be.

As I mentioned, an attribute is an instance of a type. The type must have a public constructor to create an instance of it. So, when you apply an attribute to a target, the syntax is similar to that for calling one of the type's instance constructors. In addition, a language might permit some special syntax that allows you to set any public fields or properties associated with the attribute type. Let's look at an example. Recall the application of the **DllImport** attribute as it was applied to the **GetVersionEx** method earlier:

```
[DllImport("Kernel32", CharSet=CharSet.Auto, SetLastError=true)]
```

The syntax of this line should look pretty strange to you because you could never use syntax like this when calling a constructor. If you examine the **DllImportAttribute** type in the documentation, you'll see that its constructor requires a single **String** parameter. In this example, "Kernel32" is being passed for this parameter. A constructor's parameters are called *positional parameters* and are mandatory: the parameter must be specified when the attribute is applied.

What are the other two "parameters"? This special syntax allows you to set any public fields or properties of the **DllImportAttribute** object after the object is constructed. In this example, when the **DllImportAttribute** object is constructed and "Kernel32" is passed to the constructor, the object's public instance fields, **CharSet** and **SetLastError**, are set to **CharSet.Auto** and **true**, respectively. The "parameters" that set fields or properties are called *named parameters* and are optional: the parameters don't have to be specified when you're applying an instance of the attribute. A little later on, I'll explain what causes an instance of the **DllImportAttribute** type to actually be constructed.

Also note that it's possible to apply multiple attributes to a single target. For example, the **GetVersionEx** method's **ver** parameter has both the **In** and **Out** attributes applied to it. When applying multiple attributes to a single target, be aware that the order of attributes has no significance. Also, in C# each attribute can be enclosed in square brackets or multiple attributes can be comma–separated within a single set of square brackets. The **Attribute** suffix is optional, and if the attribute type's constructor takes no parameters, the parentheses are optional. The following lines behave identically and demonstrate all the possible ways of applying multiple attributes:

```
[Serializable][Flags]
[Serializable, Flags]
[FlagsAttribute, SerializableAttribute]
[FlagsAttribute()][Serializable()]
```

## Defining Your Own Attribute

You know that an attribute is a type derived from **System.Attribute**, and you also know how to apply an attribute. Let's now look at how to define your own custom attributes. Say that you're the Microsoft employee responsible for adding the bit flag support to enumerated types. To accomplish this, the first thing you have to do is define a **FlagsAttribute** type:

```
namespace System {
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

Notice that the **FlagsAttribute** type inherits from **Attribute;** this is what makes the **FlagsAttribute** type a CLS–compliant custom attribute. In addition, all nonabstract attributes must have **public** accessibility, and by convention all attribute type names should end with "**Attribute**". Finally, all nonabstract attributes must contain at least one **public** constructor. The simple **FlagsAttribute** constructor takes no parameters and does absolutely nothing.

So far, instances of the **FlagsAttribute** class can be applied to any target, but really this attribute should be applied to enumerated types only. It doesn't make sense to apply the attribute to a property or a method. To tell the compiler where this attribute can legally be applied, you apply an instance of the **System.AttributeUsageAttribute** class to the attribute type. Here's the new code:

```
namespace System {
   [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

In this new version, I've applied an instance of **AttributeUsageAttribute** to the attribute. After all, the attribute type is just a class and a class can have attributes applied to it. The **AttributeUsageAttribute** attribute is a simple type that allows you to indicate to a compiler where your custom attribute can legally be applied. All compilers have built–in support for this attribute and generate errors when a user–defined custom attribute is applied to an invalid target. In this example, the **AttributeUsage** attribute indicates that instances of the **Flags** attribute can be applied to enumerated type targets only.

Because all attributes are just types, you can easily understand the **AttributeUsageAttribute** type. Here's what the FCL source code for the type looks like:

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
[Serializable]
public sealed class AttributeUsageAttribute : Attribute {
   internal AttributeTargets m_attributeTarget = AttributeTargets.All;
   internal Boolean          m_allowMultiple   = false;
   internal Boolean          m_inherited       = true;

   public AttributeUsageAttribute(AttributeTargets validOn) {
      m_attributeTarget = validOn;
   }

   public AttributeTargets ValidOn {
```

```
      get { return m_attributeTarget; }
   }

   public Boolean AllowMultiple {
      get { return m_allowMultiple;  }
      set { m_allowMultiple = value; }
   }

   public Boolean Inherited {
      get { return m_inherited;  }
      set { m_inherited = value; }
   }
}
```

As you can see, the **AttributeUsageAttribute** type has a constructor that allows you to pass bit flags indicating where your attribute can legally be applied. The **System.AttributeTargets** enumerated type is defined in the FCL as follows:

```
[Flags, Serializable]
public enum AttributeTargets {
   Assembly      = 0x0001,
   Module        = 0x0002,
   Class         = 0x0004,
   Struct        = 0x0008,
   Enum          = 0x0010,
   Constructor   = 0x0020,
   Method        = 0x0040,
   Property      = 0x0080,
   Field         = 0x0100,
   Event         = 0x0200,
   Interface     = 0x0400,
   Parameter     = 0x0800,
   Delegate      = 0x1000,
   ReturnValue   = 0x2000,
   All           = Assembly    | Module    | Class    | Struct | Enum  |
                   Constructor | Method    | Property | Field  | Event |
                   Interface   | Parameter | Delegate | ReturnValue
}
```

The **AttributeUsageAttribute** class offers two additional public properties that can optionally be set when the attribute is applied to an attribute type: **AllowMultiple** and **Inherited**.

For most attributes, it makes no sense to apply them to a single target more than once. For example, nothing is gained by applying the **Flags** or **Serializable** attributes more than once to a single target. In fact, if you tried to compile the code below, the compiler would report the following: "error CS0579: Duplicate ‘Flags’ attribute":

```
[Flags][Flags]
enum Color {
   Red
}
```

For a few attributes, however, it does make sense to apply the attribute multiple times to a single target. In the FCL, the **ConditionalAttribute** attribute class and lots of permission attribute classes (such as **EnvironmentPermissionAttribute**, **FileIOPermissionAttribute**, **ReflectionPermissionAttribute**, **Registry–PermissionAttribute**, and so on) allow multiple instances of themselves to be applied to a single target. If you don’t explicitly set **AllowMultiple**, your attribute will get the default behavior, which allows it to be applied no more than once to a selected target.

**AttributeUsageAttribute**'s other property, **Inherited**, indicates whether the purpose of the attribute should be considered to apply to derived classes or derived methods. Of all the attributes defined by the FCL, less than five have the **Inherited** property set to **true**. The following code demonstrates what it means for an attribute to be inherited:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
   Inherited=true)]
class TastyAttribute : Attribute {
}

[Tasty][Serializable]
public class SomeType {

   [Tasty] public virtual void DoSomething() { }
}

public class AnotherType : SomeType {
   public override void DoSomething() { }
}
```

In this code, **AnotherType** and its **DoSomething** method are both considered **Tasty** because **Tasty** attributes are marked as inherited. However, **AnotherType** is not serializable because the FCL's **SerializableAttribute** type is marked as a noninherited attribute.

Be aware that the .NET Framework only considers targets of classes, methods, properties, events, and parameters to be inheritable. So when you're defining an attribute type, you should set **Inherited** to **true** only if your targets include any of these targets. Note that inherited attributes do not cause additional metadata to be emitted for the derived types into the managed module. I'll say more about this a little later in the section "Detecting the Use of a Custom Attribute."

**Note** If you define your own attribute class and forget to apply an **AttributeUsage** attribute to your class, the compiler and the CLR will assume that your attribute can be applied to all targets, can be applied only once to a single target, and is inherited. These assumptions mimic the default field values in the **AttributeUsageAttribute** type.

## Attribute Constructor and Field/Property Data Types

When defining your own custom attribute type, you can define a constructor that takes parameters that a developer applying an instance of your attribute type must specify. In addition, you can define nonstatic, public fields and properties in your type that identify settings a developer applying an instance of your attribute type can optionally specify.

When defining an attribute type's instance constructor, fields, and properties, you must restrict yourself to a small subset of data types. Specifically, the legal set of data types is limited to any of the following: **Boolean**, **Char**, **Byte**, **SByte**, **Int16**, **UInt16**, **Int32**, **UInt32**, **Int64**, **UInt64**, **Single**, **Double**, **String**, **Type**, **Ob-ject**, or an enumerated type. In addition, you can use a single-dimension, zero-based array of any of these types.

When applying an attribute, you must pass a compile-time constant expression that matches the type defined by the attribute class. Wherever the attribute class defines a **Type** parameter, **Type** field, or **Type** property, you must use C#'s **typeof** operator as shown in the following code. Wherever the attribute class defines an **Object** parameter, **Object** field, or **Object** property, you can pass an **Int32**, a **String**, or any other constant expression. If the constant expression represents a

value type, the value type will be boxed at run time when an instance of the attribute is constructed.

Here's an example of an attribute and an application of it:

```
[AttributeUsage(AttributeTargets.All)]
class SomeAttribute : Attribute {
   public SomeAttribute(String name, Object o, Type[] types) {
      // 'name'  refers to a String.
      // 'o'     refers to one of the legal types (boxing if necessary).
      // 'types' refers to a 1-dimension, 0-based array of Types.
      Ã
   }
}

[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
public class SomeType {
   ...
}
```

Logically, when a compiler detects a custom attribute applied to a target, the compiler constructs an instance of the attribute type by calling its constructor, passing it any specified parameters. Then the compiler initializes any public fields and properties that have also been specified. Now that the custom attribute object is initialized, the compiler serializes the object out to the target's metadata table entry.

**Important** I've found that this is the best way for developers to think of custom attributes: instances of types that have been serialized to a byte stream that resides in metadata. Later, at run time, an instance of the type can be constructed by deserializing the bytes contained in the metadata.

**Note** Each parameter is written out with a 1–byte type ID followed by the value. After "serializing" the constructor's parameters, the compiler emits each of the specified field and property values by writing out the field/property name followed by a 1–byte type ID and then the value. For arrays, the count of elements is saved first, followed by each individual element.

## Detecting the Use of a Custom Attribute

Defining an attribute type is useless by itself. Sure, you could define attribute types all you want and apply them all you want; but this would just cause additional metadata to be written out to the managed module—the behavior of your application code wouldn't change.

In Chapter 13, you saw that applying the **Flags** attribute to an enumerated type altered the behavior of **System.Enum**'s **ToString**, **Format**, and **Parse** methods. The reason these methods behave differently is that they check at run time whether the enumerated type they're operating on has the **Flags** attribute metadata associated with it. Code can look for the presence of attributes using a technology called *reflection*. I'll give some brief demonstrations of reflection here, but I'll discuss it fully in Chapter 20.

If you were the Microsoft employee responsible for implementing **Enum**'s **Format** method, you would implement it like this:

```
public static String Format(Type enumType, Object value, String format) {

   // Does the enumerated type have an instance of
```

```
   // the FlagsAttribute type applied to it?
   if (enumType.IsDefined(typeof(FlagsAttribute), false)) {
      // Yes; execute code treating value as a bit flag enumerated type.
      Ã
   } else {
      // No;  execute code treating value as a normal enumerated type.
      Ã
   }
   Ã
}
```

This code calls **Type**'s **IsDefined** method, effectively asking the system to look up the metadata for the enumerated type and see whether an instance of the **FlagsAttribute** type is associated with it. If **IsDefined** returns **true**, then an instance of **FlagsAttribute** is associated with the enumerated type and the **Format** method knows to treat the value as though it contained a set of bit flags. If **Is–Defined** returns **false**, then **Format** treats the value as a normal enumerated type.

So if you define your own attribute types, you must also implement some code that checks for the existence of an instance of your attribute type (on some target) and then executes some alternate code path. This is what makes custom attributes so useful!

The FCL offers many ways to check for the existence of an attribute. If you're checking for the existence of an attribute via a **System.Type** object, you can use the **IsDefined** method as shown earlier. However, sometimes you want to check for an attribute on a target other than a type, such as an assembly, a module, or a method. For this discussion, let's concentrate on the methods defined by the **System.Attribute** type. You'll recall that all CLS–compliant attributes are derived from **System.Attribute** and this type defines three static methods for retrieving the attributes associated with a target: **IsDefined**, **GetCustomAttributes**, and **GetCustomAttribute**. Each of these functions has several overloaded versions. For example, each method has a version that works on type members (classes, structs, enums, interfaces, delegates, constructors, methods, properties, fields, events, and return types), parameters, modules, and assemblies. There are also versions that allow you to tell the system to walk up the derivation hierarchy to include inherited attributes in the results. Table 16–1 briefly describes what each method does.

If you just want to see whether an attribute has been applied to a target, you should call **IsDefined** because it's much faster than the other two methods. However, you know that when an attribute is applied to a target, you can specify parameters to the attribute's constructor and you can also optionally set fields and properties. Using **IsDefined** won't construct an attribute object, call its constructor, or set its fields and properties.

If you want to construct an attribute object, you must call either **GetCustomAttributes** or **GetCustomAttribute**. Every time one of these methods is called, it constructs instances of the specified attribute type and sets each instance's fields and properties based on the values specified in the source code. These methods return references to fully constructed instances of the applied attribute types.

When you call any of these methods, internally, they must scan the managed module's metadata, performing string comparisons to locate the specified custom attribute class. Obviously, these operations take time. If you're performance conscious, you should consider caching the result of calling these methods rather than calling them repeatedly asking for the same information.

Table 16–1: System.Attribute's Methods That Reflect over Metadata Looking for Instances of CLS–Compliant Custom Attributes

| Method | Description |
|---|---|
| **IsDefined** | Returns **true** if there is at least one instance of the specified **Attribute**–derived type associated with the target. This method is fast because it doesn't construct (deserialize) any instances of the attribute type. |
| **GetCustomAttributes** | Returns an array where each element is an instance of the specified attribute type that was applied to the target. Each instance is constructed (deserialized) using the parameters, fields, and properties specified during compilation. If the target has no instances of the specified attribute type, then an empty array is returned. This method is typically used with attributes that have **AllowMultiple** set to **true**. |
| **GetCustomAttribute** | Returns an instance of the specified attribute type that was applied to the target. The instance is constructed (deserialized) using the parameters, fields, and properties specified during compilation. If the target has no instances of the specified attribute type, then **null** is returned. If the target has multiple instances of the specified attribute applied to it, then a **System.Reflection.AmbiguousMatchException** exception is thrown. This method is typically used with attributes that have **AllowMultiple** set to **false**. |

The **System.Reflection** namespace defines several types that allow you to examine the contents of a module's metadata: **Assembly**, **Module**, **Enum**, **ParameterInfo**, **MemberInfo**, **Type**, **MethodInfo**, **ConstructorInfo**, **FieldInfo**, **EventInfo**, **PropertyInfo**, and their respective ***Builder** types. All these types also offer **IsDefined** and **GetCustomAttributes** methods. Only **System.Attribute** offers the very convenient **GetCustomAttribute** method.

The version of **GetCustomAttributes** defined by the reflection types returns an array of **Object** types (**Object[]**) instead of an array of **Attribute** types (**Attribute[]**). This is because the reflection types are able to return objects of non–CLS–compliant attribute types. You shouldn't be concerned about this inconsistency because non–CLS–compliant attributes are incredibly rare. In fact, in all the time I've been working with the .NET Framework, I've never even seen one.

> **Note** Be aware that the methods on the reflection types only consider attributes on classes and methods to be inheritable; that is, if you call **IsDefined** or **GetCustomAttributes** using an **EventInfo**, a **PropertyInfo**, or a **ParameterInfo**, the **inherit** parameter is ignored and assumed to be **false**. Only the **Attribute** type's methods honor the **inherit** parameter for events, properties, and parameters.

There's just one more thing that you should be aware of: When you pass a type to **IsDefined**, **GetCustomAttribute**, or **GetCustomAttributes**, these methods search for the application of the attribute type you specify or any attribute type derived from the specified type. If your code is looking for a specific attribute type, you should perform an additional check on the returned value to ensure that what these methods returned is the exact type you're looking for. You might also want to consider defining your attribute type to be sealed to reduce potential confusion and eliminate this extra check. I must admit, all the attribute types I've ever seen inherit directly from **System.Attribute** so I've never run into this problem personally.

Here's some sample code that gets all the methods defined within a type and displays the attributes applied to each method. The code is for demonstration purposes; normally, you wouldn't apply these particular custom attributes to these targets as I've done here.

```
using System;
using System.Diagnostics;
using System.Reflection;

[assembly:CLSCompliant(true)]

[Serializable]
[DefaultMemberAttribute("Main")]
class App {
   [Conditional("Debug")][Conditional("Release")]
   public void DoSomething() {}

   public App() {
   }

   [CLSCompliant(true)]
   [STAThread]
   public static void Main() {
      // Display the type's name.
      Console.WriteLine("Attributes applied to: {0}", typeof(App));

      // Get and show the set of attributes applied to this type.
      ShowAttributes(typeof(App).GetCustomAttributes(false));

      // Get the set of methods associated with the type.
      MemberInfo[] members = typeof(App).FindMembers(
         MemberTypes.Constructor | MemberTypes.Method,
         BindingFlags.DeclaredOnly | BindingFlags.Instance |
         BindingFlags.Public  | BindingFlags.Static,
         Type.FilterName, "*");

      foreach (MemberInfo member in members) {
         // Display the type's member name.
         Console.WriteLine("Attributes applied to: {0}", member.Name);

         // Get and show the set of attributes applied to this member.
         ShowAttributes(member.GetCustomAttributes(false));
      }
   }

   public static void ShowAttributes(Object[] attributes) {
      foreach (Object attribute in attributes) {
         // Display the type of each applied attribute.
         Console.Write("   {0}", attribute.GetType().ToString());
         if (attribute is ConditionalAttribute)
            Console.Write("  ({0})",
               ((ConditionalAttribute) attribute).ConditionString);

         if (attribute is CLSCompliantAttribute)
            Console.Write("  ({0})",
               ((CLSCompliantAttribute) attribute).IsCompliant);

         Console.WriteLine();
      }

      if (attributes.Length == 0)
         Console.WriteLine("   No attributes applied to this target.");
```

```
      Console.WriteLine();
   }
}
```

Building and running this application yields the following output:

```
Attributes applied to: App
   System.Reflection.DefaultMemberAttribute

Attributes applied to: DoSomething
   System.Diagnostics.ConditionalAttribute  (Release)
   System.Diagnostics.ConditionalAttribute  (Debug)

Attributes applied to: Main
   System.CLSCompliantAttribute  (True)
   System.STAThreadAttribute

Attributes applied to: ShowAttributes
   No attributes applied to this target.

Attributes applied to: .ctor
   No attributes applied to this target.
```

## Matching Two Attribute Instances Against Each Other

Now that your code knows how to check to see whether an instance of an attribute is applied to a target, it might want to check the fields of the attribute to see what values they have. One way to do this is to write code that checks the values of the type's fields. However, your attribute type could also override **System.Attribute**'s **Match** method. Then your code could construct an instance of the attribute type and call **Match** to compare it to the instance that was applied to the target. The following code demonstrates.

```
using System;

[Flags]
public enum Accounts {
   Savings   = 0x0001,
   Checking  = 0x0002,
   Brokerage = 0x0004
}

[AttributeUsage(AttributeTargets.Class)]
public class AccountsAttribute : Attribute {

   private Accounts accounts;

   public AccountsAttribute(Accounts accounts) {
      this.accounts = accounts;
   }
   public override Boolean Match(object obj) {
      // If the base class implements Match and the base class
      // is not Attribute, then uncomment the following line.
      // if (!base.Match(obj)) return false;

      // Because 'this' isn't null, if obj is null, then the
      // objects can't match.
      // NOTE: You can delete this line if you trust that
      // the base type implemented Match correctly.
      if (obj == null) return false;
```

```
      // If the objects are of different types, they can't match.
      // NOTE: You can delete this line if you trust that
      // the base type implemented Match correctly.
      if (this.GetType() != obj.GetType()) return false;

      // Cast obj to your type to access fields.
      // NOTE: This cast can't fail because you know that objects
      // are of the same type.
      AccountsAttribute other = (AccountsAttribute) obj;

      // Compare the fields as you see fit.
      // This example checks whether 'this' accounts is a subset
      // of other's accounts.
      if ((other.accounts & accounts) != accounts) return false;

      return true;   // Objects match
   }

   public override Boolean Equals(object obj) {
      // If the base class implements Equals and the base class
      // is not Object, then uncomment the following line.
      // if (!base.Equals(obj)) return false;

      // Because 'this' isn't null, if obj is null, then the
      // objects can't be equal.
      // NOTE: You can delete this line if you trust that
      // the base type implemented Equals correctly.
      if (obj == null) return false;

      // If the objects are of different types, they can't be equal.
      // NOTE: You can delete this line if you trust that
      // the base type implemented Equals correctly.
      if (this.GetType() != obj.GetType()) return false;

      // Cast obj to your type to access fields.
      // NOTE: This cast can't fail because you know that objects
      // are of the same type.
      AccountsAttribute other = (AccountsAttribute) obj;

      // Compare the fields to see whether they have the same value.
      // This example checks whether 'this' accounts is the same
      // as other's accounts.
      if (other.accounts != accounts) return false;

      return true;   // Objects are equal.
   }

   // Override GetHashCode because Equals is overloaded.
   public override Int32 GetHashCode() {
      return (Int32) accounts;
   }
}

[Accounts(Accounts.Savings)]
class ChildAccount { }

[Accounts(Accounts.Savings | Accounts.Checking | Accounts.Brokerage)]
class AdultAccount { }

class App {
   static void Main() {
      CanWriteCheck(new ChildAccount());
```

```
      CanWriteCheck(new AdultAccount());

      // This just demonstrates that the method works correctly on a
      // type that doesn't have any AccountsAttribute attributes
      // applied to it.
      CanWriteCheck(new App());
   }

   public static void CanWriteCheck(Object obj) {

      // Construct an instance of the attribute type, and initialize it
      // to what you're explicitly looking for.
      Attribute checking = new AccountsAttribute(Accounts.Checking);

      // Construct the attribute instance that was applied to the type.
      Attribute validAccounts = Attribute.GetCustomAttribute(
         obj.GetType(), typeof(AccountsAttribute), false);

      // If the attribute was applied to the type AND the
      // attribute specifies the "Checking" account, write
      // a check from this object.
      if ((validAccounts != null) && checking.Match(validAccounts)) {
         Console.WriteLine("{0} types can write checks.",
            obj.GetType());
      } else {
         Console.WriteLine("{0} types can NOT write checks.",
            obj.GetType());
      }
   }
}
```

Building and running this application yields the following output:

```
ChildAccount types can NOT write checks.
AdultAccount types can write checks.
App types can NOT write checks.
```

You'll notice that the code for **Match** is almost identical to the code for **Equals** (discussed in Chapter 6); that is, in both, you must cast carefully and you must remember to call the base type's **Match** method if appropriate. If you define a custom attribute and you don't override the **Match** method, you'll inherit the implementation of **Attribute**'s **Match** method. This implementation simply calls **Equals**.

## Pseudo–Custom Attributes

Certain Microsoft–defined attributes are used so frequently that emitting the full attribute information into the metadata would increase the size of the managed module significantly. These attributes get special treatment at compile time and are emitted into the metadata as bits. The CLR and the FCL know how to look in the metadata specifically for these *pseudo–custom attributes*. Examine the following code:

```
[Serializable]
class SomeType {
   Ã
}
```

The FCL does offer a **System.SerializableAttribute** type, and an instance of this type is being applied to **SomeType**. Because the **Serializable** attribute is common, it is compiled into the metadata as a bit: the full metadata for an instance of the **Serializable** attribute won't be emitted to the metadata. This is why these attributes are called *pseudo*–custom attributes: they look like regular attributes and you apply them in source code just as you do regular attributes, but they are persisted in a super–compressed way (as a bit).

The important point to remember about pseudo–custom attributes is that you can't detect their presence at run time in the same way you detect the presence of regular custom attributes. In a previous code example, I applied the **Serializable** and **DefaultMemberAttribute** attributes to the **App** type. However, when the application runs, only the **DefaultMemberAttribute** is displayed. Methods such as **IsDefined**, **GetCustomAttributes**, and **GetCustomAttribute** don't work with pseudo–custom attributes.

Ideally, the .NET Framework team would have hidden the fact that pseudo–custom attributes are handled differently than regular attributes. And in future versions of the .NET Framework, they might. You can employ other means to detect the use of these pseudo–custom attributes. For example, **System.Type** offers read–only properties such as **IsSerializable**, **IsAutoLayout**, **IsExplicitLayout**, **IsLayoutSequential**, and more. And **System.Reflection.FieldInfo** offers read–only properties such as **IsNotSerialized**. However, for most pseudo–custom attributes, there are no types or methods defined in the FCL that allow you to detect the presence of a pseudo–custom attribute. (By the way, there is a way to detect pseudo–custom attributes if you resort to unmanaged code and access the CLR's COM interface directly.)

Here's a list of the .NET FrameworkÐdefined pseudo–custom attributes:

- **System.NonSerializedAttribute**
- **System.SerializableAttribute**
- **System.Diagnostics.DebuggableAttribute**
- **System.Runtime.CompilerServices.MethodImplAttribute**
- **System.Runtime.InteropServices.DllImportAttribute**
- **System.Runtime.InteropServices.InAttribute**
- **System.Runtime.InteropServices.ComImportAttribute**
- **System.Runtime.InteropServices.FieldOffsetAttribute**
- **System.Runtime.InteropServices.GuidAttribute**
- **System.Runtime.InteropServices.InterfaceTypeAttribute**
- **System.Runtime.InteropServices.OptionalAttribute**
- **System.Runtime.InteropServices.OutAttribute**
- **System.Runtime.InteropServices.PreserveSigAttribute**
- **System.Runtime.InteropServices.StructLayoutAttribute**
- **System.Runtime.InteropServices.MarshalAsAttribute**

# Chapter 17: Delegates

In this chapter, I talk about callback functions. Callback functions are an extremely useful programming mechanism that has been around for years. The Microsoft .NET Framework exposes a callback function mechanism using delegates. Unlike callback mechanisms used in other platforms, such as unmanaged C++, delegates offer much more functionality. For example, delegates ensure that the callback method is type–safe (in keeping with one of the most important goals of the common language runtime). Delegates also integrate the ability to call multiple methods serially and support the calling of static methods as well as instance methods.

## A First Look at Delegates

The C runtime's **qsort** function takes a callback function to sort elements within an array. In Microsoft Windows, callback functions are required for window procedures, hook procedures, asynchronous procedure calls, and more. In the .NET Framework, callback methods are used for a whole slew of things. For example, you can register callback methods to get a variety of notifications, such as unhandled exceptions, window state changes, menu item selections, file system changes, and completed asynchronous operations.

In unmanaged C/C++, the address of a function is just a memory address. This address doesn't carry along any additional information, such as the number of parameters the function expects, the types of these parameters, the function's return value type, and the function's calling convention. In short, unmanaged C/C++ callback functions are not type–safe.

In the .NET Framework, callback functions are just as useful and pervasive as in unmanaged Windows programming. However, the .NET Framework provides a type–safe mechanism called *delegates*. I'll start off the discussion of delegates by showing you how to use them. The following code demonstrates how to declare, create, and use delegates.

```
using System;
using System.Windows.Forms;
using System.IO;

class Set {
   private Object[] items;

   public Set(Int32 numItems) {
      items = new Object[numItems];
      for (Int32 i = 0; i < numItems; i++)
         items[i] = i;
   }

   // Define a Feedback type.
   // NOTE: This type is nested within the Set class.
   public delegate void Feedback(
      Object value, Int32 item, Int32 numItems);

   public void ProcessItems(Feedback feedback) {
      for (Int32 item = 0; item < items.Length; item++) {
         if (feedback != null) {
            // If any callbacks are specified, call them.
            feedback(items[item], item + 1, items.Length);
         }
      }
   }
}
```

```
class App {
   static void Main() {
      StaticCallbacks();
      InstanceCallbacks();
   }


   static void StaticCallbacks() {
      // Create a set with five items in it.
      Set setOfItems = new Set(5);

      // Process the items, but give no feedback.
      setOfItems.ProcessItems(null);
      Console.WriteLine();
      // Process the items, and give feedback to the console.
      setOfItems.ProcessItems(new Set.Feedback(App.FeedbackToConsole));
      Console.WriteLine();

      // Process the items, and give feedback to a message box.
      setOfItems.ProcessItems(new Set.Feedback(App.FeedbackToMsgBox));
      Console.WriteLine();

      // Process the items, and give feedback to the
      // console AND to a message box.
      Set.Feedback fb = null;
      fb += new Set.Feedback(App.FeedbackToConsole);
      fb += new Set.Feedback(App.FeedbackToMsgBox);
      setOfItems.ProcessItems(fb);
      Console.WriteLine();
   }

   static void FeedbackToConsole(
      Object value, Int32 item, Int32 numItems) {
      Console.WriteLine("Processing item {0} of {1}: {2}.",
         item, numItems, value);
   }

   static void FeedbackToMsgBox(
      Object value, Int32 item, Int32 numItems) {
      MessageBox.Show(String.Format("Processing item {0} of {1}: {2}.",
         item, numItems, value));
   }


   static void InstanceCallbacks() {
      // Create a set with five items in it.
      Set setOfItems = new Set(5);

      // Process the items, and give feedback to a file.
      App appobj = new App();
      setOfItems.ProcessItems(new Set.Feedback(appobj.FeedbackToFile));
      Console.WriteLine();
   }

   void FeedbackToFile(
      Object value, Int32 item, Int32 numItems) {

      StreamWriter sw = new StreamWriter("Status", true);
      sw.WriteLine("Processing item {0} of {1}: {2}.",
         item, numItems, value);
      sw.Close();
```

```
    }
}
```

Now I'll describe what this code is doing. At the top, notice the **Set** class. Pretend that this class contains a set of items that will be processed individually. When you create a **Set** object, you pass the number of items it should manage to its constructor. The constructor then creates an array of **Object**s and initializes each object to an integer value.

The **Set** class also defines a public delegate: **Feedback**. The delegate indicates the signature of a callback method. In this example, a **Feedback** delegate identifies a method that takes three parameters (an **Object** and two **Int32**s) and returns **void**. In a way, a delegate is very much like an unmanaged C/C++ **typedef** that represents the address of a function.

The **Set** class also defines a public method named **ProcessItems**. This method takes one parameter, **feedback**, which is a reference to a **Feedback** delegate object. **ProcessItems** iterates through all the elements of the array, and for each element, the callback method (specified by the **feedback** variable) is called. This callback method is passed the value of the item being processed, the item number, and the total number of items in the array. The callback method can process each item any way it chooses.

## Using Delegates to Call Back Static Methods

Now that you understand how the **Set** type is designed and how it works, let's see how to use delegates to call back static methods. The **StaticCallbacks** method that appears in the previous code sample is the focus of this section.

The **StaticCallbacks** method begins by constructing a **Set** object, telling it to create an array of five objects. Then **ProcessItems** is called, passing it **null** for its **feedback** parameter. **ProcessItems** represents a method that performs some action for every item managed by the **Set**. Because the **feedback** parameter is **null** in this example, each item is processed without calling any callback methods.

For the second call to **ProcessItems**, a new **Set.Feedback** delegate object is constructed. This delegate object is a wrapper around a method, allowing that method to be called back indirectly via the wrapper. To the **Feedback** type's constructor, the name of a static method, **App.FeedbackToConsole** in this example, is passed; this indicates the method to be wrapped. The reference returned from the **new** operator is then passed to **ProcessItems**. Now, when **ProcessItems** executes, it will call the **App** type's static **FeedbackToConsole** method for each item in the set. **FeedbackToConsole** simply writes a string to the console indicating the item being processed and the item's value.

**Note** The **FeedbackToConsole** method is defined as **private** inside the **App** type, but the **Set** type's **ProcessItems** method is able to call **App**'s private method. No security problem results here because **App**'s code explicitly decided to return a delegate wrapper over a private method.

The third call to **ProcessItems** is almost identical to the second call. The only difference is that the **Feedback** delegate object wraps the static **App.FeedbackToMsgBox** method. **FeedbackToMsgBox** builds a string indicating the item being processed and the item's value. This string is then displayed in a message box.

The fourth and final call to **ProcessItems** demonstrates how delegates can be linked together to form a chain. In this example, a reference variable to a **Feedback** delegate object, **fb**, is created and initialized to **null**. This variable points to the head of a linked list of delegates. A value of **null** indicates that there are no nodes in the linked list. Then a **Feedback** delegate object that wraps a call to **App**'s **FeedbackToConsole** method is constructed. The C# **+=** operator is used to append this object to the linked list referred to by **fb**. The **fb** variable now refers to the head of the linked list.

Finally, another **Feedback** delegate object is constructed that wraps a call to **App**'s **FeedbackToMsgBox** method. Again, the C# **+=** operator is used to append this object to the linked list and **fb** is updated to refer to the new head of the linked list. Now, when **ProcessItems** is called, it is passed the head of the linked list of **Feedback** delegates. Inside **ProcessItems**, the line of code that calls the callback method actually ends up calling all the callback methods wrapped by the delegate objects in the linked list. In other words, for each item being iterated, **FeedbackToConsole** will be called immediately followed by **FeedbackToMsgBox**. I'll explain exactly how a delegate chain works later in this chapter.

Everything in this example is type−safe. For instance, when constructing a **Feedback** delegate object, the compiler ensures that **App**'s **FeedbackToConsole** and **FeedbackToMsgBox** methods have the exact prototype as defined by the **Feed**−
**back** delegate; that is, both methods must take three parameters (**Object** and two **Int32**s) and both methods must have the same return type (**void**). What would have happened if **FeedbackToConsole** had been prototyped like this?

```
void FeedbackToConsole(Object, Int32, Int32, String s) {
   Ã
}
```

The C# compiler wouldn't compile the code and would issue the following error: "error CS0123: The signature of method 'App.FeedbackToConsole()' does not match this delegate type".

## Using Delegates to Call Back Instance Methods

In the code example near the beginning of the chapter, I explained how to use delegates to call static methods. You can also use delegates to call instance methods for a specific object. To understand how calling back an instance method works, look at the **InstanceCallbacks** method from the code shown earlier:

```
static void InstanceCallbacks() {
   // Create a set with five items in it.
   Set setOfItems = new Set(5);

   // Process the items, and give feedback to a file.
   App appobj = new App();
   setOfItems.ProcessItems(new Set.Feedback(appobj.FeedbackToFile));
   Console.WriteLine();
}
```

Notice that an **App** object is constructed after the **Set** object is constructed. This **App** object doesn't have any fields or properties associated with it; I created it merely for demonstration purposes. When the new **Feedback** delegate object is constructed, its constructor is passed **appobj.FeedbackToFile**. This causes the delegate to wrap a reference to the **FeedbackToFile** method, which is an instance method (not a static method). When this instance method is called, the object **appobj** refers to is the object being operated on (passed as the hidden **this** parameter).

The **FeedbackToFile** method works like the **FeedbackToConsole** and **FeedbackToMsgBox** methods except that it opens a file and appends the processing item string to the end of the file.

Again, the purpose of this example is to demonstrate that delegates can wrap calls to instance methods as well as static methods. For instance methods, the delegate needs to know the instance of the object that the method is going to operate on.

## Demystifying Delegates

On the surface, delegates seem easy to use: you define them using C#'s **delegate** keyword, you construct instances of them using the familiar **new** operator, and you invoke the callback using familiar "method call" syntax (except instead of a method name, you use the variable that refers to the delegate object).

However, what's really going on is quite a bit more complex than what the earlier examples illustrate. The compilers and the common language runtime (CLR) do a lot of behind–the–scenes processing to hide the complexity. In this section, I'll focus on how the compiler and the CLR work together to implement delegates. Having this knowledge will improve your understanding of delegates and will teach you how to use them efficiently and effectively. I'll also touch on some additional features that delegates make available.

Let's start by reexamining this line of code:

```
public delegate void Feedback(
   Object value, Int32 item, Int32 numItems);
```

When it sees this line, the compiler actually defines a complete class definition that looks something like this:

```
public class Feedback : System.MulticastDelegate {
   // Constructor
   public Feedback(Object target, Int32 methodPtr);

   // Method with same prototype as specified by the source code
   public void virtual Invoke(
      Object value, Int32 item, Int32 numItems);

   // Methods allowing the callback to be called asynchronously
   public virtual IAsyncResult BeginInvoke(
      Object value, Int32 item, Int32 numItems,
      AsyncCallback callback, Object object);
   public virtual void EndInvoke(IAsyncResult result);
}
```

The class defined by the compiler has four methods: a constructor, **Invoke**, **BeginInvoke**, and **EndInvoke**. In this chapter, I'll concentrate on the constructor and **Invoke** methods.

In fact, you can verify that the compiler did indeed generate this class automatically by examining the resulting module with ILDasm.exe, as shown in Figure 17–1.

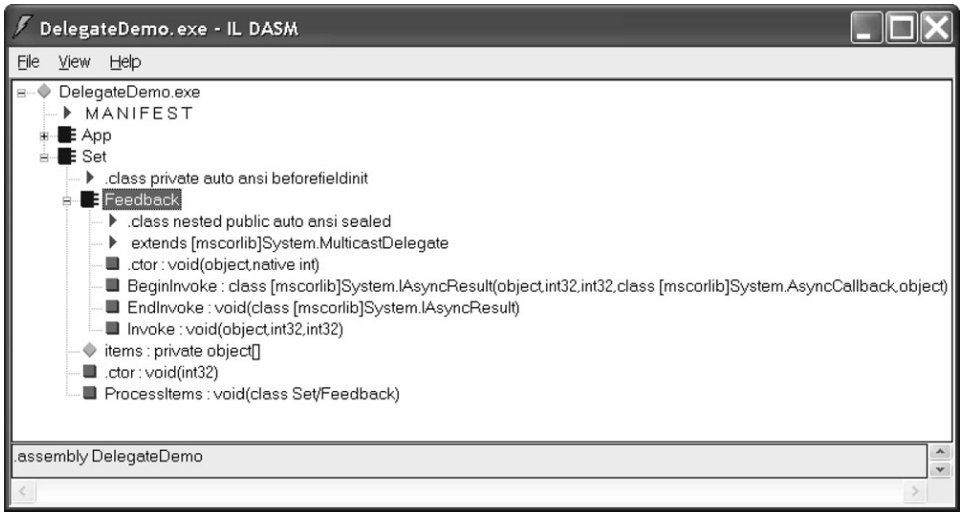



Figure 17–1 : ILDasm.exe showing the metadata produced by the compiler for the delegate

In this example, the compiler has defined a class called **Feedback** that is derived from the **System.MulticastDelegate** type defined in the .NET Framework Class Library (FCL). (All delegate types are derived from **MulticastDelegate**.) The class is public because the delegate is declared as **public** in the source code. If the source code had indicated **private** or **protected**, the **Feedback** class the compiler generated would also be private or protected, respectively. You should be aware that delegate types can be defined within a class (as in the example, **Feedback** is defined within the **Set** class) or at global scope. Basically, because delegates are classes, a delegate can be defined anywhere a class can be defined.

Because all delegate types are derived from **MulticastDelegate**, they inherit **MulticastDelegate**'s fields, properties, and methods. Of all these members, three private fields are probably most significant. Table 17–1 describes these fields.

Table 17–1: MulticastDelegate's Significant Private Fields

| Field | Type | Description |
|---|---|---|
| **_target** | **System.Object** | Refers to the object that should be operated on when the callback method is called. This field is used for instance method callbacks. |
| **_methodPtr** | **System.Int32** | An internal integer that the CLR uses to identify the method that is to be called back. |
| **_prev** | **System.MulticastDelegate** | Refers to another delegate object. This field is usually **null**. |

Notice that all delegates have a constructor that takes two parameters: a reference to an object and an integer that refers to the callback method. However, if you examine the source code, you'll see that I'm passing in values such as **App.FeedbackToConsole** or **appobj.FeedbackToFile**. All your sensibilities tell you that this code shouldn't compile!

However, the compiler knows that a delegate is being constructed and parses the source code to determine which object and method are being referred to. A reference to the object is passed for the **target** parameter, and a special **Int32** value (obtained from a MethodDef or MethodRef metadata token) that identifies the method is passed for the **methodPtr** parameter. For static methods, **null** is passed for the **target** parameter. Inside the constructor, these two parameters are saved in their corresponding private fields.

In addition, the constructor sets the **_prev** field to **null**. This **_prev** field is used to create a linked list of **MulticastDelegate** objects. I'll ignore this field for now but cover it in detail later in the chapter, in the section "Delegate Chains."

So, each delegate object is really a wrapper around a method and an object to be operated on when the method is called. The **MulticastDelegate** class defines two read–only public instance properties: **Target** and **Method**. Given a reference to a delegate object, you can query these properties. The **Target** property returns a reference to the object that will be operated on if the method is called back. If the method is a static method, **Target** returns **null**. The **Method** property returns a **System.Reflection.MethodInfo** object that identifies the callback method.

You could use this information in several ways. For example, you could check to see whether a delegate object refers to an instance method of a specific type:

```
Boolean DelegateRefersToInstanceMethodOfType(
   MulticastDelegate d, Type type) {

   return((d.Target != null) && d.Target.GetType() == type);
}
```

You could also write code to check whether the callback method has a specific name (such as **FeedbackToMsgBox**):

```
Boolean DelegateRefersToMethodOfName(
   MulticastDelegate d, String methodName) {

   return(d.Method.Name == methodName);
}
```

Now that you know how delegate objects are constructed, let's talk about how the callback method is invoked. For convenience, I've repeated the code to **Set**'s **ProcessItems** here:

```
public void ProcessItems(Feedback feedback) {
   for (Int32 item = 0; item < items.Length; item++) {
      if (feedback != null) {
         // If any callbacks are specified, call them.
         feedback(items[item], item, items.Length);
      }
   }
}
```

Just below the comment is the line of code that invokes the callback method. It might seem as if I'm calling a function named **feedback** and passing it three parameters. However, there is no function called **feedback**. Again, because it knows that **feedback** is a variable that refers to a delegate object, the compiler generates code to call the delegate object's **Invoke** method. In other words, the compiler sees this:

```
feedback(items[item], item, items.Length);
```

But the compiler generates code as though the source code said this:

```
feedback.Invoke(items[item], item, items.Length);
```

You can verify that the compiler produces code to call the delegate type's **In–voke** method by using ILDasm.exe to examine the code for the **ProcessItems** method. Figure 17–2 shows the intermediate language (IL) for the **Set** type's **ProcessItems** method. The boxed lines in the figure indicate the instruction that calls **Set.Feedback**'s **Invoke** method.

If you were to modify the source code to call **Invoke** explicitly, the C# compiler would complain and produce the following: "error CS1533: Invoke cannot be called directly on a delegate". C# doesn't allow you to call **Invoke** explicitly; however, a different compiler might require that **Invoke** be called to execute the callback method. In fact, Visual Basic requires that you explicitly call **Invoke**.

You'll recall that the compiler defined the **Invoke** method when it defined the **Feedback** class. When **Invoke** is called, it uses the private **_target** and **_methodPtr** fields to call the desired method on the specified object. Note that the signature of the **Invoke** method matches the signature of the delegate; that is, because the **Feedback** delegate takes three parameters and returns **void**, the **Invoke** method takes the same three parameters and also returns **void**.

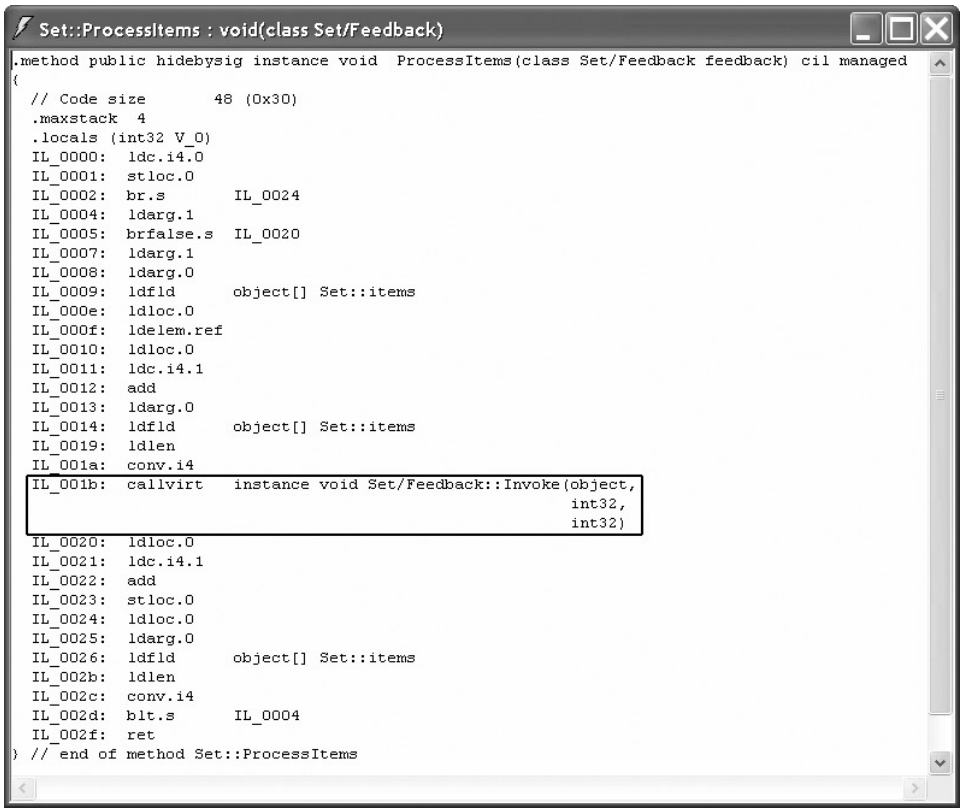


Figure 17–2 : ILDasm.exe proves that the compiler emitted a call to the Set.Feedback delegate type's Invoke method

## Some Delegate History: System.Delegate and System.MulticastDelegate

Even though the FCL defines the **System.MulticastDelegate** class, **MulticastDelegate** is actually derived from the **System.Delegate** class (also defined in the FCL), which itself is derived from **System.Object**. When originally designing the .NET Framework, Microsoft engineers felt the need to provide two different types of delegates: single–cast and multicast. **MulticastDelegate**–derived

types would represent delegate objects that could be chained together, and **Delegate**–derived types would represent objects that could not be chained together. The **System.Delegate** type was designed as the base type, and this class implemented all the functionality necessary to call back a wrapped method. The **MulticastDelegate** class was derived from the **Delegate** class and added the ability to create a linked list (or chain) of **MulticastDelegate** objects.

When compiling source code, a compiler would check the delegate's signature and select the more appropriate of the two classes for the compiler–generated delegate type's base class. For the curious, methods with a signature that indicated a non–**void** return value would be derived from **System.Delegate**, and methods with a **void** return value would be derived from **System.MulticastDelegate**. This made sense because you can get the return value only from the last method called in a linked–list chain.

During the beta testing of the .NET Framework, it became clear that having the two different base types greatly confused developers. In addition, designing delegates this way placed arbitrary limitations on them. For example, many methods have return values that you can ignore in many situations. Because these methods would have a non–**void** return value, they wouldn't be derived from the **MulticastDelegate** class, preventing them from being combined into a linked list.

To reduce developer confusion, Microsoft's engineers wanted to merge the **Delegate** and **MulticastDelegate** classes together into a single class that allowed any delegate object to participate in a linked–list chain. All compilers would generate delegate classes deriving from this one class. This change would reduce complexity and effort for the .NET Framework team, the CLR team, the compiler team, and for developers out in the field who are using delegates.

Unfortunately, the idea of merging the **Delegate** and **MulticastDelegate** classes came along a bit late in the .NET Framework development cycle, and Microsoft was concerned about the potential bugs and testing hit that would occur if these changes were made. So in version 1 of the .NET Framework, these classes haven't been merged; in a future version of the .NET Framework, I expect that these two classes will be merged into a single class.

Although Microsoft chose to delay the merging of these two classes in the FCL, they were able to modify all the Microsoft compilers. All the Microsoft compilers now generate delegate types derived from the **MulticastDelegate** class all the time. So when I said earlier in this chapter that all delegate types are derived from **MulticastDelegate**, I wasn't lying. Because of this change to the compiler, all instances of delegate types can be combined into a linked–list chain regardless of the callback method's return value.

You might be thinking, "Why do I need to know about all this?" Well, here's why: As you start working more and more with delegates, you'll undoubtedly run across both the **Delegate** and **MulticastDelegate** types in the .NET Framework SDK documentation. I want you to understand the relationship between these two classes. In addition, even though all delegate types you create have **MulticastDelegate** as a base class, you'll occasionally manipulate your types using methods defined by the **Delegate** class instead of the **MulticastDelegate** class. For example, the **Delegate** class has static methods called **Combine** and **Remove**. (I explain what these methods do later.) The signatures for both these methods indicate that they take **Delegate** parameters. Because your delegate type is derived from **MulticastDelegate**, which is derived from **Delegate**, instances of your delegate type can be passed to the **Combine** and **Remove** methods.

## Comparing Delegates for Equality

The **Delegate** base class overrides **Object**'s virtual **Equals** method. The **MulticastDelegate** type inherits **Delegate**'s implementation of **Equals**. **Delegate**'s implementation of **Equals** compares two delegate objects to see whether their **_target** and **_methodPtr** fields refer to the same object and method. If these two fields match, then **Equals** returns **true;** if they don't match, **Equals** returns **false**. The following code demonstrates:

```
// Construct two delegate objects that refer to the same target/method.
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToConsole);

// Even though fb1 and fb2 refer to two different objects, internally
// they both refer to the same callback target/method.
Console.WriteLine(fb1.Equals(fb2));   // Displays "True"
```

In addition, both the **Delegate** and **MulticastDelegate** types provide overloads for the equality (**==**) and inequality (**!=**) operators. You can use these operators instead of calling the **Equals** method. The following code's behavior is identical to that of the preceding code:

```
// Construct two delegate objects that refer to the same target/method.
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToConsole);

// Even though fb1 and fb2 refer to two different objects, internally
// they both refer to the same callback target/method.
Console.WriteLine(fb1 == fb2);   // Displays "True"
```

Understanding how to compare delegates for equality is important when you try to manipulate delegate chains, a topic I'll talk about next.

## Delegate Chains

By themselves, delegates are incredibly useful. But add in their support for chaining, and delegates become even more useful. I've already mentioned that each **MulticastDelegate** object has a private field, called **_prev**. This field holds a reference to another **MulticastDelegate** object; that is, every object of type **MulticastDelegate** (or any type derived from **MulticastDelegate**) has a reference to another **MulticastDelegate**–derived object. This field allows delegate objects to be part of a linked list.

The **Delegate** class defines three static methods that you can use to manipulate a linked–list chain of delegate objects.

```
class System.Delegate {
   // Combines the chains represented by head and tail; head is returned.
   // NOTE: head will be the last delegate called.
   public static Delegate Combine(Delegate tail, Delegate head);

   // Creates a chain represented by the array of delegates.
   // NOTE: entry 0 is the head and will be the last delegate called.
   public static Delegate Combine(Delegate[] delegateArray);

   // Removes a delegate matching value's Target/Method from the chain.
   // The new head is returned and will be the last delegate called.
   public static Delegate Remove(Delegate source, Delegate value);
```

```
}
```

When you construct a new delegate object, the object's **_prev** field is set to **null**, indicating that no other objects are in the linked list. To combine two delegates into a linked list, you call one of **Delegate**'s static **Combine** methods:

```
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToMsgBox);
Feedback fbChain = (Feedback) Delegate.Combine(fb1, fb2);
// The left side of Figure 17-3 shows what the
// chain looks like after this code executes.

App appobj = new App();
Feedback fb3 = new Feedback(appobj.FeedbackToStream);
fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
// The right side of Figure 17-3 shows what the
// chain looks like after all the code executes.
```

Figure 17–3 shows what the internal representation of delegate chains looks like.

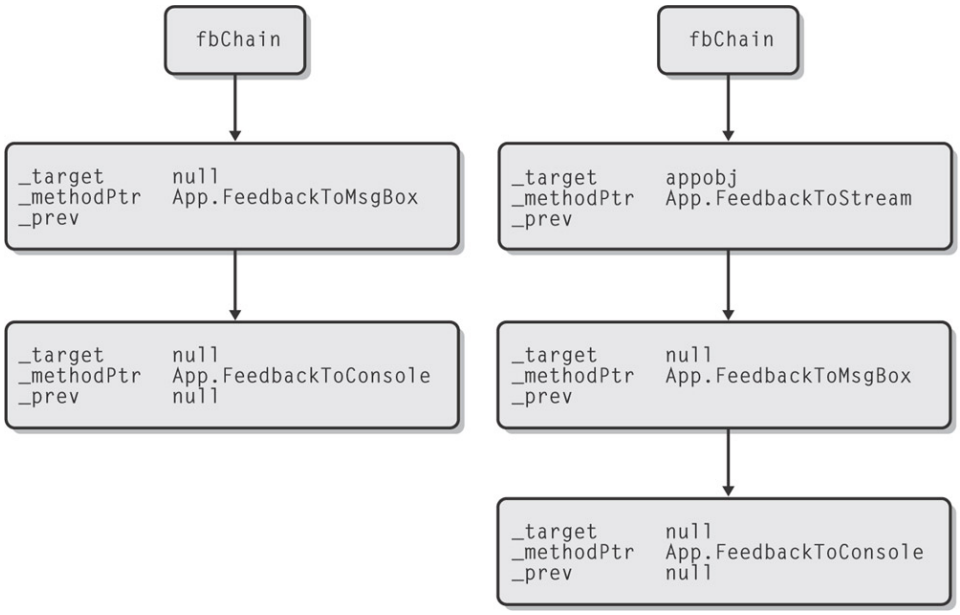



Figure 17–3 : Internal representation of delegate chains

You'll notice that the **Delegate** type offers another version of the **Combine** method that takes an array of **Delegate** references. Using this version of **Com–bine**, you could rewrite the preceding code as follows:

```
Feedback[] fbArray = new Feedback[3];
fbArray[0] = new Feedback(FeedbackToConsole);
fbArray[1] = new Feedback(FeedbackToMsgBox);
App appobj = new App();
fbArray[2] = new Feedback(appobj.FeedbackToStream);

Feedback fbChain = Delegate.Combine(fbArray);
// The right side of Figure 17-3 shows what the
// chain looks like after all the code executes.
```

When a delegate is invoked, the compiler generates a call to the delegate type's **Invoke** method (as discussed earlier in this chapter). To refresh your memory, my earlier example declared a **Feedback** delegate as follows:

```
public delegate void Feedback(
   Object value, Int32 item, Int32 numItems);
```

This caused the compiler to generate a **Feedback** class that contains an **Invoke** method that looks like this (in pseudocode):

```
class Feedback : MulticastDelegate {
   public void virtual Invoke(
      Object value, Int32 item, Int32 numItems) {

      // If there are any delegates in the chain that
      // should be called first, call them recursively.
      if (_prev != null) _prev.Invoke(value, item, numItems);

      // Call the callback method on the specified target object.
     _target.methodPtr(value, item, numItems);
   }
}
```

As you can see, invoking a delegate object causes its previous delegate to be invoked first. When the previous delegate returns, its return value is discarded. After calling its previous delegate, the delegate can then invoke the callback target/method that it wraps. The following code demonstrates.

```
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToMsgBox);
Feedback fbChain = (Feedback) Delegate.Combine(fb1, fb2);
// At this point, fbChain refers to a delegate that calls
// FeedbackToMsgBox, this delegate refers to another delegate that
// calls FeedbackToConsole, and this delegate refers to null.

// Now let's invoke the head delegate, which internally invokes
// its previous delegate and so on.
if (fbChain != null) fbChain(null, 0, 10);
// NOTE: In the preceding line, fbChain will never be null, but checking a
// delegate reference against null before attempting to invoke it is a
// very good habit to get into.
```

So far, I've shown examples where my delegate type, **Feedback**, is defined as having a **void** return value. However, I could have defined my **Feedback** delegate as follows:

```
public delegate Int32 Feedback(
   Object value, Int32 item, Int32 numItems);
```

If I had, its **Invoke** method would have internally looked like this (again, in pseudocode):

```
class Feedback : MulticastDelegate {
   public Int32 virtual Invoke(
      Object value, Int32 item, Int32 numItems) {

      // If there are any delegates in the chain that
      // should be called first, call them.
      if (_prev != null) _prev.Invoke(value, item, numItems);

      // Call the callback method on the specified target object.
     return _target.methodPtr(value, item, numItems);
   }
}
```

When the head of the delegate chain is invoked, it calls any previous delegate in the chain. However, notice that the previous delegate's return value is discarded. Your application code will receive the return value only from the delegate that is at the head of the chain (the last callback method called).

**Note** Once constructed, delegate objects are considered to be immutable; that is, delegate objects always have their **_prev** fields set to **null** and this never changes. When you **Combine** a delegate object to a chain, internally, **Combine** constructs a new delegate object that has the same **_target** and **_methodPtr** fields as the source object but the **_prev** field is set to the old head of the chain. The address of the new delegate object is returned from **Combine**.

This behavior is demonstrated in the following lines of code:

```
Feedback fb = new Feedback(FeedbackToConsole);
Feedback fbChain = (Feedback) Delegate.Combine(fb, fb);
// fbChain refers to a chain of two delegate objects.
// One of the objects is identical to the object referred
// to by fb. The other object was constructed by Combine.
// This object's _prev field refers to fb, and Combine
// returns the reference to this new object.

// Do fb and fbChain refer to the same object? "False"
Console.WriteLine(Object.ReferenceEquals(fb, fbChain);

// Do fb and fbChain refer to the same callback
// target/method? "True"
Console.WriteLine(fb.Equals(fbChain));
```

Now that you know how to build delegate chains, let's look at how to remove a delegate from a chain. To remove a delegate from a linked list, you call the **Delegate** type's static **Remove** method:

```
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToMsgBox);
Feedback fbChain = (Feedback) Delegate.Combine(fb1, fb2);
// fbChain refers to a chain of two delegates.

// Invoke the chain: two methods are called.
if (fbChain != null) fbChain(null, 0, 10);


fbChain = (Feedback)
   Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
// fbChain refers to a chain with one delegate.

// Invoke the chain: one method is called.
if (fbChain != null) fbChain(null, 0, 10);

fbChain = (Feedback)
   Delegate.Remove(fbChain, new Feedback(FeedbackToConsole));
// fbChain refers to a chain of 0 delegates. (fbChain is null.)

// Invoke the chain: 0 methods are called.
if (fbChain != null) fbChain(null, 0, 10);
// Now you see why I've been comparing fbChain to null all this time!
```

This code first builds a chain by constructing two delegate objects and then combining them into a linked list by calling the **Combine** method. The **Remove** method is then called. **Remove**'s first parameter refers to the head of the delegate object chain, and the second parameter refers to the delegate object that is to be removed from the chain. I know it seems strange to construct a new

delegate object in order to remove it from the chain. To fully understand why this is necessary requires some additional explanation.

In the call to **Remove**, I'm constructing a new delegate object. This delegate object has its **_target** and **_methodPtr** fields initialized appropriately, and the **_prev** field is set to **null**. The **Remove** method scans the chain (referred to by **fbChain**), checking whether any of the delegate objects in the chain are equal to the new delegate object. Remember that the overridden **Equals** method implemented by the **Delegate** class compares the **_target** and **_methodPtr** fields only and ignores the **_prev** field.

If a match is found, the **Remove** method removes the located delegate object from the chain by fixing up the previous delegate object's **_prev** field. **Re–move** returns the head of the new chain. If a match is not found, **Remove** does nothing (no exception is thrown) and returns the same value that was passed for its first parameter.

Note that each call to **Remove** eliminates only one object from the chain, as the following code demonstrates:

```
Feedback fb = new Feedback(FeedbackToConsole);
Feedback fbChain = (Feedback) Delegate.Combine(fb, fb);
// fbChain refers to a chain of two delegates.

// Invoke the chain: FeedbackToConsole is called twice.
if (fbChain != null) fbChain(...);

// Remove one of the callbacks from the chain.
fbChain = (Feedback) Delegate.Remove(fbChain, fb);

// Invoke the chain: FeedbackToConsole is called once.
if (fbChain != null) fbChain(...);

// Remove one of the callbacks from the chain.
fbChain = (Feedback) Delegate.Remove(fbChain, fb);

// Invoke the chain: 0 methods are called.
if (fbChain != null) fbChain(...);
```

## C#'s Support for Delegate Chains

To make things easier for C# developers, the C# compiler automatically provides overloads of the **+=** and **–=** operators for instances of delegate types. These operators call **Delegate.Combine** and **Delegate.Remove**, respectively. Using these operators simplifies the building of delegate chains. The following C# code shows how using the C# operators simplify the code to combine and remove delegate objects from a chain:

```
Feedback fb = new Feedback(FeedbackToConsole);
App appobj = new App();
fb += new Feedback(appobj.FeedbackToStream);

// Invoke the chain: FeedbackToStream and FeedbackToConsole are called.
if (fb != null) fb(...);

// Remove one of the callbacks from the chain.
fb -= new Feedback(FeedbackToConsole);

// Invoke the chain: FeedbackToStream is called.
```

```
if (fb != null) fb(...);

// Remove the last callback from the chain.
fb -= new Feedback(appobj.FeedbackToStream);

// Invoke the chain: 0 methods are called.
if (fb != null) fb(...);
```

Internally, the compiler translates all uses of **+=** on delegates to calls to **Delegate**'s **Combine** method. Likewise, all uses of the **–=** operator on delegate objects translate to calls to the **Remove** method. In fact, you can build the code I've just shown and look at its IL using ILDasm.exe. This will confirm that the C# compiler did in fact replace all **+=** and **–=** operators with calls to the **Delegate** type's static **Combine** and **Remove** methods, respectively.

## Having More Control over Invoking a Delegate Chain

At this point, you understand how to build a linked–list chain of delegate objects and how to invoke all the objects in that chain. All items in the linked–list chain are invoked because the delegate type's **Invoke** method includes code to call the previous delegate (if one exists). This is obviously a very simple algorithm. And although this simple algorithm is good enough for a lot of scenarios, it has many limitations.

For example, the return values of the callback methods are all discarded except for the last one. Using this simple algorithm, there's no way to get the return values for all the callback methods called. But this isn't the only limitation. What happens if one of the invoked delegates throws an exception or blocks for a very long time? Because the algorithm invoked each delegate in the chain serially, a "problem" with one of the delegate objects stops all the other delegates in the chain from getting called. Clearly, this algorithm isn't robust.

For those scenarios in which this algorithm is insufficient, the **MulticastDelegate** class offers an instance method, **GetInvocationList**, that you can use to call each delegate in a chain explicitly, using any algorithm that meets your needs:

```
public class MulticastDelegate {
   // Creates a delegate array; each item is a clone from the chain.
   // NOTE: entry 0 is the tail, which would normally be called first.
   public virtual Delegate[] GetInvocationList();
}
```

The **GetInvocationList** method operates on a reference to a delegate chain and returns an array of references to delegate objects. Internally, **GetInvocationList** walks the specified chain and creates a clone of each object in the chain, appending the clone to the array. Each clone has its **_prev** field set to **null**, so each object is isolated and doesn't refer to a chain of any other objects.

You can easily write an algorithm that explicitly calls each object in the array. The following code demonstrates:

```
using System;
using System.Text;

// Define a Light component.
class Light {
   // This method returns the light's status.
   public String SwitchPosition() {
```

```
      return "The light is off";
   }
}

// Define a Fan component.
class Fan {
   // This method returns the fan's status.
   public String Speed() {
      throw new Exception("The fan broke due to overheating");
   }
}

// Define a Speaker component.
class Speaker {
   // This method returns the speaker's status.
   public String Volume() {
      return "The volume is loud";
   }
}

class App {

   // Definition of delegate that allows querying a component's status
   delegate String GetStatus();

   static void Main() {
      // Declare an empty delegate chain.
      GetStatus getStatus = null;

      // Construct the three components, and add their status methods
      // to the delegate chain.
      getStatus += new GetStatus(new Light().SwitchPosition);
      getStatus += new GetStatus(new Fan().Speed);
      getStatus += new GetStatus(new Speaker().Volume);

      // Show consolidated status report reflecting
      // the condition of the three components.
      Console.WriteLine(GetComponentStatusReport(getStatus));
   }

   // Method that queries several components and returns a status report
   static String GetComponentStatusReport(GetStatus status) {

      // If the chain is empty, there's nothing to do.
      if (status == null) return null;

      // Use this to build the status report.
      StringBuilder report = new StringBuilder();

      // Get an array where each element is a delegate from the chain.
      Delegate[] arrayOfDelegates = status.GetInvocationList();

      // Iterate over each delegate in the array.
      foreach (GetStatus getStatus in arrayOfDelegates) {

         try {
            // Get a component's status string, and append it to the report.
            report.AppendFormat("{0}{1}{1}",
               getStatus(), Environment.NewLine);
         }
         catch (Exception e) {
            // Generate an error entry in the report for this component.
            Object component = getStatus.Target;
```

```
            report.AppendFormat(
               "Failed to get status from {1}{2}{0}   Error: {3}{0}{0}",
               Environment.NewLine,
               ((component == null) ? "" : component.GetType() + "."),
               getStatus.Method.Name, e.Message);
         }
      }

      // Return the consolidated report to the caller.
      return report.ToString();
   }
}
```

When you build and run this code, the following output appears:

```
The light is off

Failed to get status from Fan.Speed
   Error: The fan broke due to overheating

The volume is loud
```

## Delegates and Reflection

So far in this chapter, the use of delegates has required that the developer know up front the prototype of the method that is to be called back. For example, if **feedback** is a variable that references a **Feedback** delegate, then to invoke the delegate, the code would look like this:

```
feedback(items[item], item, items.Length);
```

As you can see, the developer must know when coding how many parameters the callback method requires and the types of those parameters. Fortunately, the developer almost always has this information and so writing code like the preceding isn't a problem.

In some rare scenarios, however, the developer doesn't have this information at compile time. I showed an example of this in Chapter 11 when I discussed the **EventHandlerSet** type. In this example, a hash table maintained a set of different delegate types. At run time, to fire an event, one of the delegates was looked up in the hash table and invoked. At compile time, it wasn't possible to know exactly which delegate would be called and which parameters were necessary to pass to the delegate's callback method.

Fortunately, **System.Delegate** offers a few methods that allow you to create and invoke a delegate when you just don't have all the necessary information about the delegate at compile time. Here are the methods that **Delegate** defines:

```
public class Delegate {
   // Construct a delType delegate wrapping the specified methodInfo.
   public static Delegate CreateDelegate(Type delType,
      MethodInfo mi);

   // Construct a delType delegate wrapping a type's static method.
   public static Delegate CreateDelegate(Type delType,
      Type type, String methodName);

   // Construct a delType delegate wrapping an object's instance method.
```

```
   public static Delegate CreateDelegate(Type delegateType,
      Object obj, String methodName);

   // Construct a delType delegate wrapping an object's instance method.
   public static Delegate CreateDelegate(Type delegateType,
      Object obj, String methodName, Boolean ignoreCase);

   public Object DynamicInvoke(Object[] args);
}
```

All the **CreateDelegate** methods here construct a new object of a **Delegate**–derived type identified by the first parameter, **delType**. The remaining parameters to **CreateDelegate** determine the callback method that the **Delegate**–derived object is to wrap. You can specify a **MethodInfo** (discussed in Chapter 20), a type's static method by **String**, or an object's instance method by **String**.

The **Delegate**'s instance **DynamicInvoke** method allows you to invoke a delegate object's callback method, passing a set of parameters that you determine at run time. When you call **DynamicInvoke**, it internally ensures that the parameters you pass are compatible with the parameters that the callback method expects. If they're compatible, the callback method is called. If they're not, then an exception is thrown. **DynamicInvoke** returns the object that the callback method returned.

The following code shows how to use these methods:

```
using System;
using System.Reflection;
using System.IO;

// Here are some different delegate definitions.
delegate Object TwoInt32s(Int32  n1, Int32  n2);
delegate Object OneString(String s1);


class App {
   static void Main(String[] args) {
      if (args.Length < 2) {
         String fileName  =
            Path.GetFileNameWithoutExtension(
               Assembly.GetEntryAssembly().CodeBase);
         Console.WriteLine("Usage:");
         Console.WriteLine("{0} delType methodName [Param1] [Param2]",
            fileName);
         Console.WriteLine("   where delType must be TwoInt32s or OneString");
         Console.WriteLine("   if delType is TwoInt32s, " +
            "methodName must be Add or Subtract");
         Console.WriteLine("   if delType is OneString, " +
            "methodName must be NumChars or Reverse");
         Console.WriteLine();
         Console.WriteLine("Examples:");
         Console.WriteLine("   {0} TwoInt32s Add 123 321", fileName);
         Console.WriteLine("   {0} TwoInt32s Subtract 123 321", fileName);
         Console.WriteLine("   {0} OneString NumChars \"Hello there\"",
            fileName);
         Console.WriteLine("   {0} OneString Reverse  \"Hello there\"",
            fileName);
         return;
      }
      Type delType = Type.GetType(args[0]);
      if (delType == null) {
```

```
         Console.WriteLine("Invalid delType argument: " + args[0]);
         return;
      }

      Delegate d;
      try {
         d = Delegate.CreateDelegate(delType, typeof(App), args[1]);
      }
      catch (ArgumentException) {
         Console.WriteLine("Invalid methodName argument: " + args[1]);
         return;
      }

      Object[] callbackArgs = new Object[args.Length - 2];
      if (d.GetType() == typeof(TwoInt32s)) {
         try {
            for (Int32 a = 2; a < args.Length; a++)
               callbackArgs[a - 2] = Int32.Parse(args[a]);
         }
         catch (FormatException) {
            Console.WriteLine("Parameters must be integers.");
            return;
         }
      }

      if (d.GetType() == typeof(OneString)) {
         Array.Copy(args, 2, callbackArgs, 0, callbackArgs.Length);
      }

      try {
         Object result = d.DynamicInvoke(callbackArgs);
         Console.WriteLine("Result = " + result);
      }
      catch (TargetParameterCountException) {
         Console.WriteLine("Incorrect number of parameters specified.");
      }
   }


   // This is the callback method that takes two Int32 parameters.
   static Object Add(Int32 n1, Int32 n2) {
      return n1 + n2;
   }
   static Object Subtract(Int32 n1, Int32 n2) {
      return n1 - n2;
   }

   // This is the callback method that takes one String parameter.
   static Object NumChars(String s1) {
      return s1.Length;
   }

   static Object Reverse(String s1) {
      Char[] chars = s1.ToCharArray();
      Array.Reverse(chars);
      return new String(chars);
   }
}
```

# Part V: Managing Types

## Chapter List

*Chapter 18: Exceptions*
*Chapter 19: Automatic Memory Management (Garbage Collection)*
*Chapter 20: CLR Hosting, AppDomains, and Reflection*

# Chapter 18: Exceptions

## Overview

In this chapter, I'll talk about a powerful mechanism that allows you to write more maintainable and robust code: *exception handling*. Here are just a few of the benefits offered by exception handling:

- **The ability to keep cleanup code in a localized location, and the assurance that this cleanup code will execute** By moving cleanup code out of an application's main logic to a localized location, the application is easier to write, understand, and maintain. The assurance that the cleanup code runs means that the application is more likely to remain in a consistent state. For example, files will get closed when the code writing to the file can no longer continue what it's doing for whatever reason.
- **The ability to keep code that deals with exceptional situations in a central place** A line of code can fail for many reasons: arithmetic overflow, stack overflow, out–of–memory status, an out–of–range argument, an out–of–range index into an array, and an attempt to access a resource (such as a file) after it has been closed, to name a few. Without using exception handling, it's very difficult, if not impossible, to write code that gracefully detects and recovers from such failures. Sprinkling the code to detect these potential failures into your application's main logic makes the code difficult to write, understand, and maintain. In addition, having code that checks for these potential failures would be a huge performance hit on an application.

  Using exception handling, you don't need to write code to detect these potential failures. Instead, you can simply write your code assuming that the failures won't occur. This certainly makes the code easier to write, understand, and maintain. In addition, the code runs fast. Then you put all your recovery code in a central location. Only if a failure occurs does the exception handling mechanism step in to execute your recovery code.
- **The ability to locate and fix bugs in the code** When a failure occurs, the common language runtime (CLR) walks up the thread's call stack, looking for code capable of handling the exception. If no code handles the exception, you receive a notification of this "unhandled exception." You can then easily locate the source code that issued the failure, determine why the failure happened, and modify the source code to fix the bug. This means that bugs will be detected during development and testing of an application and fixed prior to the application's deployment. Once deployed, the application will be more robust, improving the end–user's experience.

When used properly, exception handling is a great tool that eases the burden on software developers. However, if you use it improperly, exception handling can bring much sorrow and pain by hiding serious problems in the code or by misinforming you of the actual problem. The bulk of this chapter is dedicated to explaining how to use exception handling properly.

## The Evolution of Exception Handling

When designing the Win32 API and COM, Microsoft decided not to use exceptions to notify callers of problems. Instead, most Win32 functions return **FALSE** to indicate that something is wrong and then the caller calls **GetLastError** to find the violation. On the other hand, COM methods return an **HRESULT**. If the high bit is 1, an assumption was violated and the remaining bits represent a value to help you determine the cause of the violation.

Microsoft avoided using exceptions with Win32 and COM APIs for many reasons:

- Most developers are not familiar with exception handling.
- Many programming languages, including C and early versions of C++, don't support exceptions.
- Some developers feel that exceptions are difficult to use and understand, and Microsoft didn't want to force exceptions down every programmer's throat. (Personally, I feel the benefit of exceptions far outweighs the "exception handling learning curve.")
- Exception handling can hurt the performance of an application. In some situations, exception handling doesn't perform as well as simply returning an error code. However, if exceptions are rarely thrown, this overhead isn't noticeable. Again, I feel that the benefit of exceptions outweighs the slight performance hit that sometimes occurs. I also believe that when exception handling is used correctly throughout an application, performance can improve. I'll address performance later in this chapter. In addition, exceptions in managed code are far less expensive than in some systems, such as C++.

The old ways of reporting violations are too limiting because all the caller gets back is a 32–bit number. If the number is the code for an invalid argument, the caller doesn't know which argument is invalid. And if the number is the code for a division by zero, the caller doesn't know exactly which line of code caused the division by zero and can't fix the code easily. Developing software is hard enough without losing important information about problems! If the application code detects a problem, you want all the information possible about the problem so that you can put a proper remedy in place as quickly and easily as possible.

The exception handling mechanism offers several advantages over a 32–bit number. An exception includes a string description so that you know exactly which argument is causing the problem. The string might also include additional information giving guidance for improving the code. And an exception includes a stack trace so that you know which path the application took that caused the exception.

Another advantage is that exceptions don't have to be caught or detected at the place they occur. This simplifies coding substantially because you don't have to associate error detection and correction code with every statement or method call that might fail.

Closely related to the previous advantage is perhaps the biggest benefit of all: an exception can't easily be ignored. If a Win32 function is called and returns a failure code, it's all too easy for the caller to ignore the return code, assume that the function worked as expected, and allow the application to continue running as if all is fine. However, when a method throws an exception, it's indicating that the method couldn't work as expected. If the application doesn't catch the exception, the CLR terminates the application. To some, this behavior might seem radical and severe. However, I think it's the right thing to do.

If a method is called and an exception is thrown, it's not OK for the application to continue running. The rest of the application assumes that the previous operations all completed as expected. If this isn't the case, the application will continue running but with unpredictable results. For example, if the user's data in the application is corrupt, this data shouldn't continue to be manipulated. With the Win32 and COM 32–bit numbers, the possibility of the application running with unpredictable results is all too strong. With exceptions, it's not possible.

All the methods defined by types in the Microsoft .NET Framework throw exceptions to indicate that an assumption was violated; no 32–bit status values are returned. So, all programmers must have a full understanding of exceptions and how to handle them in their code. Microsoft made a great

decision here! Using exception handling in your code is straightforward and allows you to write code that's easy to implement, easy to read, and easy to maintain. In addition, exception handling allows you to write robust code that's capable of recovering from any application situation. Used correctly, exception handling can prevent application crashes and keep your users happy.

## The Mechanics of Exception Handling

In this section, I'll introduce the mechanics and C# constructs for using exception handling, but it's not my intention to explain them in great detail. The purpose of this chapter is to offer useful guidelines for when and how to use exception handling in your code. If you want more information about the mechanics and language constructs for using exception handling, see the .NET Framework documentation and your programming language reference. Also, the .NET Framework exception handling mechanism is built using the structured exception handling (SEH) mechanism offered by Windows. SEH has been discussed in many resources, including my own book, *Programming Applications for Microsoft Windows,* (4th ed., Microsoft Press, 1999), which contains three chapters devoted to SEH.

The following C# code shows a standard usage of the exception handling mechanism. This code gives you an idea of what exception handling blocks look like and what their purpose is. In the subsections after the code, I'll formally describe the **try**, **catch**, and **finally** blocks and their purpose and provide some notes about their use.

```
void SomeMethod() {

   try {
      // Inside the try block is where you put code requiring
      // graceful recovery or common cleanup operations.
   }
   catch (InvalidCastException) {
      // Inside this catch block is where you put code that recovers
      // from an InvalidCastException (or any exception type derived
      // from InvalidCastException).
   }
   catch (NullReferenceException) {
      // Inside this catch block is where you put code that recovers
      // from a NullReferenceException (or any exception type derived
      // from NullReferenceException).
   }
   catch (Exception e) {
      // Inside this catch block is where you put code that recovers
      // from any CLS-compliant exception.

      // When catching a CLS-compliant exception, you usually rethrow
      // the exception. I explain rethrowing later in this chapter.
      throw;
   }
   catch {
      // Inside this catch block is where you put code that recovers
      // from any exception, CLS-compliant or not.

      // When catching any exception, you usually rethrow
      // the exception. I explain rethrowing later in this chapter.
      throw;
   }
   finally {
      // Inside the finally block is where you put code that
      // cleans up any operations started within the try block.
```

```
      // The code in this block ALWAYS executes, regardless of
      // whether an exception is thrown.
   }
   // Code below the finally block executes if no exception is thrown
   // within the try block or if a catch block catches the exception
   // and doesn't throw or rethrow an exception.
}
```

This code demonstrates one possible way to use exception handling blocks. Don't let the code scare you—most methods have simply a **try** block matched with a single **finally** block or a **try** block matched with a single **catch** block. It's unusual to have as many **catch** blocks as in this example. I put them there for illustration purposes.

## The try Block

A **try** block contains code that requires common cleanup or exception recovery operations. The cleanup code should be placed in a single **finally** block. A **try** block can also contain code that might potentially throw an exception. The exception recovery code should be placed in one or more **catch** blocks. You create one **catch** block for each kind of event you think the application can recover from. A **try** block must be associated with at least one **catch** or **finally** block; it makes no sense to have a **try** block that stands by itself.

## The catch Block

A **catch** block contains code to execute in response to an exception. A **try** block can have zero or more **catch** blocks associated with it. If the code in a **try** block doesn't cause an exception to be thrown, the CLR never executes any code contained within any of its **catch** blocks. The thread skips over all the **catch** blocks and executes the code in the **finally** block (if one exists). After the code in the **finally** block executes, execution continues with the statement following the **finally** block.

The parenthetical expression appearing after the **catch** keyword is called an *exception filter*. The exception filter is a type representing an exceptional situation that the developer anticipated and can recover from. In C#, the type in a **catch** filter must be **System.Exception** or a type derived from **System.Exception**. For example, the previous code contains **catch** blocks ready to handle an **InvalidCastException** (or any exception derived from it), a **NullReferenceException** (or any exception derived from it), or any **Exception** (any type of Common Language Specification [CLS]Đcompliant exception at all).

Note that **catch** blocks are searched from top to bottom; place the more specific exceptions (types whose base class is farther from **System.Object**) at the top. In fact, the C# compiler generates an error if more specific **catch** blocks appear closer to the bottom because the **catch** block would be unreachable.

If an exception is thrown by code executing within the **try** block (or any method called from within the **try** block), the CLR starts searching for **catch** blocks whose filter recognizes the thrown exception. If none of the catch filters accepts the exception, the CLR continues searching up the call stack looking for a catch filter that will accept the exception. If after reaching the top of the call stack no **catch** block is interested in handling the exception, an unhandled exception results. I'll talk more about unhandled exceptions later in this chapter.

Once it locates an exception filter capable of handling the exception, the CLR executes the code in all **finally** blocks, starting from the **try** block whose code threw the exception and stopping with the

catch filter that matched the exception. Note that any **finally** block associated with the **catch** block that matched the exception is not executed yet. The code in this **finally** block won't execute until after the code in the handling **catch** block has executed.

**Important** C# can throw only CLS–compliant exceptions—that is, an exception type derived from **System.Exception**. However, the CLR allows any type of object to be thrown. C# offers a special type of **catch** block that allows you to catch non–CLS–compliant exceptions:

```
catch {   // Notice no exception filter is specified here.
   // Execute some recovery code here.
   Ã
   throw; // Rethrow the exception to let other code
          // know what happened.
}
```

Because you can't catch an object in this **catch** block, you can't get any information about the exception. About the only thing you can do is execute some recovery code and then rethrow the exception. By the way, this **catch** block will also catch any CLS–compliant exceptions.

For the record, the .NET Framework Class Library (FCL) never throws any non–CLS–compliant exceptions, and in fact, I've never seen any managed code throw a non–CLS–compliant exception. Of course, if you program in intermediate language (IL) assembly language, you could throw a non–CLS–compliant exception (such as an **Int32**). C++ with Managed Extensions also allows the developer to throw non–CLS–compliant exceptions. Obviously, any code you write should throw CLS–compliant exceptions only since many managed programming languages don't offer a high degree of support for non–CLS–compliant exception types.

In C#, a catch filter can specify an exception variable. When an exception is caught, this variable refers to the **System.Exception**–derived object that was thrown. The **catch** block's code can reference this variable to access information specific to the exception (such as the stack trace leading up to the exception). Although it's possible to modify this object, you shouldn't; consider the object to be read–only. I'll explain the **Exception** type and what you can do with it later in this chapter.

After all the code in **finally** blocks has executed, the code in the handling **catch** block executes. This code typically performs some operations to recover from the exception. At the end of the **catch** block, you have three choices:

- Rethrow the same exception, notifying code higher up the call stack of the exception.
- Throw a different exception, giving richer exception information to code higher up the call stack.
- Let the thread fall out the bottom of the **catch** block.

Later in this chapter, I'll offer some guidelines for when you should use each of these techniques.

If you choose either of the first two techniques, you're throwing an exception and the CLR behaves just like it did before: it walks up the call stack looking for a catch filter interested in recovering from the exception. If you pick the last technique, when the thread falls out the bottom of the **catch** block,

it immediately starts executing code contained in the **finally** block, if one exists. After all the code in the **finally** block executes, the thread drops out of the **finally** block and starts executing the statements immediately following the **finally** block. If no **finally** block exists, the thread continues execution at the statement following the last **catch** block.

### The finally Block

A **finally** block contains code that's guaranteed to execute. Typically, the code in a **finally** block performs the cleanup operations required by actions taken in the **try** block. For example, if you open a file in a **try** block, put the code to close the file in a **finally** block:

```
void ReadData(String pathname) {

   FileStream fs = null;
   try {
      fs = new FileStream(pathname, FileMode.Open);
      // Process the data in the file.
      Ã
   }
   catch (OverflowException) {
      // Inside this catch block is where you put code that recovers
      // from an OverflowException (or any exception type derived
      // from OverflowException).
      Ã
   }
   finally {
      // Make sure that the file gets closed.
      if (fs != null) fs.Close();
   }
}
```

If the code in the **try** block executes without throwing an exception, the file is guaranteed to be closed. If the code in the **try** block does throw an exception, the code in the **finally** block still executes and the file is guaranteed to be closed, regardless of whether or not the exception is caught. It's improper to put the statement to close the file after the **finally** block; the statement wouldn't execute if an exception were thrown and not caught, leaving the file open.

A **try** block doesn't have to have a **finally** block associated with it at all; sometimes the code in a **try** block just doesn't require any cleanup code. However, if you do have a **finally** block, it must appear after any and all **catch** blocks, and a **try** block can have no more than one **finally** block associated with it.

When the thread reaches the end of the code contained in a **finally** block, the thread simply falls out the bottom of the block and starts executing the statements immediately following the **finally** block. Remember that the code in the **finally** block is cleanup code. This code should execute only what is necessary to undo operations initiated in the **try** block. Avoid putting code that might throw an exception in a **finally** block. However, if an exception is thrown within a **finally** block, the world doesn't come to an end—the application isn't terminated, and the exception mechanism continues to work as though the exception were thrown after the **finally** block.

## What Exactly Is an Exception?

Over the years, I've run into many developers who think that an exception identifies something that rarely happens: "an exceptional event." I always ask them to define "exceptional event." They

respond, “You know, something you don’t expect to happen.” Then they add, “If you’re reading bytes from a file, eventually you’ll reach the end of the file. So because you expect this, an exception shouldn’t be raised when you reach the end of the file. Instead, the **Read** method should return some special value when the end of the file is reached."

Here’s my response: “I have an application that needs to read a 20–byte data structure from a file. However, for some reason, the file contains only 10 bytes. In this case, I’m not expecting to reach the end of the file while reading. But because I’ll reach the end of the file prematurely, I’d expect an exception to be thrown. Wouldn’t you?” In fact, most files contain structured data. It’s rare that applications read bytes from a file and process them one at a time until the end of the file is reached. For this reason, I think it makes more sense to have the **Read** method always throw an exception when attempting to read past the end of a file.

**Important** Many developers are misguided by the term *exception handling*. These developers believe that the word *exception* is related to how *frequently* something happens. For example, a developer designing a file **Read** method is likely to say the following: "When reading from a file, you will eventually reach the end of its data. Since reaching the end will *always* happen, I’ll design my **Read** method so that it reports the end by returning a special value; I won’t have it throw an exception." The problem with this statement is that it is being made by the developer designing the **Read** method, not by the developer calling the **Read** method.

When designing the **Read** method, it is impossible for the developer to know all the possible situations in which the method gets called. Therefore, the developer can’t possibly know how *often* the caller of the **Read** method will attempt to read past the end of the file. In fact, since most files contain structured data, attempting to read past the end of a file is something that *rarely* happens.

Another common misconception is that an “exception” identifies an “error”. The term *error* implies that the programmer did something wrong. However, again, it isn’t possible for the developer designing the **Read** method to know when the caller has called the method incorrectly for the application. Only the developer calling the method can determine this, and therefore only the caller can decide if the results of the call indicate an "error." So you should avoid thinking, "I’ll throw an exception here in my code to report an error." In fact, because exceptions don’t necessarily indicate errors, I’ve avoided using the term *error handling* throughout this entire chapter (except for this sentence, of course).

The preceding note explained what *exception* does not mean. Now I’ll describe what it does mean. An *exception* is the violation of a programmatic interface’s implicit assumptions. For example, when designing a type, you first imagine the various situations for how the type will be used. Then you define the fields, properties, methods, events, and so on for the type. The way you define these members (property data types, method parameters, return values, and so forth) becomes the programmatic interface to your type.

The interface you define carries with it some implicit assumptions. An exception occurs when an assumption made by your programming interface is violated.

Look at the following class definition:

```
public class Account {
   public static void Transfer(Account from, Account to, Decimal amount) {
      Ã
```

```
   }
}
```

The **Transfer** method accepts two **Account** objects and a **Decimal** value that identifies an amount of money to transfer between accounts. When calling the **Trans-fer** method, some obvious assumptions are implied: the **from** argument refers to a valid **Account**, and the account has enough money in it to subtract the specified amount. Also, it isn't clear from this prototype whether **amount** must be a positive value or if **amount** can be negative. In addition, what happens if the **from** argument and the **to** argument refer to the same account? Is it "legal" to transfer money to and from the same account? Also, what if the **amount** argument is outside a specific range set by the class designer? Is it "legal" to transfer an amount of 0?

The answers to the questions posed in the preceding paragraph are baked into **Transfer**'s implementation by the developer of the **Account** class. Ideally, the developer designing this class will clearly document all these assumptions so that developers using the class can implement the calling code in the most efficient way possible and so that few surprises occur at run time. Unfortunately, developers usually find that documentation lacks a description of all the implicit assumptions, causing developers to discover their violations at run time. Hopefully, all these violations will be detected and corrected during the application's test cycle so that no violations occur after the application is deployed and running in the hands of users.

How does a method notify its caller that one of its assumptions has been violated? It throws an exception. After all, an exception is simply a violation of a programming interface's assumptions.

The next point I want you to understand is that violating a programming interface's assumption isn't necessarily a bad thing. In fact, it can be a good thing because exception handling allows you to catch the exception and gracefully continue execution.

When designing a type, first imagine how the type is going to be most commonly used; then design your interface to work well with this usage. Also think about the implicit assumptions that your interface is introducing and throw exceptions when any assumption is violated. If your type is going to be used in lots of different situations, designing an interface that meets all the possible scenarios will be impossible. In this case, you must simply try your best and get feedback from users that you can take into account when designing the next version of your type's interface.

---

Implied Assumptions Developers Almost Never Think About

When accessing any method, developers make several assumptions—assumptions that we hardly ever think about. When we call a method, we assume that there is enough stack space; we assume that there is enough memory for the method's IL code to be JIT compiled into native code; and we assume that no bug will occur within the CLR itself when attempting the call. Although it happens rarely, these assumptions are sometimes violated.

As you write code to catch and deal with exceptions, keep in mind that these assumptions can be violated at any time, causing the CLR itself to throw a **System.StackOverflowException**, a **System.OutOfMemoryException**, or a **System.ExecutionEngineException** exception, respectively. For example, imagine the following method:

```
void InfiniteLoop() {
   while (true) ;
}
```

The loop in the preceding method could execute successfully 1000 times, but on the 1001st time an exception could be thrown. If the CLR needs to perform a garbage collection, it could hijack the calling thread (discussed in Chapter 19) and make it call an internal function, thereby causing a **StackOverflowException** to be thrown.

These three exceptions are unlike most others because they're usually thrown when the CLR is in a catastrophic situation and can't recover gracefully. Depending on the circumstance that causes one of these exceptions, your code might not be able to catch it and your **finally** blocks might not execute. The following list describes what happens when one of these special exceptions is thrown:

- **OutOfMemoryException** This exception is thrown when you try to new up an object and the garbage collector can't find any free memory. In this case, your application code can successfully catch this exception, and **finally** blocks will execute. This exception is also thrown when the CLR needs some internal memory and none is available. In this case, the CLR displays a message to the console and the process is immediately terminated: your application won't be able to catch this exception, and your **finally** blocks won't execute. Since the CLR can't recover gracefully when it runs out of memory internally, you should be careful when creating a server application using the .NET Framework. In particular, you should have a separate watchdog process that respawns the server automatically if it determines that the server has just terminated.
- **StackOverflowException** The CLR throws this exception when the thread has used all its stack space. Your application can catch this exception, but **finally** blocks won't execute since they would require additional stack space and none is available. Also, while a **catch** block might catch this exception (to log some information to help debugging), it should never swallow this exception. The reason is that the application is in an undefined state now because its **finally** blocks didn't execute. Any **catch** block that catches **StackOverflowException** should rethrow it—let the CLR terminate the process. If the stack overflow occurs within the CLR itself, your application code won't be able to catch the **StackOverflowException** exception and none of your finally blocks will execute. In this case, the CLR will connect a debugger to the process or, if no debugger is installed, just kill the process.
- **ExecutionEngineException** The CLR throws this exception when it detects that its internal data structures are corrupted or if it detects some bug in itself. When the CLR throws this exception, it will connect a debugger to the process; if no debugger is installed, it just kills the process. No **catch** blocks or **finally** blocks will be processed when this exception is thrown.

By the way, some other thread could always call **System.Threading.Thread**'s **Abort** method, forcing a **System.Threading.ThreadAbortException** exception to be thrown in a thread. This is another example demonstrating that an exception can get thrown at any time.

---

As you develop an application, you might try using a type whose interface isn't ideal for your situation thereby violating the type's assumptions frequently. Again, this isn't a bad thing because you can catch the exceptions and continue gracefully. For example, you might write an application that builds an index of all files on the user's drives. To build this index, you must open each file and parse its contents. However, some of the files you try to access might be secured, preventing the file from being opened. In this example, the application should expect lots of **System.Security.SecurityException** exceptions to be thrown. Don't think that exceptions always indicate a mistake (or an error) on your part—exceptions might be thrown frequently in your application because of the way your application needs to use the types designed by a different

developer. Your code can just catch these exceptions and recover from them as appropriate so that your application continues running. However, be aware that a performance penalty occurs while the system searches for a catch filter capable of handling the exception. The fewer exceptions an application throws, the faster it runs.

## The System.Exception Class

The common language runtime (CLR) allows an instance of any type to be thrown for an exception—from an **Int32** to a **String** and beyond. However, Microsoft decided against forcing all programming languages to throw and catch exceptions of any type. So Microsoft defined the **System.Exception** type and decreed that all CLS–compliant programming languages must be able to throw and catch exceptions whose type is derived from this type. Exception types that are derived from **System.Exception** are said to be CLS–compliant. C# and many other languages allow your code to throw only CLS–compliant exceptions.

The **System.Exception** type is a very simple type that contains the properties described in Table 18–1.

Table 18–1: Properties of the System.Exception Type

| Property | Access | Type | Description |
|---|---|---|---|
| **Message** | Read–only | **String** | Contains helpful text indicating why the exception was thrown. The message should be localized because a user might see this message if the application code doesn't catch the exception or if the application code does catch the exception in order to log it. |
| **Source** | Read/write | **String** | Contains the name of the assembly that generated the exception. |
| **StackTrace** | Read–only | **String** | Contains the names and signatures of methods called that led up to the exception being thrown. This property is very useful for debugging. |
| **TargetSite** | Read–only | **MethodBase** | Contains the method that threw the exception. |
| **HelpLink** | Read–only | **String** | Contains a URL (such as *file://C:\MyApp\Help.htm#–MyExceptionHelp*) to documentation that can help a user understand the exception. |
| **InnerException** | Read–only | **Exception** | Indicates the previous exception if the current exception was raised while handling an exception. This field is usually **null**. The **Exception** type also offers a public **GetBaseException** method that traverses the linked list of inner exceptions and returns the originally thrown exception. |
| **HResult** | Read/write | **Int32** | This protected property is used only for scenarios in which managed code and unmanaged COM code are interoperating. |

# FCL–Defined Exception Classes

The .NET Framework Class Library defines many exception types (all ultimately derived from **System.Exception**). The following hierarchy shows the exception types defined in the MSCorLib.dll assembly; other assemblies define even more exception types. (The application used to obtain this hierarchy is shown in Chapter 20.)

```
System.Exception
   System.ApplicationException
      System.Reflection.InvalidFilterCriteriaException
      System.Reflection.TargetException
      System.Reflection.TargetInvocationException
      System.Reflection.TargetParameterCountException
   System.IO.IsolatedStorage.IsolatedStorageException
   System.SystemException
      System.AppDomainUnloadedException
      System.ArgumentException
         System.ArgumentNullException
         System.ArgumentOutOfRangeException
         System.DuplicateWaitObjectException
      System.ArithmeticException
         System.DivideByZeroException
         System.NotFiniteNumberException
         System.OverflowException
      System.ArrayTypeMismatchException
      System.BadImageFormatException
      System.CannotUnloadAppDomainException
      System.ContextMarshalException
      System.ExecutionEngineException
      System.FormatException
         System.Reflection.CustomAttributeFormatException
      System.IndexOutOfRangeException
      System.InvalidCastException
      System.InvalidOperationException
         System.ObjectDisposedException
      System.InvalidProgramException
      System.IO.IOException
         System.IO.DirectoryNotFoundException
         System.IO.EndOfStreamException
         System.IO.FileLoadException
         System.IO.FileNotFoundException
         System.IO.PathTooLongException
      System.MemberAccessException
         System.FieldAccessException
         System.MethodAccessException
         System.MissingMemberException
            System.MissingFieldException
            System.MissingMethodException
      System.MulticastNotSupportedException
      System.NotImplementedException
      System.NotSupportedException
         System.PlatformNotSupportedException
      System.NullReferenceException
      System.OutOfMemoryException
      System.RankException
      System.Reflection.AmbiguousMatchException
      System.Reflection.ReflectionTypeLoadException
      System.Resources.MissingManifestResourceException
      System.Runtime.InteropServices.ExternalException
         System.Runtime.InteropServices.COMException
```

```
         System.Runtime.InteropServices.SEHException
      System.Runtime.InteropServices.InvalidComObjectException
      System.Runtime.InteropServices.InvalidOleVariantTypeException
      System.Runtime.InteropServices.MarshalDirectiveException
      System.Runtime.InteropServices.SafeArrayRankMismatchException
      System.Runtime.InteropServices.SafeArrayTypeMismatchException
      System.Runtime.Remoting.RemotingException
         System.Runtime.Remoting.RemotingTimeoutException
      System.Runtime.Remoting.ServerException
      System.Runtime.Serialization.SerializationException
      System.Security.Cryptography.CryptographicException
         System.Security.Cryptography.CryptographicUnexpectedOperationException
   System.Security.Policy.PolicyException
      System.Security.SecurityException
      System.Security.VerificationException
      System.Security.XmlSyntaxException
      System.StackOverflowException
      System.Threading.SynchronizationLockException
      System.Threading.ThreadAbortException
      System.Threading.ThreadInterruptedException
      System.Threading.ThreadStateException
      System.TypeInitializationException
      System.TypeLoadException
         System.DllNotFoundException
         System.EntryPointNotFoundException
      System.TypeUnloadedException
      System.UnauthorizedAccessException
```

Microsoft's idea was that **Exception** would be the base type for all exceptions and that two other types, **System.SystemException** and **System.ApplicationException**, would be derived from **Exception**.

The CLR throws types derived from **SystemException**. Most of the exceptions derived from **SystemException** (such as **DivideByZeroException**, **InvalidCastException**, and **IndexOutOfRangeException**) signal nonfatal situations that an application might be able to recover from. However, some of the exceptions, such as **StackOverflowException**, are considered fatal. An application shouldn't attempt to recover from these exceptions because it's extremely unlikely that recovery would be successful.

In addition, methods defined by FCL types are also supposed to throw exceptions derived from **SystemException**. For example, all FCL methods validate their arguments before attempting to perform any operations. If an argument doesn't live up to the method's implicit assumptions, an **ArgumentNullException**, **ArgumentOutOfRangeException**, or **DuplicateWaitObjectException** exception is thrown. All these exceptions are derived from **ArgumentException;** this allows an application to catch **ArgumentException** as a convenient way to catch any of the specific argument exception types.

Microsoft's idea for the **ApplicationException** type was that it would be a base type reserved solely for an application's use; that is, Microsoft wouldn't use **ApplicationException** as the base type for any of its own exception types.

If you examine the exception type hierarchy, you'll see that Microsoft's developers didn't exactly follow Microsoft's guidelines. The FCL defines some reflection–related exception types derived from **ApplicationException**. In addition, some exception types, such as **IsolatedStorageException**, are derived directly from **Exception** instead of **SystemException**.

You might think that these “bugs” are bad. However, before you make that judgment, you should wonder about the value of having all the CLR/FCL exception types derived from **SystemException** and all your application exception types derived from **ApplicationException**. Once you start thinking about exception hierarchies, it doesn’t take long before you realize that the sole benefit of a hierarchy is to allow code to catch a related set of exception types easily. In other words, it’s easier to write code that catches an **ArithmeticException** than it is to catch all the exception types derived from it: **DivideByZeroException**, **NotFiniteNumberException**, and **OverflowException**.

Now, would you ever want to catch all exceptions derived from **SystemException** versus all exceptions derived from **ApplicationException**? I don’t think so. On the other hand, there are times when you’ll want to know if any exception is thrown, and you can gain this knowledge easily by catching **System.Exception**. So it does make sense that all exception types are derived from **Exception;** it also makes sense that **DirectoryNotFoundException**, **EndOfStreamException**, **FileLoadException**, and **FileNotFoundException** are derived from **IOException**. However, I don’t think there is any value in having the **SystemException** and **ApplicationException** base types in the exception hierarchy. In fact, I think having them is just confusing.

Also, I think that the two special exceptions, **ExecutionEngineException** and **StackOverflowException**, should be in a special hierarchy because they are unlike any other exceptions. Only the CLR itself—never the application code—should be able to throw one of these exceptions because an application can’t recover gracefully from any of them.

## Defining Your Own Exception Class

When implementing your methods, you might come across scenarios in which you want to throw an exception. For example, I recommend that your nonprivate methods always validate their arguments, and if any argument doesn’t live up to your method’s implicit assumptions, an exception should be thrown. In this case, I recommend that you throw one of the exception classes already defined in the FCL: **ArgumentNullException**, **ArgumentOutOfRangeException**, or **DuplicateWaitObjectException**.

I strongly suggest that you throw a specific exception class, a class that has no other classes derived from it. For example, don’t throw an **ArgumentException** because it’s too vague, it could mean any of its three derived types, and it doesn’t provide as much information as possible to its catchers. You should never throw **Exception**, **ApplicationException**, or **SystemException**.

**Note** Throwing an instance of an exception class that you’ve defined provides your catch code with the capability to know exactly what happened and to recover in any way it sees fit.

Now let’s say that you’re defining a method that’s passed a reference to an object whose type must implement the **ICloneable** and **IComparable** interfaces. You might implement the method like this:

```
class SomeType {
   public void SomeMethod(Object o) {
      if (!((o is ICloneable) && (o is IComparable)))
         throw new MissingInterfaceException(...);

      // Code that operates on o goes here.
      Ã
   }
}
```

Because the FCL doesn't define an appropriate exception type, you must define the **MissingInterfaceException** type yourself. Note that by convention the name of an exception type should end with "Exception". When defining this type, you must decide what its base type will be. Should you choose **Exception**, **ArgumentException**, or a different type entirely? I've spent months thinking about this question, but unfortunately, I can't come up with a good rule of thumb to offer you, and here's why.

If you derive **MissingInterfaceException** from **ArgumentException**, any existing code that's already catching **ArgumentException** will catch your new exception, too. In some ways this is a feature, and in some ways this is a bug. It's a feature because any code that wants to catch any kind of argument exception (via **ArgumentException**) now catches this new kind of argument exception (**MissingInterfaceException**) automatically. It's a bug because a **MissingInterfaceException** identifies a new event that wasn't anticipated when code was written to catch an **ArgumentException**. When you define the **MissingInterfaceException** type, you might think it's so similar to an **ArgumentException** that it should be handled the same way. However, this unanticipated relationship might cause unpredictable behavior.

On the other hand, if you derive **MissingInterfaceException** directly from **Exception**, the code throws a new type that the application couldn't have known about. Most likely, this will become an unhandled exception that causes the application to terminate. I could easily consider this desired behavior because an implicit assumption was violated and the application never considered a remedy for it. Catching this new exception, swallowing it, and continuing execution might cause the application to run with unpredictable results.

Answering questions like these is one of the reasons that application design is more of an art form than a science. When defining a new exception type, carefully consider how application code will catch your type (or base types of your type), and then choose a base type that has the least negative effect on your callers.

When defining your own exception types, feel free to define your own subhierarchies, if applicable to what you're doing. You can define them directly from **Exception** or from some other base type. Again, just make sure that where you're putting the subhierarchy makes some sense for your callers. As a general rule, hierarchies should be broad and shallow: exception types should be derived from a type close to **Exception** and should typically be no more than two or three levels deep. If you define an exception type that's not going to be the base of other exception types, then mark the type as **sealed**.

The **Exception** base type defines three public constructors:

- A parameterless (default) constructor that creates an instance of the type and sets all fields and properties to default values.
- A constructor taking a **String** that creates an instance of the type and sets a specific message.
- A constructor taking a **String** and an instance of an **Exception**–derived type that creates an instance of the type and sets a specific message and an inner exception. Unfortunately, there are a lot of **Exception**–derived types in the FCL that don't have this constructor. Microsoft will rectify this oversight in a future version of the .NET Framework.

When defining your own exception type, your type should implement a set of three matching constructors and call the base type's corresponding constructor.

Of course, your exception type will inherit all the fields and properties defined by **Exception**. In

addition, you might add fields and properties of your own. For example, the **System.ArgumentException** exception adds a **String** property called **ParamName**. The **ArgumentException** type also defines new constructors (in addition to the three public constructors) that take an extra **String** as a parameter so that the **ParamName** property can be initialized to identify the name of the parameter that violated the method's implicit assumptions.

When an **ArgumentException** is caught, the **ParamName** property can be read to determine exactly which parameter caused the problem. Let me tell you, this is incredibly handy when you're trying to debug an application! If you do add fields to your exception type, make sure you define some constructors that allow the fields to be initialized.

Exception types should always be serializable so that the exception object can be marshaled across an AppDomain or a machine boundary and rethrown in the client's code. Making an exception type serializable also allows it to persist in a log or a database. To make your exception type serializable, you must apply the **[Serializable]** custom attribute to the type, and if the type defines any fields of its own, you must also implement the **ISerializable** interface along with its **GetObjectData** method and special protected constructor, both of which take **SerializationInfo** and **StreamingContext** parameters. The following code shows how to properly define your own exception type:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

// Allow instances of DiskFullException to be serialized.
[Serializable]
sealed class DiskFullException : Exception, ISerializable {
   // The three public constructors
   public DiskFullException()
      : base() {  // Call base constructor.
   }

   public DiskFullException(String message)
      : base(message) { // Call base constructor.
   }


   public DiskFullException(String message, Exception innerException)
      : base(message, innerException) {   // Call base constructor.
   }

   // Define a private field.
   private String diskpath;

   // Define a read-only property that returns the field.
   public String DiskPath { get { return diskpath; } }

   // Override the public Message property so that the
   // field is included in the message.
   public override String Message {
      get {
         String msg = base.Message;
         if (diskpath != null)
            msg += Environment.NewLine + "Disk Path: " + diskpath;
         return msg;
      }
   }

   // Because at least one field is defined, define the
```

```
   // special deserialization constructor. Because this
   // class is sealed, this constructor is private. If this
   // class is not sealed, this constructor should be protected.
   private DiskFullException(SerializationInfo info,
      StreamingContext context)
      : base(info, context) { // Let the base deserialize its fields.

      // Deserialize each field.
      diskpath = info.GetString("DiskPath");
   }

   // Because at least one field is defined,
   // define the serialization method.
   void ISerializable.GetObjectData(SerializationInfo info,
      StreamingContext context) {
      // Serialize each field.
      info.AddValue("DiskPath", diskpath);

      // Let the base type serialize its fields.
      base.GetObjectData(info, context);
   }

   // Define additional constructors that set the field.
   public DiskFullException(String message, String diskpath)
      : this(message) { // Call another constructor.
      this.diskpath = diskpath;
   }

   public DiskFullException(String message, String diskpath,
      Exception innerException)
      : this(message, innerException) {   // Call another constructor.
      this.diskpath = diskpath;
   }
}


// The following code tests the serialization of the exception.
class App {
   static void Main() {

      // Construct a DiskFullException object, and serialize it.
      DiskFullException e =
         new DiskFullException("The disk volume is full", @"C:\");
      FileStream fs = new FileStream(@"Test", FileMode.Create);
      IFormatter f = new SoapFormatter();
      f.Serialize(fs, e);
      fs.Close();

      // Deserialize the DiskFullException object, and check its fields.
      fs = new FileStream(@"Test", FileMode.Open);
      e = (DiskFullException) f.Deserialize(fs);
      fs.Close();
      Console.WriteLine("Type: {1}{0}DiskPath: {2}{0}Message: {3}",
         Environment.NewLine, e.GetType(), e.DiskPath, e.Message);
   }
}
```

## How to Use Exceptions Properly

Understanding the exception mechanism is certainly important; equally important is understanding

how to use exceptions wisely. All too often I see library developers catching all kinds of exceptions, preventing the application developer from knowing that a problem occurred. In this section, I offer some guidelines that all developers should be aware of when using exceptions.

**Important** If you're a *class library developer*, developing types that will be used by other developers, take these guidelines very seriously. You have a huge responsibility: you're trying to design the type's interface so that its implicit assumptions make sense for a wide variety of applications. Remember that you don't have intimate knowledge of the code you're calling (via delegates, virtual methods, or interface methods). And you don't know what code is calling you. It's not feasible to anticipate every situation in which your type will be used, so don't make any policy decisions. Your code must not decide what is an error; let the caller make this decision. Follow the guidelines in this chapter, or application developers will have a difficult time using the types in your class library.

If you're an *application developer*, define whatever policy you think is appropriate. Following the design guidelines in this chapter will help you discover problems in your code sooner, allowing you to fix them and make your application more robust. However, feel free to diverge from these guidelines after careful consideration. You get to set the policy. For example, application code can get more aggressive about catching exceptions.

## You Can't Have Too Many finally Blocks

I think **finally** blocks are awesome! They allow you to specify a block of code that's guaranteed to execute no matter what kind of exception the thread throws You should use **finally** blocks to clean up from any operation that successfully started before returning to your caller or continuing to execute code following the **finally** block. You also frequently use **finally** blocks to explicitly dispose of any objects to avoid resource leaking. Here's an example that has all cleanup code (closing the file) in a **finally** block:

```
class SomeType {
   void SomeMethod() {

      // Open a file.
      FileStream fs = new FileStream(@"C:\ReadMe.txt", FileMode.Open);
      try {
         // Display 100 divided by the first byte in the file.
         Console.WriteLine(100 / fs.ReadByte());
      }
      finally {
         // Put cleanup code in a finally block to ensure that
         // the file gets closed regardless of whether or not an
         // exception occurs (for example, the first byte was 0).
         fs.Close();
      }
   }
}
```

Ensuring that cleanup code always executes is so important that many programming languages offer constructs that make coding easier. For example, the C# language provides the **lock** and **using** statements. (I'll touch on **using** statements again in Chapter 19.) These statements provide the developer with a simple syntax that causes the compiler to automatically generate **try** and **finally** blocks, where the **finally** block contains the cleanup code. For example, the following C# code takes advantage of the **using** statement. This code is shorter than the code shown in the

previous example, but the code the compiler generates is identical to the code generated in the previous example.

```
class SomeType {
   void SomeMethod() {

      // Open a file.
      using (FileStream fs =
         new FileStream(@"C:\ReadMe.txt", FileMode.Open)) {

         // Display 100 divided by the first byte in the file.
         Console.WriteLine(100 / fs.ReadByte());
      }
   }
}
```

## Don't Catch Everything

A common mistake is to use **catch** blocks too often and to use them improperly. When you catch an exception, you're stating that you expected this exception, you understand why it occurred, and you know how to deal with it. In other words, you're defining a policy for the application. All too often, I see code like this:

```
catch (Exception) {
   Ã
}
```

This code indicates that it was expecting *any* and *all* exceptions and knows how to recover from *any* and *all* situations. How can this possibly be? A type that's part of a class library should never catch all exceptions because there is no way for the type to know exactly how the application intends to respond to an exception. In addition, the type will frequently call out to application code via a delegate or a virtual method. If the application code throws an exception, another part of the application is probably expecting to catch this exception. The exception should filter its way up the call stack and let the application code handle the exception as it sees fit.

You might identify specific places in your code where you want to catch everything that can go wrong and gracefully recover. The temptation is to add a **catch** block that catches **System.Exception**. As an illustration of what *not* to do, consider the virtual **Equals** method (defined by **System.Object**). This method should return **false** if two objects don't have the same logical value. So if some code tries to compare an **Apple** to an **Orange**, **Equals** should return **false**. Here's how you might (incorrectly) implement **Apple**'s **Equals** method:

```
sealed class Apple : Object {
   Color c = Color.Red;   // The color of the apple

   public override Boolean Equals(Object o) {
      Boolean equal = false;  // Assume objects aren't equal.

      try {
         // Cast o to an Apple.
         Apple a = (Apple) o;

         // Compare the fields of 'this' and a.
         // If any fields aren't equal, leave the try block.
         if (this.c != a.c) goto leave;

         // If all fields have the same value, they're equal.
```

```
            equal = true;

            // Leave the try block. NOTE: C# doesn't offer a
            // leave keyword, so you have to goto a label.
            leave:;
         }
         catch (Exception) {
         }
         return equal;
      }
   }
```

Inspecting this code, you see that attempting to cast **o** to an **Apple** might cause the CLR to throw an **InvalidCastException** exception. In fact, this is the only real exception I can imagine the method throwing. Because the **InvalidCastException** is the only exception you expect, you might feel that it's OK to catch **Exception** as shown in the code. In fact, you might think that it's OK to catch **Exception** even if anything else were to throw an exception. After all, **Equals** should return **true** or **false**.

However, you shouldn't catch **Exception** here because a **StackOverflowException** or **OutOfMemoryException** exception could be thrown at any time. Because of the way the previous code is written, **Equals** would catch either of these two exceptions and simply return **false** to its caller. **Equals** is now hiding these fatal problems and allowing the application to continue running with unpredictable results. This situation is certainly not desired! To fix the code, catch **InvalidCastException** instead of **Exception**. An **InvalidCastException** is the only exception that this code knows how to recover from gracefully; all other exceptions are not anticipated by **Equals** and should be allowed to propagate out of the method.

## Gracefully Recovering from an Exception

Sometimes you call a method knowing in advance some of the exceptions that the method might throw. Because you expect these exceptions, you might want to have some code that allows your application to recover gracefully from the situation and continue running. Here's an example in pseudocode.

```
public String CalculateSpreadsheetCell(Int32 row, Int32 column) {
   String result;
   try {
      result = /* Code to calculate value of a spreadsheet's cell */
   }
   catch (DivideByZeroException) {
      result = "Can't show value: Divide by zero";
   }
   return result;
}
```

This pseudocode calculates the contents of a cell in a spreadsheet and returns a string representing the value back to the caller so that the caller can display the string in the application's window. However, a cell's contents might be the result of dividing one cell by another cell. If the cell containing the denominator contains 0, the CLR will throw a **DivideByZeroException** exception. In this case, the method catches this specific exception and returns a special string that will be displayed to the user.

When you catch specific exceptions, fully understand the circumstances that cause the exception to be thrown and know what exception types are derived from the exception type you're catching. Don't catch and handle **System.Exception** because it's not feasible for you to know all the possible

exceptions that could be thrown within your **try** block (especially if you consider **OutOfMemoryException**, **OverflowException**, **StackOverflowException**, or even **ExecutionEngineException**, to name a few).

## Backing Out of a Partially Completed Operation When an Unrecoverable Exception Occurs

Usually methods call several other methods to perform a single abstract operation. Some of the individual methods might complete successfully, and some might not. For example, a method that transfers money from one account to another account might first add money to one account and then subtract money from the second account. If the first operation completes successfully but the second operations fails (for any reason), the money must be subtracted from the first account so that the accounts balance.

Here is another, perhaps more meaningful, example: Let's say that you're serializing a set of objects to a disk file. After serializing 10 objects, an exception is thrown. (Perhaps the disk is full or the next object to be serialized isn't marked with the **Serializable** custom attribute.) At this point, the exception should filter up to the caller, but what about the state of the disk file? The file is now corrupt because it contains a partially serialized object graph. It would be great if the application could back out of the partially completed operation so that the file would be in the state it was before any objects were serialized into it. The following code demonstrates the correct way to implement this:

```
public void SerializeObjectGraph(FileStream fs,
   IFormatter formatter, Object rootObj) {

   // Save the current position of the file.
   Int64 beforeSerialization = fs.Position;

   try {
      // Attempt to serialize the object graph to the file.
      formatter.Serialize(fs, rootObj);
   }
   catch {  // Catch all CLS and non-CLS exceptions.
      // If ANYTHING goes wrong, reset the file back to a good state.
      fs.Position = beforeSerialization;

      // Truncate the file.
      fs.SetLength(fs.Position);

      // NOTE: The preceding code isn't in a finally block because
      // the stream should be reset only when serialization fails.

      // Let the caller(s) know what happened by
      // rethrowing the SAME exception.
      throw;
   }
}
```

To properly back out of the partially completed operation, write code that catches all exceptions. Yes, catch *all* exceptions here because you don't care what kind of error occurred; you need to put your data structures back into a consistent state. After you've caught and handled the exception, don't swallow it—let the caller know that the exception occurred. You do this by rethrowing the same exception. In fact, C# and many other languages make this easy. Just use C#'s **throw** keyword without specifying anything after **throw**, as shown in the previous code.

Notice that the **catch** block in the previous example doesn't specify any exception type because I want to catch non–CLS–compliant exceptions *and* CLS–compliant exceptions. Fortunately, C# lets me do this easily by just not specifying any exception type and by making the **throw** statement rethrow whatever object is caught.

## Hiding an Implementation Detail

In some situations, you might find it useful to catch one exception and rethrow a different exception. Here's an example:

```
public Int32 SomeMethod(Int32 x){
   try {
      return 100 / x;
   }
   catch (DivideByZeroException e) {
      throw new ArgumentOutOfRangeException("x", x, "x can't be 0", e);
   }
}
```

When **SomeMethod** is called, the caller passes in an **Int32** value and the method returns 100 divided by this value. Upon entry into the method, code could check to see whether **x** is 0, and if it is, throw an **ArgumentOutOfRangeException** exception at that point. However, this check would be performed every time, and because there is an implicit assumption that **x** is rarely 0, the check would cause a performance hit. So, this method assumes that **x** is not 0 and attempts to divide 100. Now, if **x** does happen to be 0, then the specific **DivideByZeroException** exception is caught and is rethrown as an **ArgumentOutOfRangeException** exception. Note that the **DivideByZeroException** exception is set as the **ArgumentOutOfRangeException**'s **InnerException** property via the constructor's fourth argument.

**Important** In this discussion, I've shown how to catch an exception and throw a different exception. When you use this technique, the new exception should have its inner exception property refer to the original exception. The preceding code demonstrates how you should do this correctly. Unfortunately, that code won't compile. The problem is that many of the FCL exception types don't offer constructors that take an **innerException** parameter. These missing constructors are bugs that Microsoft says they will correct in future versions of the .NET Framework. Personally, I've found these bugs to be quite frustrating, and there is no workaround because the **Exception** type offers no way to set an inner exception.

This technique results in catches similar to those discussed in the earlier section "Gracefully Recovering from an Exception." You catch specific exceptions, you fully understand the circumstances that cause the exception to be thrown, and you know what exception types are derived from the exception type you're catching.

Again, class library developers shouldn't catch **System.Exception** and the like. Doing so means that you're converting all exception types into a single exception type. It discards all meaningful information (the type of the exception) and throws a single exception type that doesn't contain any useful information about what really happened. Without this information, it's much harder for code higher up the call stack to catch and handle a specific exception. Give the code higher up the call stack a chance to catch **System.Exception** or some other exception that's a base type for more specific exceptions.

Basically, the only time to catch an exception and rethrow a different exception is to improve the meaning of a method's abstraction. Also, the new exception type you throw should be a specific

exception (an exception that's not used as the base type of any other exception type). Imagine a **PhoneBook** type that defines a method that looks up a phone number from a name, shown in the following pseudocode:

```
class PhoneBook {
   String pathname;  // pathname for file containing the address book

   // Other methods go here.

   public String GetPhoneNumber(String name) {
      String phone;
      FileStream fs = null;
      try {
         fs = new FileStream(pathname, FileMode.Open);
         (Code to read from fs until name is found)
         phone = /* the phone # found */
      }
      catch (FileNotFoundException e) {
         // Throw a different exception containing the name, and
         // set the originating exception as the inner exception.
         throw new NameNotFoundException(name, e);
      }
      catch (IOException e) {
         // Throw a different exception containing the name, and
         // set the originating exception as the inner exception.
         throw new NameNotFoundException(name, e);
      }
      finally {
         if (fs != null) fs.Close();
      }
      return phone;
   }
}
```

The phonebook data is obtained from a file (versus a network connection or database). However, the user of the **PhoneBook** type doesn't know this. So if the file isn't found or can't be read for any reason, the caller would see a **FileNotFoundException** or **IOException**, which wouldn't be anticipated. In other words, the file's existence and ability to be read is part of the method's implied assumptions. However, there is no way that the caller could have guessed this. So the **GetPhoneNumber** method catches these two exception types and throws a new **NameNotFoundException**.

Throwing an exception still lets the caller know that an implied assumption was violated, and the **NameNotFoundException** type gives the caller an abstracted view of the implied assumption that was violated. Setting the inner exception to **FileNotFoundException** or **IOException** is important so that the real cause of the exception isn't lost; knowing what caused the exception could be useful to the developer of the **PhoneBook** type.

Now let's say that the **PhoneBook** type was implemented a little differently. Assume that the type offers a public **PhoneBookPathname** property that allows the user to set or get the pathname of the file in which to look up a phone number. Because the user is aware that the phone book data comes from a file, I would modify the **GetPhoneNumber** method so that it doesn't catch any exceptions; instead, I let whatever exception is thrown propagate out of the method. Note that I'm not changing any parameters of the **GetPhoneNumber** method but I am changing how it's abstracted to users of the **PhoneBook** type.

Again, let me emphasize that class library developers should follow all the design guidelines

mentioned here very carefully. As I said earlier, an application developer, however, should always be able to define whatever policy he or she thinks is appropriate. Following the design guidelines in this chapter will probably help you discover problems in your code sooner, allowing you to fix them and make your application more robust. Application developers should feel free to diverge from these guidelines after careful consideration. They get to set any policy they think is best; the application code can get more aggressive about catching exceptions, for example.

## What's Wrong with the FCL

In this chapter, I've given you my recommendations for working with exceptions. These recommendations are based on conversations with many developers and my own experience writing code for many years. In the previous section, I mentioned that many of the FCL exception types are missing a constructor that allows the inner exception to be set. This is just one type of bug in the FCL. Unfortunately, the FCL contains many more bugs related to exception handling. In this section, my intent is to make you aware of these bugs so that you won't waste as much time as I have trying to figure out what's going on when your code doesn't work quite as you expect.

Microsoft hasn't followed a lot of the guidelines that I describe in this chapter. In fact, quite a bit of Microsoft's code violates Microsoft's own guidelines, a situation that can make working with the FCL difficult at times. The problem is exacerbated by the fact that the FCL documentation doesn't always describe what exceptions a developer can expect or how to recover from them.

The first problem is that Microsoft's FCL code is peppered with the following constructs:

```
catch { ... }

catch (Exception) { ... }

catch (ApplicationException) { ... }
```

As I explained previously, these constructs catch and swallow exceptions that shouldn't be caught and swallowed by a class library.

Here's an example: the **System.IO.Directory** and **System.IO.File** types have static **Exists** methods. Both methods return **true** or **false** depending on whether the path argument identifies a directory/file that exists on the user's disk drive. If anything goes wrong inside these methods, the methods catch **Exception** and return **false** to their caller. The caller has no way to tell whether the exception occurred because the file doesn't exist or whether the caller doesn't have sufficient access to the directory or file. Also, if a **StackOverflowException** or an **OutOfMemoryException** is thrown, then **false** is also returned!

The second problem is that Microsoft's FCL code frequently catches an exception and throws a new exception. As explained previously, this can be useful if the new exception provides necessary information, but the FCL often hides what actually happened from the application developer.

For example, the **System.Array** type's **Sort** method sorts an array of objects by calling each object's **CompareTo** method. If the **CompareTo** method throws an exception (any exception), the **Sort** method catches it and throws a new **InvalidOperationException**. This is terrible! It means that your code could throw an exception that your code can never catch. There's no reason for **Array**'s **Sort** method to do this. By the way, **Array**'s **BinarySearch** method also catches any and all exceptions and throws a new **InvalidOperationException**, which is equally as troublesome.

The third problem with the FCL is the way it handles reflection (a topic I'll discuss more in Chapter 20). If you obtain a **MethodInfo** for a method and call its **Invoke** method, the method being invoked could throw an exception. Unfortunately, the **MethodInfo**'s **Invoke** method catches any and all exceptions and throws a new **System.Reflection.TargetInvocationException**. For this reason, you can't catch the actual exception thrown by the method you invoked.

Another example of this third problem is that the FCL code frequently catches one exception and throws a new **System.Exception**. Again, useful information is lost because the caller must now catch **Exception** and then figure out what really went wrong.

Here's another problem with the FCL code: the **System.Windows.Forms.DataGrid** control has a public **CurrentCell** property. If your code tries to set the value in the current cell and an exception is thrown, the **DataGrid** control catches all **Exception**–derived exceptions and displays a message box! I couldn't believe this when I discovered it. Not only can you not catch the exception, but the user is also confronted with a message box that your application can't control.

I've run into these and other problems the hard way: writing code to handle what I expected, discovering that my code didn't work right, and then finding out that the FCL code had been swallowing my exceptions. It's my fervent hope that Microsoft will examine the FCL source code and correct it so that it follows the design guidelines set out in this chapter.

Experience is the best way to learn which FCL methods will cause you problems. The only real workaround is to define exception types that the FCL code can't catch.

You might reasonably conclude that you should define and throw non–CLS–compliant exceptions, which the FCL code rarely catches. However, I don't recommend this for several reasons:

- Programmers working in other languages might not be able to catch non–CLS–compliant exceptions.
- C# and many other languages don't let you throw non–CLS–compliant exceptions.
- C# and many other languages can catch non–CLS–compliant exceptions, but they don't let you get any information about the exception. You can't tell what type the exception is, nor can you get a stack trace or helpful string message.

Even if you did use non–CLS–compliant exceptions, there's no guarantee that the FCL won't catch them and swallow them or throw a new exception type in its place. Unfortunately, there is no satisfactory remedy until Microsoft fixes the FCL exception handling code.

## Performance Considerations

The developer community actively debates the performance of exception handling. My experience is that the benefit of exception handling far outweighs any performance penalties. In this section, I'll address some of the performance issues related to exception handling.

It's difficult to compare performance between exception handling and the more conventional means of reporting exceptions (**HRESULT**s, special return codes, and so forth). If you write code to check the return value of every method call and filter the return value up to your own callers, your application's performance will be seriously affected. But performance aside, the amount of additional coding you must do and the potential for mistakes is incredibly high. Exception handling is a much better alternative.

Unmanaged C++ compilers must generate code that tracks which objects have been constructed successfully. The compiler must also generate code that, when an exception is caught, calls the destructor for each of the successfully constructed objects. It's great that the compiler takes on this burden, but it generates a lot of bookkeeping code in your application, adversely affecting code size and execution time.

On the other hand, managed compilers have it much easier because managed objects are allocated in the managed heap, which is monitored by the garbage collector (GC). If an object is successfully constructed and an exception is thrown, the GC will eventually deallocate the object. Compilers don't need to emit any bookkeeping code to track which objects are constructed successfully and to ensure that a destructor is called (especially because managed objects are destroyed at a nondeterministic time). Compared to unmanaged C++, this means that less code is generated by the compiler and less code has to execute at run time: your application's performance is better.

Over the years, I've used exception handling in different languages, different operating systems, and different CPU architectures. In each case, exception handling is implemented differently. Each implementation has its pros and cons with respect to performance. Some implementations compile exception handling constructs directly into a method, while other implementations store information related to exception handling in a data table associated with the method—this table is accessed only if an exception is thrown. Some compilers can't inline methods that contain exception handlers, and some compilers won't enregister variables if the method contains exception handlers.

The point is that you can't determine how much additional overhead using exception handling adds to an application. In the managed world, it's even more difficult to tell because your assembly's code can run on any platform that supports the .NET Framework. So the code produced by the JIT compiler to manage exception handling when your assembly is running on an x86 machine will be very different than the code produced by the JIT compiler when your code is running on an IA64 processor or the code produced by the .NET Compact Framework's JIT compiler.

Actually, I've been able to test some of my own code with a few different JIT compilers that Microsoft has internally, and the difference in performance that I've observed has been quite dramatic and surprising. The point is that you must test your code on the various platforms that you expect your users to run on and make changes accordingly. Again, I wouldn't worry about the performance of using exception handling; as I've said, the benefits far outweigh any negative performance impact.

If you're interested in seeing how exception handling impacts the performance of your code, you can use PerfMon.exe or the System Monitor ActiveX control that comes with Windows NT 4, Windows 2000, Windows XP, and the Windows .NET Server product family. The screen in Figure 18–1 shows the exception–related counters that get installed when the .NET Framework is installed.

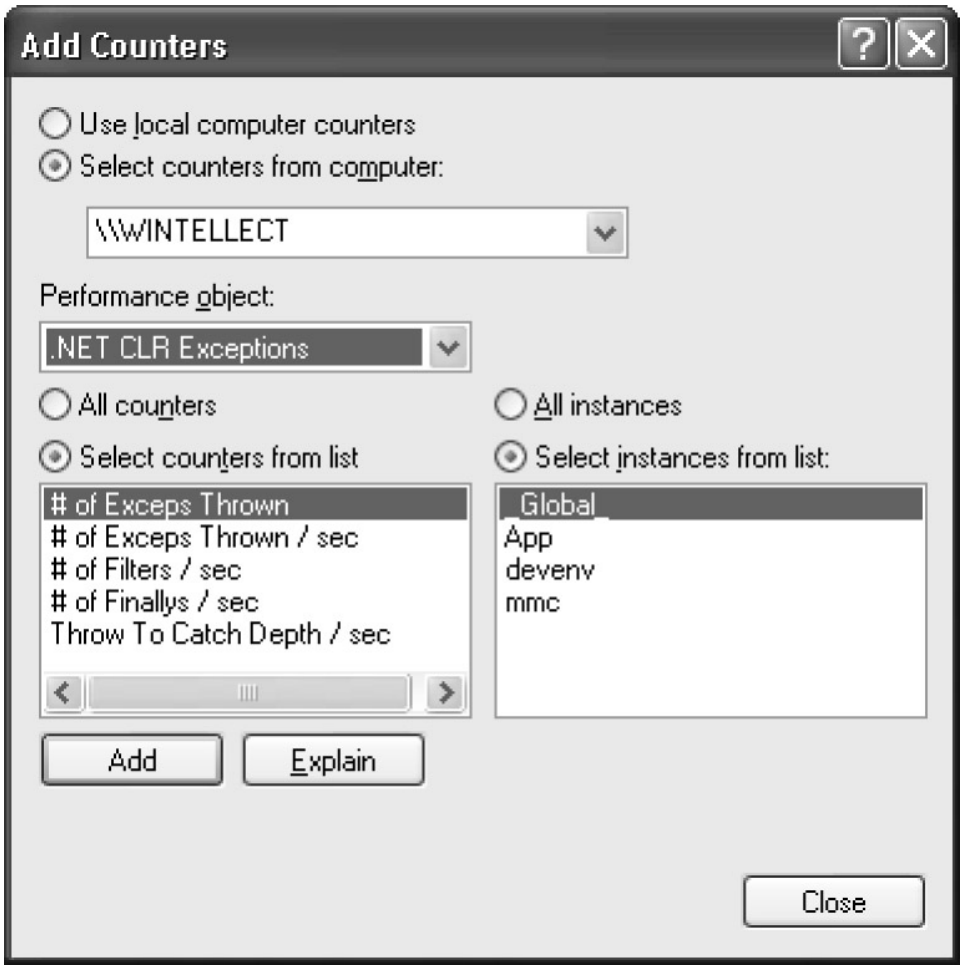


Figure 18–1 : PerfMon.exe showing the .NET CLR exception counters
Here's what each counter means:

- **# Of Exceps Thrown** Displays the total number of exceptions thrown since the application started. These include both .NET exceptions and unmanaged exceptions that get converted into .NET exceptions. For example, a null pointer reference exception in unmanaged code would get rethrown in managed code as a .NET **System.NullReferenceException** exception; this counter includes both handled and unhandled exceptions. Exceptions that are rethrown would get counted again.
- **# Of Exceps Thrown/Sec** Displays the number of exceptions thrown per second. These include both .NET exceptions and unmanaged exceptions that get converted into .NET exceptions. For example, a null pointer reference exception in unmanaged code would get re–thrown in managed code as a .NET **System.NullReferenceException** exception; this counter includes both handled and unhandled exceptions. This counter was designed as an indicator of potential performance problems caused by a large (>100s) number of exceptions being thrown. This counter isn't an average over time; rather, it displays the difference between the values observed in the last two samples divided by the duration of the sample interval.
- **# Of Filters/Sec** Displays the number of .NET exception filters executed per second. An exception filter evaluates whether or not an exception should be handled. This counter tracks the rate of exception filters evaluated, regardless of whether or not the exception was handled. As with the preceding counter, this counter isn't an average over time; rather, it displays the difference between the values observed in the last two samples divided by the duration of the sample interval.

- **# Of Finallys/Sec** Displays the number of **finally** blocks executed per second. A **finally** block is guaranteed to be executed regardless of how the **try** block was exited. Only the **finally** blocks that are executed for an exception are counted; this counter doesn't count **finally** blocks on normal code paths. Again, this counter isn't an average over time; it displays the difference between the values observed in the last two samples divided by the duration of the sample interval.
- **Throw To Catch Depth/Sec** Displays the number of stack frames traversed from the frame that threw the .NET exception to the frame that handled the exception per second. This counter resets to 0 when an exception handler is entered, so nested exceptions would show the handler–to–handler stack depth. Again, this counter isn't an average over time; it displays the difference between the values observed in the last two samples divided by the duration of the sample interval.

## Catch Filters

When an exception is thrown, the CLR walks up the call stack looking at each **catch** block's catch filter—the exception type in a **catch** block's parentheses. The following code shows a **try** block with three **catch** blocks.

```
public void SomeMethod() {
   try {
      // Do something in here.
   }
   catch (NullReferenceException e) {
      // Handle a null reference exception.
   }
   catch (InvalidCastException e) {
      // Handle an Invalid cast exception.
   }
   catch {  // In C#, this filter catches everything.
      // Handle any kind of exception.
   }
}
```

When compiling this code, the compiler emits a tiny "catch filter funclet" for each **catch** block contained inside the **SomeMethod** method. When an exception is thrown, the CLR calls the **NullReferenceException** funclet and passes it the object that identifies the thrown exception. The catch filter funclet checks whether the type of the object is a **NullReferenceException** or a type derived from **NullReferenceException**. If there isn't a match, then the catch filter funclet returns a special value telling the CLR to continue searching. In my example, the **InvalidCastException** funclet is asked next, followed by the "catch all" filter funclet. If the "catch all" **catch** block didn't exist, then the CLR would continue walking up the call stack.

When a catch filter funclet recognizes the thrown object's type, it returns a special value to inform the CLR. The CLR executes all the **finally** blocks necessary to unwind and clean up the started operations farther down the call stack. Then the CLR passes execution to the code contained inside the **catch** block.

In C# and many other languages designed for the .NET Framework, a catch filter is simply a data type. The catch filter funclet matches the thrown object's type against the filter's specified type. However, the CLR supports more complex catch filters. Microsoft Visual Basic, C++ with Managed Extensions, and IL assembly language are the only languages that I'm aware of that allow more complex catch filters. Here is some contrived Visual Basic code that demonstrates a complex catch

filter:

```
Imports System
Public Module MainMod

Function HundredDivX(x as Int32) As String
   Try
      x = 100 / x
      HundredDivX = x.ToString()

      Dim a as Object
      Console.WriteLine(a.ToString())

   Catch e as Exception When x = 0
      HundredDivX = "Problem: x was 0"

   Catch e as Exception
      HundredDivX = "Problem: I don't know what the problem is"

   End Try
End Function


Sub Main()
   Console.WriteLine(HundredDivX(0))
   Console.WriteLine(HundredDivX(2))
End Sub

End Module
```

When you compile and execute this code, you get the following output:

```
Problem: x was 0
Problem: I don't know what the problem is
```

Here's what's happening. **Main** begins executing and calls **HundredDivX**, passing it **0**. Inside **HundredDivX**, **100** is divided by **0**, which causes the CLR to throw an **OverflowException**. (An **OverflowException** is thrown instead of a **DivideByZeroException** because Visual Basic treats the **100** as an 8–byte real number instead of an integer.)

When the exception is thrown, the CLR calls the first catch filter funclet (generated by the compiler) corresponding to the first **Catch** block. This complex catch filter funclet checks whether the **OverflowException** object is derived from **Exception**. Because it is, the filter then asks if **x** is **0**, by way of the Visual Basic **When** statement. Because this is also true, the catch filter funclet returns a value telling the CLR that it wants to handle the exception. The CLR now unwinds the stack by executing any **Finally** blocks farther down the call stack. In this example, there aren't any. Once all the **Finally** blocks have executed, execution passes to the code inside the handling **Catch** block. The return string is set to "Problem: x was 0"; the thread falls out of the **Catch** block and then out of the **Try** block; lastly, the thread returns from the method.

Now back inside **Main**, **HundredDivX** is called again, but this time **2** is passed. Inside **HundredDivX**, **100** is divided by **2**, which sets **x** to **50** without causing an exception to be thrown. But then, **a**, a reference to a **System.Object**, is declared and initialized to **Nothing**. Attempting to call **ToString** causes the CLR to throw a **NullReferenceException** exception. The first catch filter funclet examines the type and sees a match. But because **x** is now **50** (not **0**), the catch filter funclet tells the CLR to continue searching.

The CLR checks the second catch filter. Because **NullReferenceException** is derived from **Exception** and because this second catch filter has no **When** statement associated with it, the filter funclet returns a special value telling the CLR that it will handle the exception. Again, the CLR unwinds any **Finally** blocks that might exist and then passes execution to the code inside the active **Catch** block. This block sets the return string to "Problem: I don't know what the problem is". The thread now falls out of the **Catch** block, then out of the **Try** block, and then returns from the method.

You should take away two important points from this section:

- The CLR supports complex filters, but many languages don't. If you're working in a language that doesn't support complex filters but you want to use one, write that part of your code in Visual Basic, IL assembly, or some other language that supports it. Because code in the various programming languages integrates seamlessly, this is a viable solution.
- Notice from the previous code example how the CLR manages an exception. First, the CLR locates a catch filter that accepts the exception. Then the CLR unwinds the call stack by calling **Finally** blocks. Finally, the CLR passes execution to the code contained inside the handling **Catch** block.

## Unhandled Exceptions

As explained in the previous section, when an exception is thrown, the CLR starts searching for a catch filter interested in handling the exception. If no catch filter accepts the exception object, an *unhandled exception* occurs. An unhandled exception identifies a situation that the application didn't anticipate.

When developing an application, you should set a policy for dealing with unhandled exceptions. In addition, it's quite common to have one policy for debug versions of your application and a different policy for release versions. Normally, in a debug version, you want the debugger to start and attach itself to your application so that you can find out exactly what went wrong and correct the code.

For a release version, the unhandled exception is occurring while the user is using the application. The user probably doesn't have the knowledge (or the source code) to debug the application. So the best thing to do is log some information about the unhandled exception and allow the application to recover as gracefully as possible. For a client–side application, a graceful recovery might mean trying to save the user's data and terminating the application. For a server–side application, it might mean aborting the current client's request and preparing for a new client request—normally, an unhandled exception in a server–side application shouldn't terminate the server.

**Note** Class library developers shouldn't consider setting a policy for unhandled exceptions. The application developer should have complete control over defining and implementing this policy.

Personally, I like what the Microsoft Office XP applications do when they experience an unhandled exception—even though they're not .NET Framework applications. They save the document that the user was currently editing and then display a dialog box notifying the user that a problem occurred. Figure 18–2 shows such a dialog box for Microsoft PowerPoint.

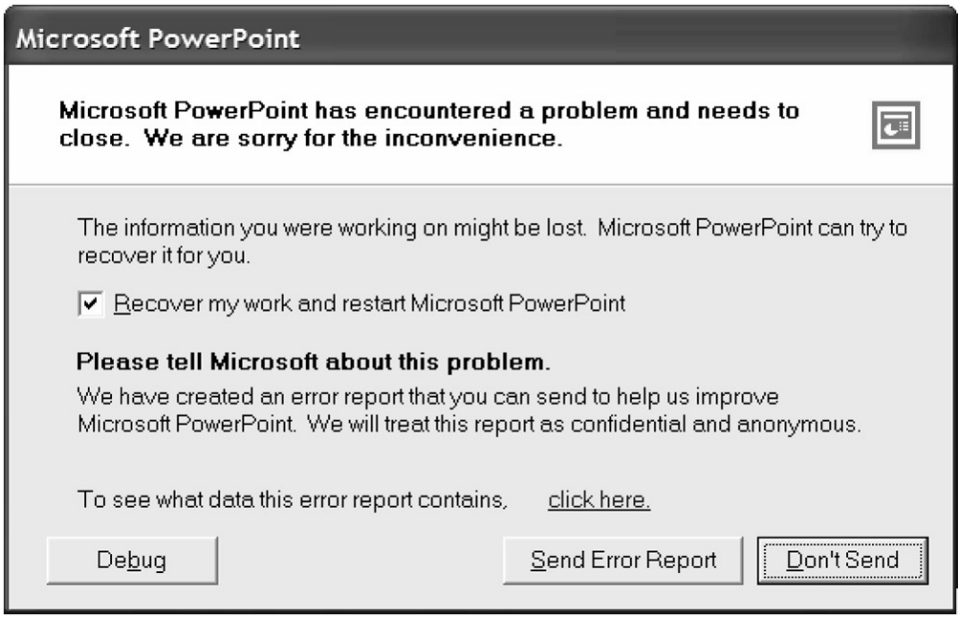


Figure 18–2 : Microsoft PowerPoint showing its Unhandled Exception dialog box

The Send Error Report button allows the user to send an error report to Microsoft over the Internet. The Recover My Work And Restart Microsoft PowerPoint check box offers the user the option to relaunch the application (after clicking the Send Error Report or Don't Send button). If the application is relaunched, it automatically reloads the document that was being edited. If the machine has a debugger installed, the Debug button appears in the dialog box, allowing the user to debug the application. Later in this chapter, I'll discuss how an application can set up a policy similar to Office XP's policy of handling unhandled exceptions.

Here's another example of a policy that deals with an unhandled exception: When making a request of a remote object or remote service, a thread (from a thread pool) is awakened to handle the client's request. The server's code is executed in a **try** block that has a **catch** block associated with it that catches all exceptions. If this **catch** block catches an exception, the information about the exception (including a stack trace) is sent back to the client as the server's response. This is possible because the exception type is serializable, as described in the "Defining Your Own Exception Class" section earlier in this chapter.

When the client code detects that an exception was returned from the server, it deserializes the exception object and throws it. This allows the client code to catch an exception thrown by the server—very cool indeed! It effectively hides the AppDomain, process, or machine boundary and allows the client code to believe that the remote call was made locally.

When thinking about unhandled exceptions, you should know what kind of thread you're dealing with. There are five kinds of threads:

- **Main thread** The thread that executes a console (CUI) or Windows Forms (GUI) application's **Main** method is a managed main thread.
- **Manual threads** Application code can explicitly create threads by constructing a **System.Threading.Thread** object.
- **Pool threads** A lot of features in the .NET Framework take advantage of the thread pool that's built into the CLR. Thread pool threads typically execute code started by methods of the **System.Threading.ThreadPool** class or the **System.Threading.Timer** class. In addition, methods that expose the CLR's asynchronous programming model (such as a delegate's **BeginInvoke** and **EndInvoke** methods) typically complete using a thread pool thread.

- **Finalizer thread** The managed heap has a thread dedicated to executing an object's **Finalize** method when a garbage collection determines the object to be unreachable.
- **Unmanaged threads** Some threads are created without the knowledge of the CLR. Using P/Invoke or COM interoperability, these unmanaged threads can transition into the CLR to execute managed code. Because the CLR itself doesn't create these threads, any unhandled exceptions are thrown outside the CLR and the unmanaged thread code can handle these unhandled exceptions any way it desires; the CLR doesn't get involved.

For all five kinds of threads, you can implement an unhandled exception policy using code similar to the following:

```
using System;
using System.Diagnostics;
using System.Windows.Forms;

class App {
   static void Main() {
      // Register the MgdUEFilter callback method with the AppDomain
      // so that it gets called when an unhandled exception occurs.
      AppDomain.CurrentDomain.UnhandledException +=
         new UnhandledExceptionEventHandler(MgdUEPolicy);

      // The rest of the application code goes here.
      try {
         // Simulate an exception here for testing purposes:
         Object o = null;
         o.GetType();  // throws a NullReferenceException
      }
      finally {
         Console.WriteLine("In finally");
      }
   }

////////////////////////////////////////////////////////////////////////////////

   // The following method is called when an unhandled exception is
   // encountered.
   static void MgdUEPolicy(object sender, UnhandledExceptionEventArgs e) {
      // This string contains the information to display or log.
      String info;

      // Initialize the contents of the string.
      Exception ex = e.ExceptionObject as Exception;
      if (ex != null) {
         // An unhandled CLS-compliant exception was thrown.
         // Do whatever: you can access the fields of Exception.
         // (Message, StackTrace, HelpLink, InnerException, etc.)
         info = ex.ToString();
      } else {
         // An unhandled non-CLS-compliant exception was thrown.
         // Do whatever: all you can call are the methods defined by Object.
         // (ToString, GetType, etc.)
         info = String.Format("Non-CLS-Compliant exception: Type={0}, String={1}",
            e.ExceptionObject.GetType(), e.ExceptionObject.ToString());
      }

      #if DEBUG

      // For debug builds of the application, launch the debugger
      // to understand what happened and to fix it.
      if (!e.IsTerminating) {
```

```
            // Unhandled exception occurred in a thread pool or finalizer thread.
            Debugger.Launch();

         } else {

            // Unhandled exception occurred in a managed thread.
            // By default, the CLR will automatically attach a debugger, but
            // you can force it with the following line:
            Debugger.Launch();
         }

         #else

         // For release builds of the application, display or log the
         // exception so that the user can report it back to you.
         if (!e.IsTerminating) {
            // Unhandled exception occurred in a thread pool or finalizer thread.
            // For thread pool or finalizer threads, you might just log the
            // exception and not display the problem to the user. However, each
            // application should do whatever makes the most sense for it.

         } else {

            // Unhandled exception occurred in a managed thread
            // The CLR is going to kill the application; you should display
            // and/or log the exception.
            String msg = String.Format("{0} has encountered a problem and " +
               "needs to close. We are sorry for the inconvenience.\n\n" +
               "Please tell {1} about this problem.\n" +
               "We have created an error report that you can send to " +
               "help us improve {0}. " +
               "We will treat this report as confidential and anonymous.\n\n" +
               "Would you like to send the report?",
               "(AppName)", "(CompanyName)");

            if (MessageBox.Show(msg, "(AppName)",
                MessageBoxButtons.YesNo) == DialogResult.Yes) {
               // The user has chosen to send the error report to you.
               // Send yourself the contents of the info variable and any
               // additional information you think you'd find useful to help you
               // correct this problem.

               // For testing purposes only, I'll display info here:
               MessageBox.Show(info, "Error Report");
            }
         }

         #endif
      }
   }
```

During application initialization (in the **Main** method), this code constructs a **System.UnhandledExceptionEventHandler** delegate as a wrapper around the static **MgdUEPolicy** method; this delegate is then registered with the **UnhandledException** event offered by the **System.AppDomain** type. Whenever a thread has an unhandled exception, the CLR will invoke the **MgdUEPolicy** method. If an unmanaged thread has an unhandled exception that was thrown by unmanaged code, the CLR won't invoke the **MgdUEPolicy** method.

For a release build of the application, the code in the **MgdUEPolicy** method should display or log the unhandled exception information (including the stack trace) so that this information can be sent back to the company developing the application. The code could also try to send the exception

information over the Internet back to the company (similar to what Office XP applications do) so that the users don't have to spend their own time reporting the problem. The company can then revise their source code so that the next version of the application properly anticipates this exception and responds to it appropriately. For a debug build of the application, the debugger should start up and attach itself to the process so that the developer can determine the reason of the exception and correct the code.

The callback method is considered a catch filter; that is, no deeper catch filter accepted the exception and therefore no **finally** blocks have executed yet. Also note that the callback receives a **System.UnhandledExceptionEventArgs** object. This object has two public, read–only properties, **ExceptionObject** (of type **System.Object**) and **IsTerminating** (of type **System.Boolean**). **ExceptionObject** identifies the exception object that was thrown. Note that the type of **ExceptionObject** is **Object**, not **Exception**. The reason is that the object thrown might not be CLS–compliant.

The **IsTerminating** property tells you if the CLR would terminate the application because of this unhandled exception. Normally, for manual threads, pool threads, and the finalizer thread, the CLR swallows any unhandled exceptions and either kills the thread, returns the thread to the pool, or moves on to call the **Finalize** method of the next object. If an unhandled exception occurs in any of these kinds of threads, the **IsTerminating** property will be **false**. Should an application's main thread or an unmanaged thread have an unhandled exception, **IsTerminating** will be **true**.

If you want to debug your application when an unhandled exception occurs in a nonterminating thread, you should place a call to **System.Diagnostics.Debugger** type's static **Launch** method inside the **MgdUEPolicy** method. The sample code demonstrates this.

For a main thread or an unmanaged thread, the system offers to either connect a debugger to the AppDomain or terminate the process. This is why the **IsTerminating** property will be **true**.

## Controlling What the CLR Does When an Unhandled Exception Occurs

When a managed thread has an unhandled exception, the CLR examines some settings to determine whether it should launch a debugger. To make this determination, the CLR checks the following registry subkey for the **DbgJITDebugLaunchSetting** value: HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework

If this value exists, its value must be one of those listed in Table 18–2.

Table 18–2: Possible Values of DbgJITDebugLaunchSetting

| Value | Description |
|---|---|
| 0 | Display a dialog box asking the user whether he would like to debug the process. If the user chooses not to debug the process, the CLR fires the **AppDomain**'s **UnhandledException** event and then, if the exception occurred in the main thread or in an unmanaged thread, terminates the process along with all the AppDomains in it. If no callback methods have registered with **AppDomain**'s **UnhandledException** event and if the process is a CUI application, the CLR displays the stack trace to the console. If the user chooses to debug the application, the CLR spawns a debugger, which will attach itself to the AppDomain. The CLR determines the command line used to spawn the debugger by examining the **DbgManagedDebugger** registry value contained in the save registry subkey. |

| | |
|---|---|
| 1 | In this case, no dialog box is displayed to the user. The CLR fires the **AppDomain**'s **UnhandledException** event and then, if the exception occurred in the main thread or in an unmanaged thread, terminates the process along with all the AppDomains in it. If no callback methods have registered with **AppDomain**'s **UnhandledException** event and if the process is a CUI application, the CLR displays the stack trace to the console. |
| 2 | In this case, no dialog box is displayed to the user and **AppDomain**'s **UnhandledException** event doesn't fire. Instead, the CLR just spawns the debugger attaching it to the application. |

By default, the CLR terminates the application or offers to spawn a debugger only for an application's main thread or for an unmanaged thread. For manual threads, pool threads, or finalizer threads, the CLR just swallows the exception and allows the thread to continue running—it doesn't terminate the process.

To help with debugging, you might want to know when an unhandled exception occurs in any thread. To get this information, just set the top three bytes of the **DbgJITDebugLaunchSetting** registry value to 0xFFFFFF. Use 0, 1, or 2 from Table 18–2 for the bottom–most byte.

**Note** In my opinion, Microsoft should check the registry value only if no callback method is registered with **AppDomain**'s **UnhandledException** event. Currently, when an unhandled exception occurs, the CLR examines the bottom–most byte of the **DbgJITDebugLaunchSetting** registry value. If this byte contains 0 or 2, the user is presented with the CLR's dialog box or the debugger just starts running—this happens even if the application has registered a callback method with **AppDomain**'s **UnhandledException** event. In the future, I'd like to see the CLR consider an exception unhandled only if no method is registered with **AppDomain**'s **UnhandledException** event.

## Unhandled Exceptions and Windows Forms

In a Windows Forms application, the **System.Windows.Forms.Application** class has a static **Run** method that's responsible for the thread's message loop. This loop dispatches window messages to a private method defined by the **System.Windows.Forms.NativeWindow** type. This method simply sets up a **try**/**catch** block and inside the **try** block, the protected **WndProc** method is called.

If a **System.Exception**–derived unhandled exception occurs while processing a window message, the **catch** block calls the window type's virtual **OnThreadException** method (passing it the exception object). **System.Windows.Forms.Control** type's implementation of **OnThreadException** ends up calling **Application**'s **OnThreadException**. By default, this method displays a dialog box like the one shown in Figure 18–3.

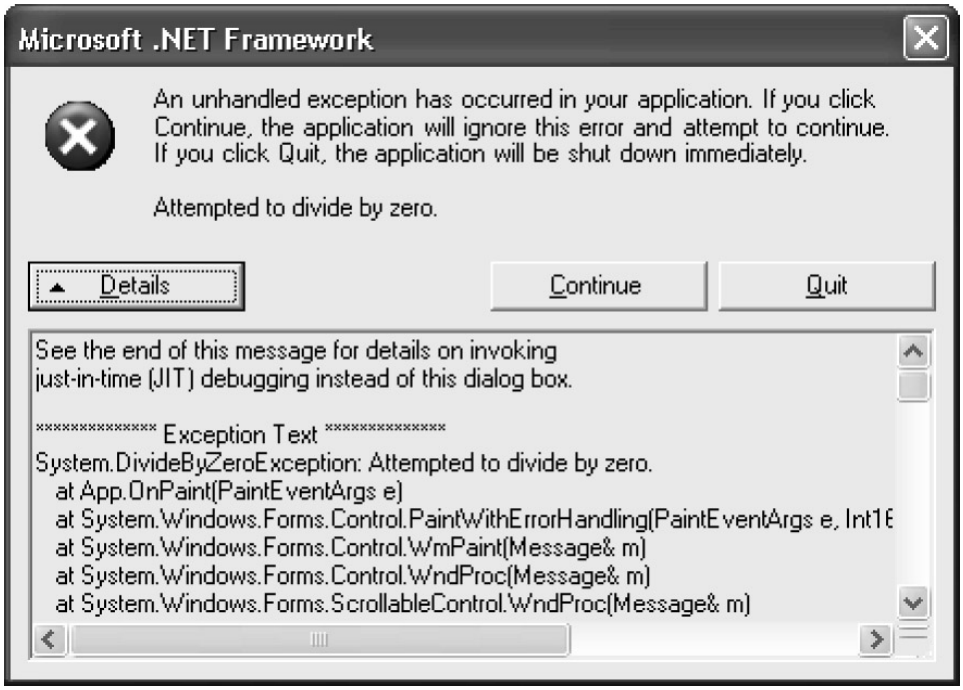


Figure 18–3 : An unhandled exception in a window procedure causes Windows Forms to display this dialog box.

This dialog box appears when a window procedure has an unhandled CLS–compliant exception, notifying the user that an unhandled exception has occurred and giving the user the option of ignoring the window message and continuing to run or killing (quitting) the application. If the user ignores the exception, the application will continue running, but it has probably been corrupted and will behave unpredictably at this point. If the application processes a data file, the user should save the work to a new file.

Then the user should exit the application and restart it. Once restarted, the user should load the new file and check it to make sure it's not corrupted. If it is corrupted, then any new work would have been lost but the user can go back to the original file and make the changes again. If the new file seems OK, then the user can continue editing it and at some point delete the original file or keep it as a backup.

You can override the built–in dialog box by defining a method that matches the **System.Threading.ThreadExceptionEventHandler** delegate (defined in the System.dll assembly) and then registering your method with the **Application** type's static **ThreadException** event.

**Note** I think it's sad that this event didn't mimic the names and behavior of **AppDomain**'s **UnhandledException** event; that is, **Application**'s event should have been called **UnhandledException** instead of **ThreadException** and it should have used the **UnhandledExceptionEventHandler** delegate instead.

You might have realized by now that Windows Forms deals only with CLS–compliant exceptions; non–CLS–compliant exceptions continue to propagate outside the thread's message loop and up the call stack. So if you want to display or log both CLS–compliant and non–CLS–compliant exceptions, you must define two callback methods and register one with the **Application** type's **ThreadException** event and register the other with **AppDomain** type's **UnhandledException** event.

As I explained, the behavior of **NativeWindow**'s internal method is to catch all CLS–compliant exceptions and display a dialog box or call any callback method that you've registered with the **ThreadException** event. However, some situations can change this default behavior. First, if a debugger is attached to your Windows Forms application, any unhandled exception thrown from a window procedure isn't caught and the exception is allowed to propagate up the call stack. Second, if a JIT debugger is installed and if the **jitDebugging** configuration setting is specified in the application's XML .config file, then again, the exception isn't caught and the exception is allowed to propagate up the call stack.

### Unhandled Exceptions and ASP.NET Web Forms

ASP.NET executes any of your Web Forms code inside its own **try** block. If your code throws an unhandled exception, ASP.NET catches the exception and determines how to handle it. ASP.NET offers a few mechanisms that you can use to receive a notification when an unhandled exception occurs. First, you can define a callback method that will get called when a particular Web page experiences an unhandled exception. You register the callback method using the **Error** event offered by the **System.Web.UI.TemplateControl** class; this class is the base class of the **System.Web.UI.Page** and **System.Web.UI.UserControl** classes.

In addition to allowing you to receive notifications of unhandled exceptions for a specific page, ASP.NET also lets you register a callback method that will receive notifications of unhandled exceptions on any page in your Web Forms application. You register the application–wide callback method using the **Error** event offered by the **System.Web.HTTPApplication** class. You typically add this code to your Global.asax file.

ASP.NET also offers tracing options that dump stack traces for unhandled exceptions out to a Web page to help you detect problems and correct your code. For more details on ASP.NET and exceptions, consult the .NET Framework SDK documentation.

### Unhandled Exceptions and ASP.NET XML Web Services

For ASP.NET XML Web services, the unhandled exception story is easy. When your XML Web service's method throws an unhandled exception, ASP.NET catches the exception and throws a new **System.Web.Services.Protocols.SoapException** object. A **SoapException** object is serialized into XML, representing a SOAP fault. This SOAP fault XML can be parsed and understood by any machine acting as an XML Web service client. This allows for XML Web service client/server interoperability.

If the client is a .NET Framework client, the SOAP fault XML is deserialized into a **SoapException** object and this new object is thrown in the client's thread. The client code can now catch this exception and proceed any way it chooses.

## Exception Stack Traces

As I mentioned earlier, the **System.Exception** type offers a public, read–only **StackTrace** property. An exception filter or a **catch** block can read this property to obtain the stack trace indicating what events occurred up to the exception. This information can be extremely valuable when you're trying to detect the cause of an exception so that you can correct your code. In this section, I'll discuss some issues related to the stack trace that aren't immediately obvious.

The **Exception** type's **StackTrace** property is magical. When you access this property, you're actually calling into code in the CLR; the property doesn't simply return a string. When you construct a new object of an **Exception**–derived type, the **StackTrace** property is initialized to **null**. If you were to read the property, you wouldn't get back a stack trace; you would get back **null**.

When an exception is thrown, the CLR internally records where the **throw** instruction occurred. When a catch filter accepts the exception, the CLR records where the exception was caught. If, inside a **catch** block, you now access the thrown exception object's **StackTrace** property, the code that implements the property calls into the CLR, which builds a string identifying all the methods between the place where the exception was thrown and the filter that caught the exception. Note that a catch filter can't access the stack information because it's not built until after a catch filter accepts the exception.

**Important** When you throw an exception, the CLR resets the starting point for the exception; that is, the CLR remembers only the location where the most recent exception object was thrown. The following code throws the same exception object that it caught and causes the CLR to reset its starting point for the exception:

```
void SomeMethod() {
   try { ... }
   catch (Exception e) {
      Ã
      throw e;  // CLR thinks this is where exception originated.
   }
}
```

In contrast, if you rethrow an exception object, the CLR doesn't reset the stack's starting point. The following code rethrows the same exception object that it caught, causing the CLR to not reset its starting point for the exception:

```
void SomeMethod() {
   try { ... }
   catch (Exception e) {
      ¤
      throw;  // This has no effect on where the CLR thinks
              // the exception originated.
   }
}
```

In fact, the only difference between these two code fragments is what the CLR thinks is the original location where the exception was thrown.

The string returned from the **StackTrace** property doesn't include any of the methods in the call stack that are above the point where the catch filter accepted the exception object. If you want the complete stack trace from the top of the thread to the exception handler, you can call **System.Environment**'s static **StackTrace** method and merge the two strings.

**Note** The FCL also offers a **System.Diagnostics.StackTrace** type that defines some properties and methods that allow a developer to programmatically manipulate a stack trace and the frames that make up the stack trace. You can construct a **StackTrace** object using several different constructors. Some constructors build a **StackTrace** object representing the frames from the top of the thread to the point where the object is constructed. Other constructors initialize the frames of the **StackTrace** object using an **Exception**–derived object.

Calling **Environment**'s static **StackTrace** property internally constructs a **StackTrace** object by calling the constructor that takes a single **Boolean** parameter; **true** is passed for this parameter. **Environment** then builds a string from the **StackTrace** object's frames.

If the CLR can find debug symbols for your assemblies, then the string returned by **Exception**'s **StackTrace** property or **Environment**'s **StackTrace** property will include source code file pathnames and line numbers. This information is incredibly useful for debugging. Unfortunately, the **System.Diagnostics.StackTrace**– type's **ToString** method doesn't support source code file pathnames and line numbers. I hope this bug will be fixed in a future version of the .NET Framework.

Whenever you obtain a stack trace, you might find that some methods in the call stack don't appear in the stack trace. The reason for their absence is that the JIT compiler can inline methods to avoid the overhead of calling and returning from a separate method. Many compilers (including the C# compiler), offer a /**debug** command–line switch. When this switch is turned on, these compilers embed information into the resulting assembly that tells the JIT compiler not to inline any of the assembly's methods so that stack traces are more complete and meaningful to the developer debugging the code.

**Note** The JIT compiler examines the **System.Diagnostics.DebuggableAttribute** custom attribute applied to the assembly. Your compiler of choice normally applies this attribute automatically. If this attribute has **true** specified for the **DebuggableAttribute** constructor's **isJITOptimizerDisabled** parameter, then the JIT compiler won't inline the assembly's methods. Using the C# compiler's /**debug** switch sets this parameter to **true**. By applying the **System.Runtime.CompilerServices.MethodImplAttribute** custom attribute to a method, you can forbid the JIT compiler from inlining the method for both debug and release builds. The following method definition shows how to forbid the method from being inlined:

```
using System.Runtime.CompilerServices;
Ã
class SomeType {

   [MethodImpl(MethodImplOptions.NoInlining)]
   public void SomeMethod() {
      Ã
   }
}
```

## Remoting Stack Traces

If a client makes a request of a server and the server code throws an exception, the exception object can be marshaled back to the client and rethrown in the client's thread. This capability is cool because the client code can simply react to the server–thrown exception the same way that it would react to a client–thrown exception. But what about the stack trace?

Well, the full stack trace is also marshaled back to the client. If the client were to examine the stack trace, the trace would contain all the frames from where the exception was thrown (on the server) to where the exception was caught (on the client). This is another neat capability: developers building

distributed applications can easily detect where problems occur and fix them regardless of whether the problem occurred on the client or the server.

Here is how the server's stack trace gets marshaled back to the client. When an **Exception**–derived object is serialized, the stack trace information is serialized as a string. When the object is deserialized, the new **Exception**–derived object saves the string in a private field. From now on, when you request the **StackTrace** property on this new object, the string returned contains the stack from the point where the exception is caught to the point where the new exception object was thrown. Appended to this string is the string from the internal field. In other words, the string returned from the **StackTrace** property contains the complete trace from where the server originally threw the exception to the point where the client catches the exception.

## Debugging Exceptions

The Microsoft Visual Studio .NET debugger offers special support for exceptions. To access this support, choose the Debug.Exceptions menu item and you'll see the dialog box shown in Figure 18–4.

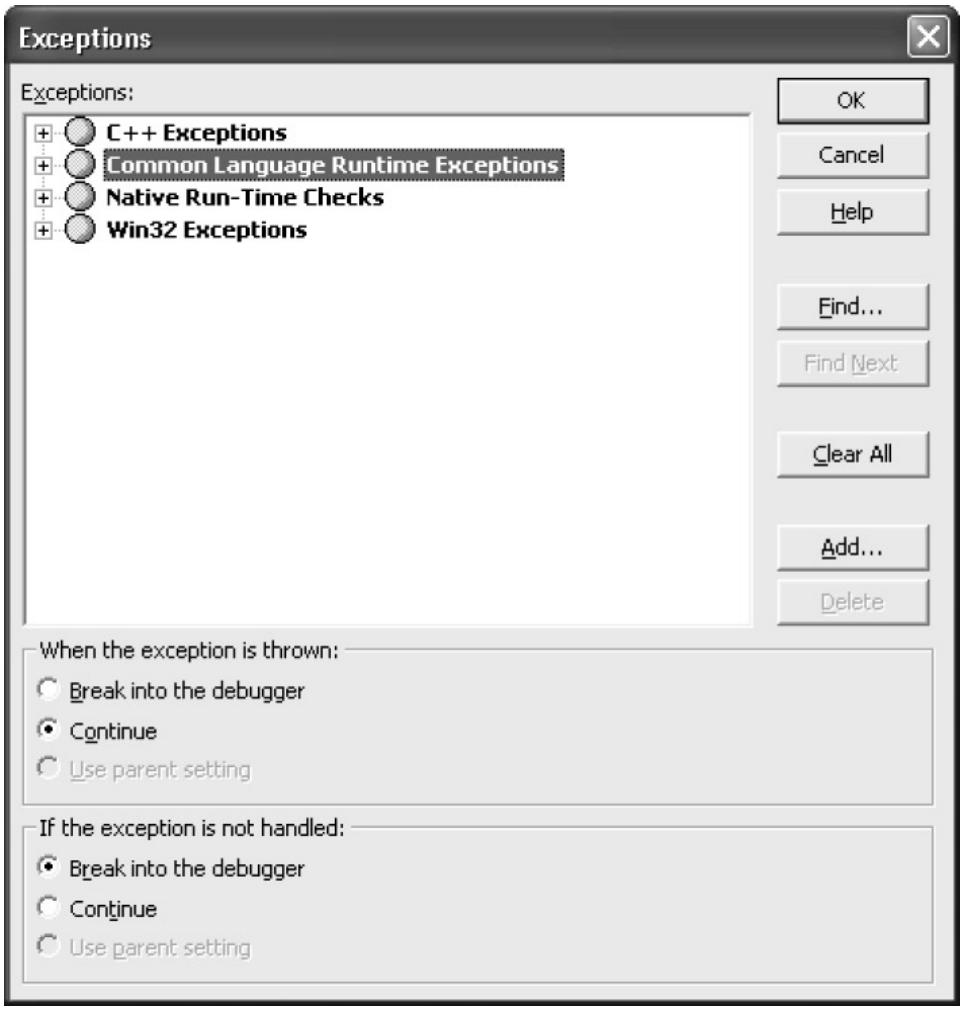



Figure 18–4 : Visual Studio .NET Exceptions dialog box showing the different kinds of exceptions

This dialog box shows the different kinds of exceptions that Visual Studio is aware of:

- **C++ Exceptions** For working with unmanaged C++ exceptions.

- **Common Language Runtime Exceptions** For working with managed exceptions, the focus of this chapter.
- **Native Run–Time Checks** For working with a new feature offered by the unmanaged Microsoft Visual C++ compiler. Only useful when you're compiling the code with the /**RTC** compiler switch.
- **Win32 Exceptions** For working with Windows unmanaged 32–bit exception codes.

For Common Language Runtime Exceptions, expanding the corresponding branch in the dialog box, as in Figure 18–5, shows the set of namespaces that the Visual Studio debugger is aware of.

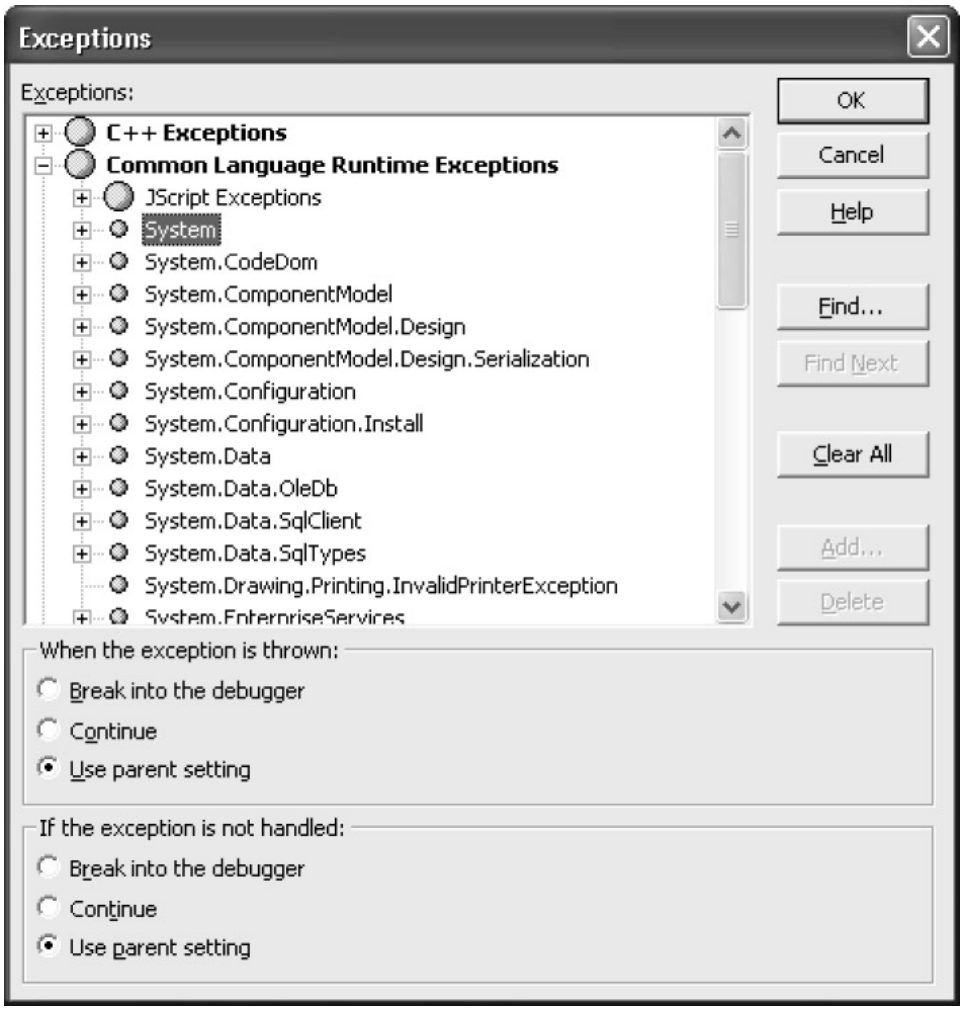



Figure 18–5 : Visual Studio .NET Exceptions dialog box showing CLR exceptions by namespace

If you expand a namespace, you'll see all the **System.Exception**–derived types defined within that namespace. For example, Figure 18–6 shows what you'll see if you open the **System** namespace.

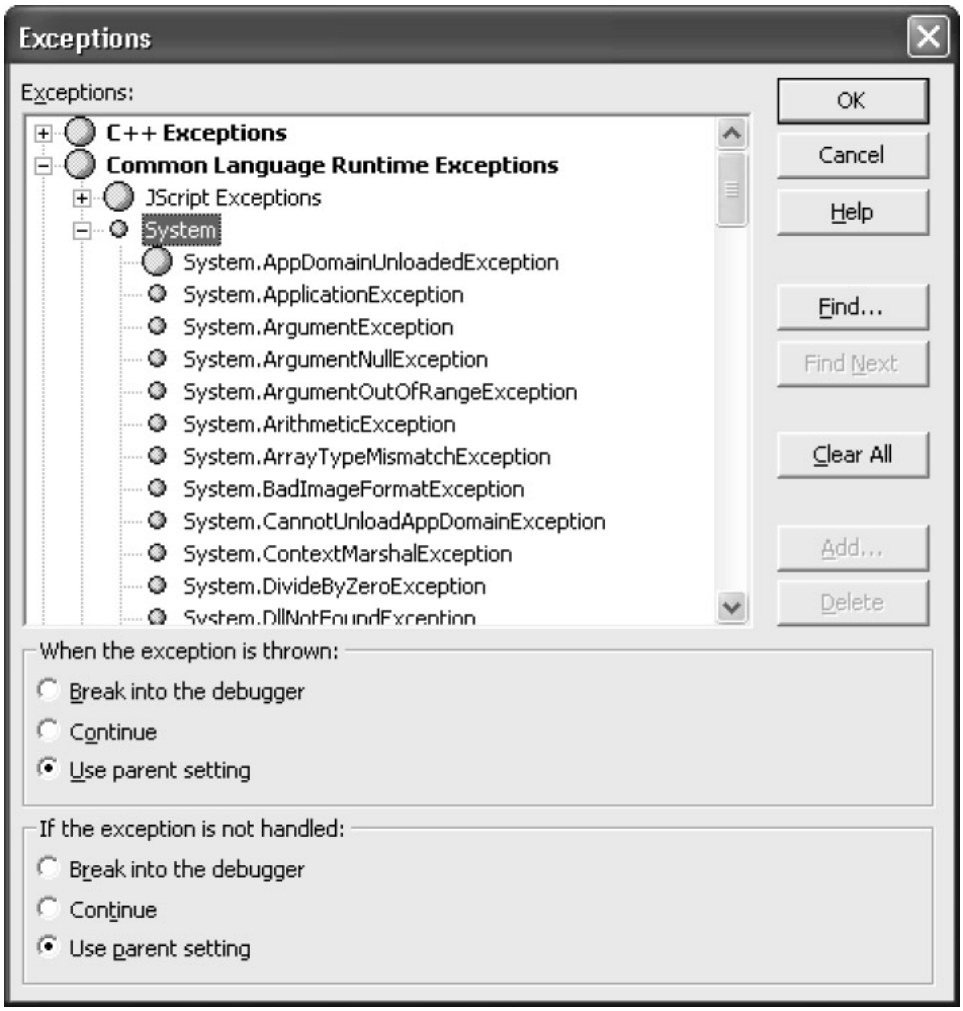


Figure 18–6 : Visual Studio .NET Exceptions dialog box showing CLR exceptions defined in the System namespace

After selecting an exception type, you can tell the debugger what you want it to do as soon as the exception is thrown:

- **Break Into The Debugger** Tells the debugger to notify you as soon as the exception is thrown. The CLR hasn't tried to walk up the call stack yet looking for any catch filters. This option is useful if you want to debug your code that catches and handles an exception.
- **Continue** Tells the debugger not to notify you as soon as the exception is thrown. The CLR will walk up the call stack looking for a catch filter that accepts the exception. If a **catch** block handles the exception, the debugger will never notify you that the exception occurred. This option is the most common one to select because a handled exception indicates that the application anticipated the situation and dealt with it; the application continues running normally. If no exception filter accepts the exception object, an unhandled exception occurs. In this case, the debugger will notify you of the unhandled exception even if the Continue option is selected.
- **Use Parent Setting** Tells the debugger to use the setting associated with the parent node in the tree. In my opinion, this setting is meaningless as is. It's unlikely that you'd ever want to choose Break Into The Debugger or Continue for all exception types in a namespace. What Microsoft should have done was show the exception type hierarchy, not the namespace hierarchy. If this dialog box showed the exception type hierarchy, this setting would make some sense: I could choose a setting for **ArgumentException**, and the exception types derived from it (**ArgumentNullException**, **ArgumentOutOfRangeException**, and **DuplicateWaitObjectException**) could inherit this

setting. This way I could break into the debugger whenever any of these related exception types were thrown.

At the bottom of the dialog box in Figure 18–6, you can tell the debugger how to behave when an unhandled exception is encountered. Here are the options:

- **Break Into The Debugger** Tells the debugger to notify you that an unhandled exception occurred and allows you to debug the application code. For managed application, this is by far the most useful option. In fact, this is the only option I'd ever choose.
- **Continue** For managed applications, tells the debugger to let the application die. In other words, the debugger won't present you a notification of the unhandled exception; the application will just terminate. When debugging a managed application, this option is useless because you're typically debugging the application in order to fix unhandled exceptions, not to ignore them. However, if you're debugging a script code (in Internet Explorer or some other HTML host), then the script code stops executing, the error is reported to the host, and the host can keep running.
- **Use Parent Setting** Tells the debugger to use the setting associated with the parent node in the tree. Again, because the tree shows the namespace hierarchy instead of the type hierarchy, this option isn't that useful.

If you define your own exception types, you can add them to this dialog box by selecting the Common Language Runtime Exceptions node and clicking the Add button. This causes the dialog box in Figure 18–7 to appear.

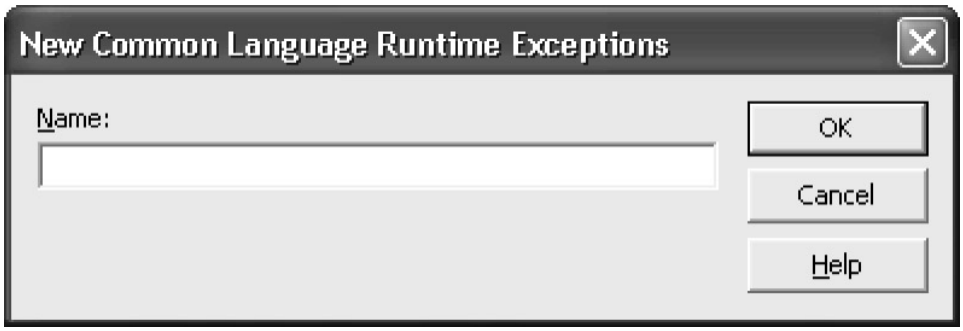



Figure 18–7 : Making Visual Studio .NET aware of your own exception type

In this dialog box, you can enter the fully qualified name of your own exception type. Note that the type you enter doesn't have to be a type derived from **System.Exception;** non–CLS–compliant types are fully supported. If you have two or more types with the same name but in different assemblies, there is no way to distinguish the types from one another. Fortunately, this situation rarely happens.

If your assembly defines several exception types, you must add them one at a time. In the future, I'd like to see this dialog box allow me to browse for an assembly and automatically import all **Exception**–derived types into Visual Studio's debugger. Each type could then be identified by assembly as well, which would fix the problem of having two types with the same name in different assemblies.

In addition, it might be nice if this dialog box also allowed me to individually select types not derived from **Exception** so that I could add any non–CLS–compliant exceptions that I might define. However, non–CLS–compliant exception types are strongly discouraged, so this isn't a must–have feature.

## Telling Visual Studio What Kind of Code to Debug

When using Visual Studio to debug an application, you must tell the debugger what kind of code you want to debug. When you attach the Visual Studio debugger to a process, it displays the dialog box shown in Figure 18–8.

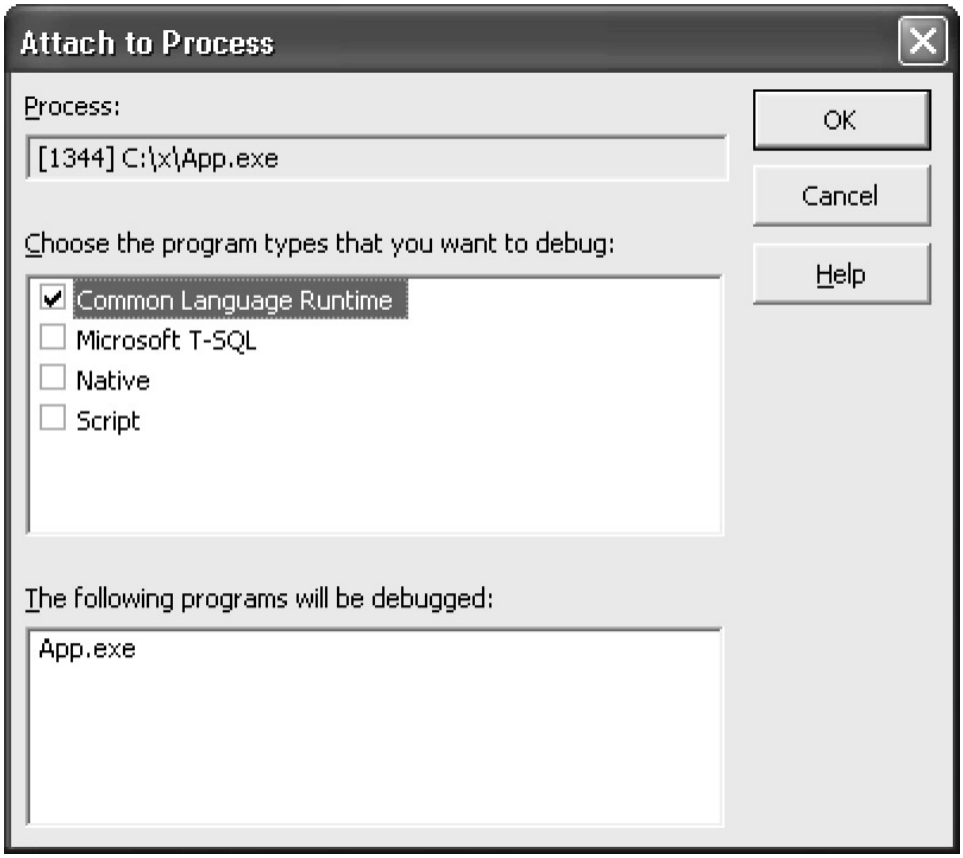



Figure 18–8 : Attaching Visual Studio .NET's debugger to a process

Now let me explain what it means to debug the different kinds of code:

- **Common Language Runtime** Allows you to debug managed code. The debugger shows managed symbols and stack traces. If this option isn't checked, the debugger won't show any symbols for managed code, making it very tedious to step through managed code.
- **Microsoft T–SQL** Allows you to debug T–SQL procedures stored in a SQL server database.
- **Native** Allows you to debug unmanaged code. The debugger shows unmanaged symbols and stack traces. If your managed application uses unmanaged code but you don't need to debug the unmanaged code, you should make sure that this option isn't checked. With this option off, single–stepping through the managed portion of your code will be much faster and unmanaged threads aren't suspended when a breakpoint is hit. If a thread is executing unmanaged code, it continues to run.

  With this option turned on, detaching the debugger from the process is possible only on Windows XP and Windows .NET Server platforms. With this option turned off, the debugger can detach from a process on any Windows platform.
- **Script** Allows you to debug scripting code executing within a host such as Internet Explorer.

The only time you should check both Common Language Runtime and Native is when you're debugging the portions of your code that interoperate between managed and unmanaged code

(P/Invoke and COM interoperability). Doing both managed and unmanaged debugging causes single–stepping to be quite slow. In addition, the debugger can't be detached regardless of the version of Windows you're running.

You should also be aware that the debugger sometimes loses control of the application when stepping over a managed/unmanaged transition. In other words, you might try to single step over a transition and the thread might just start running (or possibly hang). Most of the time, stepping over these transitions works fine, but occasionally it doesn't. The loss of control occurs because of decisions made in the CLR team to favor performance over "accurate debugging."

For a Visual Studio project, you can also indicate what kinds of code you want to debug. You do this by displaying the property page dialog box for the project. Figure 18–9 shows the property page for a C# console application project. Because the project is a C# project, Visual Studio assumes that you always want to debug managed code. Using this dialog box, I could enable unmanaged debugging and/or SQL debugging.

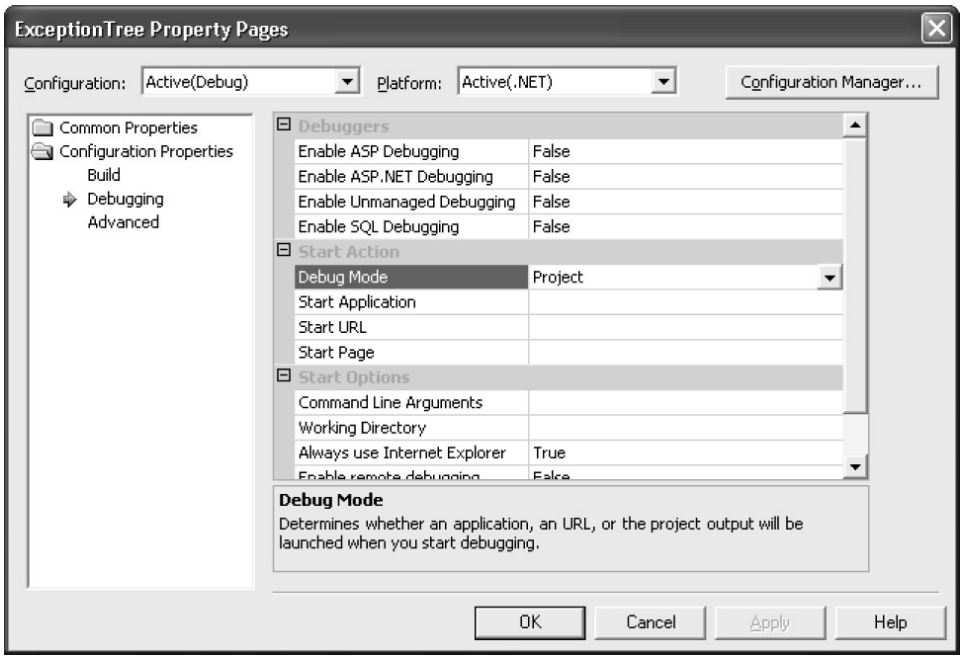


Figure 18–9 : Selecting the kind of code I want to debug for my project

# Chapter 19: Automatic Memory Management (Garbage Collection)

In this chapter, I'll discuss how managed applications construct new objects, how the managed heap controls the lifetime of these objects, and how the memory for these objects gets reclaimed. In short, I'll explain how the .NET Framework's garbage collector works and various performance issues related to it.

## Understanding the Basics of Working in a Garbage–Collected Platform

Every program uses resources of one sort or another, be they files, memory buffers, screen space, network connections, database resources, and so on. In fact, in an object–oriented environment, every type identifies some resource available for a program's use. To use any of these resources requires that memory be allocated to represent the type. The following steps are required to access a resource:

1. Allocate memory for the type that represents the resource by calling the intermediate language's **newobj** instruction, which is emitted when you use the **new** operator in C#, Microsoft Visual Basic, and other programming languages.
2. Initialize the memory to set the initial state of the resource and to make the resource usable. The type's constructor is responsible for setting this initial state.
3. Use the resource by accessing the type's members (repeating as necessary).
4. Tear down the state of a resource to clean up. I'll address this topic in the section "The Dispose Pattern: Forcing an Object to Clean Up" later in this chapter.
5. Free the memory. The garbage collector is solely responsible for this step.

This seemingly simple paradigm has been one of the major sources of programming errors. How many times have programmers forgotten to free memory when it is no longer needed? How many times have programmers attempted to use memory after it had already been freed?

These two application bugs are worse than most others because you usually can't predict the consequences or the timing of them. For other bugs, when you see your application misbehaving, you just fix the problem. But these two bugs cause resource leaks (memory consumption) and object corruption (destabilization), making the application perform unpredictably—and at unpredictable times. In fact, there are many tools (such as the Microsoft Windows Task Manager, the System Monitor ActiveX Control, NuMega BoundsChecker from Compuware, and Rational's Purify) specifically designed to help developers locate these types of bugs.

Proper resource management is very difficult and quite tedious. It distracts developers from concentrating on the real problems they're trying to solve. It would be wonderful if some mechanism existed that simplified the mind–numbing memory management task for developers. Fortunately, there is: garbage collection.

Garbage collection completely absolves the developer from having to track memory usage and know when to free memory. However, the garbage collector doesn't know anything about the resource represented by the type in memory, which means that a garbage collector can't know how to perform step 4 in the preceding list: tear down the state of a resource to clean up. To get a resource to clean up properly, the developer must write code that knows how to properly clean up a resource. The developer writes this code in **Finalize**, **Dispose**, and **Close** methods, as described

later in this chapter. However, as you'll see, the garbage collector can offer some assistance here too, allowing developers to skip step 4 in many circumstances.

Also, most types, such as **Int32**, **Point**, **Rectangle**, **String**, **ArrayList**, and **SerializationInfo**, represent resources that don't require any special cleanup. For example, a **Point** resource can be completely cleaned up simply by destroying its **x** and **y** fields maintained in the object's memory.

On the other hand, a type that represents (or wraps) an unmanaged (operating system) resource, such as a file, a database connection, a socket, a mutex, a bitmap, an icon, and so on, always requires the execution of some cleanup code when the object is to be destroyed. In this chapter, I'll explain how to properly define types that require explicit cleaning up, and I'll also show you how to properly use types that offer this explicit cleanup. For now, let's examine how memory is allocated and how resources are initialized.

The common language runtime (CLR) requires that all resources be allocated from a heap called the *managed heap*. This heap is similar to a C–runtime heap except that you never free objects from the managed heap—objects are automatically freed when the application no longer needs them. This, of course, raises the question, "How does the managed heap know when the application is no longer using an object?" I'll address this question shortly.

Several garbage collection algorithms are in practice today. Each algorithm is fine–tuned for a particular environment to provide the best performance. In this chapter, I'll concentrate on the garbage collection algorithm used by the Microsoft .NET Framework's CLR. Let's start off with the basic concepts.

When a process is initialized, the CLR reserves a contiguous region of address space that initially contains no backing storage. This address space region is the managed heap. The heap also maintains a pointer, which I'll call **NextObjPtr**. This pointer indicates where the next object is to be allocated within the heap. Initially, **NextObjPtr** is set to the base address of the reserved address space region.

The **newobj** intermediate language (IL) instruction creates an object. Many languages (including C# and Visual Basic) offer a **new** operator, which causes the compiler to emit a **newobj** instruction into the method's IL code. The **newobj** instruction causes the CLR to perform the following steps:

1. Calculate the number of bytes required for the type's (and all its base type's) fields.
2. Add the bytes required for an object's overhead. Each object has two overhead fields: a method table pointer and a SyncBlockIndex. On a 32–bit system, each of these fields requires 32 bits, adding 8 bytes to each object. On a 64–bit system, each is 64 bits, adding 16 bytes to each object.
3. The CLR then checks that the bytes required to allocate the object are available in the reserved region (committing storage if necessary). If the object fits, it is allocated at the address pointed to by **NextObjPtr**. The type's constructor is called (passing **NextObjPtr** for the **this** parameter), and the **newobj** IL instruction (or the **new** operator) returns the address of the object. Just before the address is returned, **NextObjPtr** is advanced past the object and indicates the address where the next object will be placed in the heap.

Figure 19–1 shows a managed heap consisting of three objects: A, B, and C. If a new object were to be allocated, it would be placed where **NextObjPtr** points (immediately after object C).

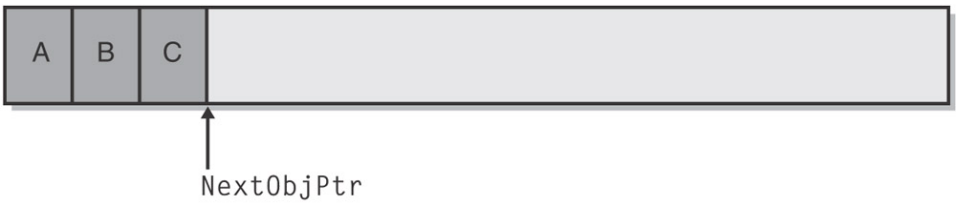


Figure 19–1 : Newly initialized managed heap with three objects constructed in it

By contrast, let's look at how the C–runtime heap allocates memory. In a C–runtime heap, allocating memory for an object requires walking through a linked list of data structures. Once a large enough block is found, that block is split and pointers in the linked–list nodes are modified to keep everything intact. For the managed heap, allocating an object simply means adding a value to a pointer—this is blazingly fast by comparison. In fact, allocating an object from the managed heap is nearly as fast as allocating memory from a thread's stack! In addition, most heaps (like the C–runtime heap) allocate objects wherever they find free space. Therefore, if I create several objects consecutively, it's quite possible that these objects will be separated by megabytes of address space. In the managed heap, however, allocating several objects consecutively ensures that the objects are contiguous in memory.

In many applications, objects allocated around the same time tend to have strong relationships to each other and are frequently accessed around the same time. For example, it's very common to allocate a **FileStream** object immediately followed by a **BinaryWriter** object. Then the application would use the **BinaryWriter** object, which internally uses the **FileStream** object. In a garbage–collected environment, new objects are allocated contiguously in memory, providing performance gains resulting from locality of reference. Specifically, this means that your process's working set will be smaller, and it's also likely that the objects your method is using can all reside in the CPU's cache. Your application will access these objects with phenomenal speed because the CPU will be able to perform most of its manipulations without having cache misses forcing RAM access.

So far, it sounds like the managed heap is far superior to the C–runtime heap because of its simplicity of implementation and speed. But there's one little detail you should know about before getting too excited. The managed heap gains these advantages because it makes one really big assumption: that address space and storage are infinite. Obviously, this assumption is ridiculous, and the managed heap must employ a mechanism that allows it to make this assumption. This mechanism is the garbage collector. Here's how it works.

When an application calls the **new** operator to create an object, there might not be enough address space left in the region to allocate to the object. The heap detects this lack of space by adding the bytes the object requires to the address in **NextObjPtr**. If the resulting value is beyond the end of the address space region, the heap is full and a garbage collection must be performed.

**Important** What I've just said is an oversimplification. In reality, a garbage collection occurs when generation 0 is full. Some garbage collectors use generations, a mechanism whose sole purpose is to improve performance. The idea is that newly created objects are part of a young generation, and objects created early in the application's lifecycle are in an old generation. Separating objects into generations can allow the garbage collector to collect specific generations instead of collecting all the objects in the managed heap. I'll explain generations in more detail later in this chapter. Until then, it's easiest for you to think that a garbage collection occurs when the heap is full.

# The Garbage Collection Algorithm

The garbage collector checks to see whether there are any objects in the heap that are no longer being used by the application. If such objects exist, the memory used by these objects can be reclaimed. (If no more memory is available in the heap, then **new** throws an **OutOfMemoryException** exception.) How does the garbage collector know whether or not the application is using an object? As you might imagine, this isn't a simple question to answer.

Every application has a set of *roots*. A single root is a storage location containing a memory pointer to a reference type. This pointer either refers to an object in the managed heap or is set to **null**. For example, all global or static reference type variables are considered roots. In addition, any reference type local variable or parameter variable on a thread's stack is also considered a root. Finally, within a method, a CPU register that refers to a reference type object is also considered a root.

When the JIT compiler compiles a method's IL, in addition to producing the native CPU code, the JIT compiler also creates an internal table. Logically, each entry in the table indicates a range of byte offsets in the method's native CPU instructions, and for each range, a set of memory addresses (or CPU registers) that contain roots. For example, the table could logically look like Table 19–1.

Table 19–1: Sample of a JIT CompilerÐProduced Table Showing Mapping of Native Code Offsets to a Method's Roots

| Starting Byte Offset | Ending Byte Offset | Roots |
|---|---|---|
| 0x00000000 | 0x00000020 | **this**, **arg1**, **arg2**, **ECX**, **EDX** |
| 0x00000021 | 0x00000122 | **this**, **arg2**, **fs**, **EBX** |
| 0x00000123 | 0x00000145 | **fs** |

If a garbage collection were to start while code was executing between offset 0x00000021 and 0x00000122 in the method, then the garbage collector would know that the objects referred to by the **this** parameter, **arg2** parameter, **fs** local variable, and the **EBX** register were all roots and refer to objects in the heap that shouldn't be considered garbage. In addition, the garbage collector can walk up the thread's call stack and determine the roots for all the calling methods by examining each method's internal table. The garbage collector uses other means to obtain the set of roots stored in global and static reference type variables.

**Note** In Table 19–1, notice that the method's **arg1** argument isn't referred to after the CPU instruction at offset 0x00000020. This means that the object **arg1** refers to can be collected anytime after this instruction executes (assuming that there are no other roots in the application that also refer to this object). In other words, as soon as an object becomes unreachable, it is a candidate for collection—objects aren't guaranteed to live throughout a method's lifetime.

However, when an application is running under a debugger or when an assembly contains the **System.Diagnostics.DebuggableAttribute** attribute with its constructor's **isJITOptimizerDisabled** parameter set to **true**, the JIT compiler extends the life of all variables (value type and reference type) until the end of their scope, which is usually the end of the method. (By the way, Microsoft's C# compiler offers a /**debug** command–line switch that adds the **DebuggableAttribute** attribute to the assembly, setting the **isJITOptimizerDisabled** parameter to **true**.) This extension prevents a garbage collection from collecting reference objects while executing the code in the scope and is useful when

you're debugging your code. It would be very odd if you called an object's method, got the wrong result, and then couldn't even look at the object!

When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage. In other words, it assumes that none of the application's roots refer to any objects in the heap. The garbage collector then starts walking the roots, building a graph of all reachable objects. For example, the garbage collector might locate a global variable that points to an object in the heap. Figure 19–2 shows a heap with several allocated objects where the application's roots refer directly to objects A, C, D, and F. All these objects become part of the graph. When adding object D, the garbage collector notices that this object refers to object H and also adds object H to the graph. The garbage collector continues to walk through all reachable objects recursively.

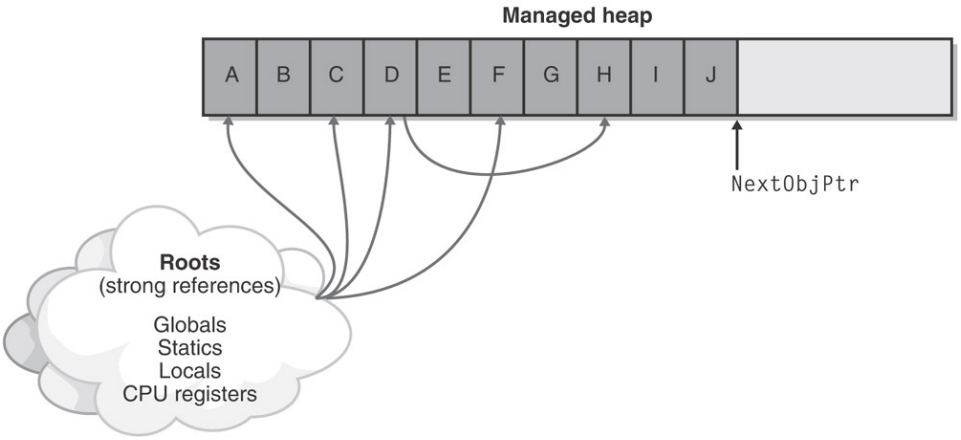



Figure 19–2 : Managed heap before a collection

Once this part of the graph is complete, the garbage collector checks the next root and walks the objects again. As the garbage collector walks from object to object, if it attempts to add an object to the graph that it previously added, it can stop walking down that path. This behavior serves two purposes. First, performance is enhanced significantly because the garbage collector doesn't walk through a set of objects more than once. Second, infinite loops are prevented should you have any circular linked lists of objects.

Once all the roots have been checked, the garbage collector's graph contains the set of all objects that are somehow reachable from the application's roots; any objects that aren't in the graph aren't accessible by the application and are therefore garbage. The garbage collector now traverses the heap linearly looking for contiguous blocks of garbage objects (now considered free space). If small blocks are found, the garbage collector leaves the blocks alone.

If large free contiguous blocks are found, however, the garbage collector shifts the nongarbage objects down in memory (using the standard **memcpy** function that you've known for years) to compact the heap. Naturally, moving the objects in memory invalidates all pointers to the objects. So the garbage collector must modify the application's roots so that the roots point to the objects' new locations. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well. After the heap memory is compacted, the managed heap's **NextObjPtr** pointer is set to point just after the last nongarbage object. Figure 19–3 shows the managed heap after a collection.

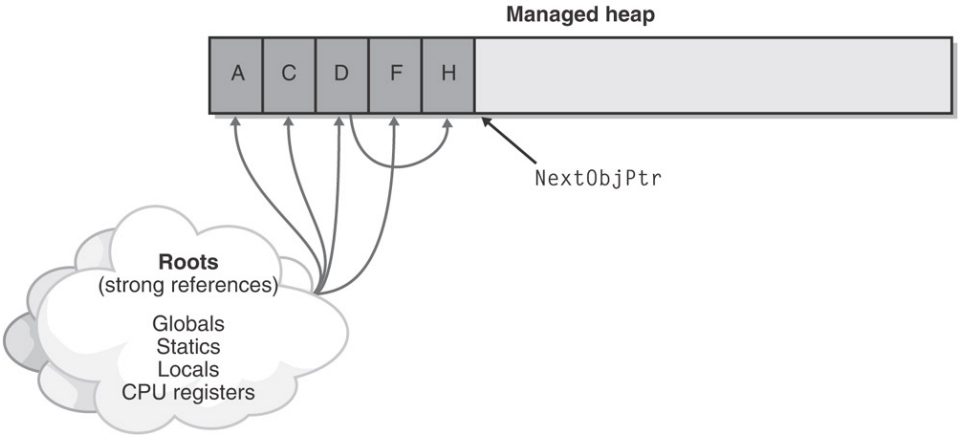



Figure 19–3 : Managed heap after a collection

As you can see, a garbage collection generates a considerable performance hit, which is the major downside of using a managed heap. But keep in mind that garbage collections occur only when generation 0 is full, and until then, the managed heap is significantly faster than a C–runtime heap. Finally, the CLR's garbage collector offers some optimizations that greatly improve the performance of garbage collection. I'll discuss these optimizations later in this chapter, in the "Generations" and "Other Garbage Collector Performance Issues" sections.

As a programmer, you should take away a couple important points from this discussion. To start, you no longer have to implement any code that manages the lifetime of objects your application uses. And notice how the two bugs described at the beginning of this chapter no longer exist. First, it's not possible to leak objects because any object not accessible from your application's roots can be collected at some point. Second, it's not possible to access an object that is freed because the object won't be freed if it is reachable, and if it's not reachable, your application has no way to access it.

The following code demonstrates how objects are allocated and managed:

```
class App {
   static void Main() {

      // ArrayList object created in heap, a is now a root.
      ArrayList a = new ArrayList();

      // Create 10000 objects in the heap.
      for (Int32 x = 0; x < 10000; x++) {
         a.Add(new Object());   // Object created in heap
      }

      // Right now, a is a root (on the thread's stack). So a is
      // reachable and the 10000 objects it refers to are reachable.
      Console.WriteLine(a.Length);

      // After a.Length returns, a isn't referred to in the code and is no
      // longer a root. If another thread were to start a garbage collection
      // before the result of a.Length were passed to WriteLine, the 10001
      // objects would have their memory reclaimed.

      Console.WriteLine("End of method");
   }
}
```

**Note** If garbage collection is so great, you might be wondering why it isn't in ANSI C++. The reason is that a garbage collector must be able to identify an application's roots

and must also be able to find all object pointers. The problem with unmanaged C++ is that it allows casting a pointer from one type to another, and there's no way to know what a pointer refers to. In the CLR, the managed heap always knows the actual type of an object and uses the metadata information to determine which members of an object refer to other objects.

# Finalization

At this point, you should have a basic understanding of garbage collection and the managed heap, including how the garbage collector reclaims an object's memory. Fortunately for us, most types only need memory to operate. For example, the **Int32**, **Point**, **Rectangle**, **String**, and **ArrayList** types are really just types that manipulate bytes in memory. However, some types require more than just memory to be useful.

The **System.IO.FileStream** type, for example, needs to open a file and save the file's handle. Then the type's **Read** and **Write** methods use this handle to manipulate the file. Similarly, the **System.Threading.Mutex** type opens a Windows mutex kernel object and saves its handle, using it when the **Mutex**'s methods are called.

Any type that wraps an unmanaged resource, such as a file, network connection, socket, mutex, and so on, must support *finalization*. Finalization allows a resource to gracefully clean up after itself when it is being collected. Basically, the type implements a method named **Finalize**. When the garbage collector determines that an object is garbage, it calls the object's **Finalize** method (if it exists). The **Finalize** method is usually implemented to call **CloseHandle**, passing in the handle of the unmanaged resource. Because **FileStream** defines a **Finalize** method, every **FileStream** object is guaranteed to have its unmanaged resource freed when the managed object is freed. If a type that wraps an unmanaged resource fails to define a **Finalize** method, the unmanaged resource won't be closed and will be a resource leak that exists until the process terminates, at which point, the operating system will reclaim the unmanaged resources.

The following **OSHandle** type demonstrates how to define a type that wraps an unmanaged resource. When the garbage collector determines that the object is garbage, it calls the **Finalize** method, which in turn calls the Win32 **CloseHandle** function, ensuring that the unmanaged resource is freed. The **OSHandle** class can also be used for any unmanaged resource that is freed by calling **CloseHandle**. If you're working with some unmanaged resource that requires a different function to clean it up, you'd have to modify the **Finalize** method accordingly.

```
public sealed class OSHandle {

   // This field holds the Win32 handle of the unmanaged resource.
   private IntPtr handle;

   // This constructor initializes the handle.
   public OSHandle(IntPtr handle) {
      this.handle = handle;
   }

   // When garbage collected, the Finalize method, which
   // will close the unmanaged resource's handle, is called.
   protected override void Finalize() {
      try {
         CloseHandle(handle);
      }
      finally {
         base.Finalize();
      }
```

```
   }

   // Public method returns the value of the wrapped handle
   public IntPtr ToHandle() { return handle; }

   // Public implicit cast operator returns the value of the wrapped handle
   public static implicit operator IntPtr(OSHandle osHandle) {
      return osHandle.ToHandle();
   }

   // Private method called to free the unmanaged resource
   [System.Runtime.InteropServices.DllImport("Kernel32")]
   private extern static Boolean CloseHandle(IntPtr handle);
}
```

When an **OSHandle** object is garbage collected, the garbage collector will call its **Finalize** method. This method will perform any required cleanup operations and then call the base type's **Finalize** method so that it has a chance to perform any cleanup that the base type finds necessary. The call to the base type's **Finalize** method is inside a **finally** block. This ensures that it gets called even if **OSHandle**'s cleanup code throws an exception for some reason. In this example, **System.Object**'s **Finalize** method gets called. **Object**'s **Finalize** method does nothing except return, so you could omit the exception handling code and the call to **base.Finalize** from the preceding code to improve performance without losing any "correctness."

The C# compiler won't actually compile the previous source code. The C# compiler team found that many developers code their **Finalize** methods improperly. Specifically, many developers forget to use exception handling and also forget to call the base type's **Finalize** method. To make things easier for the developer, C# offers special syntax to define a **Finalize** method. The following C# code is identical to the preceding code except that it will compile because it uses C#'s special syntax to define the **Finalize** method.

```
public sealed class OSHandle {

   // This field holds the Win32 handle of the unmanaged resource.
   private IntPtr handle;

   // This constructor initializes the handle.
   public OSHandle(IntPtr handle) {
      this.handle = handle;
   }

   // When garbage collected, the destructor (Finalize) method, which
   // will close the unmanaged resource's handle, is called.
   ~OSHandle() {
      CloseHandle(handle);
   }

   // Public method  returns the value of the wrapped handle
   public IntPtr ToHandle() { return handle; }

   // Public implicit cast operator returns the value of the wrapped handle
   public static implicit operator IntPtr(OSHandle osHandle) {
      return osHandle.ToHandle();
   }

   // Private method called to free the unmanaged resource
   [System.Runtime.InteropServices.DllImport("Kernel32")]
   private extern static Boolean CloseHandle(IntPtr handle);
}
```

If you were to compile this code and examine the resulting assembly with ILDasm.exe, you'd see that the C# compiler did, in fact, emit a method named **Finalize** into the module's metadata. If you examined the **Finalize** method's IL code, you'd also see that the call to **CloseHandle** was emitted into a **try** block and that a call to **base.Finalize** was emitted into a **finally** block.

To create an instance of the **OSHandle** object, you'd first have to call a Win32 function that returns a handle to an unmanaged resource, such as **CreateFile**, **CreateMutex**, **CreateSemaphore**, **CreateEvent**, **socket**, **CreateFileMapping**, and so on. Then you'd use C#'s **new** operator to construct an instance of **OSHandle**, passing the Win32 handle to the constructor.

Sometime in the future, the garbage collector will determine that this object is garbage. When that happens, the garbage collector will see that the type has a **Finalize** method and will call the method, allowing **CloseHandle** to close the unmanaged resource. Sometime after **Finalize** returns, the memory occupied in the managed heap by the **OSHandle** object will be reclaimed.

**Important** If you're familiar with C++, you'll notice that the special syntax that C# requires for defining a **Finalize** method looks just like the syntax you'd use to define a C++ destructor. In fact, the C# Programming Language Specification even calls this method a destructor. However, a **Finalize** method doesn't work at all like an unmanaged C++ destructor.

I think it was a mistake for the C# compiler team to mimic the C++ destructor syntax and to call this method a destructor. Using the unmanaged C++ nomenclature has been incredibly confusing to C++ programmers who are now adopting C# and the .NET Framework. These developers mistakenly believe that using the C# destructor syntax means that the type's objects will be deterministically destructed, just as they would be in C++. However, the CLR doesn't support deterministic destruction and therefore C# can't provide this mechanism. In fact, no programming language that supports the CLR can offer this. Even if you define a managed type using Microsoft's C++ with Managed Extensions, defining a "destructor" method causes the compiler to emit a **Finalize** method that gets called only when a garbage collection occurs.

Don't let a language's destructor syntax fool you—a **Finalize** method gets called *only* when a garbage collection occurs; it won't get called when a method exits or at the point when the object goes out of scope.

When designing a type, it's best if you avoid using a **Finalize** method, for several reasons:

- Finalizable objects get promoted to older generations, which increases memory pressure and prevents the object's memory from being collected at the time when the garbage collector determines the object is garbage. In addition, all objects referred to directly or indirectly by this object get promoted as well. (I'll talk about generations and promotions later in this chapter.)
- Finalizable objects take longer to allocate because pointers to them must be placed on the finalization list (which I'll discuss in the "Finalization Internals" section a little later).
- Forcing the garbage collector to execute a **Finalize** method can hurt performance significantly. Remember, each object is finalized. So if I have an array of 10,000 objects, each object must have its **Finalize** method called.
- A finalizable object can refer to other (finalizable and nonfinalizable) objects, prolonging their lifetime unnecessarily. In fact, you might want to consider breaking a type into two different types: a lightweight type with a **Finalize** method that doesn't refer to any other objects (just like the **OSHandle** type shown earlier) and a separate type without a **Finalize** method that

does refer to other objects.

- You have no control over when the **Finalize** method will execute. The object might hold on to resources until the next time the garbage collector runs.
- The CLR doesn't make any guarantees as to the order in which **Finalize** methods are called. For example, let's say an object contains a pointer to another object, which I'll call the inner object. The garbage collector has detected that both objects are garbage. Let's further say that the inner object's **Finalize** method gets called first. Now, the outer object's **Finalize** method is allowed to access the inner object and call methods on it, but the inner object has been finalized and the results can be unpredictable. For this reason, Microsoft strongly recommends that **Finalize** methods do not access any inner, member objects. The dispose pattern, which I'll cover later in this chapter, offers a way to clean up an object where this restriction doesn't apply.

If you determine that your type must implement a **Finalize** method, then make sure that the code executes as quickly as possible. Avoid all actions that would block the **Finalize** method, including any thread synchronization operations. Also, if you let any exceptions escape the **Finalize** method, the CLR swallows the exception and continues calling other objects' **Finalize** methods.

**Note** By far, the most common reason to implement a **Finalize** method is to free an unmanaged resource that the object itself owns. In fact, during finalization, you should avoid writing code that accesses other managed objects or managed static methods. You should avoid accessing other managed objects because the object's type can implement a **Finalize** method, which might get called first, putting the object in an unpredictable state. You should avoid calling any managed static methods because these methods can internally access objects that have been finalized, first causing the behavior of the static method to be unpredictable.

**Finalize** methods can also be used for other purposes. Here's a class that causes the computer to beep every time the garbage collector performs a collection.

```
public sealed class GCBeep {
   ~GCBeep() {
      // We're being finalized, beep.
      MessageBeep(-1);

      // If the AppDomain isn't unloading, create a new object
      // that will get finalized at the next collection.
      // I'll discuss IsFinalizingForUnload in the next section.
      if (!AppDomain.CurrentDomain.IsFinalizingForUnload())
         new GCBeep();
   }

   [System.Runtime.InteropServices.DllImport("User32.dll")]
   private extern static Boolean MessageBeep(Int32 uType);
}
```

To use this class, you just need to construct one instance of the class. Then whenever a garbage collection occurs, the object's **Finalize** method is called, which calls **MessageBeep** and constructs a new **GCBeep** object. This new **GCBeep** object will have its **Finalize** method called when the next garbage collection occurs. Here's a sample program that demonstrates the **GCBeep** class:

```
class App {
   static void Main() {
      // Constructing a single GCBeep object causes a beep to
      // occur every time a garbage collection starts.
      new GCBeep();
```

```
      // Construct a lot of 100-byte objects.
      for (Int32 x = 0; x < 10000; x++) {
         Console.WriteLine(x);
         Byte[] b = new Byte[100];
      }
   }
}
```

Also be aware that a type's **Finalize** method is called even if the type's instance constructor throws an exception. So your **Finalize** method shouldn't assume that the object is in a good, consistent state. The following code demonstrates.

```
class TempFile {
   String filename = null;
   public FileStream fs;

   public TempFile(String filename) {
      // The following line might throw an exception.
      fs = new FileStream(filename, FileMode.Create);

      // Save the name of this file.
      this.filename = filename;
   }

   ~TempFile() {
      // The right thing to do here is to test filename
      // against null because you can't be sure that
      // filename was initialized in the constructor.
      if (filename != null)
         File.Delete(filename);
   }
}
```

Alternatively, you could write the code as follows instead.

```
class TempFile {
   String filename;
   public FileStream fs;

   public TempFile(String filename) {
      try {
         // The following line might throw an exception.
         fs = new FileStream(filename, FileMode.Create);

         // Save the name of this file.
         this.filename = filename;
      }
      catch {
         // If anything goes wrong, tell the garbage collector
         // not to call the Finalize method. I'll discuss
         // SuppressFinalize later in this chapter.
         GC.SuppressFinalize(this);

         // Let the caller know something failed.
         throw;
      }
   }

   ~TempFile() {
      // No if statement is necessary now because this code
      // executes only if the constructor ran successfully.
```

```
            File.Delete(filename);
        }
    }
```

## What Causes Finalize Methods to Get Called

Four events cause an object to have its **Finalize** method called:

- **Generation 0 is full** This event is by far the most common way for **Finalize** methods to be called because it occurs naturally as the application code runs, allocating new objects.
- **Code explicitly calls System.GC's static Collect method** Code can explicitly request that the CLR perform a collection. Although Microsoft strongly discourages such requests, at times it might make sense for an application to force a collection. (I'll talk about this later in the chapter.)
- **The CLR is unloading an AppDomain** When an AppDomain unloads, the CLR considers nothing in the AppDomain to be a root and calls the **Finalize** method for all objects that were created in the AppDomain. I'll discuss AppDomains in Chapter 20.
- **The CLR is shutting down** When a process is gracefully terminating, it tries to shut down the CLR gracefully as well. At this point, the CLR considers nothing in the process to be a root and calls the **Finalize** method for all objects in the managed heap.

The CLR uses a special, dedicated thread to call **Finalize** methods. For the first, second, and third events, if a **Finalize** method enters an infinite loop, this special thread is blocked and no more **Finalize** methods can be called. This is a very bad situation because the application will never be able to reclaim the memory occupied by finalizable objects—the application will leak memory for as long as it runs.

For the fourth event, each **Finalize** method is given approximately 2 seconds to return. If a **Finalize** method doesn't return within 2 seconds, the CLR just kills the process—no more **Finalize** methods are called. Also, if it takes more than 40 seconds to call all objects' **Finalize** methods, then again, the CLR just kills the process.

> **Note** These timeout values were correct at the time I wrote this text, but Microsoft might change them in the future.

Code in a **Finalize** method can construct new objects. If this happens during CLR shutdown, the CLR continues collecting objects and calling their **Finalize** methods until no more objects exist or until the 40 seconds have expired.

Recall the **GCBeep** type presented earlier in this chapter. If a **GCBeep** object is being finalized because of the first or second event, a new **GCBeep** object is constructed. This is OK because the application continues to run, assuming that more collections will occur in the future. However, if a **GCBeep** object is being finalized because of the third or fourth event, a new **GCBeep** object shouldn't be constructed because this object would be created while the AppDomain was unloading or the CLR was shutting down. If these new objects were created, the CLR would have a bunch of useless work to do because it would continue to call **Finalize** methods.

To prevent the construction of new **GCBeep** objects, **GCBeep**'s **Finalize** method contains a call to **AppDomain**'s **IsFinalizingForUnload** method. This method returns **true** if the object's **Finalize** method is being called because of the AppDomain unloading. This solution works for AppDomain unloading, but what about CLR shutdown?

To know whether the CLR is shutting down, Microsoft added the **HasShutdownStarted** read–only property to the **System.Environment** class. **GCBeep**'s **Finalize** method should be reading this property and shouldn't be constructing a new object when the property returns **true**. Unfortunately, **GCBeep**'s **Finalize** method doesn't do this. Why not? The reason **GCBeep**'s **Finalize** method doesn't read this property is that the property is inaccessible! A Microsoft developer made a mistake and implemented the property as an instance property, but the **Environment** class has only a private constructor, preventing you from creating an instance of it. So there's no way to access this property—what a bad bug! I hope Microsoft will fix this in a future version of the .NET Framework Class Library (FCL).

There is no real workaround for this problem. **GCBeep**'s **Finalize** method does construct new objects during CLR shutdown. The new objects are continuously finalized, and after 40 seconds, the CLR gives up and terminates the process.

## Finalization Internals

On the surface, finalization seems pretty straightforward: you create an object; when the object is collected, the object's **Finalize** method is called. But once you dig in, finalization is more complicated than this.

When an application creates a new object, the **new** operator allocates the memory from the heap. If the object's type defines a **Finalize** method, a pointer to the object is placed on the *finalization list* just before the type's instance constructor is called. The finalization list is an internal data structure controlled by the garbage collector. Each entry in the list points to an object that should have its **Finalize** method called before the object's memory can be reclaimed.

Figure 19–4 shows a heap containing several objects. Some of these objects are reachable from the application's roots, and some are not. When objects C, E, F, I, and J were created, the system detected that these objects' types defined **Finalize** methods and added pointers to these objects to the finalization list.

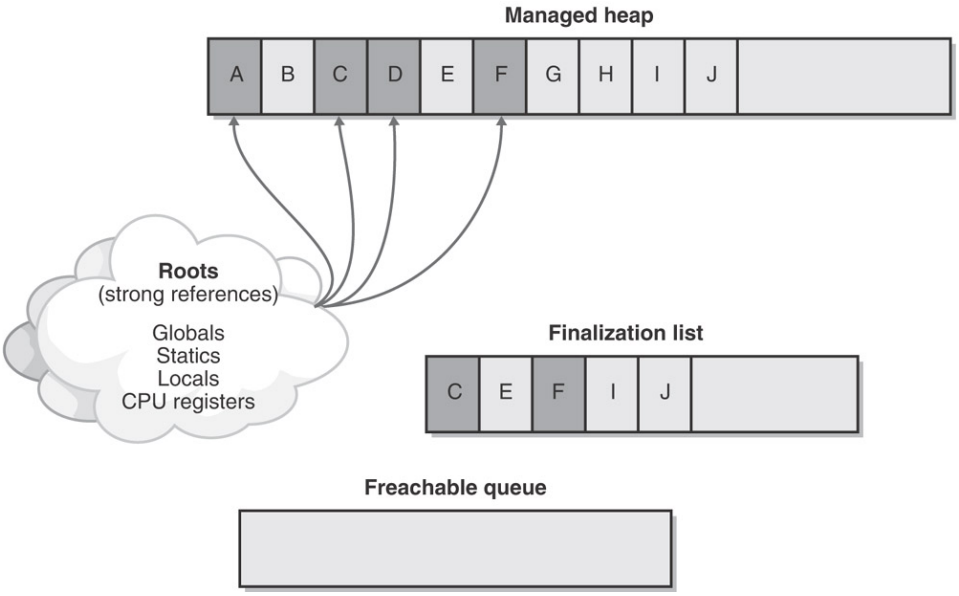



Figure 19–4 : The managed heap showing pointers in its finalization list

**Note** Even though **System.Object** defines a **Finalize** method, the CLR knows to ignore it; that is, when constructing an instance of a type, if the type's **Finalize** method is the one inherited from **System.Object**, the object isn't considered finalizable. One of the derived types must

override **Object**'s **Finalize** method.

When a garbage collection occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalization list looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalization list and appended to the *freachable queue*. The freachable queue (pronounced "F–reachable") is another of the garbage collector's internal data structures. Each pointer in the freachable queue identifies an object that is ready to have its **Finalize** method called. After the collection, the managed heap looks like Figure 19–5.

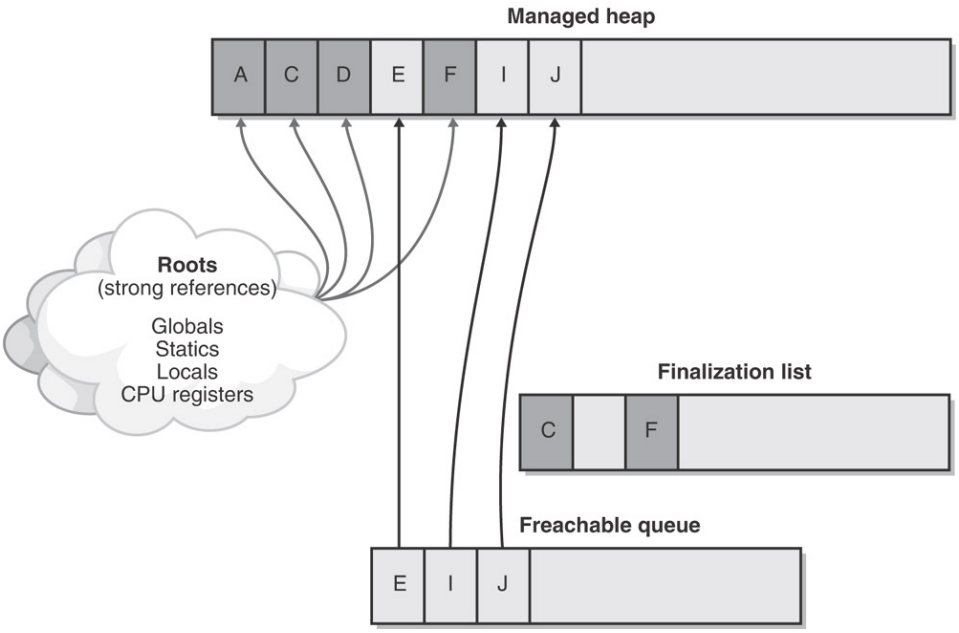



Figure 19–5 : The managed heap showing pointers that moved from the finalization list to the freachable queue

In this figure, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects didn't have a **Finalize** method that needed to be called. However, the memory occupied by objects E, I, and J couldn't be reclaimed because their **Finalize** methods haven't been called yet.

A special high–priority CLR thread is dedicated to calling **Finalize** methods. A dedicated thread is used to avoid potential thread synchronization situations that could arise if one of the application's threads was used instead. When the freachable queue is empty (the usual case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's **Finalize** method. Because of the way this thread works, you shouldn't execute any code in a **Finalize** method that makes any assumptions about the thread that's executing the code. For example, avoid accessing thread local storage in the **Finalize** method.

The interaction between the finalization list and the freachable queue is fascinating. First I'll tell you how the freachable queue got its name. Well, the "f" is obvious and stands for "finalization": every entry in the freachable queue should have its **Finalize** method called. But the "reachable" part of the name means that the objects are reachable. To put it another way, the freachable queue is considered a root just as global and static variables are roots. So if an object is on the freachable queue, the object is reachable and is *not* garbage.

In short, when an object isn't reachable, the garbage collector considers the object garbage. Then when the garbage collector moves an object's entry from the finalization list to the freachable queue, the object is no longer considered garbage and its memory can't be reclaimed. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been reclassified as not garbage: in a sense, the object has become *resurrected*. The garbage

collector compacts the reclaimable memory, and the special CLR thread empties the freachable queue, executing each object's **Finalize** method.

The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage because the application's roots don't point to it and the freachable queue no longer points to it. The memory for the object is simply reclaimed. The important point to get from all of this is that two garbage collections are required to reclaim memory used by objects that require finalization. In reality, more than two collections might be necessary because the objects get promoted to another generation (which I'll explain later). Figure 19–6 shows what the managed heap looks like after the second garbage collection.

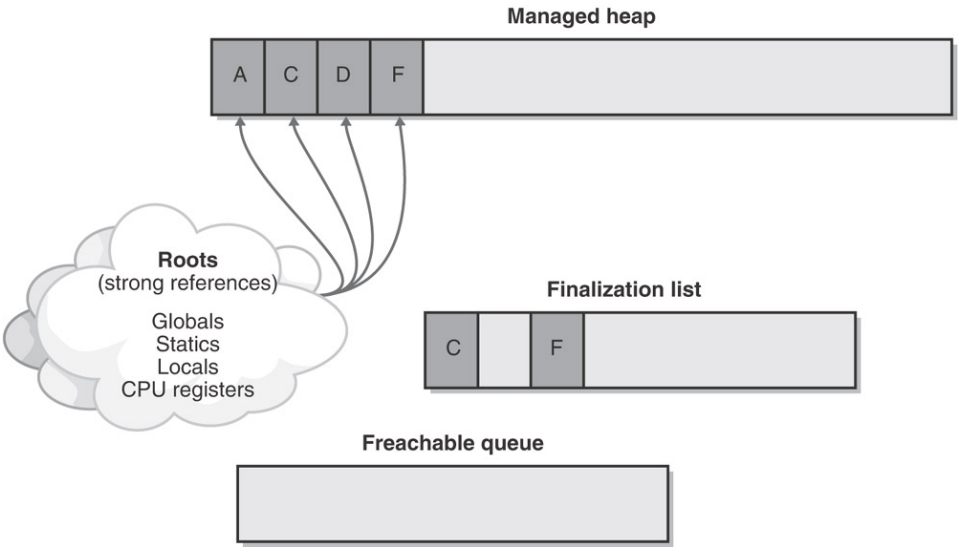



Figure 19–6 : Status of managed heap after second garbage collection

## The Dispose Pattern: Forcing an Object to Clean Up

The **Finalize** method is incredibly useful because it ensures that unmanaged resources aren't leaked when managed objects have their memory reclaimed. However, the problem with the **Finalize** method is that you have no guarantee when it will be called, and because it isn't a public method, a user of the class can't call it explicitly.

The capability to deterministically dispose of or close an object is frequently useful when you're working with unmanaged resources such as mutexes and bitmaps; this capability is even more useful when you're working with files and database connections. For example, you might want to open a database connection, query some records, and close the database connection—you don't want the database connection to stay open until the next garbage collection occurs, especially because the next garbage collection could occur hours or even days after you retrieve the database records.

Types that offer the capability to be deterministically disposed of or closed implement what is known as the *dispose pattern*. The dispose pattern defines conventions that a developer should adhere to when defining a type that wants to offer explicit cleanup to a user of the type. In addition, if a type implements the dispose pattern, a developer using the type knows exactly how to explicitly dispose of the object when it's no longer needed.

**Note** Any type that defines a **Finalize** method should also implement the dispose pattern as described in this section so that users of the type have a lot of control. However, a type can implement the dispose pattern and not define a **Finalize** method. For example, the

**System.IO.BinaryWriter** class falls into this category. I'll explain the reason for this exception in the section "An Interesting Dependency Issue" later in this chapter.

Earlier, I showed you the **OSHandle** type. That version of the code implemented a **Finalize** method, so the unmanaged resource that the object wrapped was closed when the object was collected. However, a developer using an **OSHandle** object had no way to explicitly close the unmanaged resource—the unmanaged resource got closed only when the object got garbage collected.

The following code shows a new and better version of the **OSHandle** class. This new version implements the well–defined dispose pattern.

```
using System;

// Implementing the IDisposable interface signals users of
// this class that it offers the dispose pattern.
public sealed class OSHandle : IDisposable {

   // This field holds the Win32 handle of the unmanaged resource.
   private IntPtr handle;

   // This constructor initializes the handle.
   public OSHandle(IntPtr handle) {
      this.handle = handle;
   }

   // When garbage collected, this Finalize method, which
   // will close the unmanaged resource's handle, is called.
   ~OSHandle() {
      Dispose(false);
   }

   // This public method can be called to deterministically
   // close the unmanaged resource's handle.
   public void Dispose() {
      // Because the object is explicitly cleaned up, stop the
      // garbage collector from calling the Finalize method.
      GC.SuppressFinalize(this);

      // Call the method that actually does the cleanup.
      Dispose(true);
   }

   // This public method can be called instead of Dispose.
   public void Close() {
      Dispose();
   }

   // The common method that does the actual cleanup.
   // Finalize, Dispose, and Close call this method.
   // Because this class is sealed, this method is private.
   // If this class weren't sealed, this method wouldn't be protected.
   private void Dispose(Boolean disposing) {
      // Synchronize threads calling Dispose/Close simultaneously.
      lock (this) {
         if (disposing) {
            // The object is being explicitly disposed of/closed, not
            // finalized. It is therefore safe for code in this if
            // statement to access fields that reference other
            // objects because the Finalize method of these other objects
            // hasn't yet been called.
```

```
            // For the OSHandle class, there is nothing to do in here.
         }

         // The object is being disposed of/closed or finalized.
         if (IsValid) {
            // If the handle is valid, close the unmanaged resource.
            // NOTE: Replace CloseHandle with whatever function is
            // necessary to close/free your unmanaged resource.
            CloseHandle(handle);

            // Set the handle field to some sentinel value. This precaution
            // prevents the possibility of calling CloseHandle twice.
            handle = InvalidHandle;
         }
      }
   }

   // Public property to return the value of an invalid handle.
   // NOTE: Make this property return an invalid value for
   // whatever unmanaged resource you're using.
   public IntPtr  InvalidHandle { get { return IntPtr.Zero; } }

   // Public method to return the value of the wrapped handle
   public IntPtr ToHandle() { return handle; }

   // Public implicit cast operator returns the value of the wrapped handle
   public static implicit operator IntPtr(OSHandle osHandle) {
      return osHandle.ToHandle();
   }


   // Public properties to return whether the wrapped handle is valid.
   public Boolean IsValid   { get { return (handle != InvalidHandle); }  }
   public Boolean IsInvalid { get { return !IsValid; } }

   // Private method called to free the unmanaged resource.
   [System.Runtime.InteropServices.DllImport("Kernel32")]
   private extern static Boolean CloseHandle(IntPtr handle);
}
```

This **OSHandle** code is hardly trivial: a lot of code is necessary to support the dispose pattern, and this class just wraps a simple unmanaged resource. The good news is that this class is almost all you'll ever need; that is, if you're designing a type that wraps an unmanaged resource, you can pretty much take the **OSHandle** code and copy it directly into your own project as is. In fact, it would be great if Microsoft had incorporated this class into the FCL itself.

Now let me explain what all this code does. First, the **OSHandle** type implements the **System.IDisposable** interface. This interface is defined in the FCL as follows:

```
public interface IDisposable {
   void Dispose();
}
```

Any type that implements this interface is stating that it adheres to the dispose pattern. Simply put, this means that the type offers a public, parameterless **Dispose** method that can be explicitly called to free the resource wrapped by the object. Note that the memory for the object itself is *not* freed from the managed heap's memory; the garbage collector is still responsible for freeing the object's memory, and there's no telling exactly when this will happen.

**Note** You might notice that this **OSHandle** type also offers a public **Close** method. This method simply calls **Dispose**. Some classes that offer the dispose pattern also offer a **Close** method for convenience; but the dispose pattern doesn't require this method. For example, the **System.IO.FileStream** class offers the dispose pattern, and this class also offers a **Close** method. Programmers find it more natural to "close" a file rather than "dispose of" a file. However, the **System.Threading.Timer** class doesn't offer a **Close** method even though it adheres to the dispose pattern.

The parameterless **Dispose** and **Close** methods should both be public and nonvirtual. However, because the **Dispose** method is implementing a method of an interface, the method is virtual by default. So the best you can do is seal the **Dispose** method. Fortunately, when you implement an interface method in C#, the compiler defaults the virtual method to public and sealed. But if you're using a different programming language, you should ensure that **Dispose** is sealed.

So now you know three ways to clean up an **OSHandle** object: a programmer can write code to call **Dispose**, a programmer can write code to call **Close**, or the garbage collector can call the object's **Finalize** method. The cleanup code is the same no matter which option you choose, so it's placed in a separate, private, nonvirtual method, which is also called **Dispose;** but this **Dispose** method takes a **Boolean** parameter, **disposing**.

This **Dispose** method is where you put all the cleanup code. In the **OSHandle** example, the method simply calls the Win32 **CloseHandle** function and changes the **handle** field to an invalid value. Setting the field to an invalid value ensures that the handle isn't closed multiple times; because the parameterless **Dispose** and **Close** methods are public, application code could call them multiple times. And because these methods can be called by multiple threads simultaneously, the Boolean **Dispose** method uses C#'s **lock** statement to ensure that the code in the method is thread–safe.

When an object's **Finalize** method is called, the **Dispose** method's **disposing** parameter is **false**. This tells the **Dispose** method that it shouldn't execute any code that references other managed objects. Imagine that the CLR is shutting down, and inside a **Finalize** method, you attempt to write to a **FileStream**. This might not work because the **FileStream** might have already had its **Finalize** method called.

On the other hand, when **Dispose** or **Close** is called, the **Dispose** method's **disposing** parameter will be set to **true**. This indicates that the object is being explicitly disposed of, not finalized. In this case, the **Dispose** method is allowed to execute code that references another object (such as a **FileStream**); because you have control over the program's logic, you know that the **FileStream** object is still open.

By the way, if the **OSHandle** class were not sealed, the Boolean **Dispose** method would be implemented as a protected virtual method instead of a private nonvirtual method. Any class that derives from **OSHandle** would implement the Boolean **Dispose** method only; it wouldn't implement the parameterless **Dispose** or **Close** method, and it wouldn't implement a **Finalize** method. The derived class would simply inherit the implementation of all these methods.

**Important** You need to be aware of some versioning issues here. If, in version 1, a base type doesn't implement the **IDisposable** interface, it can never implement this interface in a later version. If the base type were to add the **IDisposable** interface in the future, all the derived types wouldn't know to call the base type's methods and the base type

wouldn't get a chance to clean itself up properly. Likewise, if, in version 1, a base type implements the **IDisposable** interface, it can never remove this interface in a later version because the derived type would be trying to call methods that no longer existed in the base type.

In fact, if the derived class didn't have any cleanup to do itself, it wouldn't have to do anything with respect to the dispose pattern. But if the derived class did need to do some cleanup, it would just override the virtual Boolean **Dispose** method. In this method, you would write code to do the necessary explicit cleanup and then call the base class's Boolean **Dispose** method. Here's an example of a class that overrides the **OSHandle** class (assuming that **OSHandle** isn't sealed):

```
class SomeType : OSHandle {
   // This field holds the Win32 handle of the unmanaged resource.
   private IntPtr handle;

   protected override void Dispose(Boolean disposing) {
      // Synchronize threads calling Dispose/Close simultaneously.
      lock (this) {
         try {
            if (disposing) {
               // The object is being explicitly disposed of/closed, not
               // finalized. It is therefore safe for code in this if
               // statement to access fields that reference other
               // objects because the Finalize method of these other
               // objects hasn't been called.

               // For this class, there is nothing to do in here.
            }

            // The object is being disposed of/closed or finalized.
            if (IsValid) {
               // If the handle is valid, close the unmanaged resource.
               // NOTE: Replace CloseHandle with whatever function is
               // necessary to close/free your unmanaged resource.
               CloseHandle(handle);

               // Set the handle field to some sentinel value. This precaution
               // prevents the possibility of calling CloseHandle twice.
               handle = InvalidHandle;
            }
         }
         finally {
            // Let the base class do its cleanup.
            base.Dispose(disposing);
         }
      }
   }
}
```

Another noteworthy part of this code is the call to **GC**'s static **Suppress**–**Finalize** method inside the parameterless **Dispose** method. You see, if code using an **OSHandle** object explicitly calls **Dispose** or **Close**, the object's **Finalize** method should not execute. For if **Finalize** didn't execute, **CloseHandle** would be called multiple times. The call to **GC**'s **SuppressFinalize** turns on a bit flag associated with the object referred to by **this**. When this flag is on, the CLR knows not to move this object's pointer from the finalization list to the freachable queue, preventing the object's **Finalize** method from being called. When the garbage collector determines the object to be garbage, its memory is simply reclaimed.

## Using a Type That Implements the Dispose Pattern

Now that you know how a type implements the dispose pattern, let's take a look at how a developer uses a type that offers the dispose pattern. Instead of talking about the **OSHandle** class, let's talk about the more common **System.IO.FileStream** class. The **FileStream** class offers the ability to open a file, read bytes from the file, write bytes to the file, and close the file. Internally, the **FileStream** class is implemented identically to the **OSHandle** class except that its constructor calls the Win32 **CreateFile** function and the result is saved in the private handle field. The **FileStream** class also offers several additional properties (such as **Length**, **Position**, **CanRead**, **Handle**, and so on) and methods (such as **Read**, **Write**, **Flush**, and so on).

Let's say you want to write some code that creates a temporary file, writes some bytes to the file, and then deletes the file. You might start writing the code like this:

```
using System;
using System.IO;

class App {
   static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Write the bytes to the temporary file.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Delete the temporary file.
      File.Delete("Temp.dat");  // Throws an IOException
   }
}
```

Unfortunately, if you build and run this code, it might work, but most likely it won't. The problem is that the call to **File**'s static **Delete** method requests that Windows delete a file that is open. And so, **Delete** throws a **System.IO.IOException** exception with the following string message: "The process cannot access the file "Temp.dat" because it is being used by another process."

Be aware that in some cases, the file might actually get deleted! If, somehow, another thread caused a garbage collection to start after the call to **Write** and before the call to **Delete**, the **FileStream** object would get its **Finalize** method called, which would close the file and allow **Delete** to work. The likelihood of this situation is extremely rare, however, and the previous code will fail more than 99 percent of the time.

Fortunately, the **FileStream** class implements the dispose pattern, allowing me to modify the source code to explicitly close the file. Here's the corrected source code:

```
using System;
using System.IO;

class App {
   static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);
```

```
      // Write the bytes to the temporary file.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Explicitly close the file when done writing to it.
      ((IDisposable) fs).Dispose();

      // Delete the temporary file.
      File.Delete("Temp.dat");  // This always works now.
   }
}
```

The only difference here is that I've added a call to **FileStream**'s **Dispose** method. The **Dispose** method calls the Boolean **Dispose** method, which calls **CloseHandle**: Windows closes the file. Now, when **File**'s **Delete** method is called, Windows sees that the file isn't open and successfully deletes it.

Note that the **FileStream** object still exists in the managed heap, so you could still call methods on the object. Eventually, the garbage collector will run and the **FileStream** object will be determined to be garbage. At this point, the garbage collector would normally call its **Finalize** method, but because the **Dispose** method called **GC**'s **SuppressFinalize** method, the **Finalize** method won't be called—the object's memory will just be reclaimed.

**Note** The previous code casts the **fs** variable to an **IDisposable** before calling **Dispose**. Most classes that implement the dispose pattern won't require this cast. However, for a **FileStream** object, the cast is required because Microsoft's developers implemented the **Dispose** method as an explicit interface method implementation (as described in Chapter 15). I felt that this was a poor decision because it just makes things more complex without adding any value. In general, you should use explicit interface method implementations only when you have multiple methods with the same name. Because **Dispose** and **Close** are different method names, I think that they should be publicly available and that no casting should be required.

Because the **FileStream** class also offers a public **Close** method, the earlier code could be written as follows with identical results:

```
using System;
using System.IO;

class App {
   static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Write the bytes to the temporary file.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Explicitly close the file when done writing to it.
      fs.Close();

      // Delete the temporary file.
      File.Delete("Temp.dat");  // This always works now.
   }
}
```

**Note** Again, remember that the **Close** method isn't officially part of the dispose pattern: some types will offer it and some won't.

Keep in mind that calling **Dispose** or **Close** simply gives the programmer a way to force the object to do its cleanup at a deterministic time; these methods have no control over the lifetime of the memory used by the object in the managed heap. This means that you can still call methods on the object even though it has been cleaned up. The following code calls the **Write** method after the file is closed, attempting to write more bytes to the file. Obviously, the bytes can't be written, and when the code executes, the second call to the **Write** method throws a **System.ObjectDisposedException** exception with the following string message: "Cannot access a closed file."

```
using System;
using System.IO;

class App {
   static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Write the bytes to the temporary file.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Explicitly close the file when done writing to it.
      fs.Close();

      // Try to write to the file after closing it.
      // The following line throws an ObjectDisposedException.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Delete the temporary file.
      File.Delete("Temp.dat");  // This always works now.
   }
}
```

No memory corruption has occurred here because the memory for the **FileStream** object still exists; it's just that the object can't successfully execute its methods after it is explicitly disposed.

**Important** When defining your own type that implements the dispose pattern, be sure to write code in all your methods to throw a **System.ObjectDisposedException** exception if the object has been explicitly cleaned up. The **Dispose** and **Close** methods should never throw an **ObjectDisposedException** exception if called multiple times, though; these methods should just return (as shown earlier in the **OSHandle** type).

**Important** In general, I strongly discourage the use of calling a **Dispose** or **Close** method. The reason is that the CLR's garbage collector is well written and you should let it do its job. The garbage collector knows when an object is no longer accessible from application code, and only then will it collect the object. When application code calls **Dispose** or **Close**, it is effectively saying that it knows when the application no longer has a need for the object. For many applications, it is impossible to know for sure when an object is no longer required.

For example, if you have code that constructs a new object and you then pass a reference to this object to another method, the other method could save a reference to the object in some internal field variable (a root). There is no way for the calling method

to know that this has happened. Sure, the calling method can call **Dispose** or **Close**, but later, some other code might try to access the object, causing **ObjectDisposedException** exceptions to be thrown.

I recommend that you call **Dispose** or **Close** at a place in your code where you know you must clean up the resource (as in the case of attempting to delete an open file) or at a place where you know it is safe and you want to improve performance by preventing object promotion so that the **Finalize** method can run.

## C#'s using Statement

The previous code examples show how to explicitly call a type's **Dispose** or **Close** method. If you decide to call either of these methods explicitly, I highly recommend that you place the call in an exception handling **finally** block. This way, the cleanup code is guaranteed to execute. So it would be better to write the previous code example as follows:

```
using System;
using System.IO;

class App {
   static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = null;
      try {
         fs = new FileStream("Temp.dat", FileMode.Create);

         // Write the bytes to the temporary file.
         fs.Write(bytesToWrite, 0, bytesToWrite.Length);
      }
      finally {
         // Explicitly close the file when done writing to it.
         if (fs != null)
            ((IDisposable) fs).Dispose();
      }

      // Delete the temporary file.
      File.Delete("Temp.dat");  // This always works now.
   }
}
```

Adding the exception handling code is the right thing to do, and you must have the diligence to do it. If you're using C#, you can take advantage of C#'s **using** statement, which offers a simplified syntax that produces code identical to that just shown. Here's how the preceding code would be rewritten using C#'s **using** statement:

```
using System;
using System.IO;

class App {
   static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      using (FileStream fs =
```

```
            new FileStream("Temp.dat", FileMode.Create)) {

            // Write the bytes to the temporary file.
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }

        // Delete the temporary file.
        File.Delete("Temp.dat");  // This always works now.
    }
}
```

In the **using** statement, you initialize an object and save its reference in a variable. Then you access the variable via code contained inside **using**'s braces. When you compile this code, the compiler automatically emits the **try** block and the **finally** block. Inside the **finally** block, the compiler emits code to cast the object to an **IDisposable** and calls the **Dispose** method. Obviously, the **using** statement is usable only with types that implement the **IDisposable** interface.

**Note** C#'s **using** statement supports the capability to initialize multiple variables as long as the variables are all of the same type. It also supports the capability to just use an already initialized variable. For more information about this topic, see the C# Programmer's Reference.

I'm not aware of any other programming language that offers a similar convenient syntax for using types that implement the dispose pattern. In other programming languages, you must manually code the **try** and **finally** exception handling frames.

**Important** As I stated at the end of the previous section, you should call **Dispose** or **Close** only in situations where you need the object to be cleaned up before the next statement executes. For this reason, you should use C#'s **using** statement cautiously. I've seen many developers use C#'s **using** statement liberally, only to find out later that they have been explicitly cleaning up objects too early, causing another part of their application to throw **ObjectDisposedException** exceptions.

## An Interesting Dependency Issue

The **System.IO.FileStream** type allows the user to open a file for reading and writing. To improve performance, the type's implementation makes use of a memory buffer. Only when the buffer fills does the type flush the contents of the buffer to the file. A **FileStream** supports the writing of bytes only. If you want to write more complex data types (such as an **Int32**, a **Double**, a **String**, and so on), you can use a **System.IO.BinaryWriter** as demonstrated in the following code:

```
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
BinaryWriter bw = new BinaryWriter(fs);
bw.Write("Hi there");

// The following call to Close is what you should do.
bw.Close();
// NOTE: BinaryWriter.Close closes the FileStream. The FileStream
// shouldn't be explicitly closed in this scenario.
```

Notice that the **BinaryWriter**'s constructor takes a reference to a **FileStream** object as a parameter. Internally, the **BinaryWriter** object saves the **FileStream**'s reference. When you write to a **BinaryWriter** object, it internally buffers the data in its own memory buffer. When the buffer is full, the **BinaryWriter** object writes the data to the **FileStream**.

When you're done writing data via the **BinaryWriter** object, you should call **Dispose** or **Close**. (Because the **BinaryWriter** type implements the dispose pattern, you can also use it with C#'s **using** statement.) Both of these methods do exactly the same thing: cause the **BinaryWriter** object to flush its data to the **FileStream** object and close the **FileStream** object. When the **FileStream** object is closed, it flushes its buffer to disk just prior to calling the Win32 **CloseHandle** function.

> **Note** You don't have to explicitly call **Dispose** or **Close** on the **FileStream** object because the **BinaryWriter** calls it for you. However, if you do call **Dispose**/**Close** explicitly, the **FileStream** will see that the object has already been cleaned up—the methods do nothing and just return.

What do you think would happen if there was no code that explicitly called **Dispose** or **Close**? Well, at some point, the garbage collector would correctly detect that the objects were garbage and finalize them. But the garbage collector doesn't guarantee the order in which the **Finalize** methods are called. So if the **FileStream** got finalized first, it would close the file. Then when the **BinaryWriter** got finalized, it would attempt to write data to the closed file, throwing an exception. If, on the other hand, the **BinaryWriter** got finalized first, the data would be safely written to the file.

How was Microsoft to solve this problem? Making the garbage collector finalize objects in a specific order would have been impossible because objects could contain references to each other and there would be no way for the garbage collector to correctly guess the order in which to finalize these objects. Here is Microsoft's solution: the **BinaryWriter** type doesn't implement a **Finalize** method. This means that if you forget to explicitly close the **BinaryWriter** object, data is guaranteed to be lost. Microsoft expects developers to see this consistent loss of data and fix the code by inserting an explicit call to **Close**/**Dispose**.

## Weak References

When a root points to an object, the object can't be collected because the application's code can reach the object. When a root points to an object, a *strong reference* to the object is said to exist. However, the garbage collector also supports *weak references*. Weak references allow the garbage collector to collect the object but also allow the application to access the object. It all comes down to timing.

If only weak references to an object exist and the garbage collector runs, the object is collected, and when the application later attempts to access the object, the access will fail. On the other hand, to access a weakly referenced object, the application must obtain a strong reference to the object. If the application obtains this strong reference before the garbage collector collects the object, the garbage collector can't collect the object because a strong reference to the object exists.

Confused? Let's examine some code to clarify what all this really means:

```
void SomeMethod() {
   // Create a strong reference to a new Object.
   Object o = new Object();

   // Create a strong reference to a short WeakReference object.
   // The WeakReference object tracks the Object's lifetime.
   WeakReference wr = new WeakReference(o);

   o = null;   // Remove the strong reference to the object.

   o = wr.Target;
```

```
   if (o == null) {
      // A garbage collection occurred and Object's memory was reclaimed.
   } else {
      // A garbage collection did not occur and I can successfully
      // access the Object using o.
   }
}
```

Why might you want to use weak references? Well, some data structures are easy to create but require a lot of memory. For example, you might have an application that needs to know all the directories and files on the user's hard drive. You can easily build a tree that reflects this information, and as your application runs, refer to the tree in memory instead of accessing the user's hard disk. This greatly improves the performance of your application.

The problem is that the tree could be extremely large, requiring quite a bit of memory. If the user starts accessing a different part of your application, the tree might no longer be necessary but be wasting valuable memory nonetheless. You could give up the reference to the tree's root object, but if the user switches back to the first part of your application, you'll need to reconstruct the tree again. A weak reference allows you to handle this scenario easily and efficiently.

When the user switches away from the first part of the application, you can create a weak reference to the tree's root object and give up all strong references. If the other subcomponent's memory load is low, the garbage collector won't reclaim the tree's objects. When the user switches back to the first component, the application attempts to obtain a strong reference to the tree's root object. If successful, the application doesn't have to traverse the user's hard drive again.

The **System.WeakReference** type offers two public constructors:

```
public WeakReference(Object target);
public WeakReference(Object target, Boolean trackResurrection);
```

The **target** parameter identifies the object that the **WeakReference** object should track. The **trackResurrection** parameter indicates whether the **WeakReference** object should track the object after its **Finalize** method has been called. Usually, **false** is passed for the **trackResurrection** parameter, and the first constructor creates a **WeakReference** that doesn't track resurrection.

For convenience, a **WeakReference** that doesn't track resurrection is called a *short weak reference;* a **WeakReference** that does track resurrection is called a *long weak reference*. If an object's type doesn't offer a **Finalize** method, short and long weak references behave identically. I strongly recommend that you avoid using long weak references. Long weak references allow you to resurrect an object after it has been finalized and the state of the object is unpredictable.

Once you've created a weak reference to an object, you usually set the strong reference to the object to **null**. If any strong reference remains, the garbage collector will be unable to collect the object.

To use the object again, you must turn the weak reference into a strong reference. You accomplish this by simply querying the **WeakReference** object's **Target** property and assigning the result to one of your application's roots. If the **Target** property returns **null**, the object was collected. If the property doesn't return **null**, the root is a strong reference to the object and the code can manipulate the object. As long as the strong reference exists, the object can't be collected.

## Weak Reference Internals

From the preceding discussion, it should be obvious that **WeakReference** objects don't behave like other object types. Normally, if your application has a root that refers to an object and that object refers to another object, both objects are reachable and the garbage collector can't reclaim the memory either object is using. However, if your application has a root that refers to a **WeakReference** object, the object referred to by the **WeakReference** object isn't considered reachable and can be collected.

To fully understand how weak references work, let's look inside the managed heap again. The managed heap contains two internal data structures whose sole purpose is to manage weak references: the short weak reference table and the long weak reference table. These two tables simply contain pointers to objects allocated within the managed heap.

Initially, both tables are empty. When you create a **WeakReference** object, an object isn't allocated from the managed heap. Instead, an empty slot in one of the weak reference tables is located; short weak references use the short weak reference table, and long weak references use the long weak reference table.

Once an empty slot is found, the value in the slot is set to the address of the object you want to track—the object's pointer is passed to the **WeakReference**'s constructor. The value returned from the **new** operator is the address of the slot in the weak reference table. Obviously, the two weak reference tables aren't considered part of an application's roots or the garbage collector wouldn't be able to reclaim the objects the tables point to.

Here's what happens when the garbage collector runs:

1. The garbage collector builds a graph of all the reachable objects. I already explained how the garbage collector does this.
2. The garbage collector scans the short weak reference table. If a pointer in the table refers to an object that isn't part of the graph, then the pointer identifies an unreachable object and the slot in the short weak reference table is set to **null**.
3. The garbage collector scans the finalization list. If a pointer in the list refers to an object that isn't part of the graph, then the pointer identifies an unreachable object and the pointer is moved from the finalization list to the freachable queue. At this point, the object is added to the graph because the object is now considered reachable.
4. The garbage collector scans the long weak reference table. If a pointer in the table refers to an object that isn't part of the graph (which now contains the objects pointed to by entries in the freachable queue), then the pointer identifies an unreachable object and the slot is set to **null**.
5. The garbage collector compacts the memory, squeezing out the holes left by the unreachable objects. Note that the garbage collector sometimes decides not to compact memory if it determines that the amount of fragmentation isn't worth the time to compact.

Once you understand the logic behind how the garbage collector works, it's easy to understand how weak references work. Querying the **WeakReference**'s **Target** property causes the system to return the value in the appropriate weak reference table's slot. If **null** is in the slot, the object was collected.

A short weak reference doesn't track resurrection. This means that the garbage collector sets the pointer to **null** in the short weak reference table as soon as the garbage collector has determined the object to be unreachable. If the object has a **Finalize** method, the method hasn't been called

yet, so the object still exists. If the application accesses the **WeakReference** object's **Target** property, then **null** will be returned even though the object still exists.

A long weak reference tracks resurrection. This means that the garbage collector sets the pointer to **null** in the long weak reference table when the object's storage is reclaimable. If the object has a **Finalize** method, the **Finalize** method has been called and the object was not resurrected.

## Resurrection

I'm sure you'd agree that finalization is fascinating. And there's even more to it than what I've already described. You'll notice that when an object requiring finalization is considered dead, the garbage collector forces the object back to life so that its **Finalize** method can be called. Then after its **Finalize** method is called, the object is permanently dead. To summarize: an object requiring finalization dies, lives, and then dies again. This very interesting phenomenon is called *resurrection*. Resurrection, as its name implies, allows an object to come back from the dead.

The act of preparing to call an object's **Finalize** method is a form of resurrection. When the garbage collector places a reference to the object on the freachable queue, the object is reachable from a root and has come back to life. Eventually, the object's **Finalize** method is called, no roots point to the object, and the object is dead forever after. But what if an object's **Finalize** method executed code that placed a pointer to the object in a global or static variable, as demonstrated in the following code?

```
class SomeType {
   ~SomeType() {
      Application.ObjHolder = this;
   }
}

class Application {
   public static Object ObjHolder;    // Defaults to null
   :
}
```

In this case, when the object's **Finalize** method executes, a reference to the object is placed in a root and the object is reachable from the application's code. This object is now resurrected, and the garbage collector won't consider the object to be garbage. The application is free to use the object—but you must remember that the object *has* been finalized, so using it can cause unpredictable results. Also keep in mind that if **SomeType** contained fields that referenced other objects (either directly or indirectly), all objects would be resurrected because they are all reachable from the application's roots. However, be aware that some of these other objects might also have had their **Finalize** method called.

**Note** Any type you define might at some point be resurrected out of your control; that is, an object of your type might have members accessed after your **Finalize** method has been called. In an ideal world, you might consider adding code that checks to see whether your **Finalize** method has already executed and handle the member access gracefully, perhaps by throwing an appropriate exception. In practice, however, you'd have to write a lot of very tedious code, and I'm not aware of any type that has been implemented to handle this situation gracefully.

If some other piece of code sets **Application.ObjHolder** to **null**, the object is unreachable. Eventually, the garbage collector will consider the object to be garbage and will reclaim the object's storage. The object's **Finalize** method won't be called because no pointer to the object exists on the

finalization list.

As cool as resurrection sounds, there are very few good uses of it, and you should really avoid using it if possible. When developers do use resurrection, they usually want the object to gracefully clean itself up every time the object dies. To make this possible, the **GC** type offers a static method named **ReRegisterForFinalize**, which takes a single parameter: a reference to an object. The following code is a better version of the code shown earlier:

```
class SomeType {
   ~SomeType() {
      Application.ObjHolder = this;
      GC.ReRegisterForFinalize(this);
   }
}
```

When the **Finalize** method is called, it resurrects the object by making a root refer to the object. The **Finalize** method then calls **ReRegisterForFinalize**, which appends the address of the specified object (**this**) to the end of the finalization list. When the garbage collector detects that this object is unreachable (sometime in the future), it will move the object's pointer to the freachable queue and the **Finalize** method will get called again.

This example shows how to create an object that constantly resurrects itself and never dies—but you don't usually want objects to do this. It's far more common to conditionally set a root to reference the object inside the **Finalize** method.

> **Note** Make sure that you call **ReRegisterForFinalize** no more than once per resurrection or the object will have its **Finalize** method called multiple times. The reason is that each call to **ReRegisterForFinalize** appends a new entry to the end of the finalization list. When an object is determined to be garbage, all these entries move from the finalization list to the freachable queue, calling the object's **Finalize** method multiple times.

## Designing an Object Pool Using Resurrection

Here's a great scenario that really shows the value of resurrection. Let's say that you want to create a pool of **Expensive** objects. These objects are expensive because they take a lot of time to construct. For performance reasons, the application is going to construct a bunch of **Expensive** objects during startup and then reuse them over and over again during the lifetime of the application.

The code to manage a pool of **Expensive** objects would look like this:

```
using System;
using System.Collections;

// Instances of this class are expensive to construct.
class Expensive {

   // This static ArrayList contains references
   // to the objects available in the pool.
   static Stack pool = new Stack();


   // This static method returns an object from the pool.
   public static Expensive GetObjectFromPool() {
      // Get a reference to an object in the pool, and
```

```
      // remove the object from the pool.
      return (Expensive) pool.Pop();
   }


   // This static method is called when the application shuts down,
   // to kill the pool of objects.
   public static void ShutdownThePool() {

      // This will stop finalized objects from
      // adding themselves back into the pool.
      pool = null;
   }


   // This constructor creates an object and adds it to the pool.
   public Expensive() {
      // It takes a long time to construct this object.
      :

      // After the object is constructed, it adds itself to the pool.
      pool.Push(this);
   }

   // Finalize is called when the application no longer needs this object.
   ~Expensive() {

      // If the application isn't shutting down,
      // add the object back into the pool.
      if (pool != null) {

         // Call ReRegisterForFinalize so that the object will
         // get added back to the pool after subsequent uses.
         GC.ReRegisterForFinalize(this);

         // Add the object back into the pool.
         pool.Push(this);
   }
}

class App {
   static void Main() {
      // Populate the pool with a bunch of Expensive objects.
      for (Int32 i = 0; i < 10; i++)
         new Expensive();
      :

      // When you need an object, grab one out of the pool.
      Expensive e = Expensive.GetObjectFromPool();
      // The application can now use e.
      :

      // To shut down the application cleanly, shut down the pool.
      Expensive.ShutdownThePool();
   }
}
```

The **Expensive** type has a private static **System.Collections.Stack** object, which is used to manage the available objects in the pool. In **Main**, a loop constructs 10 **Expensive** objects. As each one is constructed, it adds itself to the pool. **Expensive**'s static **pool** field is a root and refers to the set of **Expensive** objects—none of which can be collected.

When the application needs to work with an object, it calls **Expensive**'s static **GetObjectFromPool** method. This method returns a reference to an object in the pool and removes the reference from the **Stack**. The application can now use the object.

At some point in the future, the application will no longer hold a reference to the object and a garbage collection will occur, causing the **Expensive** object to be finalized. When the object's **Finalize** method is called, it adds the reference to the object back into the pool, resurrecting the object and preventing the garbage collector from reclaiming the object's memory. In addition, the **Finalize** method calls **GC**'s **ReRegisterForFinalize** method, passing it a reference to the object. At some point in the future, this object will be given to the application again. Sometime later, the application will give up its reference to the object and the garbage collector will collect it again. Because **ReRegisterForFinalize** was called, the **Finalize** method will again execute and add the object back into the pool so that it can be used again.

Just before **Main** exits, it calls **Expensive**'s static **ShutdownThePool** method, which sets the **pool** field to **null**. When the application shuts down, the CLR calls the **Finalize** methods for all the objects remaining in the heap. When these objects are finalized, you don't want to call **ReRegisterForFinalize** because doing so would create an infinite loop (that the CLR would forcibly terminate after 40 seconds). So when the application is shutting down, **Expensive**'s **Finalize** method checks the **pool** field. If this field is **null**, the objects aren't reregistered for finalization and they're not added back into the pool—they are allowed to die and have their memory reclaimed.

As you can see, resurrection offers a very easy and efficient way to implement object pooling.

## Generations

As I mentioned near the beginning of the chapter, generations is a mechanism within the CLR garbage collector that's sole reason for being is to improve an application's performance. A *generational garbage collector* (also known as an *ephemeral garbage collector*, though I don't use this latter term in this book) makes the following assumptions:

- The newer an object is, the shorter its lifetime will be.
- The older an object is, the longer its lifetime will be.
- Collecting a portion of the heap is faster than collecting the whole heap.

Numerous studies have demonstrated the validity of these assumptions for a very large set of existing applications, and these assumptions have influenced how the garbage collector is implemented. In this section, I'll describe how generations work.

When initialized, the managed heap contains no objects. Objects added to the heap are said to be in generation 0. Stated simply, objects in generation 0 are newly constructed objects that the garbage collector has never examined. Figure 19–7 shows a newly started application that has had five objects allocated (A through E). After a while, objects C and E become unreachable.

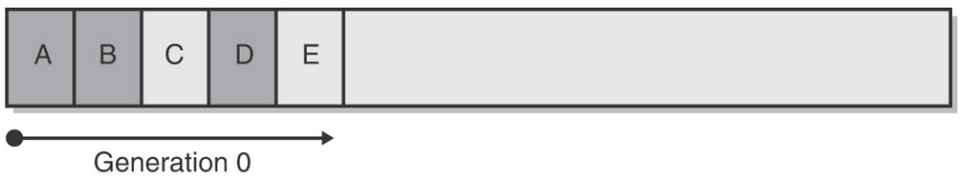



Figure 19–7 : A newly initialized heap containing some objects; all in generation 0. No collections have occurred yet.

When the CLR initializes, it selects a threshold size for generation 0, say, 256 KB. (The exact size is subject to change.) So if allocating a new object causes generation 0 to surpass its threshold, a garbage collection must start. Let's say that objects A through E occupy 256 KB. When object F is allocated, a garbage collection must start. The garbage collector will determine that objects C and E are garbage and will compact object D so that it is adjacent to object B. The objects that survive the garbage collection (objects A, B, and D) are said to be in generation 1. Objects in generation 1 have been examined by the garbage collector once. The heap now looks like Figure 19–8.

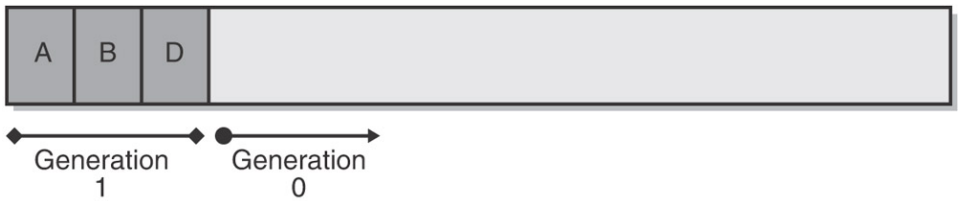



Figure 19–8 : After one collection: generation 0 survivors are promoted to generation 1; generation 0 is empty.

After a garbage collection, generation 0 contains no objects. As always, new objects will be allocated in generation 0. Figure 19–9 shows the application running and allocating objects F through K. In addition, while the application was running, objects B, H, and J became unreachable and should have their memory reclaimed at some point.

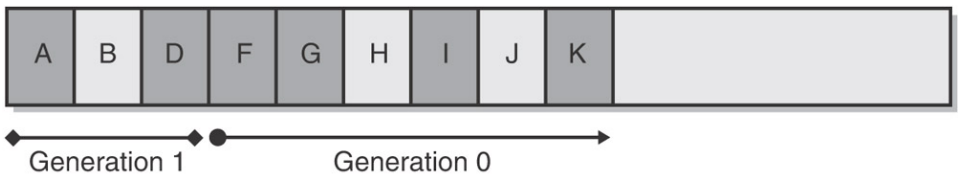



Figure 19–9 : New objects are allocated in generation 0; generation 1 has some garbage.

Now let's say that attempting to allocate object L would put generation 0 over its 256–KB threshold. Because generation 0 has reached its threshold, a garbage collection must start. When starting a garbage collection, the garbage collector must decide which generations to examine. Earlier, I said that when the CLR initializes, it selects a threshold for generation 0. Well, it also selects a threshold for generation 1. Let's say that the threshold selected for generation 1 is 2 MB.

When starting a garbage collection, the garbage collector also sees how much memory is occupied by generation 1. In this case, generation 1 occupies much less than 2 MB, so the garbage collector examines only the objects in generation 0. Look again at the assumptions that a generational garbage collector makes. The first assumption is that newly created objects have a short lifetime. So generation 0 is likely to have a lot of garbage in it, and collecting generation 0 will therefore reclaim a lot of memory. The garbage collector will just ignore the objects in generation 1, which will speed up the garbage collection process.

Obviously, ignoring the objects in generation 1 improves the performance of the garbage collector. However, the garbage collector improves performance more because it doesn't traverse every object in the managed heap. If a root or an object refers to an object in an old generation, the garbage collector can ignore any of the older objects' inner references, decreasing the amount of time required to build the graph of reachable objects. Of course, it's possible that an old object refers to a new object. To ensure that these "old" objects are examined, the garbage collector uses a mechanism internal to the JIT compiler that sets a bit when an object's reference field changes. This support lets the garbage collector know which old objects (if any) have been written to since the last collection. Only old objects that have had fields changed need to be examined to see whether they refer to any new objects in generation 0.

**Note**

Microsoft's performance tests show that it takes less than 1 millisecond on a 200–MHz Pentium to perform a garbage collection of generation 0. Microsoft's goal is to have garbage collections take no more time than an ordinary page fault.

A generational garbage collector also assumes that objects that have lived a long time will continue to live. So it's likely that the objects in generation 1 will continue to be reachable from the application. Therefore, if the garbage collector were to examine the objects in generation 1, it probably wouldn't find a lot of garbage and it wouldn't be able to reclaim much memory. So collecting generation 1 is likely to be a waste of time. If any garbage happens to be in generation 1, it just stays there. The heap now looks like Figure 19–10.

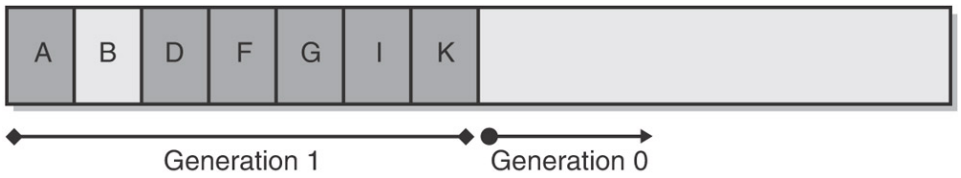



Figure 19–10 : After two collections: generation 0 survivors are promoted to generation 1 (growing the size of generation 1); generation 0 is empty.

As you can see, all the generation 0 objects that survived the collection are now part of generation 1. Because the garbage collector didn't examine generation 1, object B didn't have its memory reclaimed even though it was unreachable at the time of the last garbage collection. Again, after a collection, generation 0 contains no objects and is where new objects will be placed. In fact, let's say that the application continues running and allocates objects L through O. And while running, the application stops using objects G, L, and M, making them all unreachable. The heap now looks like Figure 19–11.

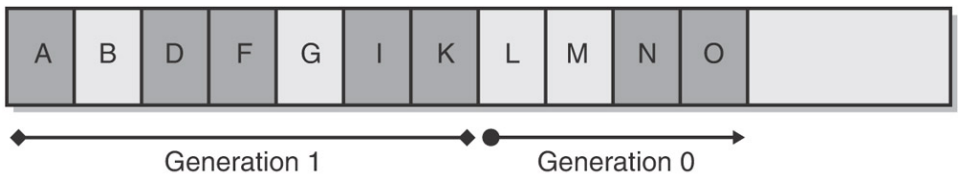



Figure 19–11 : New objects are allocated in generation 0; generation 1 has more garbage.

Let's say that allocating object P causes generation 0 to pass its threshold, causing a garbage collection to occur. Because the memory occupied by all the objects in generation 1 is less than 2 MB, the garbage collector again decides to collect only generation 0, ignoring the unreachable objects in generation 1 (objects B and G). After the collection, the heap looks like Figure 19–12.

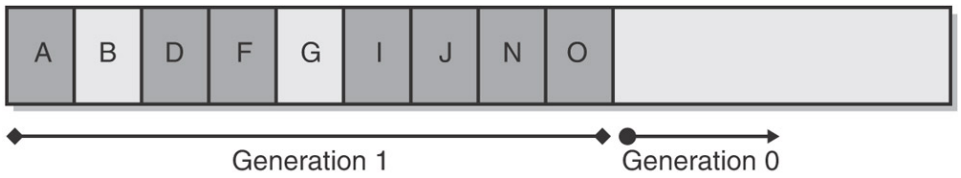



Figure 19–12 : After three collections: generation 0 survivors are promoted to generation 1 (growing the size of generation 1 again); generation 0 is empty.

In Figure 19–12, you see that generation 1 keeps growing slowly. In fact, let's say that generation 1 has now grown to the point where all the objects in it occupy 2 MB of memory. At this point, the application continues running (because a garbage collection just finished) and starts allocating objects P through S, which fill generation 0 up to its threshold. The heap now looks like Figure 19–13.



Figure 19–13 : New objects are allocated in generation 0; generation 1 has more garbage.
When the application attempts to allocate object T, generation 0 is full and a garbage collection must start. This time, however, the garbage collector sees that the objects in generation 1 are occupying so much memory that generation 1's 2–MB threshold has been reached. Over the several generation 0 collections, it's likely that a number of objects in generation 1 have become unreachable (as in our example). So this time, the garbage collector decides to examine all the objects in generation 1 and generation 0. After both generations have been garbage collected, the heap now looks like Figure 19–14.

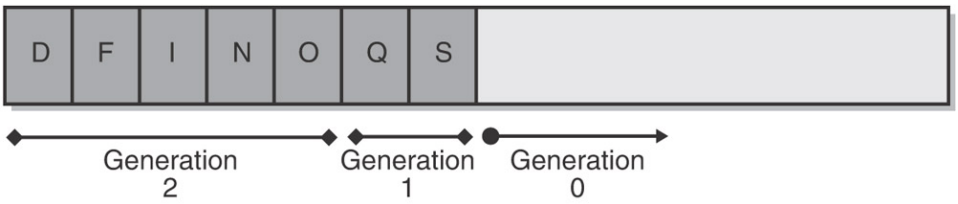


Figure 19–14 : After four collections: generation 1 survivors are promoted to generation 2, generation 0 survivors are promoted to generation 1, and generation 0 is empty.
Like before, any objects that were in generation 0 that survived the garbage collection are now in generation 1; any objects that were in generation 1 that survived the collection are now in generation 2. As always, generation 0 is empty immediately after a garbage collection and is where new objects will be allocated. Objects in generation 2 are objects that the garbage collector has examined at least twice. There might have been several collections, but the objects in generation 1 are examined only when generation 1 reaches its threshold, which usually requires several garbage collections of generation 0.

The managed heap supports only three generations: generation 0, generation 1, and generation 2; there is no generation 3. When the CLR initializes, it selects thresholds for all three generations. As I mentioned earlier, the threshold for generation 0 is about 256 KB, and the threshold for generation 1 is about 2 MB. The threshold for generation 2 is around 10 MB. Again, the threshold sizes are selected to improve performance. The larger the threshold, the less frequently a garbage collection will occur. And again, the performance improvement comes because of the initial assumptions: new objects have short lifetimes, and older objects are likely to live longer.

The CLR's garbage collector is a self–tuning collector. This means that the garbage collector learns about your application's behavior whenever it performs a garbage collection. For example, if your application constructs a lot of objects and uses them for a very short period of time, it's possible that garbage collecting generation 0 will reclaim a lot of memory. In fact, it's possible that the memory for all objects in generation 0 can be reclaimed.

If the garbage collector sees that there are very few surviving objects after collecting generation 0, it might decide to reduce the threshold of generation 0 from 256 KB to 128 KB. This reduction in the allotted space will mean that garbage collections occur more frequently but will require less work for the garbage collector, so your process's working set won't grow much. In fact, if all objects in generation 0 are garbage, then a garbage collection doesn't have to compact any memory; it can simply set **NextObjPtr** back to the beginning of generation 0 and the garbage collection is done. This is a fast way to reclaim memory!

**Note** The garbage collector works extremely well with ASP.NET Web Forms and XML Web service applications. For ASP.NET applications, a client request comes in, a bunch of new objects are constructed, the objects perform work on the client's behalf, and the result is sent back to the client. At this point, all the objects used to satisfy the client's request are garbage. In other words, each ASP.NET application request causes a lot of garbage to be created. Because these objects are unreachable almost immediately after they're created, each garbage collection reclaims a lot of memory. This keeps the process's working set very low, and the garbage collector's performance is phenomenal.

On the other hand, if the garbage collector collects generation 0 and sees that there are a lot of surviving objects, not a lot of memory was reclaimed in the garbage collection. In this case, the garbage collector will grow generation 0's threshold to something like 512 KB. Now, fewer collections will occur, but when they do, a lot more memory should be reclaimed.

Throughout this discussion, I've been talking about how the garbage collector dynamically modifies generation 0's threshold after every collection. But the garbage collector also modifies the thresholds of generation 1 and generation 2 using similar heuristics. When these generations get garbage collected, the garbage collector again sees how much memory is reclaimed and how many objects survive. Based on the garbage collector's findings, it might grow or shrink the thresholds of these generations as well to improve the overall performance of the application.

## Programmatic Control of the Garbage Collector

The **System.GC** type allows your application some direct control over the garbage collector. For starters, you can query the maximum generation supported by the managed heap by reading the **GC.MaxGeneration** property. Currently, this property always returns 2.

You can also force the garbage collector to perform a garbage collection by calling one of the following two static methods:

```
void GC.Collect(Int32 Generation)
void GC.Collect()
```

The first method allows you to specify which generation to collect. You can pass any integer from 0 to **GC.MaxGeneration** inclusive. Passing 0 causes generation 0 to be collected, passing 1 causes generations 1 and 0 to be collected, and passing 2 causes generations 2, 1, and 0 to be collected. The version of the **Collect** method that takes no parameters forces a full collection of all generations and is equivalent to calling

```
GC.Collect(GC.MaxGeneration);
```

Under most circumstances, you should avoid calling any of the **Collect** methods; it's best just to let the garbage collector run on its own accord and fine–tune its generation thresholds based on actual application behavior. However, if you're writing a CUI or GUI application, your application code "owns" the process and the CLR in that process. For these application types, you *might* want to force a garbage collection to occur at certain times.

For example, it might make sense for your application to force a full garbage collection of all generations after the user saves a data file. I'd also imagine Internet browsers performing a full collection each time a page is unloaded. You might also want to force a garbage collection when your application is performing other lengthy operations. The basic idea here is to hide the time

required by a collection when your application is doing something that's already time–consuming. In the three scenarios described, the user will never feel a garbage collection because the time is overshadowed by the other work the application is doing.

The **GC** type also offers a **WaitForPendingFinalizers** method. This method simply suspends the calling thread until the thread processing the freachable queue has emptied the queue, calling each object's **Finalize** method. In most applications, it's unlikely that you'll ever have to call this method. Occasionally, though, I've seen code like this:

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

This code forces a garbage collection. When the collection is complete, the memory for objects that don't require finalization is reclaimed. But the objects that do require finalization can't have their memory reclaimed yet. After the first call to **Collect** returns, the special, dedicated finalization thread is calling **Finalize** methods asynchronously. The call to **WaitForPendingFinalizers** puts the application's thread to sleep until all **Finalize** methods have been called. When **WaitForPendingFinalizers** has returned, all the finalized objects are now truly garbage. At this point, the second call to **Collect** forces another garbage collection, which reclaims all the memory that was occupied by the now finalized objects.

Finally, the **GC** class offers two static methods that allow you to determine which generation an object is currently in:

```
Int32 GetGeneration(Object obj)
Int32 GetGeneration(WeakReference wr)
```

The first version of **GetGeneration** takes an object reference as a parameter, and the second version takes a **WeakReference** reference as a parameter. The value returned will be between **0** and **GC.MaxGeneration** inclusively.

The following code will help you understand how generations work. The code also demonstrates the use of the **GC** methods just discussed.

```
using System;

class GenObj {
   ~GenObj() {
      Console.WriteLine("In Finalize method");
   }
}


class App {
   static void Main() {
      Console.WriteLine("Maximum generations: " + GC.MaxGeneration);

      // Create a new BaseObj in the heap.
      Object o = new GenObj();

      // Because this object is newly created, it is in generation 0.
      Console.WriteLine("Gen " + GC.GetGeneration(o)); // 0

      // Performing a garbage collection promotes the object's generation.
      GC.Collect();
      Console.WriteLine("Gen " + GC.GetGeneration(o)); // 1
```

```
      GC.Collect();
      Console.WriteLine("Gen " + GC.GetGeneration(o)); // 2

      GC.Collect();
      Console.WriteLine("Gen " + GC.GetGeneration(o)); // 2 (max)


      o = null; // Destroy the strong reference to this object.

      Console.WriteLine("Collecting Gen 0");
      GC.Collect(0);                    // Collect generation 0.
      GC.WaitForPendingFinalizers();    // Finalize is NOT called.

      Console.WriteLine("Collecting Gen 1");
      GC.Collect(1);                    // Collect generation 1.
      GC.WaitForPendingFinalizers();    // Finalize is NOT called.

      Console.WriteLine("Collecting Gen 2");
      GC.Collect(2);                    // Same as Collect()
      GC.WaitForPendingFinalizers();    // Finalize IS called.
   }
}
```

Building and running this code yields the following output:

```
Maximum generations: 2
Gen 0
Gen 1
Gen 2
Gen 2
Collecting Gen 0
Collecting Gen 1
Collecting Gen 2
In Finalize method
```

## Other Garbage Collector Performance Issues

Earlier in this chapter, I explained the garbage collection algorithm. However, I made a big assumption during that discussion: that only one thread is running. In the real world, it's likely that multiple threads will be accessing the managed heap or at least manipulating objects allocated within the managed heap. When one thread sparks a garbage collection, other threads must not access any objects (including object references on its own stack) because the garbage collector is likely to move these objects, changing their memory locations.

So when the garbage collector wants to start a garbage collection, all threads executing managed code must be suspended. The CLR has a few different mechanisms that it uses to safely suspend threads so that a garbage collection can be done. The reason there are multiple mechanisms is to keep threads running as long as possible and to reduce overhead as much as possible. I don't want to get into all the details here, but suffice it to say that Microsoft has done a lot of work to reduce the overhead involved with doing a garbage collection. And Microsoft will continue to modify these mechanisms over time to ensure efficient garbage collections in the future.

When the CLR wants to start a garbage collection, it immediately suspends all threads in the process that have ever executed managed code. The CLR then examines each thread's instruction pointer to determine where the thread is executing. The instruction pointer address is then

compared with the JIT compiler—produced tables in an effort to determine what code the thread is executing.

If the thread's instruction pointer is at an offset identified by a table, the thread is said to have reached a *safe point*. A safe point is a place where it's OK to suspend the thread until a garbage collection completes. If the thread's instruction pointer isn't at an offset identified by an internal method table, then the thread isn't at a safe point and the CLR can't perform a garbage collection. In this case, the CLR *hijacks* the thread: the CLR modifies the thread's stack so that the return address points to a special function implemented inside the CLR. The thread is then resumed. When the currently executing method returns, this special function will execute, suspending the thread.

However, the thread might not return from its method for quite some time. So after the thread resumes execution, the CLR waits about 250 milliseconds for the thread to be hijacked. After this time, the CLR suspends the thread again and checks its instruction pointer. If the thread has reached a safe point, the garbage collection can start. If the thread still hasn't reached a safe point, the CLR checks to see whether another method has been called; if one has, the CLR modifies the stack again so that the thread is hijacked when it returns from the most recently executing method. Then the CLR resumes the thread and waits another few milliseconds before trying again.

When all the threads have reached a safe point or have been hijacked, the garbage collection can begin. When the garbage collection is completed, all threads are resumed and the application continues running. The hijacked threads return to the method that originally called them.

**Note** This algorithm has one small twist. If the CLR suspends a thread and detects that the thread is executing unmanaged code, the thread's return address is hijacked and the thread is allowed to resume execution. However, in this case, the garbage collection is allowed to start even though the thread is still executing. This isn't a problem because unmanaged code isn't accessing objects on the managed heap unless the objects are *pinned*. A pinned object is one that the garbage collector isn't allowed to move in memory. If a thread currently executing unmanaged code returns to managed code, the thread is hijacked and is suspended until the garbage collection has completed.

In addition to the mechanisms mentioned earlier (generations, safe points, and hijacking), the garbage collector offers some additional mechanisms that improve the performance of object allocations and collections.

## Synchronization–Free Allocations

On a multiprocessor system running the workstation (MSCorWks.dll) or server (MSCorSvr.dll) version of the execution engine, generation 0 of the managed heap is partitioned into multiple memory arenas, one arena per thread. This allows multiple threads to make allocations simultaneously so that exclusive access to the heap isn't required.

## Scalable Parallel Collections

On a multiprocessor system running the server version of the execution engine (MSCorSvr.dll), the managed heap is split into several sections, one per CPU. When a garbage collection is initiated, the garbage collector has one thread per CPU; each thread collects its own section in parallel with the other threads. Parallel collections work well for server applications where the worker threads tend to exhibit uniform behavior. The workstation version of the execution engine (MSCorWks.dll)

doesn't support this feature.

## Concurrent Collections

On a multiprocessor system running the workstation version of the execution engine (MSCorWks.dll), the garbage collector has an additional background thread that collects objects concurrently, while the application runs. When a thread allocates an object that pushes generation 0 over its threshold, the garbage collector first suspends all threads and then determines which generations to collect. If the garbage collector needs to collect generation 0 or 1, then it proceeds as normal. However, if generation 2 needs collecting, the size of generation 0 grows beyond its threshold to allocate the new object and the application's threads are resumed.

While the application threads are running, the garbage collector has a normal priority background thread that builds the graph of unreachable objects. This thread competes for CPU time with the application's threads, causing the application's tasks to execute more slowly; however, the concurrent collector runs only on multiprocessor systems, so you shouldn't see much of a degradation. Once the graph is built, the garbage collector suspends all threads again and decides whether or not to compact memory. If the garbage collector decides to compact memory, then memory is compacted, root references are fixed up, and the application's threads are resumed—this garbage collection takes less time than usual because the graph of unreachable objects has already been built. However, the garbage collector might decide not to compact memory; in fact, the garbage collector favors this approach. If you have a lot of free memory, the garbage collector won't compact the heap; this improves performance but grows your application's working set. When using the concurrent garbage collector, you'll typically find that your application is consuming more memory than it would compared with the nonconcurrent garbage collection.

To summarize, concurrent collection makes for a better interactive experience for users and is therefore best for interactive CUI or GUI applications. For some applications, however, concurrent collection will actually hurt performance and will cause more memory to be used. When testing your application, you should experiment with and without concurrent collection and see which approach gives the best performance and memory usage for your application.

You can tell the CLR not to use the concurrent collector by creating a configuration file (as discussed in Chapters 2 and 3) that contains a **gcConcurrent** element for the application. Here's an example of a configuration file:

```
<configuration>
   <runtime>
      <gcConcurrent enabled="false"/>
   </runtime>
</configuration>
```

You can also use the Microsoft .NET Framework Configuration administrative tool to create an application configuration file containing the **gcConcurrent** element. To do this, open Control Panel, select Administrative Tools, and then invoke the Microsoft .NET Framework Configuration tool. In the tool, go to the Applications node in the left–hand tree pane and add an application or select an existing application. Then right–click on the application and select Properties. The dialog box shown in Figure 19–15 will appear.

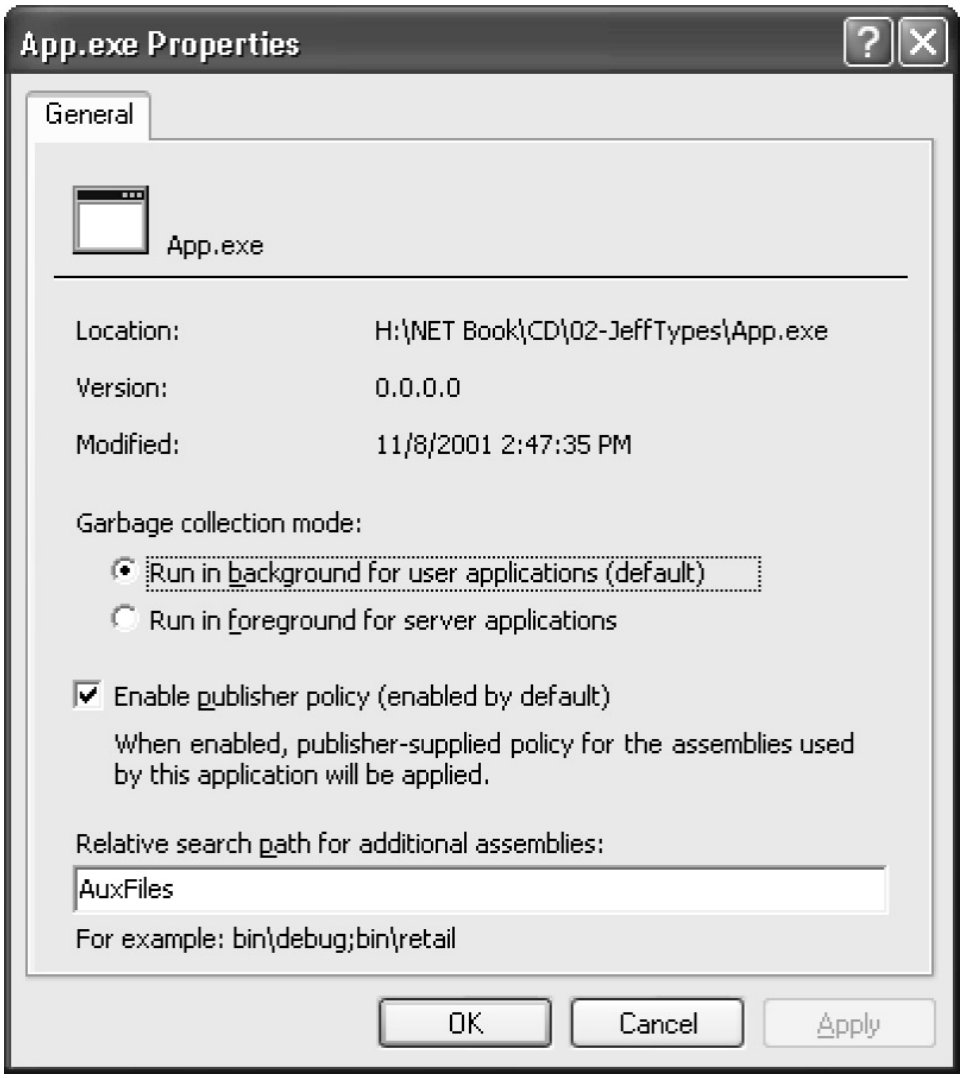


Figure 19–15 : Configuring an application to use the concurrent garbage collector using the Microsoft .NET Framework Configuration administrative tool
The radio buttons under the Garbage Collection Mode selection set the **gcConcurrent** element's **enabled** attribute to **true** or **false**, respectively.

## Large Objects

There is one more performance improvement that you might want to be aware of. Any objects that are 85,000 bytes or more in size are considered to be *large objects*. Large objects are allocated from a special large object heap. Objects in this heap are finalized and freed just like the small objects I've been talking about. However, large objects are never compacted because it would waste too much CPU time shifting 85,000 byte blocks of memory down in the heap.

Large objects are always considered part of generation 2, so you should create large objects only for resources that you need to keep alive for a long time. Allocating short–lived large objects will cause generation 2 to be collected more frequently, which will hurt performance.

All of these mechanisms are transparent to your application code. To you, the developer, it appears as if there is just one managed heap; these mechanisms exist simply to improve application performance.

# Monitoring Garbage Collections

When you install the .NET Framework, it installs a set of performance counters that offer a lot of real–time statistics about the CLR's operations. These statistics are visible via the PerfMon.exe tool or the System Monitor ActiveX control that ships with Windows. The easiest way to access the System Monitor control is to run PerfMon.exe and select the + toolbar button, which causes the Add Counters dialog box shown in Figure 19–6 to appear.

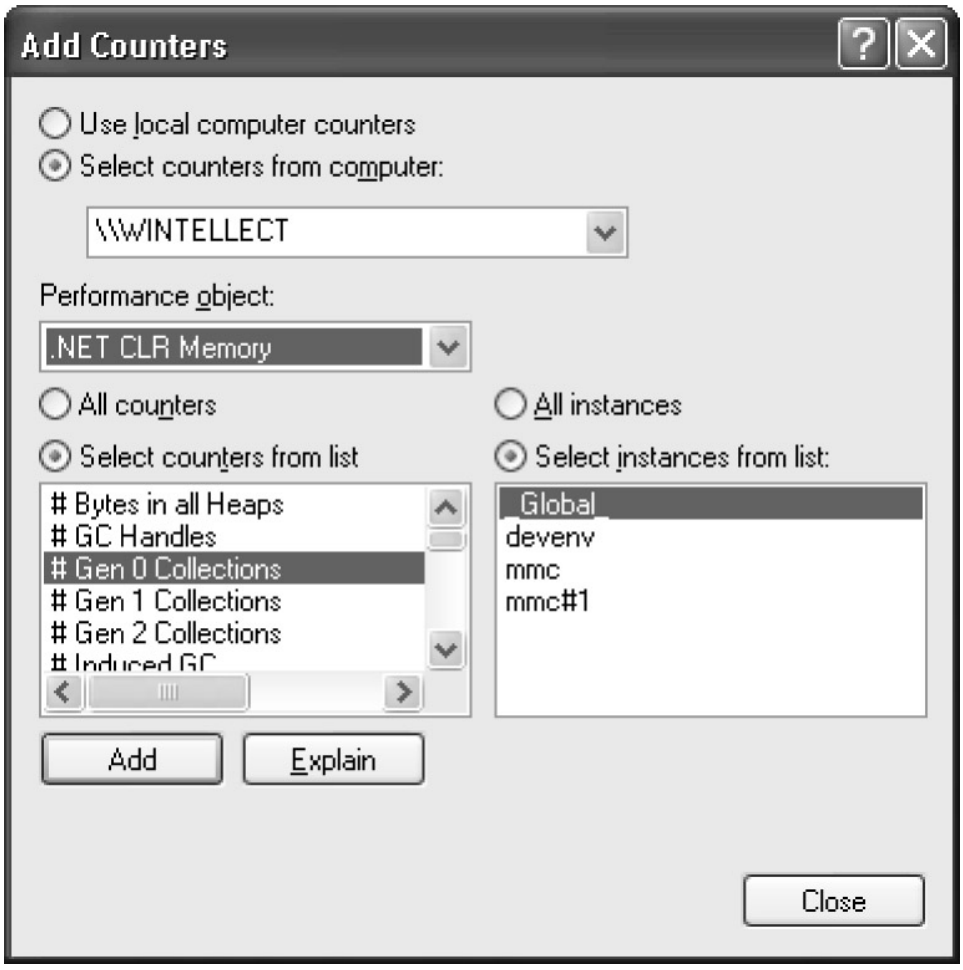



Figure 19–16 : PerfMon.exe showing the .NET CLR memory counters

To monitor the CLR's garbage collector, select the .NET CLR Memory performance object. Then select a specific application from the instance list box. Finally, select the set of counters that you're interested in monitoring and press the Add button followed by the Close button. At this point, the System Monitor will graph the selected real–time statistics. For an explanation of a particular counter, select the desired counter and press the Explain button.

# Chapter 20: CLR Hosting, AppDomains, and Reflection

In this chapter, I'll discuss three main topics that really show off the incredible value provided by the Microsoft .NET Framework. In particular, I'll explain how many of Microsoft's existing application and server products intend to leverage the common language runtime (CLR) in the future. You'll see that your investment in learning the .NET Framework today will certainly pay off down the line. I'll also talk about *AppDomains*, a mechanism offered by the CLR to reduce memory usage and improve system performance. And finally, I'll discuss *reflection*, a mechanism that allows you to design dynamically extensible applications that your types or another party's types can easily enhance.

## Metadata: The Cornerstone of the .NET Framework

By now, it should be obvious to you that metadata is the cornerstone technology of the .NET Framework development platform. Metadata describes a type's fields along with its methods. Metadata is what allows a type developed in one programming language to be consumed by code developed in a completely different programming language. In addition, the garbage collector uses metadata to determine what objects are reachable; the metadata indicates what other objects an object can refer to. Development tools, such as Microsoft Visual Studio's editor, use metadata to provide IntelliSense and other help–related assistance. And, of course, metadata is used to serialize and deserialize objects so that they can be persisted to disk or sent over the network. In fact, this ability to use metadata to easily serialize and deserialize objects over the wire is what makes building XML Web services with the .NET Framework child's play.

Throughout this book, I've been using the ILDasm.exe tool that ships with the .NET Framework SDK. This tool parses the metadata contained inside a managed module or assembly and shows the metadata information in human readable format. The act of examining metadata is called *reflection;* in other words, ILDasm.exe reflects over the module's or assembly's metadata and shows the results to the user.

Reflection is an incredibly powerful tool for developers. Reflection allows developers to build dynamically extensible applications. For example, anyone can produce a type and package it in an assembly. However, if that type follows certain rules, the Visual Studio .NET Windows Forms and Web Forms designers can integrate the type (component) into the designers. Visual Studio can add the type to the Toolbox window, and when an instance of the type is dropped on a form, the Properties window will show the properties that are exposed by the type. This rich level of integration and the ease with which it's produced are unparalleled in earlier technologies such as Win32 and COM. In this chapter, I'll demonstrate how to use reflection to accomplish this level of integration.

A method can use reflection to alter its behavior based on facts learned about another piece of code. You saw an example of this in Chapter 13. If an enumerated type has an instance of the **System.FlagsAttribute** applied to it, then calling **ToString** on an instance of the enumerated type causes the value to be treated as a bit flag instead of a single numeric value. In fact, reflection is really what custom attributes are all about.

Using reflection, a method can alter its behavior based on its caller. For example, it's possible to implement a method that performs a certain operation when called from code in the same assembly. The same method could perform slightly different operations when called from code

outside the assembly. The possibilities are endless.

Before getting too deep into reflection, you need to become familiar with CLR hosting and AppDomains. So I'll spend some time addressing these topics before delving back into reflection to explain how all this fits together.

## CLR Hosting

The .NET Framework runs on top of Microsoft Windows. This means that the .NET Framework must be built using technologies that Windows understands. For starters, all managed module and assembly files must use the Windows portable executable (PE) file format and be either a Windows EXE file or a dynamic–link library (DLL).

When developing the CLR, Microsoft implemented it as a COM server contained inside a DLL; that is, Microsoft defined a standard COM interface for the CLR and assigned GUIDs to this interface and the COM server. When you install the .NET Framework, the COM server representing the CLR is registered in the Windows registry just like any other COM server. If you want more information about this topic, refer to the MSCorEE.h C++ header file that ships with the .NET Framework SDK. This header file defines the GUIDs and the unmanaged **ICorRuntimeHost** interface definition.

Any Windows application can host the CLR. However, you shouldn't create an instance of the CLR COM server by calling **CoCreateInstance;** instead, your unmanaged host should call the **CorBindToRuntimeEx** function (prototyped in MSCorEE.h). The **CorBindToRuntimeEx** function is implemented in the MSCorEE.dll, which is usually found in the C:\Windows\System32 directory. This DLL is called the *shim*, and its job is to determine which version of the CLR to create; the shim DLL doesn't contain the CLR COM server itself.

You see, version 1.0 of the .NET Framework comes with two versions of the CLR COM server. The MSCorWks.dll file contains the workstation version; this version is tuned to offer better performance in single–processor, workstation environments. The MSCorSvr.dll file contains the server version, which is tuned to offer better performance in multiprocessor, server environments. In the future, Microsoft will be producing new versions of the CLR, and these can be installed on a user's hard disk as well.

When **CorBindToRuntimeEx** is called, its parameters allow the host to specify which version of the CLR it would like to create. The version information indicates workstation vs. server as well as a version number. **CorBindToRuntimeEx** uses the specified version information and gathers some additional information of its own (such as how many CPUs are installed in the machine and which versions of the CLR are installed) to decide which version of the CLR to load—the shim might not load the version that the host requested.

By default, the shim examines the managed executable file and extracts the information indicating what version of the CLR the application was built and tested with. However, an application can override the default by placing entries in its XML configuration file (as described in Chapters 2 and 3). The following sample XML configuration file shows how an application can tell the shim to load a particular CLR version:

```
<configuration>
   <startup>
      <requiredRuntime version="v1.0.0.0" safemode="true"/>
   </startup>
</configuration>
```

After examining this information, the shim loads the corresponding CLR version. If **safemode** is set to **false** (the default), the shim loads the most recently installed version of the CLR that is compatible with the version the application wants.

Microsoft determines which versions of the CLR are compatible with other versions by creating some registry settings. You can see the CLR version policy settings by examining the values under the following registry subkey:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\Policy
```

You should never modify any of the values under this registry subkey; when you install a new version of the .NET Framework, the setup program will modify these settings based on Microsoft's compatibility testing.

The **CorBindToRuntimeEx** function returns a pointer to an unmanaged **ICorRuntimeHost** interface (also defined in MSCorEE.h). The hosting application can call methods defined by this interface to initialize the CLR. The host can also call methods telling the CLR what assembly to load and what method to start executing. The section "Loading the Common Language Runtime" on page 9 in Chapter 1 explains how all this works for managed console and Windows Forms applications.

## AppDomains

When the CLR COM server is loaded into a Windows process, it initializes. Part of this initialization is to create the managed heap that all reference objects get allocated in and garbage collected from. In addition, the CLR creates a thread pool usable by any of the managed types whose assemblies are loaded into the process. While initializing, the CLR also creates an *AppDomain*. An AppDomain is a logical container for a set of assemblies. The first AppDomain created when the CLR initializes is called the *default AppDomain;* this AppDomain is destroyed only when the Windows process terminates.

**Note** In version 1 of the .NET Framework, no more than one CLR COM server object can exist in a Windows process, and this CLR COM server object can't be destroyed until the hosting process terminates; that is, **CorBindToRuntimeEx** creates an instance of a CLR COM server and returns a pointer to an **ICorRuntimeHost** interface. Calling the **AddRef** and **Release** methods on this interface has no effect. In addition, a single host process can create only one instance of a CLR COM server. If a single host process calls **CorBindToRuntimeEx** multiple times, the same **ICorRuntimeHost** pointer is returned every time.

In addition to the default AppDomain, a host can instruct the CLR to create additional AppDomains. Moreover, code in a managed assembly can also tell the CLR to create additional AppDomains. Three characteristics of AppDomains make them useful:

- **AppDomains are isolated from one another** An AppDomain can't see the objects created by a different AppDomain. This enforces a clean separation because code in one AppDomain can't have a direct reference to an object created in a different AppDomain. This isolation allows AppDomains to easily be unloaded from a process.
- **AppDomains can be unloaded** The CLR doesn't support the ability to unload a single assembly. However, you can tell the CLR to unload an AppDomain and all the assemblies currently contained in it.

- **AppDomains can be individually secured and configured** When created, an AppDomain can have *evidence* applied to it. Evidence is a security−related feature that determines the maximum rights granted to assemblies running in the AppDomain. More common, however, is that an AppDomain will have a security policy applied to it using **AppDomain**'s **SetAppDomainPolicy** method. In addition, the **System.AppDomainSetup** class allows you to set and query an AppDomain's settings. These configuration settings are used to fine−tune how the CLR locates and loads assemblies. These settings include the following:

  - **ApplicationName** A friendly string name used to identify an AppDomain.
  - **ApplicationBase** A directory where the CLR will look to locate assemblies.
  - **PrivateBinPath** A set of directories where the CLR will look to locate weakly named assemblies.
  - **ConfigurationFile** The pathname of a configuration file containing rules that the CLR will use to locate assemblies. The file also contains remoting settings, Web applications settings, and more.
  - **LoaderOptimization** A flag telling the CLR whether to treat loaded assemblies as domain neutral or single−domain.

**Important** It would be dangerous to run multiple unmanaged applications in a single process. The reason is that the different applications have access to each other's data and code, making it all too easy for one application to corrupt the other application. However, this isn't a concern with managed code because the managed IL code is type−safe and the IL code is verified. This makes it impossible for code in one AppDomain to corrupt code in another AppDomain. Of course, an administrator could turn off verification and allow the managed code to make calls to unmanaged functions. If an administrator does this, all bets are off and AppDomain corruption is entirely possible.

Figure 20−1 shows a single Windows process that has one CLR COM server running in it. This CLR is managing two AppDomains. Each AppDomain has its own loader heap, which maintains a record of which types have been accessed since the AppDomain was created. Each type in the loader heap has a method table, and each entry in the method table points to JIT−compiled x86 code if that method has been executed at least once.

In addition, each AppDomain has some assemblies loaded into it. AppDomain #1 (the default AppDomain) has three assemblies: MyApp.exe, TypeLib.dll, and System.dll. The TypeLib.dll assembly consists of three modules: TypeLib.dll (which contains the manifest), FUT.netmodule, and RUT.netmodule. AppDomain #2 has three single−module assemblies loaded into it: Wintellect.dll, System.dll, and Microsoft.dll.

By default, an assembly is loaded once per AppDomain; that is, the System.dll assembly is actually loaded into both AppDomain #1 and AppDomain #2. This means that the information about the System.dll assembly is reconstructed in each AppDomain's loader heap. Even the JIT−compiled code for the methods defined by System.dll's types exists twice in this process's address space. The advantage of this is that an AppDomain can be completely unloaded from the process without affecting any other AppDomain.

Some assemblies are expected to be used by several AppDomains. The best example is MSCorLib.dll, which is a single−module assembly created by Microsoft. This assembly contains **System.Object**, **System.Int32**, and all the other types that are so integral to the .NET Framework. This assembly is automatically loaded when the CLR initializes, and all AppDomains share the types in this assembly. To reduce resource usage, MSCorLib.dll is loaded in an AppDomain−neutral fashion; that is, the CLR maintains a special loader heap for assemblies that are loaded in a

domain–neutral fashion. Assemblies loaded this way can't be unloaded until the process terminates.

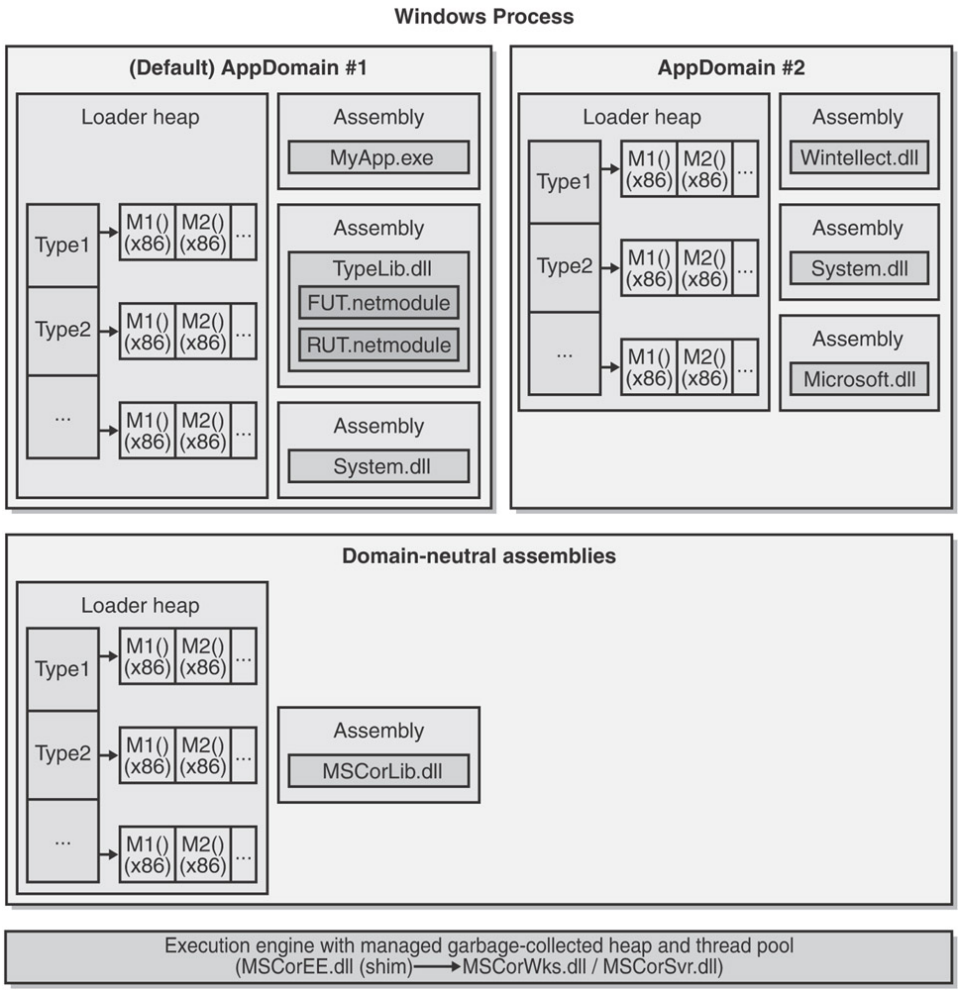



Figure 20–1 : A single Windows process hosting the CLR and two AppDomains

## Accessing Objects Across AppDomain Boundaries

Code in one AppDomain can communicate with types and objects contained in another AppDomain. However, the access to these types and objects is only through well–defined mechanisms. Most types are marshaled by value across AppDomain boundaries. In other words, if an object is constructed in one AppDomain and a reference to this object is passed to another AppDomain, the CLR must first serialize the object's fields into a block of memory. This block of memory is then passed to the other AppDomain, which deserializes the block to produce a new object. The destination AppDomain uses the reference to this new object. The destination AppDomain has no access to the original AppDomain's object. For objects to be remoted by value, the object's type must have the **System.Serializable** custom attribute applied to it.

> **Note** Deserializing an object causes the CLR to load the type's assembly if necessary. If the CLR can't locate the assembly using the destination AppDomain's policies (for example, the AppDomain can have a different AppBase directory), the object can't be deserialized and an exception will be thrown.

Types that are derived from **System.MarshalByRefObject** can also be accessed across AppDomain boundaries. However, access to the object is accomplished by reference rather than by value. Let's say that an AppDomain has an object created in it whose type is derived from **MarshalByRefObject**. When a reference to this object is passed to a destination AppDomain, the

CLR actually creates an instance of a proxy type in the destination AppDomain, and a reference to this proxy object is what code in the destination AppDomain will use. The original object and its fields remain in the original AppDomain. The proxy object is a wrapper that knows how to call instance methods on the original object in the original AppDomain. Again, the destination doesn't have direct access to the original AppDomain's object.

Obviously, accessing objects across AppDomain boundaries has some performance costs associated with it. When possible, you should avoid manipulating objects across AppDomain boundaries.

A one–to–one correspondence doesn't exist between threads and AppDomains. When a thread in one AppDomain calls a method in another AppDomain, the thread transitions between the two AppDomains. This means that method calls across AppDomain boundaries are executed synchronously. However, at any given time, a thread is considered to be in just one AppDomain. You can call **System.Threading.Thread**'s static **GetDomain** method to obtain a reference to the **System.AppDomain** object to identify the AppDomain that the thread is currently executing in.

When unloading an AppDomain, the CLR knows which threads are in the AppDomain, and the CLR forces a **ThreadAbortException** exception in the threads so that they unwind out of the AppDomain. Once the threads have left the AppDomain, the CLR can invalidate all proxy objects that refer to objects in the unloaded AppDomain. At this point, any calls to methods using an invalid proxy will cause a **System.AppDomainUnloadedException** exception to be thrown because the original object no longer exists.

The AppDomainRunner sample application discussed in this chapter's "Explicitly Unloading Assemblies: Unloading an AppDomain" section demonstrates how to marshal an object by reference across an AppDomain boundary.

## AppDomain Events

I want to briefly mention that each AppDomain exposes a number of events that can be incredibly useful. Table 20–1 lists the events that your code can register interest in.

Table 20–1: AppDomain Events

| Event Name | Description |
|---|---|
| **AssemblyLoad** | This event is fired every time the CLR loads an assembly into the AppDomain. The handler receives a **System.Reflection.Assembly** object identifying the loaded assembly. |
| **DomainUnload** | This event is fired just before the AppDomain is unloading. This event isn't fired when the process containing the AppDomain is terminating. |
| **ProcessExit** | This event is fired just before the process terminates. This event is fired only for the default AppDomain; any other AppDomain that registers interest in this event won't receive a notification. |
| **UnhandledException** | This event is fired when an unhandled exception occurs in an AppDomain. This event is covered in Chapter 18. |
| **AssemblyResolve** | This event is fired when the CLR can't locate an assembly required by the AppDomain. The handler receives a string |

| | |
|---|---|
| | identifying the name of the missing assembly. |
| **ResourceResolve** | This event is fired when the CLR can't locate a resource required by the AppDomain. The handler receives a string identifying the name of the missing resource. |
| **TypeResolve** | This event is fired when the CLR can't locate a type required in one of the AppDomain's assemblies. The handler receives a string identifying the name of the missing type and can tell the CLR what type to use by returning a reference to a **Type** object. Frequently, the handler decides what **Type** object to return based on the client's location or operating system. |

## Applications and How They Host the CLR and Manage AppDomains

So far, I've talked about hosts and how they load the CLR, AppDomains, and how the host can tell the CLR to create and unload AppDomains. To make the discussion more concrete, I thought I'd describe some common hosting and AppDomain scenarios. In particular, I'll explain to you how different application types host the CLR and how they manage AppDomains.

### Console and Windows Forms Applications

When invoking a managed console or Windows Forms application, the shim examines the CLR header information contained in the application's assembly. The header information indicates what version of the CLR was used to build and test the application. The shim uses this information to determine which CLR COM server to create. After the CLR loads and initializes, it again examines the assembly's CLR header to determine which method is the application's entry point (**Main**). The CLR invokes this method, and the application is now up and running.

As the code runs, it accesses other types. When referencing a type contained in another assembly, the CLR locates the necessary assembly and loads it into the same AppDomain. Any additionally referenced assemblies also load into the same AppDomain. When the application's **Main** method returns, the default AppDomain unloads and the Windows process terminates.

**Note** By the way, you can call **System.Environment**'s static **Exit** method if you want to shut down the Windows process, including all its AppDomains. **Exit** is the most graceful way of terminating a process because it first calls the **Finalize** methods of all the objects on the managed heap and then releases all the unmanaged COM objects held by the CLR. Finally, **Exit** calls the Win32 **ExitProcess** function.

It's possible for a console or Windows Forms application to tell the CLR to create additional AppDomains in the process's address space. However, these application types rarely ever use or require multiple AppDomains.

### ASP.NET Web Forms and XML Web Services Applications

ASP.NET is an ISAPI DLL (implemented in ASPNet_ISAPI.dll). When a client requests a URL handled by the ASP.NET ISAPI DLL, ASP.NET creates what is called a *worker process* (ASPNet_wp.exe). A worker process is a Windows process that hosts a CLR COM server.

When a client makes a request of a Web application, ASP.NET determines if this is the first time a request has been made. If it is, ASP.NET tells the CLR to create a new AppDomain for this Web application; each Web application is identified by its virtual root directory. ASP.NET then tells the

CLR to load the assembly that contains the type exposed by the Web application into this new AppDomain, creates an instance of this type, and starts calling methods in it to satisfy the client's Web request. If the code references more types, the CLR will load the required assemblies into the Web application's AppDomain. By the way, strongly named assemblies (such as System.Web.dll) are loaded in a domain–neutral fashion to conserve operating system resources.

When future clients make requests of an already running Web application, ASP.NET doesn't create a new AppDomain; instead, it just uses the existing AppDomain, creates a new instance of the Web application's type, and starts calling methods. The methods will already be JIT compiled into native code, so the performance of processing all subsequent client requests is excellent.

If a client makes a request of a different Web application, ASP.NET tells the CLR to create a new AppDomain. This new AppDomain is typically created inside the same worker process as the other AppDomains. This means that many Web applications run in a single Windows process, which improves the efficiency of the system overall. Again, the assemblies required by the different Web application are loaded into its own AppDomain.

**Microsoft Internet Explorer**

When you install the .NET Framework, it installs a MIME filter (MSCorIE.dll) that gets hooked into Internet Explorer versions 5.01 and later. This MIME filter handles downloaded content marked with a MIME type of "application/octet–stream" or "application/x–msdownload". When the MIME filter detects a managed assembly being downloaded, it calls the **CorBindToRuntimeEx** function to create a CLR COM server; this makes Internet Explorer's process a host.

The MIME filter is in control of the CLR and ensures that all assemblies from one Web site are loaded into their own AppDomain. This allows an administrator to treat assemblies downloaded from different Web sites in different ways, say, trusting assemblies from one Web site but not those from another. This also allows assemblies used by one Web application to be unloaded when the user surfs to a different Web application.

## "Yukon"

The next version of Microsoft SQL Server (code–named "Yukon") is an unmanaged application because most of its code is still written in unmanaged C++. During initialization, however, "Yukon" will create a CLR COM server. "Yukon" allows stored procedures to be written in any managed programming language (C#, Visual Basic, Smalltalk, and so on). These stored procedures will run in their own AppDomain that has special evidence applied to it prohibiting the stored procedures from adversely affecting the database server. "Yukon" can instruct the CLR to load just the desired assembly and to call methods in that assembly that will execute under the necessary security restrictions.

This functionality is absolutely incredible! It means that developers will be able to write stored procedures in the programming language of their choice. The stored procedure can use strongly typed data objects in its code. The code will also be JIT compiled into native code when executed, instead of interpreted. And developers can take advantage of any types defined in the .NET Framework Class Library (FCL) or in any other assembly to help them. The result is that our job becomes much easier and our applications perform much better. What more could a developer ask for?!

In the future, productivity applications such as word processors and spreadsheets will also allow users to write macros in any programming language they choose. These macros will have access to

all the assemblies and types that work with the CLR. They will be compiled, so they will execute fast, and, most important, these macros will run in a secure AppDomain so that users don't get hit with any unwanted surprises.

## The Gist of Reflection

As you know, metadata is a bunch of tables. When you build an assembly or a module, the compiler you're using creates a type definition table, a field definition table, a method definition table, and so on. The FCL's **System.Reflection** namespace contains several types that allow you to write code that reflects over (or parses) these metadata tables. In effect, the types in this namespace offer an object model over the metadata contained in an assembly or a module.

Using these object model types, you can easily enumerate all the types in a type definition metadata table. Then for each type, you can obtain its base type, what interfaces it implements, and what flags are associated with the type. Additional types in the **System.Reflection** namespace allow you to query the type's fields, methods, properties, and events by parsing the corresponding metadata tables. You can also discover any custom attributes (covered in Chapter 16) that have been applied to any of the metadata entities. With this information, you could build a tool very similar to Microsoft's ILDasm.exe.

**Important** The types in the **System.Reflection** namespace allow you to query definition metadata tables. No types are offered that allow you to query reference metadata tables. Nor do the reflection types offer the capability to read a method's IL code. The ILDasm.exe tool parses the file's bytes directly to obtain this information. Fortunately, the file format for managed modules and assemblies is public. In fact, this file format is being standardized by the ECMA technical committee. So if you're writing an application that wants to query reference metadata tables or get the IL bytes from a managed method, you'll have to write code that manually parses the file's format because reflection doesn't offer you this capability.

In addition, some definition metadata information isn't obtainable via reflection. For example, there's no way to determine the default value for optional arguments (a feature offered in Visual Basic but not in C#). Microsoft will address these "holes" in the reflection types in future versions of the .NET Framework.

Finally, you should be aware that some of the reflection types and some of the members defined by these types are designed specifically for use by developers who are producing compilers for the CLR. Application developers don't typically use these types and members. The .NET Framework documentation doesn't explicitly point out which of these types and members are for compiler developers rather than application developers, but if you realize that not all reflection types and their members are for everyone, the documentation can be less confusing.

In reality, very few applications will have the need to use the reflection types. Reflection is typically used for class libraries that need to understand a type's definition in order to provide some rich functionality. For example, the FCL's serialization mechanism uses reflection to determine what fields a type defines. The serialization formatter can then obtain the values of these fields and write them into a byte stream for sending across the Internet. Similarly, Visual Studio's designers use reflection to determine which properties should be shown to developers when laying out controls on their Web Forms or Windows Forms at design time.

Reflection is also used when an application needs to load a specific type from a specific assembly at run time to accomplish some task. For example, an application might ask the user to provide the name of an assembly and a type. The application could then explicitly load the assembly, construct an instance of the type, and call methods on the type. This usage is conceptually similar to calling Win32's **LoadLibrary** and **GetProcAddress** functions. Binding to types and calling methods in this way is frequently referred to as *late binding*. (*Early binding* is when the types and methods an application uses are determined at compile time.)

The remainder of this chapter contains sample applications, all of which use reflection. Each application demonstrates a different use of reflection and addresses topics that you'll need to know about to use reflection effectively and efficiently.

## Reflecting Over an Assembly's Types

Reflection is frequently used to determine what types an assembly defines. The Reflector sample code (which can be downloaded from *http://www.Wintellect.com/*) demonstrates how to do this:

```
using System;
using System.Reflection;

class App {
   static void Main() {
      Assembly assem = Assembly.GetExecutingAssembly();
      Reflector.ReflectOnAssembly(assem);
   }
}

public class Reflector {
   public static void ReflectOnAssembly(Assembly assem) {
      WriteLine(0, "Assembly: {0}", assem);

      // Find Modules
      foreach (Module m in assem.GetModules()) {
         WriteLine(1, "Module: {0}", m);

         // Find Types
         foreach (Type t in m.GetTypes()) {
            WriteLine(2, "Type: {0}", t);

            // Find Members
            foreach (MemberInfo mi in t.GetMembers())
               WriteLine(3, "{0}: {1}", mi.MemberType, mi);
         }
      }
   }

   private static void WriteLine(Int32 indent, String format,
      params Object[] args) {

      Console.WriteLine(new String(' ', 3 * indent) + format, args);
   }
}

class SomeType {
   public class InnerType {}
   public Int32 SomeField = 0;
   private static String goo = null;
```

```
   private void SomeMethod() { }

   private TimeSpan SomeProperty {
      get { return new TimeSpan(); }
      set { }
   }

   public static event System.Threading.ThreadStart SomeEvent;

   private void NoCompilerWarnings() {
      // This code is here just to make the compiler warnings go away.
      SomeEvent.ToString();
      goo.ToString();
   }
}
```

In **Main**, the application calls **System.Reflection.Assembly**'s static **GetExecutingAssembly** method. This method determines which assembly contains the method that's making the call and returns a reference to this **Assembly** object. This reference is then passed to the **Reflector** type's static **ReflectOnAssembly** method. This method displays the full name of the assembly and then calls **GetModules**, which returns an array of **System.Reflection.Module** classes. Each element in the array identifies a module that is part of the assembly.

A loop then enumerates over each module in the assembly. In the loop, the module's name is displayed and then **GetTypes** is called. **GetTypes** returns an array of **System.Type** elements; each element identifies a type defined within the assembly's module. A loop then iterates over each type. In this loop, the type's name is displayed and then **GetMembers** is called. This method returns an array of **System.Reflection.MemberInfo** elements, where each element identifies a member (constructor, method, field, property, event, or nested type) of the type.

In the Reflector application, the **SomeType** type exists purely for demonstration purposes. This type defines a bunch of different members showing what you can expect when reflecting over metadata. If you build and run the Reflector application, you'll see the following output:

```
Assembly: Reflector, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
   Module: Reflector.exe
      Type: App
         Method: Int32 GetHashCode()
         Method: Boolean Equals(System.Object)
         Method: System.String ToString()
         Method: System.Type GetType()
         Constructor: Void .ctor()
      Type: Reflector
         Method: Int32 GetHashCode()
         Method: Boolean Equals(System.Object)
         Method: System.String ToString()
         Method: Void ReflectOnAssembly(System.Reflection.Assembly)
         Method: System.Type GetType()
         Constructor: Void .ctor()
      Type: SomeType
         Field: Int32 SomeField
         Method: Int32 GetHashCode()
         Method: Boolean Equals(System.Object)
         Method: System.String ToString()
         Method: Void add_SomeEvent(System.Threading.ThreadStart)
         Method: Void remove_SomeEvent(System.Threading.ThreadStart)
         Method: System.Type GetType()
         Constructor: Void .ctor()
         Event: System.Threading.ThreadStart SomeEvent
```

```
      NestedType: SomeType+InnerType
   Type: SomeType+InnerType
      Method: Int32 GetHashCode()
      Method: Boolean Equals(System.Object)
      Method: System.String ToString()
      Method: System.Type GetType()
      Constructor: Void .ctor()
```

Here you see that the assembly is called **Reflector**, it has a version of 0.0.0.0 and has no specific culture associated with it. In addition, notice that there is no public key token associated with the assembly, making this a weakly named assembly instead of a strongly named assembly. This assembly consists of just one module: Reflector.exe. This module defines four types: **App**, **Reflector**, **SomeType**, and **SomeType+InnerType** (**SomeType**'s nested type). For each type, you see the members defined by the type. Because **SomeType** defines a variety of members, its members are the most interesting.

You'll notice that only publicly defined members are shown in the preceding output. I'll explain how to obtain all the public and nonpublic members when I get to binding flags (coming up shortly).

## Reflecting Over an AppDomain's Assemblies

The Reflector sample is a good introduction to reflection because it shows how to reflect over an assembly's metadata. At times, however, you might want to reflect over all the assemblies contained within an AppDomain. To demonstrate how to reflect over all the assemblies in an AppDomain, I took the Reflector sample application and made a small change to the **Main** method. Here's the new **Main** method (nothing else has changed):

```
static void Main() {
   foreach (Assembly assem in
      AppDomain.CurrentDomain.GetAssemblies()) {

      Reflector.ReflectOnAssembly(assem);
   }
}
```

In this version of **Main**, **System.AppDomain**'s static **CurrentDomain** property is called. This property returns a reference to an **AppDomain** object that identifies the AppDomain containing the calling code. Then the **AppDomain** object's **GetAssemblies** method is called, which returns an array of **System.Reflection.Assembly** elements—one element for each assembly loaded into the AppDomain at the time **GetAssemblies** is called.

At this point, a loop iterates over the array of assemblies, calling **Reflector**'s static **ReflectOnAssembly** method for each one. This version of the Reflector application shows all the assemblies in the AppDomain, all the modules that make up each assembly, all the types defined by each module, and all the members defined by each type.

If you build and run this version of the Reflector sample application, you'll see that two assemblies are shown in the output: MSCorLib.dll and Reflector.exe. Because MSCorLib.dll defines over 1400 types, the output produced by this application is much too long to reprint in this book.

## Reflecting Over a Type's Members: Binding

In the second version of the Reflector application, a type's members are obtained by calling the **Type** object's **GetMembers** method. **Type** actually offers two versions of the **GetMembers** method. The first overload takes no parameters. This version returns the type's publicly defined static and instance members only. The second overload of **GetMembers** takes a single parameter: an instance of a **System.Reflection.BindingFlags** enumerated type. Table 20–2 shows the relevant symbols defined by the **BindingFlags** enumerated type.

Table 20–2: Search Symbols Defined by the BindingFlags Enumerated Type

| Symbol | Value | Description |
|---|---|---|
| **Default** | 0x00 | A placeholder for no flags specified. Use this flag when you don't want to specify any of the flags in the remainder of this table. |
| **IgnoreCase** | 0x01 | Search using case–insensitivity. |
| **DeclaredOnly** | 0x02 | Only search members on the declared type. (Ignore inherited members.) |
| **Instance** | 0x04 | Search instance members. |
| **Static** | 0x08 | Search static members. |
| **Public** | 0x10 | Search public members. |
| **NonPublic** | 0x20 | Search nonpublic members. |
| **FlattenHierarchy** | 0x40 | Search static members defined by base types. |

Earlier, I pointed out that Reflector's output doesn't include a type's private members. You can specify exactly which members the **GetMembers** method should return by passing the desired **BindingFlags** when calling this method.

To reflect over the nonpublic members of a type, I've modified the original Reflector sample application again. In this version, I've now told **GetMembers** to return all public and nonpublic, and static and instance members that are declared by the type itself. Any members inherited from a base type are not displayed. Here's what the new code to call **GetMembers** looks like (nothing else in the source code has changed):

```
BindingFlags bf = BindingFlags.DeclaredOnly |
   BindingFlags.NonPublic | BindingFlags.Public |
   BindingFlags.Instance  | BindingFlags.Static;

foreach (MemberInfo mi in t.GetMembers(bf))
   WriteLine(3, "{0}: {1}", mi.MemberType, mi);
```

Here, **bf** is initialized to a set of flags, indicating that I want to iterate over the type's public and nonpublic, and static and instance members. Furthermore, the **DeclaredOnly** flag indicates that I only want to iterate over the members that the type defines—not any of the members that the type inherits. Once **bf** is initialized, it is passed to the **GetMembers** method, which now knows exactly what members I'm interested in processing.

After making the above change to the Reflector source code, building and running the new version of this application yields the following output:

```
Assembly: Reflector, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```

```
Module: Reflector.exe
   Type: App
      Method: Void Main()
      Constructor: Void .ctor()
   Type: Reflector
      Method: Void ReflectOnAssembly(System.Reflection.Assembly)
      Method: Void WriteLine(Int32, System.String, System.Object[])
      Constructor: Void .ctor()
   Type: SomeType
      Field: Int32 SomeField
      Field: System.String goo
      Field: System.Threading.ThreadStart SomeEvent
      Method: Void SomeMethod()
      Method: System.TimeSpan get_SomeProperty()
      Method: Void set_SomeProperty(System.TimeSpan)
      Method: Void add_SomeEvent(System.Threading.ThreadStart)
      Method: Void remove_SomeEvent(System.Threading.ThreadStart)
      Method: Void NoCompilerWarnings()
      Constructor: Void .cctor()
      Constructor: Void .ctor()
      Property: System.TimeSpan SomeProperty
      Event: System.Threading.ThreadStart SomeEvent
      NestedType: SomeType+InnerType
   Type: SomeType+InnerType
      Constructor: Void .ctor()
```

## Explicitly Loading Assemblies

So far, I've shown you how to reflect on the assemblies that are loaded into an AppDomain. Knowing how to do this is useful, but you must remember that the CLR decides when to load an assembly: the first time a method is called, the CLR examines the method's IL code to see what types are referenced. The CLR then loads all the assemblies that define the referenced types. If a required assembly is already available in the AppDomain, the CLR knows not to load the assembly again.

But let's say you want to write an application that counts the number of types that implement a particular interface. To accomplish this, you'd have to implement a method that references at least one type in each assembly you want loaded. Then you'd have to call this method to force the CLR to load all the assemblies that define these types. Once the assemblies are loaded, you can use the reflection methods already discussed.

This isn't the most ideal way to write this application. Instead, you'd like to have a way to explicitly load assemblies into your AppDomain so that you can reflect over their types. The technique I'm about to describe is similar to using Win32's **LoadLibrary** function.

The **System.Reflection.Assembly** type offers three static methods that allow you to explicitly load an assembly: **Load**, **LoadFrom**, and **LoadWithPartialName**. (Each method offers several overloaded versions.) Of the three methods, I highly recommend that you use **Load** whenever possible and avoid **LoadFrom** and **LoadWithPartialName**.

The **Load** method takes an assembly identity and loads it. By assembly identity, I mean an assembly name, a version, a culture, and a public key token, if the assembly is strongly named. If the assembly is weakly named, the identity is just the name of the assembly (without a file extension). **Load** uses the same algorithm that the CLR uses to implicitly load an assembly. If a strongly named assembly is specified, **Load** causes the CLR to apply policy to the assembly and

look for the assembly in the global assembly cache (GAC) followed by the application's base directory and private path directories. If you call **Load** passing a weakly named assembly, **Load** doesn't apply policy to the assembly and the CLR won't look in the GAC for the assembly. In either case, if the specified assembly can't be found, a **System.IO.FileNotFoundException** exception is thrown.

**Assembly**'s static **LoadFrom** method works differently. When calling **LoadFrom**, you pass the pathname of an assembly file (including the file's extension) and the CLR will load the exact assembly file you specify. The string that you pass to **LoadFrom** can't contain any strong–name information; that is, the string can't include version, culture, or public key information. The CLR doesn't apply any policy to the file you specify, and the CLR doesn't search for the file. If the file doesn't exist at the specified path, a **System.IO.FileNotFoundException** exception is thrown.

The last static method that loads an assembly is **LoadWithPartialName**. You should never use this method because an application won't know for sure what version of an assembly it is loading. The method exists only to help some customers who were using some behavior offered by the .NET Framework during its beta cycle; this behavior was later removed because of its unpredictability.

For those of you who care, here's how **LoadWithPartialName** works. When calling this method, you pass an assembly identity, which includes the assembly's name (without file extension), and you can optionally pass version, culture, and public key token information. When calling **LoadWithPartialName**, the CLR first checks the application's XML configuration file looking for a **qualifyAssembly** element. If this element exists, it tells the CLR how to map a partial assembly identity to a fully qualified assembly identity—the CLR will now use its normal rules to locate the assembly. If no **qualifyAssembly** element exists, the CLR searches in the AppBase and private bin path directories looking for an assembly with the specified name. If a matching assembly is found, the assembly is loaded.

If no matching assembly is found, the CLR looks in the GAC using the portions of the assembly's identity specified by the caller. If the culture or public key information wasn't specified, the behavior of **LoadWithPartialName** is undefined and you're not guaranteed to load any particular assembly. However, if only the version wasn't specified, **LoadWithPartialName** loads the assembly from GAC that has the highest version number.

**Important** Some developers notice that **System.AppDomain** offers a **Load** method. Unlike **Assembly**'s static **Load** method, AppDomain's **Load** method is an instance method that allows you to load an assembly into the specified AppDomain. This method was designed to be called by unmanaged code, and it allows a host to inject an assembly into a specific AppDomain. Managed code developers generally shouldn't call this method. Here's why.

When AppDomain's **Load** method is called, you pass it a string that identifies an assembly. The method then applies policy and searches the normal places—the user's disk or codebase references—looking for the assembly. Recall that an AppDomain has settings associated with it that tell the CLR how to look for assemblies. To load this assembly, the CLR will use the settings associated with the specified AppDomain, not the calling AppDomain.

However, AppDomain's **Load** method returns a reference to an assembly. Because the **System.Assembly** class isn't derived from **System.MarshalByRefObject**, the assembly object must be marshaled by value back to the calling AppDomain. But the CLR will now use the calling AppDomain's settings to locate the assembly and load it. If

the assembly can't be found using the calling AppDomain's policy and search locations, a **FileNotFoundException** exception is thrown. This behavior is usually undesirable and is the reason you should avoid AppDomain's **Load** method.

## Loading Assemblies as "Data Files"

Imagine a user has installed some utility application on her hard drive. The utility consists of two assemblies: SomeTool.exe (the main application's assembly file) and Component.dll (an assembly file containing some components that SomeTool.exe uses). These two assemblies are installed as follows:

```
C:\SomeTool\SomeTool.exe
C:\SomeTool\Component.dll
```

Let's further suppose that the user has an assembly that she wants to process using SomeTool.exe. The assembly file to be processed happens to be named Component.dll and resides in the following location:

```
C:\Component.dll
```

Now, let's say that the user executes the following command line:

```
C:\>C:\SomeTool\SomeTool.exe C:\Component.dll
```

SomeTool.exe's **Main** method starts running and calls **Assembly.LoadFrom**, passing the command–line argument indicating the assembly that the tool is to process. The CLR loads the C:\Component.dll assembly. Now let's say that **Main** calls some other method that references an object defined in the C:\SomeTool\Component.dll assembly. What do you think should happen? Should the CLR realize that an assembly named Component.dll is already loaded in the AppDomain and just use this assembly? Or should the CLR realize that the C:\Component.dll assembly file was loaded "as a data file" and wouldn't normally have been loaded into the AppDomain?

The CLR can't assume that the Component.dll file loaded from C:\ is the same Component.dll file that is in C:\SomeTool because the two assembly files can define completely different types and methods. You'll be glad to hear that the CLR does the right thing here: it loads C:\SomeTool\Component.dll into the AppDomain, and the code in SomeTool.exe will access types and methods defined in the correct assembly.

Here's how this works. Internally, when **Assembly**'s **LoadFrom** method is called, the CLR opens the specified file and extracts the assembly's version, culture, and public key token information from the file's metadata. Then the CLR internally calls **Load**, passing all this information. **Load** applies all the policy information and searches for the assembly. If a matching assembly is found, the CLR compares the full pathname of the assembly file specified by **LoadFrom** with the full pathname of the assembly file found by **Load**. If the pathnames are the same, the assembly is considered to be a normal part of the application. If the pathnames are different or if **Load** doesn't find a matching file, the assembly is considered a "data file" and isn't considered a normal part of the application.

When the CLR needs to find a dependency of an assembly loaded via **LoadFrom**, the CLR goes through the regular probing logic, and if it can't find the dependant assembly in any of those locations, it looks in the directory where the referring assembly was found (and in any subdirectory whose name matches that of the dependant assembly).

Earlier, I mentioned that you should always use **Load** and avoid using **LoadFrom** whenever possible. One reason is that **LoadFrom** is much slower than **Load** because it calls **Load** internally, which has to apply policy and scan several disk locations. Another reason is that an assembly loaded with **LoadFrom** is treated like a "data file," and if your AppDomain does load two identical assembly files from different paths, you're wasting a lot of memory and also hurting run-time performance. Calling **Load** ensures that performance is as good as it can be and that an assembly isn't loaded more than once into an AppDomain.

Whenever you're about to place a call to **LoadFrom**, think about your application and try to figure out a way to change it so that **Load** will work instead. Of course, there will be times when **Load** simply won't do and using **LoadFrom** is necessary (such as in the SomeTool.exe example). Certainly, if you can't use **Load**, use **LoadFrom** instead—just be careful.

## Building a Hierarchy of Exception-Derived Types

The ExceptionTree sample application (source code shown below) displays all classes that are ultimately derived from **System.Exception**. However, I wanted to examine types in several assemblies to produce this tree. To accomplish this, the application must explicitly load all the assemblies whose types I want to consider. This is the job of the application's **LoadAssemblies** method. After loading all the assemblies, the application gets an array of all the AppDomain's assemblies. Then I get an array for all types defined in each assembly.

For each type, I check its base type by querying **Type**'s **BaseType** property. If the **Type** returned is **System.Exception**, this type is an exception type. If the **Type** returned is **System.Object**, the type isn't an exception type. If the **Type** returned isn't either of these types, I check the base type's type recursively until I find **Exception** or **Object**.

The following code is for the ExceptionTree application. The output from this application is shown in Chapter 18 (page 408), so I won't display it here.

```
using System;
using System.Text;
using System.Reflection;
using System.Collections;

class App {
   static void Main() {
      // Explicitly load the assemblies that I want to reflect over.
      LoadAssemblies();

      // Initialize the counters and the exception type list.
      Int32 totalTypes = 0, totalExceptionTypes = 0;
      ArrayList exceptionTree = new ArrayList();

      // Iterate through all assemblies loaded in this AppDomain.
      foreach (Assembly a in AppDomain.CurrentDomain.GetAssemblies()) {

         // Iterate through all types defined in this assembly.
         foreach (Type t in a.GetTypes()) {

            totalTypes++;

            // Ignore type if not a public class.
            if (!t.IsClass || !t.IsPublic) continue;

            // Build a string of the type's derivation hierarchy.
            StringBuilder typeHierarchy =
```

```
            new StringBuilder(t.FullName, 5000);

         // Assume that the type isn't an Exception-derived type.
         Boolean derivedFromException = false;

         // See if System.Exception is a base type of this type.
         Type baseType = t.BaseType;
         while ((baseType != null) && !derivedFromException) {
            // Append the base type to the end of the string.
            typeHierarchy.Append("-" + baseType);

            derivedFromException =
               (baseType == typeof(System.Exception));
            baseType = baseType.BaseType;
         }

         // No more bases and not Exception-derived, so try next type.
         if (!derivedFromException) continue;

         // I found an Exception-derived type.
         totalExceptionTypes++;

         // For this Exception-derived type, reverse the order
         // of the types in the hierarchy.
         String[] h = typeHierarchy.ToString().Split('-');
         Array.Reverse(h);

         // Build a new string with the hierarchy in order
         // from Exception -> Exception-derived type.
         // Add the string to the list of Exception types.
         exceptionTree.Add(String.Join("-", h, 1, h.Length - 1));
      }
   }

   // Sort the Exception types together in order of their hierarchy.
   exceptionTree.Sort();

   // Display the Exception tree.
   foreach (String s in exceptionTree) {
      // For this Exception type, split its base types apart.
      String[] x = s.Split('-');

      // Indent based on the number of base types,
      // and then show the most derived type.
      Console.WriteLine(
         new String(' ', 3 * x.Length) + x[x.Length - 1]);
   }

   // Show the final status of the types considered.
   Console.WriteLine("\n-> Of {0} types, {1} are " +
      "derived from System.Exception.",
      totalTypes, totalExceptionTypes);
   Console.ReadLine();
}

static void LoadAssemblies() {
   String[] assemblies = {
      "System,                       PublicKeyToken={0}",
      "System.Data,                  PublicKeyToken={0}",
      "System.Design,                PublicKeyToken={1}",
      "System.DirectoryServices,     PublicKeyToken={1}",
      "System.Drawing,               PublicKeyToken={1}",
      "System.Drawing.Design,        PublicKeyToken={1}",
```

```
         "System.EnterpriseServices,     PublicKeyToken={1}",
         "System.Management,             PublicKeyToken={1}",
         "System.Messaging,              PublicKeyToken={1}",
         "System.Runtime.Remoting,       PublicKeyToken={0}",
         "System.Security,               PublicKeyToken={1}",
         "System.ServiceProcess,         PublicKeyToken={1}",
         "System.Web,                    PublicKeyToken={1}",
         "System.Web.RegularExpressions, PublicKeyToken={1}",
         "System.Web.Services,           PublicKeyToken={1}",
         "System.Windows.Forms,          PublicKeyToken={0}",
         "System.Xml,                    PublicKeyToken={0}",
      };

      String EcmaPublicKeyToken = "b77a5c561934e089";
      String MSPublicKeyToken   = "b03f5f7f11d50a3a";

      // Get the version of the assembly containing System.Object.
      // I'll assume the same version for all the other assemblies.
      Version version =
         typeof(System.Object).Assembly.GetName().Version;

      // Explicitly load the assemblies that I want to reflect over.
      foreach (String a in assemblies) {
         String AssemblyIdentity =
            String.Format(a, EcmaPublicKeyToken, MSPublicKeyToken) +
            ", Culture=neutral, Version=" + version;

         Assembly.Load(AssemblyIdentity);
      }
   }
}
```

## Explicitly Unloading Assemblies: Unloading an AppDomain

The CLR doesn't support the ability to unload an assembly. Instead, you can unload an AppDomain, which causes all the assemblies contained within it to be unloaded. Unloading an AppDomain is very easy: you just call **AppDomain**'s static **Unload** method, passing it a reference to the **AppDomain** you want unloaded.

**Note** As I mentioned previously, assemblies that are loaded in a domain–neutral fashion can never be unloaded from an AppDomain. To "unload" these assemblies, the process must be terminated.

The AppDomainRunner sample application demonstrates how to create a new AppDomain, use a type in it, and then unload the AppDomain along with all its assemblies. The code also shows how to define a type that can be marshaled by reference across AppDomain boundaries. Finally, it shows what happens if you attempt to access a marshal–by–reference object that used to exist in an AppDomain that's been unloaded.

```
using System;
using System.Reflection;
using System.Threading;

class App {
   static void Main() {
      // Create a new AppDomain.
      AppDomain ad =
         AppDomain.CreateDomain("MyNewAppDomain", null, null);
```

```
      // Create a new MarshalByRef object in the new AppDomain.
      MarshalByRefType mbrt = (MarshalByRefType)
         ad.CreateInstanceAndUnwrap(
            Assembly.GetCallingAssembly().FullName,
            "MarshalByRefType");

      // Call a method on this object. The proxy remotes
      // the call to the other AppDomain.
      mbrt.SomeMethod(Thread.GetDomain().FriendlyName);

      // I'm done using the other AppDomain, so
      // I'll unload it and all its assemblies.
      AppDomain.Unload(ad);

      // Try to call a method on the other AppDomain's object.
      // The object was destroyed when the AppDomain was unloaded,
      // so an exception is thrown.
      try {
         mbrt.SomeMethod(Thread.GetDomain().FriendlyName);

         // The following line should NOT be displayed.
         Console.WriteLine(
            "Called SomeMethod on object in other AppDomain.\n" +
            "This shouldn't happen.");
      }
      catch (AppDomainUnloadedException) {
         // I'll catch the exception here, and the
         // following line should be displayed.
         Console.WriteLine(
            "Fail to call SomeMethod on object in other AppDomain.\n" +
            "This should happen.");
      }

      Console.ReadLine();
   }
}


// This type is derived from MarshalByRefObject.
class MarshalByRefType : MarshalByRefObject {

   // This instance method can be called via a proxy.
   public void SomeMethod(String sourceAppDomain) {

      // Display the name of the calling AppDomain and my AppDomain.
      // NOTE: The application's thread has transitioned between AppDomains.
      Console.WriteLine(
         "Code from the '{0}' AppDomain\n" +
         "called into the '{1}' AppDomain.",
         sourceAppDomain, Thread.GetDomain().FriendlyName);
   }
}
```

If you build and run this application, you'll see the following output:

```
Code from the 'AppDomainRunner.exe' AppDomain
called into the 'MyNewAppDomain' AppDomain.
Fail to call SomeMethod on object in other AppDomain.
This should happen.
```

# Obtaining a Reference to a System.Type Object

Reflection is most commonly used to learn about types or to manipulate objects using information that is typically known only at run time, not at compile time. Obviously, this dynamic exploring and manipulation of types and objects comes at a performance hit, so you should use it sparingly. In addition, a compiler can't help you locate and fix programming errors related to type safety when you're using reflection.

The **System.Type** type is your starting point for doing type and object manipulations. **System.Type** is an abstract base type derived from **System.Reflection.MemberInfo** (because a **Type** can be a member of another type). The FCL provides a few types that are derived from **System.Type**: **System.RuntimeType**, **System.Reflection.TypeDelegator**, some types defined in the **System.Reflection.Emit** namespace, **EnumBuilder**, and **TypeBuilder**. Aside from the few classes in the FCL, Microsoft doesn't expect to define any other types that derive from **Type**.

> **Note** The **TypeDelegator** class allows code to dynamically subclass a **Type** by encapsulating the **Type**, allowing you to override some of the functionality while having the original **Type** handle most of the work. In general, the **TypeDelegator** type isn't useful. In fact, Microsoft isn't aware of anyone actually using the **TypeDelegator** type for anything.

Of all these types, the **System.RuntimeType** is by far the most interesting. **RuntimeType** is a type that is internal to the FCL, which means that you won't find it documented in the .NET Framework documentation. The first time a type is accessed in an AppDomain, the CLR constructs an instance of a **RuntimeType** and initializes the object's fields to reflect (pun intended) information about the type.

Recall that **System.Object** defines a method named **GetType**. When you call this method, the CLR determines the specified object's type and returns a reference to its **RuntimeType** object. Because there is only one **RuntimeType** object per type in an AppDomain, you can use equality and inequality operators to see whether two objects are of the same type:

```
Boolean AreObjectsTheSameType(Object o1, Object o2) {
   return o1.GetType() == o2.GetType();
}
```

In addition to calling **Object**'s **GetType** method, the FCL offers several more ways to obtain a **Type** object:

- The **System.Type** type offers several overloaded versions of a static **GetType** method. All versions of this method take a **String**. The string must specify the full name of the type (including its namespace), and compiler primitive types (such as C#'s **int**, **string**, **bool**, and so on) aren't allowed. If the string is simply the name of a type, the method checks the calling assembly to see whether it defines a type of the specified name. If it does, a reference to the appropriate **RuntimeType** object is returned.

  If the calling assembly doesn't define the specified type, the types defined by MSCorLib.dll are checked. If a type with a matching name still can't be found, **null** is returned or a **System.TypeLoadException** exception is thrown, depending on which **GetType** method you call and what parameters you pass to it. The .NET Framework documentation fully explains this method.

You can pass an assembly qualified type string, such as “System.Int32, mscorlib, Version=1.0.2411.0, Culture=neutral, PublicKeyToken=b77a5c561934e089”, to **GetType**. In this case, **GetType** will look for the type in the specified assembly (loading the assembly if necessary).

- The **System.Type** type offers the following instance methods: **GetNestedType** and **GetNestedTypes**.
- The **System.Reflection.Assembly** type offers the following instance methods: **GetType**, **GetTypes**, and **GetExportedTypes**.
- The **System.Reflection.Module** type offers the following instance methods: **GetType**, **GetTypes**, and **FindTypes**.

Many programming languages also offer an operator that allows you to obtain a **Type** object from a type name. When possible, you should use this operator to obtain a reference to a **Type** instead of using any of the methods in the preceding list because the operator generally produces faster code. In C#, the operator is called **typeof**. The following code demonstrates how to use it:

```
static void SomeMethod() {
   Type t = typeof(MyType);
   Console.WriteLine(t.ToString());   // Displays "MyType"
}
```

I compiled this code and obtained its IL code using ILDasm.exe. I’ll explain what’s going on in the annotated IL code here:

```
.method private hidebysig static void  SomeMethod() cil managed {
  // Code size       23 (0x17)
  .maxstack  1
  .locals ([0] class [mscorlib]System.Type t)

  // Look up the MyType metadata token, and place a "handle" to
  // the internal data structure on the stack.
  IL_0000:  ldtoken    MyType

  // Look up the RuntimeTypeHandle, and put a reference to
  // the corresponding RuntimeType object on the stack.
  IL_0005:  call       class [mscorlib]System.Type
      [mscorlib]System.Type::GetTypeFromHandle(
         valuetype [mscorlib]System.RuntimeTypeHandle)

  // Save the RuntimeType reference in the local variable t.
  IL_000a:  stloc.0

  // Load the reference in t on the stack.
  IL_000b:  ldloc.0

  // Call the RuntimeType object’s ToString method.
  IL_000c:  callvirt   instance string[mscorlib]System.Type::ToString()

  // Pass the String to Console.WriteLine.
  IL_0011:  call       void [mscorlib]System.Console::WriteLine(string)

  // Return from the method.
  IL_0016:  ret
} // End of method App::Foo
```

The **ldtoken** IL instruction has a metadata token specified as an operand. The **ldtoken** instruction causes the CLR to look for the internal data structure representing the specified metadata token. If an internal data structure for this metadata token doesn’t exist, the CLR will create it on the fly. A

"handle" to this internal structure, represented as a **System.RuntimeTypeHandle** (a value type), is then pushed on the virtual stack. This "handle" is really the memory address of the internal data structure, but you should never access these internal structures directly.

Now **System.Type**'s static **GetTypeFromHandle** method is called. This method takes the "handle" and returns a reference to the **RuntimeType** object for the type. The rest of the IL code just saves the **RuntimeTime** reference in the variable **t** and then calls **ToString** on **t;** the resulting string is passed to **Console.WriteLine** and the method then returns.

**Note** The **ldtoken** IL instruction allows you to specify a metadata token representing an entry in the type definition/reference table, the method definition/reference table, or the field definition/reference table. Keep in mind that C#'s **typeof** operator accepts only the name of a type defined in the module or a type referenced in another module; you can't specify a field or method name.

It's extremely rare that you'd ever need to obtain a "handle" for a field or method, which is why most compilers won't offer operators that emit an **ldtoken** instruction that has a field or method metadata token. Field and method "handles" are most useful to compiler writers, not to application developers. If you're interested in field handles, however, see the **System.RuntimeFieldHandle** type and **System.Reflection.FieldInfo**'s static **GetFieldFromHandle** method and instance **Handle** property. For method handles, see the **System.RuntimeMethodHandle** type and **System.Reflection.MethodBase**'s static **GetMethodFromHandle** method and instance **MethodHandle** property.

Once you have a reference to a **Type** object, you can query many of the type's properties to learn more about it. Most of the properties, such as **IsPublic**, **IsSealed**, **IsAbstract**, **IsClass**, **IsValueType**, and so on, indicate flags associated with the type. Other properties, such as **Assembly**, **AssemblyQualifiedName**, **FullName**, **Module**, and so on, return the name of the type's defining assembly or module and the full name of the type. You can also query the **BaseType** property to obtain the type's base type, and a slew of methods will give you even more information about the type.

The .NET Framework documentation describes all the methods and properties that **Type** exposes. Be aware that there are a lot of them. For example, the **Type** type offers about 45 public instance properties. This doesn't even include the methods and fields that **Type** also defines. I'll be covering some of these methods in the next section.

**Note** By the way, if you need to obtain a **Type** object that identifies a reference to a type, you can call one of the **GetType** methods, passing the name of the type suffixed with an ampersand, as demonstrated in the following code:

```
using System;
using System.Reflection;

class App {
   static void Main() {
      // Get the array of SomeMethod's parameters.
      ParameterInfo[] p =
         typeof(App).GetMethod("SomeMethod").GetParameters();

      // Get a reference to a type that identifies a String reference.
      Type stringRefType = Type.GetType("System.String&");

      // Is SomeMethod's first parameter a String reference?
      Console.WriteLine(p[0].ParameterType == stringRefType);
```

```
      // "True"
   }

   // You get identical results if 'ref' is changed to 'out' here:
   public void SomeMethod(ref String s) {
      s = null;
   }
}
```

## Reflecting Over a Type's Members

Fields, constructors, methods, properties, events, and nested types can all be defined as members within a type. The FCL contains a type called **System.Reflection.MemberInfo**. The various versions of the Reflector sample application discussed earlier in this chapter used this type to demonstrate how to discover what members a type defines. Table 20–3 shows several properties and methods offered by the **MemberInfo** type. These properties and methods are common to all type members.

Most of the properties mentioned in Table 20–3 are self–explanatory. However, developers frequently confuse the **DeclaringType** and **ReflectedType** properties. To fully understand these properties, let's define the following type:

```
class MyType {
   public override String ToString() { return "Hi"; }
}
```

What would happen if the following line of code executed?

```
MemberInfo[] members = typeof(MyType).GetMembers();
```

Table 20–3: Properties and Methods Common to All MemberInfo–Derived Types

| Member Name | Member Type | Description |
|---|---|---|
| **Name** | **String** property | Returns a **String** representing the member. |
| **MemberType** | **MemberTypes** (enum) property | Returns the kind of member (field, constructor, method, property, event, type (non–nested type), or nested type). |
| **DeclaringType** | **Type** property | Returns the **Type** that defines the member. |
| **ReflectedType** | **Type** property | Returns the **Type** that was used to obtain this member. |
| **GetCustomAttributes** | Method returning<br><br>Object[] | Returns an array in which each element identifies an instance of a custom attribute applied to this member. Custom attributes can be applied |

| | | |
|---|---|---|
| | | to any member. |
| **IsDefined** | Method returning<br><br>Boolean | Returns **true** if one and only one instance of the specified custom attribute is applied to the member. |

The **members** variable is a reference to an array in which each element identifies a public member defined by **MyType** and any of its base types, such as **System.Object**. If you were to query the **DeclaringType** property for the **MemberInfo** element identifying the **ToString** method, you'd see **MyType** returned because **MyType** declares or defines a **ToString** method. On the other hand, if you were to query the **DeclaringType** property for the **MemberInfo** element identifying the **Equals** method, you'd see **System.Object** returned because **Equals** is declared by **System.Object**, not by **MyType**.

The **ReflectedType** property always returns **MyType** because this was the type specified when **GetMembers** was called to perform the reflection. If you look up the **MemberInfo** type in the .NET Framework documentation, you'll see that it is a class derived immediately from **System.Object**. Figure 20–2 shows the hierarchy of the reflection types.

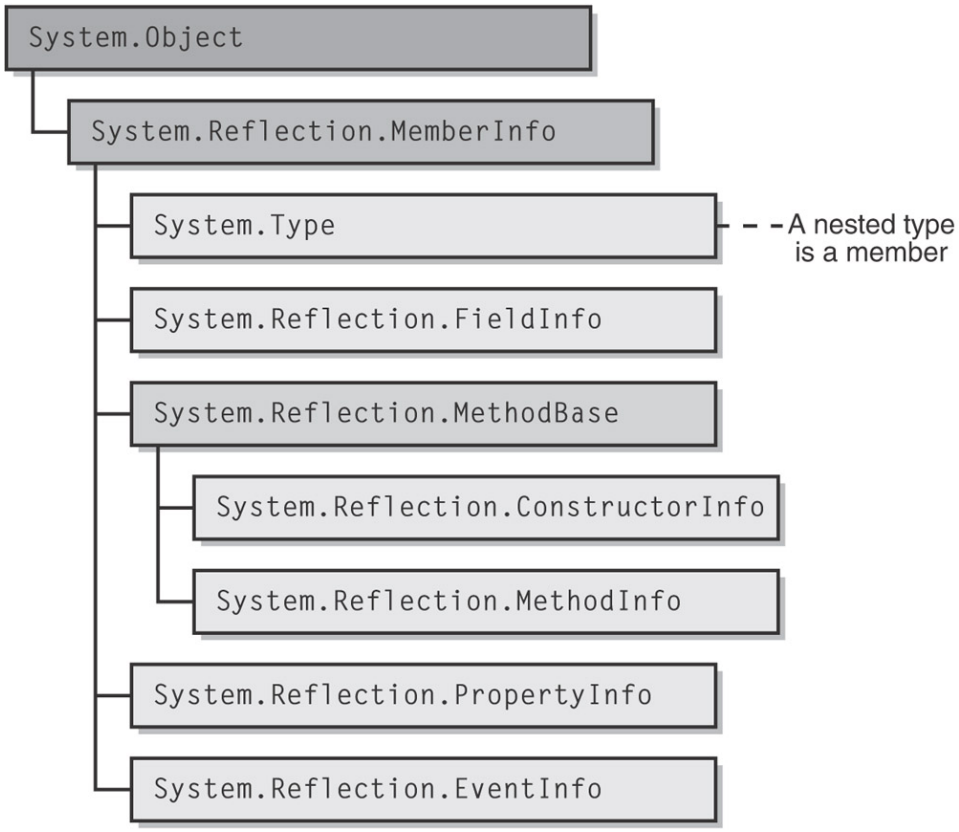



Figure 20–2 : Hierarchy of the reflection types

**Note** Don't forget that **System.Type** is derived from **MemberInfo** and therefore, **Type** also offers all the properties shown in Table 20–3.

Each element of the array returned by calling **GetMembers** is a reference to one of the concrete types in this hierarchy. While **Type**'s **GetMembers** method returns all the type's members, **Type** also offers methods that return specific member types. For example, **Type** offers **GetNestedTypes**, **GetFields**, **GetConstructors**, **GetMethods**, **GetProperties**, and **GetEvents**. These methods all return arrays where each element is a **Type**, **FieldInfo**, **ConstructorInfo**, **MethodInfo**, **PropertyInfo**, or **EventInfo**, respectively.

Figure 20–3 summarizes the types used by an application to walk reflection's object model. From an AppDomain, you can discover the assemblies loaded into it. From an assembly, you can discover the modules that make it up. From an assembly or a module, you can discover the types that it defines. From a type, you can discover its nested types, fields, constructors, methods, properties, and events.

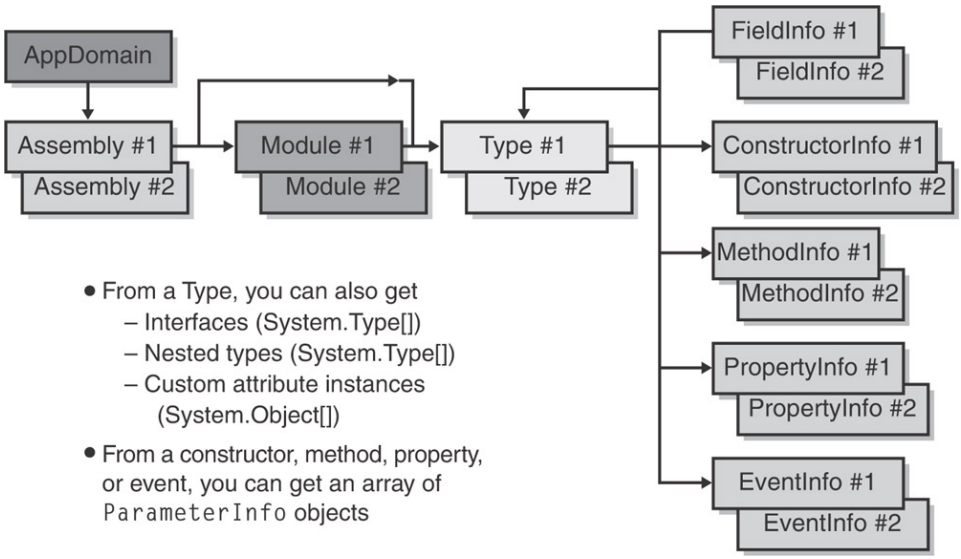



Figure 20–3 : Types an application uses to walk reflection's object model

## Creating an Instance of a Type

Once you have a reference to a **Type**–derived object, you might want to create an instance of this type. The FCL offers several mechanisms to accomplish this.

- **System.Activator's CreateInstance methods** This class offers several overloads of a static **CreateInstance** method. When you call this method, you can pass either a reference to a **Type** object or a **String** that identifies the type of object you want to create. The versions that take a type are simpler. You get to pass a set of arguments to the type's constructor and the method returns a reference to the new object.

  The versions of this method in which you specify the desired type with a string are a bit more complex. First, you must also specify a string identifying the assembly that defines the type. Second, these methods allow you to construct a remote object if you have remoting options configured properly. Third, these versions don't return a reference to the new object. Instead, they return a **System.Runtime.Remoting.ObjectHandle** (which is derived from **System.MarshalByRefObject**).

  An **ObjectHandle** is a type that allows an object created in one AppDomain to be passed around to other AppDomains without forcing the assembly that defines the type to be loaded into these AppDomains. When you're ready to access the object, you call **ObjectHandle**'s **Unwrap** method. Only when **Unwrap** is called does the assembly that contains the type's metadata get loaded. If the assembly can't get loaded, **Unwrap** throws a **System.Runtime.Remoting.RemotingException** exception. Note that **Unwrap** must be called before the object's lifetime expires; this is 5 minutes by default.
- **System.Activator's CreateInstanceFrom methods** The **Activator** class also offers a set of static **CreateInstanceFrom** methods. These methods behave just like the **CreateInstance** method except that you must always specify the type and its assembly via string parameters. The assembly is loaded into the calling AppDomain using **Assembly**'s **LoadFrom** method (instead of **Load**). Because none of these methods takes a **Type**

parameter, all the **CreateInstanceFrom** methods return a reference to an **ObjectHandle**, which must be unwrapped.

- **System.AppDomain's methods** The AppDomain type offers four instance methods that construct an instance of a type: **CreateInstance**, **CreateInstanceAndUnwrap**, **CreateIntanceFrom**, and **CreateInstanceFromAndUnwrap**. These methods work just like **Activator**'s methods except that methods are instance methods allowing you to specify which AppDomain the object should be constructed in. The methods that end with **Unwrap** exist for convenience so that you don't have to make an additional method call.
- **System.Type's InvokeMember instance method** Using a reference to a **Type** object, you can call the **InvokeMember** method. This method locates a constructor matching the parameters you pass and constructs the type. The type is always created in the calling AppDomain and a reference to the new object is returned. I'll discuss this method in more detail later in this chapter.
- **System.Reflection.ConstructorInfo's Invoke instance method** Using a reference to a **Type** object, you can bind to a particular constructor and obtain a reference to the constructor's **ConstructorInfo** object. Then you can use the reference to the **ConstructorInfo** object to call its **Invoke** method. The type is always created in the calling AppDomain and a reference to the new object is returned. I'll also discuss this method in more detail later in this chapter.

**Note** The CLR doesn't require that value types define any constructors. However, this is a problem because all the mechanisms in the preceding list construct an object by calling its constructor. To fix this problem, Microsoft "enhanced" some of **Activator**'s **CreateInstance** methods so that they can create an instance of a value type without calling a constructor. If you want to create an instance of a value type without calling a constructor, you must call the version of the **CreateInstance** method that takes a single **Type** parameter or the version that takes **Type** and **Boolean** parameters.

The mechanisms just listed allow you to create an object for all types except for arrays (**System.Array**–derived types) and delegates (**System.MulticastDelegate**–derived types).

To create an array, you should call **Array**'s static **CreateInstance** method (several overloaded versions exist). The first parameter to all versions of **CreateInstance** is a reference to the **Type** of elements you want in the array. **CreateInstance**'s other parameters allow you to specify various combinations of dimensions and bounds.

To create a delegate, you should call **Delegate**'s static **CreateDelegate** method (several overloads exist). The first parameter to all versions of **CreateDelegate** is a reference to the **Type** of delegate you want to create. **CreateDelegate**'s other parameters allow you to specify which instance method of an object or which static method of a type the delegate should wrap.

## Calling a Type's Method

The easiest way to call a method is to use **Type**'s **InvokeMember** method. This method is quite powerful in that it lets you do lots of stuff. There are several overloaded versions of **InvokeMember**. I'll discuss the one that has the most parameters; the other overloads simply pick defaults for certain parameters, making them easier to call.

```
class Type {
   public Object InvokeMember(
      String name,               // Name of member
      BindingFlags invokeAttr,   // How to look up members
      Binder binder,             // How to match members and arguments
```

```
      Object target,             // Object to invoke member on
      Object[] args,             // Arguments to pass to method
      CultureInfo culture);      // Culture used by some binders
   :
}
```

When you call **InvokeMember**, it looks at the type's members for a match. If no match is found, a **System.MissingMethodExcpetion** exception is thrown. If a match is found, **InvokeMember** calls the method. Whatever the method returns is what **InvokeMember** returns back to you. If the method has a return type of **void**, **InvokeMember** returns **null**. If the method you call throws an exception, **InvokeMember** catches the exception and throws a new **System.Reflection.TargetInvocationException** exception. The **TargetInvocationException** object's **InnerException** property will contain the actual exception that the invoked method threw. Personally, I don't like this behavior. I'd prefer it if **InvokeMember** didn't wrap the exception and just allowed it to come through.

Internally, **InvokeMember** performs two operations. First, it must select the appropriate member to be called—this is known as *binding*. Second, it must actually invoke the member—this is known as *invoking*.

When you call **InvokeMember**, you pass a string as the **name** parameter, indicating the name of the member you want **InvokeMember** to bind to. However, the type might offer several members with a particular name. After all, there might be several overloaded versions of a method, or a method and a field might have the same name. Of course, **InvokeMember** must bind to a single member before it can invoke it. All the parameters passed to **InvokeMember** (except for the **target** parameter) are used to help **InvokeMember** decide which member to bind to. Let's take a closer look at these parameters.

The **binder** parameter identifies an object whose type is derived from the abstract **System.Reflection.Binder** type. A **Binder**–derived type is a type that encapsulates the rules for how **InvokeMember** should select a single member. The **Binder** base type defines abstract virtual methods such as **BindToField**, **BindToMethod**, **ChangeType**, **ReorderArgumentArray**, **SelectMethod**, and **SelectProperty**. Internally, **InvokeMember** calls these methods using the **Binder** object passed via **InvokeMember**'s **binder** parameter.

Microsoft has defined an internal (undocumented) concrete type, called **DefaultBinder**, which is derived from **Binder**. This **DefaultBinder** type ships with the FCL, and Microsoft expects that almost everyone will use this binder. When you pass **null** to **InvokeMember**'s **binder** parameter, it will use the **DefaultBinder**. **Type** offers a public read–only property, **DefaultBinder**, that you can query to obtain a reference to a **DefaultBinder** object should you want one for some reason.

If you're concerned about **DefaultBinder**'s rules, you must define your own **Binder**–derived type and pass an instance of this type via **InvokeMember**'s **binder** parameter. Sample source code for a simple **Binder**–derived type can be obtained by downloading the code associated with this book from *http://www.Wintellect.com/*.

When a binder object has its methods called, the methods will be passed parameters to help the binder make a decision. Certainly, the binder is passed the name of the member that is being looked for. In addition, the binder's methods are passed the specified **BindingFlags** as well as the types of all the parameters that need to be passed to the member being invoked.

Earlier in this chapter, I showed a table (Table 20–2) that described the following **BindingFlags**: **Default**, **IgnoreCase**, **DeclaredOnly**, **Instance**, **Static**, **Public**, **NonPublic**, and

**FlattenHierarchy**. The presence of these flags tells the binder which members to include in the search.

In addition to these flags, the binder examines the number of arguments passed via **InvokeMember**'s **args** parameter. The number of arguments limits the set of possible matches even further. The binder then examines the types of the arguments to limit the set even more. However, when it comes to the argument's types, the binder applies some automatic type conversions to make things a bit more flexible. For example, a type can define a method that takes a single **Int64** parameter. If you call **InvokeMember** and for the **args** parameter pass a reference to an array containing an **Int32** value, the **DefaultBinder** considers this a match. When invoking the method, the **Int32** value will be converted to an **Int64** value. The **DefaultBinder** supports the conversions listed in Table 20–4.

Table 20–4: Conversions That DefaultBinder Supports

| Source Type | Target Type |
|---|---|
| Any type | Its base type |
| Any type | The interface it implements |
| **Char** | **UInt16**, **UInt32**, **Int32**, **UInt64**, **Int64**, **Single**, **Double** |
| **Byte** | **Char**, **UInt16**, **Int16**, **UInt32**, **Int32**, **UInt64**, **Int64**, **Single**, **Double** |
| **SByte** | **Int16**, **Int32**, **Int64**, **Single**, **Double** |
| **UInt16** | **UInt32**, **Int32**, **UInt64**, **Int64**, **Single**, **Double** |
| **Int16** | **Int32**, **Int64**, **Single**, **Double** |
| **UInt32** | **UInt64**, **Int64**, **Single**, **Double** |
| **Int32** | **Int64**, **Single**, **Double** |
| **UInt64** | **Single**, **Double** |
| **Int64** | **Single**, **Double** |
| **Single** | **Double** |
| Nonreference | By reference |

There are two more **BindingFlags** that you can use to fine–tune the **DefaultBinder**'s behavior. These are described in Table 20–5.

Table 20–5: BindingFlags Used with DefaultBinder

| Symbol | Value | Description |
|---|---|---|
| **ExactBinding** | 0x010000 | The binder will look for a member whose parameters match the types of the arguments passed. This flag can be used only with the **DefaultBinder** type because a custom binder would be implemented to select the appropriate member. Note that binders are free to ignore this flag. In fact, when the **DefaultBinder** type doesn't find a match, it doesn't fail; it considers a match where the passed parameter can be coerced to a compatible type without loss of precision. |
| **OptionalParamBinding** | 0x040000 | |

| | | |
|---|---|---|
| | | The binder will consider any member whose count of parameters matches the number of arguments passed. This flag is useful when there are members whose parameters have default values and for methods that take a variable number of arguments. Only **Type**'s **InvokeMember** method honors this flag. |

**InvokeMember**'s last parameter, **culture**, is also used for binding. However, the **DefaultBinder** type completely ignores this parameter. If you define your own binder, you could use the culture parameter to help with argument type conversions. For example, the caller could pass a **String** argument with a value of "1,23". The binder could examine this string, parse it using the specified **culture**, and convert the argument's type to a **Single** (if the culture is "de–DE") or continue to consider the argument a **String** (if the culture is "en–US").

At this point, I've gone through all **InvokeMember**'s parameters related to binding. The one parameter I haven't discussed yet is **target**. This parameter is a reference to the object whose method you want to call. If you want to call a **Type**'s static method, you should pass **null** for this parameter.

The **InvokeMember** method is a very powerful method. It allows you to call a method (as I've been discussing), construct an instance of a type (basically by calling a constructor method), and get or set a field. You tell **InvokeMember** which of these actions you want to perform by specifying one of the **BindingFlags** in Table 20–6.

Table 20–6: BindingFlags Used with InvokeMember

| Symbol | Value | Description |
|---|---|---|
| **InvokeMethod** | 0x0100 | Tells **InvokeMember** to call a method |
| **CreateInstance** | 0x0200 | Tells **InvokeMember** to create a new object and call its constructor |
| **GetField** | 0x0400 | Tells **InvokeMember** to get a field's values |
| **SetField** | 0x0800 | Tells **InvokeMember** to set a field's value |
| **GetProperty** | 0x1000 | Tells **InvokeMember** to call a property's get accessor method |
| **SetProperty** | 0x2000 | Tells **InvokeMember** to call a property's set accessor method |

For the most part, the flags in Table 20–6 are mutually exclusive—you must pick one and only one when calling **InvokeMember**. However, you can specify both **GetField** and **GetProperty**, in which case **InvokeMember** searches for a matching field first and then for a matching property if it doesn't find a matching field. Likewise, **SetField** and **SetProperty** can both be specified and are matched the same way. The binder uses these flags to narrow down the set of possible matches. If you specify the **BindingFlags.CreateInstance** flag, the binder knows that it can select only a constructor method.

**Important** With what I've told you so far, it would seem that reflection makes it easy to bind to a nonpublic member and invoke the member, allowing application code a way to access private members that a compiler would normally prohibit the code from accessing. However, reflection uses code access security to ensure that its power isn't abused or

exploited.

When you call a method to bind to a member, the method first checks to see whether the member you're trying to bind to would be visible to you at compile time. If it would be, the bind is successful. If the member wouldn't normally be accessible to you, the method demands the **System.Security.Permissions.ReflectionPermission** permission, checking to see whether the **System.Security.Permissions.ReflectionPermissionFlags**'s **TypeInformation** bit is set. If this flag is set, the method will bind to the member. If the demand fails, a **System.Security.SecurityException** exception is thrown.

When you call a method to invoke a member, the member performs the same kind of check that it would when binding to a member. But this time, it checks whether the **ReflectionPermission** has **ReflectionPermission Flag**'s **MemberAccess** bit set. If the bit is set, the member is invoked; otherwise, a **SecurityException** exception is thrown.

## Bind Once, Invoke Multiple Times

**Type**'s **InvokeMember** method gives you access to all a type's members. However, you should be aware that every time you call **InvokeMember**, it must bind to a particular member and then invoke it. Having the binder select the right member each time you want to invoke a member is time–consuming, and if you do it a lot, your application's performance will suffer. So if you plan on accessing a member frequently, you're better off binding to the desired member once and then accessing that member as often as you want.

You bind to a member (without invoking it) by calling one of the following **Type**'s methods: **GetFields**, **GetConstructors**, **GetMethods**, **GetProperties**, **GetEvents** methods, or any similar method. All these methods return references to objects whose type offers methods to access the specific member directly. Table 20–7 summarizes the types and what methods you call to access the member.

Table 20–7: Types Used to Bind to a Member

| Type | Member Description |
|---|---|
| **FieldInfo** | Call **GetValue** to get a field's value. Call **SetValue** to set a field's value. |
| **ConstructorInfo** | Call **Invoke** to construct an instance of the type. |
| **MethodInfo** | Call **Invoke** to call a method of the type. |
| **PropertyInfo** | Call **GetValue** to call a property's get accessor method. Call **SetValue** to call a property's set accessor method. |
| **EventInfo** | Call **AddEventHandler** to call an event's add accessor method. Call **RemoveEventHandler** to call an event's remove accessor method. |

The **PropertyInfo** type represents a property's metadata information only (as discussed in Chapter 10); that is, **PropertyInfo** offers **CanRead**, **CanWrite**, and **PropertyType** read–only properties. These properties indicate whether a property is readable or writeable and what data type the property is. **PropertyInfo** has a **GetAccessors** method that returns an array of **MethodInfo** elements: one for the get accessor method (if it exists), and one for the set accessor method (if it exists). Of more value are **PropertyInfo**'s **GetGetMethod** and **GetSetMethod** methods, each of

which returns just one **MethodInfo** object. **PropertyInfo**'s **GetValue** and **SetValue** methods exist for convenience; internally, they get the appropriate **MethodInfo** and call it.

The **EventInfo** type represents an event's metadata information only (as discussed in Chapter 11). The **EventInfo** type offers an **EventHandlerType** read–only property that returns the **Type** of the event's underlying delegate. The **EventInfo** also has **GetAddMethod** and **GetRemoveMethod** methods, which return the appropriate **MethodInfo**. **EventInfo**'s **AddEventHandler** and **RemoveEventHandler** methods exist for convenience; internally, they get the appropriate **MethodInfo** and call it.

When you call one of the methods listed in the right column of Table 20–7, you're not binding to a member; you're just invoking the member. You can call any of these methods multiple times, and because binding isn't necessary, the performance will be pretty good.

You might notice that **ConstructorInfo**'s **Invoke**, **MethodInfo**'s **Invoke**, and **PropertyInfo**'s **GetValue** and **SetValue** methods offer overloaded versions that take a reference to a **Binder**–derived object and some **BindingFlags**. This would lead you to believe that these methods bind to a member. However, they don't.

When calling any of these methods, the **Binder**–derived object is used to perform type conversions such as converting an **Int32** argument to an **Int64** so that the already selected method can be called. As for the **BindingFlags** parameter, the only flag that can be passed here is **BindingFlags.SuppressChangeType**. Like the **ExactBinding** flag, binders are free to ignore this flag. However, **DefaultBinder** doesn't ignore this flag. When **DefaultBinder** sees this flag, it won't convert any arguments. If you use this flag and the arguments passed don't match the arguments expected by the method, an **ArgumentException** exception is thrown.

Usually when you use the **BindingFlags.ExactBinding** flag to bind to a member you'll specify the **BindingFlags.SuppressChangeType** flag to invoke the member. If you don't use these two flags in tandem, it's unlikely that invoking the member will be successful unless the arguments you pass happen to be exactly what the method expects. By the way, if you call **MemberInfo**'s **InvokeMethod** to bind and invoke a member, you'll probably want to specify both or neither of the two binding flags.

The following sample application demonstrates the various ways to use reflection to access a type's members. The code shows how to use **Type**'s **InvokeMember** to both bind and invoke a member. It also shows how to bind to a member, invoking it later.

```
// Comment out the following line to test,
// bind, and invoke as separate steps.
#define BindAndInvokeTogether

using System;
using System.Reflection;
using System.Threading;

// This class is used to demonstrate reflection. It has a
// a field, a constructor, a method, a property, and an event.
class SomeType {
   Int32 someField;
   public SomeType(ref Int32 x) { x *= 2; }
   public override String ToString() { return someField.ToString(); }
   public Int32 SomeProp {
      get { return someField; }
      set {
         if (value < 1)
```

```
            throw new ArgumentOutOfRangeException(
               "value", value, "value must be > 0");
         someField = value;
      }
   }
   public event ThreadStart SomeEvent;
   private void NoCompilerWarnings() {
      SomeEvent.ToString();
   }
}

class App {
   static void Main() {
      Type t = typeof(SomeType);
      BindingFlags bf = BindingFlags.DeclaredOnly |
         BindingFlags.Public | BindingFlags.NonPublic |
         BindingFlags.Instance;

      #if BindAndInvokeTogether

      // Construct an instance of the Type.
      Object[] args = new Object[] { 12 };  // Constructor arguments
      Console.WriteLine("x before constructor called: " + args[0]);
      Object obj = t.InvokeMember(null,
         bf | BindingFlags.CreateInstance, null, null, args);
      Console.WriteLine("Type: " + obj.GetType().ToString());
      Console.WriteLine("x after constructor returns: " + args[0]);

      // Read and write to a field.
      t.InvokeMember("someField",
         bf | BindingFlags.SetField, null, obj, new Object[] { 5 });
      Int32 v = (Int32) t.InvokeMember("someField",
         bf | BindingFlags.GetField, null, obj, null);
      Console.WriteLine("someField: " + v);

      // Call a method.
      String s = (String) t.InvokeMember("ToString",
         bf | BindingFlags.InvokeMethod, null, obj, null);
      Console.WriteLine("ToString: " + s);

      // Read and write a property.
      try {
         t.InvokeMember("SomeProp",
            bf | BindingFlags.SetProperty, null, obj, new Object[] { 0 });
      }
      catch (TargetInvocationException e) {
         if (e.InnerException.GetType() !=
            typeof(ArgumentOutOfRangeException))
            throw;
         Console.WriteLine("Property set catch.");
      }
      t.InvokeMember("SomeProp",
         bf | BindingFlags.SetProperty, null, obj, new Object[] { 2 });
      v = (Int32) t.InvokeMember("SomeProp",
         bf | BindingFlags.GetProperty, null, obj, null);
      Console.WriteLine("SomeProp: " + v);

      // NOTE: InvokeMember doesn't support events.

      #else

      // Construct an instance.
      ConstructorInfo ctor = t.GetConstructor(
```

```
         new Type[] { Type.GetType("System.Int32&") });
      Object[] args = new Object[] { 12 };  // Constructor arguments
      Console.WriteLine("x before constructor called: " + args[0]);
      Object obj = ctor.Invoke(args);
      Console.WriteLine("Type: " + obj.GetType().ToString());
      Console.WriteLine("x after constructor returns: " + args[0]);

      // Read and write to a field.
      FieldInfo fi = obj.GetType().GetField("someField", bf);
      fi.SetValue(obj, 33);
      Console.WriteLine("someField: " + fi.GetValue(obj));

      // Call a method.
      MethodInfo mi = obj.GetType().GetMethod("ToString", bf);
      String s = (String) mi.Invoke(obj, null);
      Console.WriteLine("ToString: " + s);

      // Read and write a property.
      PropertyInfo pi =
         obj.GetType().GetProperty("SomeProp", typeof(Int32));
      foreach (MethodInfo m in pi.GetAccessors())
         Console.WriteLine(m);
      try {
         pi.SetValue(obj, 0, null);
      }
      catch (TargetInvocationException e) {
         if (e.InnerException.GetType() !=
            typeof(ArgumentOutOfRangeException))
            throw;
         Console.WriteLine("Property set catch.");
      }
      pi.SetValue(obj, 2, null);
      Console.WriteLine("SomeProp: " + pi.GetValue(obj, null));

      // Add and remove a delegate from the event.
      EventInfo ei = obj.GetType().GetEvent("SomeEvent", bf);
      Console.WriteLine("AddMethod: "        + ei.GetAddMethod());
      Console.WriteLine("RemoveMethod: "     + ei.GetRemoveMethod());
      Console.WriteLine("EventHandlerType: " + ei.EventHandlerType);

      ThreadStart ts = new ThreadStart(Main);
      ei.AddEventHandler(obj, ts);
      ei.RemoveEventHandler(obj, ts);

      #endif
   }
}
```

If you build and run this code with **BindAndInvokeTogether** defined, you'll see the following output:

```
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 5
ToString: 5
Property set catch.
SomeProp: 2
```

Notice that **SomeType**'s constructor takes an **Int32** reference as its only parameter. The previous code shows how to call this constructor and how to examine the modified **Int32** value after the constructor returns.

If you build and run the previous code without **BindAndInvokeTogether** defined, you'll see the following output:

```
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 33
ToString: 33
Void set_SomeProp(Int32)
Int32 get_SomeProp()
Property set catch.
SomeProp: 2
AddMethod: Void add_SomeEvent(System.Threading.ThreadStart)
RemoveMethod: Void remove_SomeEvent(System.Threading.ThreadStart)
EventHandlerType: System.Threading.ThreadStart
```

## Reflecting Over a Type's Interfaces

To obtain the set of interfaces that a type inherits, you can call **Type**'s **FindInterfaces**, **GetInterface**, or **GetInterfaces** method. All these methods return **Type** objects that represent an interface. Note that these methods only check interfaces that the type directly inherits; these methods won't return interfaces that an interface inherits. To determine an interface's base interface, you must call one of these methods again using the interface type and so on to walk the interface inheritance tree.

Determining which members of a type implement a particular interface is a little complicated because multiple interface definitions can all define the same method. For example, the **IBookRetailer** and **IMusicRetailer** interfaces might both define a method named **Purchase**. To get the **MethodInfo** objects for a specific interface, you call **Type**'s **GetInterfaceMap** instance method. This method returns an instance of a **System.Reflection.InterfaceMapping** (a value type). The **InterfaceMapping** type defines the four public fields listed in Table 20–8.

Table 20–8: Public Fields Defined by the InterfaceMapping Type

| Field Name | Data Type | Description |
|---|---|---|
| **TargetType** | **Type** | This is the type that was used to call **GetInterfaceMapping**. |
| **InterfaceType** | **Type** | This is the type of the interface passed to **GetInterfaceMapping**. |
| **InterfaceMethods** | **MethodInfo[]** | An array in which each element exposes information about an interface's method. |
| **TargetMethods** | **MethodInfo[]** | An array in which each element exposes information about the method that the type defines to implement the corresponding interface's method. |

The **InterfaceMethods** and **TargetMethods** arrays run parallel to each other; that is, **InterfaceMethods[0]** identifies a **MethodInfo** object that reflects information about the member as defined in the interface. **TargetMethods[0]** identifies a **MethodInfo** object that reflects information about the interface's member as defined by the **Type**.

```
using System;
using System.Reflection;

// Define two interfaces for testing.
public interface IBookRetailer : IDisposable {
   void Purchase();
   void ApplyDiscount();
}
public interface IMusicRetailer {
   void Purchase();
}

// This class implements two interfaces defined by this
// assembly and one interface defined by another assembly.
class MyRetailer : IBookRetailer, IMusicRetailer {
   public void Purchase() { }
   public void Dispose()  { }
   void IBookRetailer.Purchase()  { }
   public void ApplyDiscount() {}
   void IMusicRetailer.Purchase() { }
}

class App {
   static void Main() {
      // Find the interfaces implemented by MyRetailer where
      // the interface is defined in your own assembly. This
      // is accomplished using a delegate to a filter method
      // that you create and pass to FindInterfaces.
      Type t = typeof(MyRetailer);
      Type[] interfaces = t.FindInterfaces(
         new TypeFilter(App.TypeFilter),
         Assembly.GetCallingAssembly().GetName());
      Console.WriteLine("MyRetailer implements the following " +
         "interfaces (defined in this assembly):");

      // Show information about each interface.
      foreach (Type i in interfaces) {
         Console.WriteLine("\nInterface: " + i);

         // Get the type methods that map to the interface's methods.
         InterfaceMapping map = t.GetInterfaceMap(i);

         for (Int32 m = 0; m < map.InterfaceMethods.Length; m++) {
            // Display the interface method name and which type
            // method implements the interface method.
            Console.WriteLine("   {0} is implemented by {1}",
               map.InterfaceMethods[m], map.TargetMethods[m]);
         }
      }
   }

   // This filter delegate method takes a type and an object, performs
   // some check, and returns true if the type is to be included
   // in the array of returned types.
   static Boolean TypeFilter(Type t, Object filterCriteria) {
      // Return true if the interface is defined in the same
      // assembly identified by filterCriteria.
      return t.Assembly.GetName().ToString() ==
         filterCriteria.ToString();
   }
}
```

## Reflection Performance

In general, using reflection to invoke a method or access a field or property is slow, for several reasons:

- Binding causes many string comparisons to be performed while looking for the desired member.
- Passing arguments requires that an array be constructed and that the array's elements be initialized. Internally, invoking a method requires that the arguments be extracted from the array and placed on the stack.
- The CLR must check that the parameters being passed to a method are of the correct number and type.
- The CLR ensures that the caller has the proper security permission to access the member.

For all these reasons, it's best to avoid using reflection to access a member. If you're writing an application that will dynamically locate and construct types, you should take one of the following approaches:

- Have the types derive from a base type that is known at compile time. At run time, construct an instance of the type, place the reference in a variable that is of the base type (casting if your language requires it), and call virtual methods defined by the base type.
- Have the type implement an interface that is known at compile time. At run time, construct an instance of the type, place the reference in a variable that is of the interface type (casting if your language requires it), and call the methods defined by the interface. I prefer this technique over the base type technique because the base type technique doesn't allow the developer to choose the base type that works best in a particular situation.
- Have the type implement a method whose name and prototype match a delegate that is known at compile time. At run time, construct an instance of the type and then construct an instance of the delegate type using the object and the name of the method. Then call the method via the delegate as you desire. This technique is the most work of the three and quickly becomes a lot more work if you need to call more than one of the type's methods. Also, calling a method via a delegate is slower than calling a type's method or an interface method directly.

When you use any of these three techniques, I strongly suggest that the base type, interface, or delegate type be defined in its own assembly. This will reduce versioning issues. For more information about how to do this, see the section "Designing an Application That Supports Plug–in Components" on page 331 of Chapter 15.

# List of Figures

Chapter 1: The Architecture of the .NET Framework Development Platform

Figure 1–1 Compiling source code into managed modules
Figure 1–2 Combining managed modules into assemblies
Figure 1–3 Loading and initializing the CLR
Figure 1–4 Calling a method for the first time
Figure 1–5 Calling a method for the second time
Figure 1–6 Languages offer a subset of the CLR/CTS and a superset of the CLS (but not necessarily the same superset)
Figure 1–7 ILDasm showing Test type's fields and methods (obtained from metadata)
Figure 1–8 ILDasm showing MgdCApp.exe assembly's metadata

Chapter 2: Building, Packaging, Deploying, and Administering Applications and Types

Figure 2–1 A multifile assembly consisting of two managed modules, one with a manifest
Figure 2–2 Add Reference dialog box in Visual Studio .NET
Figure 2–3 A multifile assembly consisting of three managed modules, one with a manifest
Figure 2–4 Version tab of the JeffTypes.dll Properties dialog box
Figure 2–5 Resource editor in Visual Studio .NET
Figure 2–6 Applications node of the Microsoft .NET Framework Configuration tool
Figure 2–7 Configuring an application using the Microsoft .NET Framework Configuration tool

Chapter 3: Shared Assemblies

Figure 3–1 Signing an assembly
Figure 3–2 Using Explorer's shell extension to see the assemblies installed into the GAC
Figure 3–3 General tab of the System Properties dialog box
Figure 3–4 Version tab of the System Properties dialog box
Figure 3–5 An application that requires different versions of the Calculus.dll assembly in order to run
Figure 3–6 Flowchart showing how the CLR uses metadata to locate the proper assembly file that defines a type, given IL code that refers to a method or type
Figure 3–7 General tab of the System.Drawing Properties dialog box
Figure 3–8 Binding Policy tab of the System.Drawing Properties dialog box
Figure 3–9 Codebases tab of the System.Drawing Properties dialog box
Figure 3–10 .NET Application Configuration tool showing all applications that have had assembly load information recorded at one time or another
Figure 3–11 .NET Application Configuration tool showing the dates when loaded assemblies differed

Chapter 4: Type Fundamentals

Figure 4–1 Requirements section showing namespace and assembly information for a type

Chapter 5: Primitive, Reference, and Value Types

Figure 5–1 Memory layout differences between reference and value types

Chapter 7: Type Members and Their Accessibility

Figure 7–1: ILDasm.exe output showing metadata from preceding code

Chapter 9: Methods

Figure 9–1 The IL code for SomeType's constructor method
Figure 9–2 The IL code for SomeType's type constructor method

Chapter 11: Events

Figure 11–1 Architecting an application to use events

Chapter 12: Working with Text

Figure 12–1: StringSorting results
Figure 12–2: Installing East Asian Language files using the Regional And Language Options Control Panel dialog box
Figure 12–3: Result of GetTextElementEnumerator
Figure 12–4: Result of ParseCombiningCharacters
Figure 12–5: Numeric value formatted correctly to represent Vietnamese currency
Figure 12–6: Numeric value formatted to represent a culture–neutral currency

Chapter 14: Arrays

Figure 14–1 Arrays of value and reference types in the managed heap

Chapter 17: Delegates

Figure 17–1 ILDasm.exe showing the metadata produced by the compiler for the delegate
Figure 17–2 ILDasm.exe proves that the compiler emitted a call to the Set.Feedback delegate type's Invoke method
Figure 17–3 Internal representation of delegate chains

Chapter 18: Exceptions

Figure 18–1 PerfMon.exe showing the .NET CLR exception counters
Figure 18–2 Microsoft PowerPoint showing its Unhandled Exception dialog box
Figure 18–3 An unhandled exception in a window procedure causes Windows Forms to display this dialog box.
Figure 18–4 Visual Studio .NET Exceptions dialog box showing the different kinds of exceptions
Figure 18–5 Visual Studio .NET Exceptions dialog box showing CLR exceptions by namespace
Figure 18–6 Visual Studio .NET Exceptions dialog box showing CLR exceptions defined in the System namespace
Figure 18–7 Making Visual Studio .NET aware of your own exception type
Figure 18–8 Attaching Visual Studio .NET's debugger to a process
Figure 18–9 Selecting the kind of code I want to debug for my project

Chapter 19: Automatic Memory Management (Garbage Collection)

Figure 19–1 Newly initialized managed heap with three objects constructed in it
Figure 19–2 Managed heap before a collection

Figure 19−3 Managed heap after a collection
Figure 19−4 The managed heap showing pointers in its finalization list
Figure 19−5 The managed heap showing pointers that moved from the finalization list to the freachable queue
Figure 19−6 Status of managed heap after second garbage collection
Figure 19−7 A newly initialized heap containing some objects; all in generation 0. No collections have occurred yet.
Figure 19−8 After one collection: generation 0 survivors are promoted to generation 1; generation 0 is empty.
Figure 19−9 New objects are allocated in generation 0; generation 1 has some garbage.
Figure 19−10 After two collections: generation 0 survivors are promoted to generation 1 (growing the size of generation 1); generation 0 is empty.
Figure 19−11 New objects are allocated in generation 0; generation 1 has more garbage.
Figure 19−12 After three collections: generation 0 survivors are promoted to generation 1 (growing the size of generation 1 again); generation 0 is empty.
Figure 19−13 New objects are allocated in generation 0; generation 1 has more garbage.
Figure 19−14 After four collections: generation 1 survivors are promoted to generation 2, generation 0 survivors are promoted to generation 1, and generation 0 is empty.
Figure 19−15 Configuring an application to use the concurrent garbage collector using the Microsoft .NET Framework Configuration administrative tool
Figure 19−16 PerfMon.exe showing the .NET CLR memory counters

Chapter 20: CLR Hosting, AppDomains, and Reflection

Figure 20−1 A single Windows process hosting the CLR and two AppDomains
Figure 20−2 Hierarchy of the reflection types
Figure 20−3 Types an application uses to walk reflection’s object model

# List of Tables

Chapter 1: The Architecture of the .NET Framework Development Platform

Table 1–1: Parts of a Managed Module
Table 1–2: Some General FCL Namespaces
Table 1–3: Some Application–Specific FCL Namespaces
Table 1–4: Test Type's Fields and Methods (obtained from metadata)

Chapter 2: Building, Packaging, Deploying, and Administering Applications and Types

Table 2–1: Common Definition Metadata Tables
Table 2–2: Common Reference Metadata Tables
Table 2–3: Manifest Metadata Tables
Table 2–4: Version Resource Fields and Their Corresponding AL.exe Switches and Custom Attributes
Table 2–5: Format of Version Numbers
Table 2–6: Examples of Assembly Culture Tags

Chapter 3: Shared Assemblies

Table 3–1: How Weakly and Strongly Named Assemblies Can Be Deployed

Chapter 4: Type Fundamentals

Table 4–1 Public Methods of System.Object
Table 4–2 Protected Methods of System.Object
Table 4–4 Type–Safety Quiz

Chapter 5: Primitive, Reference, and Value Types

Table 5–1: FCL Types with Corresponding C# Primitives

Chapter 7: Type Members and Their Accessibility

Table 7–1: Accessibility Modifiers for Types, Fields, or Methods
Table 7–2: Predefined Attributes for Types
Table 7–3: Predefined Attributes for Fields
Table 7–4: Predefined Attributes for Methods

Chapter 9: Methods

Table 9–1: C# Operators and Their CLS–Compliant Method Names

Chapter 12: Working with Text

Table 12–1: Methods for Comparing Strings
Table 12–2: Methods for Examining String Characters
Table 12–3: Methods for Copying Strings
Table 12–4: StringBuilder's Members
Table 12–5: Bit Symbols Defined by the NumberStyles Type
Table 12–6: Symbols for NumberStyles's Bit Combinations

Table 12–7: Bit Symbols Defined by the DateTimeStyles Type
Table 12–8: Properties of Encoding–Derived Classes
Table 12–9: Methods of the Encoding–Derived Classes

Chapter 14: Arrays

Table 14–1: Members of System.Array

Chapter 16: Custom Attributes

Table 16–1: System.Attribute's Methods That Reflect over Metadata Looking for Instances of CLS–Compliant Custom Attributes

Chapter 17: Delegates

Table 17–1: MulticastDelegate's Significant Private Fields

Chapter 18: Exceptions

Table 18–1: Properties of the System.Exception Type
Table 18–2: Possible Values of DbgJITDebugLaunchSetting

Chapter 19: Automatic Memory Management (Garbage Collection)

Table 19–1: Sample of a JIT CompilerÐProduced Table Showing Mapping of Native Code Offsets to a Method's Roots

Chapter 20: CLR Hosting, AppDomains, and Reflection

Table 20–1: AppDomain Events
Table 20–2: Search Symbols Defined by the BindingFlags Enumerated Type
Table 20–3: Properties and Methods Common to All MemberInfo–Derived Types
Table 20–4: Conversions That DefaultBinder Supports
Table 20–5: BindingFlags Used with DefaultBinder
Table 20–6: BindingFlags Used with InvokeMember
Table 20–7: Types Used to Bind to a Member
Table 20–8: Public Fields Defined by the InterfaceMapping Type

List of Sidebars

Chapter 1: The Architecture of the .NET Framework Development Platform

IL and Protecting Your Intellectual Property
Standardizing the .NET Framework
Is Your Code Safe?

Chapter 2: Building, Packaging, Deploying, and Administering Applications and Types

Probing for Assembly Files

Chapter 3: Shared Assemblies

The .NET Framework Configuration Tool

Chapter 4: Type Fundamentals

How Namespaces and Assemblies Relate

Chapter 5: Primitive, Reference, and Value Types

How the CLR Controls the Layout of a Type's Fields

Chapter 9: Methods

Jeff's Opinion About Microsoft's Operator Method Name Rules

Chapter 10: Properties

Selecting the Primary Parameterful Property

Chapter 12: Working with Text

Japanese Characters

Chapter 15: Interfaces

Be Careful with Explicit Interface Method Implementations

Chapter 18: Exceptions

Implied Assumptions Developers Almost Never Think About