

CLR via C#

Fourth Edition

Developer Reference



Jeffrey Richter

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

Copyright © 2012 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 Control Number: 2012951989
ISBN: 978-0-7356-6745-7

Printed and bound in the United States of America.

Third Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspininput@microsoft.com. Please tell us what you think of this book at <http://www.microsoft.com/learning/booksurvey>.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

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

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Devon Musgrave

Developmental Editor: Devon Musgrave

Project Editor: Carol Dillingham

Editorial Production: Online Training Solutions, Inc.

Technical Reviewer: Christophe Nasarre; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Copieditor: Candace Sinclair

Indexer: Jan Bednarczuk

Cover: Twist Creative • Seattle

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

Aidan (age 9) and Grant (age 5), you both have been an inspiration to me and have taught me to play and have fun. Watching the two of you grow up has been so rewarding and enjoyable for me. I am lucky to be able to partake in your lives. I love and appreciate you more than you could ever know.

Contents at a Glance

<i>Foreword</i>	<i>xxi</i>
<i>Introduction</i>	<i>xxiii</i>

<hr/> PART I CLR BASICS <hr/>		
CHAPTER 1	The CLR's Execution Model	3
CHAPTER 2	Building, Packaging, Deploying, and Administering Applications and Types	33
CHAPTER 3	Shared Assemblies and Strongly Named Assemblies	65
<hr/> PART II DESIGNING TYPES <hr/>		
CHAPTER 4	Type Fundamentals	91
CHAPTER 5	Primitive, Reference, and Value Types	111
CHAPTER 6	Type and Member Basics	151
CHAPTER 7	Constants and Fields	175
CHAPTER 8	Methods	181
CHAPTER 9	Parameters	209
CHAPTER 10	Properties	227
CHAPTER 11	Events	249
CHAPTER 12	Generics	265
CHAPTER 13	Interfaces	295
<hr/> PART III ESSENTIAL TYPES <hr/>		
CHAPTER 14	Chars, Strings, and Working with Text	317
CHAPTER 15	Enumerated Types and Bit Flags	361
CHAPTER 16	Arrays	373
CHAPTER 17	Delegates	391
CHAPTER 18	Custom Attributes	421
CHAPTER 19	Nullable Value Types	441

PART IV	CORE FACILITIES	
CHAPTER 20	Exceptions and State Management	451
CHAPTER 21	The Managed Heap and Garbage Collection	505
CHAPTER 22	CLR Hosting and AppDomains	553
CHAPTER 23	Assembly Loading and Reflection	583
CHAPTER 24	Runtime Serialization	611
CHAPTER 25	Interoperating with WinRT Components	643
PART V	THREADING	
CHAPTER 26	Thread Basics	669
CHAPTER 27	Compute-Bound Asynchronous Operations	691
CHAPTER 28	I/O-Bound Asynchronous Operations	727
CHAPTER 29	Primitive Thread Synchronization Constructs	757
CHAPTER 30	Hybrid Thread Synchronization Constructs	789
	<i>Index</i>	823

Contents

<i>Foreword</i>	xxi
<i>Introduction</i>	xxiii

PART I CLR BASICS

Chapter 1 The CLR's Execution Model	3
Compiling Source Code into Managed Modules	3
Combining Managed Modules into Assemblies	6
Loading the Common Language Runtime	8
Executing Your Assembly's Code	11
IL and Verification	16
Unsafe Code	17
The Native Code Generator Tool: NGen.exe	19
The Framework Class Library	22
The Common Type System	24
The Common Language Specification	26
Interoperability with Unmanaged Code	30
Chapter 2 Building, Packaging, Deploying, and Administering Applications and Types	33
.NET Framework Deployment Goals	34
Building Types into a Module	35
Response Files	36
A Brief Look at Metadata	38

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Combining Modules to Form an Assembly	45
Adding Assemblies to a Project by Using the Visual Studio IDE	51
Using the Assembly Linker.....	52
Adding Resource Files to an Assembly.....	53
Assembly Version Resource Information	54
Version Numbers	58
Culture	59
Simple Application Deployment (Privately Deployed Assemblies)	60
Simple Administrative Control (Configuration)	62

Chapter 3 Shared Assemblies and Strongly Named Assemblies 65

Two Kinds of Assemblies, Two Kinds of Deployment	66
Giving an Assembly a Strong Name	67
The Global Assembly Cache	72
Building an Assembly That References a Strongly Named Assembly....	74
Strongly Named Assemblies Are Tamper-Resistant	75
Delayed Signing	76
Privately Deploying Strongly Named Assemblies.....	79
How the Runtime Resolves Type References	80
Advanced Administrative Control (Configuration)	83
Publisher Policy Control	86

PART II DESIGNING TYPES

Chapter 4 Type Fundamentals 91

All Types Are Derived from System.Object	91
Casting Between Types	93
Casting with the C# is and as Operators	95
Namespaces and Assemblies	97
How Things Relate at Run Time	101

Chapter 5 Primitive, Reference, and Value Types	111
Programming Language Primitive Types	111
Checked and Unchecked Primitive Type Operations.....	115
Reference Types and Value Types	118
Boxing and Unboxing Value Types	124
Changing Fields in a Boxed Value Type by Using Interfaces (and Why You Shouldn't Do This)	136
Object Equality and Identity	139
Object Hash Codes.....	142
The dynamic Primitive Type	144
Chapter 6 Type and Member Basics	151
The Different Kinds of Type Members	151
Type Visibility.....	154
Friend Assemblies	154
Member Accessibility.....	156
Static Classes	158
Partial Classes, Structures, and Interfaces.....	159
Components, Polymorphism, and Versioning	160
How the CLR Calls Virtual Methods, Properties, and Events	162
Using Type Visibility and Member Accessibility Intelligently.....	166
Dealing with Virtual Methods When Versioning Types.....	169
Chapter 7 Constants and Fields	175
Constants	175
Fields	177
Chapter 8 Methods	181
Instance Constructors and Classes (Reference Types)	181
Instance Constructors and Structures (Value Types).....	184
Type Constructors.....	187

Operator Overload Methods	191
Operators and Programming Language Interoperability.....	193
Conversion Operator Methods.....	195
Extension Methods.....	198
Rules and Guidelines	200
Extending Various Types with Extension Methods	201
The Extension Attribute	203
Partial Methods.....	204
Rules and Guidelines	207
Chapter 9 Parameters	209
Optional and Named Arguments.....	209
Rules and Guidelines	210
The DefaultParameterValue and Optional Attributes.....	212
Implicitly Typed Local Variables.....	212
Passing Parameters by Reference to a Method.....	214
Passing a Variable Number of Arguments to a Method	220
Parameter and Return Type Guidelines.....	223
Const-ness	224
Chapter 10 Properties	227
Parameterless Properties.....	227
Automatically Implemented Properties.....	231
Defining Properties Intelligently	232
Object and Collection Initializers	235
Anonymous Types	237
The System.Tuple Type	240
Parameterful Properties.....	242
The Performance of Calling Property Accessor Methods	247
Property Accessor Accessibility	248
Generic Property Accessor Methods.....	248

Chapter 11 Events **249**

Designing a Type That Exposes an Event	250
Step #1: Define a type that will hold any additional information that should be sent to receivers of the event notification	251
Step #2: Define the event member.....	252
Step #3: Define a method responsible for raising the event to notify registered objects that the event has occurred	253
Step #4: Define a method that translates the input into the desired event	256
How the Compiler Implements an Event	256
Designing a Type That Listens for an Event	258
Explicitly Implementing an Event.....	260

Chapter 12 Generics **265**

Generics in the Framework Class Library	270
Generics Infrastructure.....	271
Open and Closed Types	272
Generic Types and Inheritance	274
Generic Type Identity	275
Code Explosion.....	277
Generic Interfaces.....	277
Generic Delegates.....	278
Delegate and Interface Contra-variant and Covariant Generic Type Arguments.....	279
Generic Methods.....	281
Generic Methods and Type Inference	283
Generics and Other Members	284
Verifiability and Constraints	284
Primary Constraints	287
Secondary Constraints.....	288
Constructor Constraints	289
Other Verifiability Issues.....	290

Chapter 13 Interfaces	295
Class and Interface Inheritance	296
Defining an Interface	296
Inheriting an Interface	298
More About Calling Interface Methods	300
Implicit and Explicit Interface Method Implementations (What's Happening Behind the Scenes).....	301
Generic Interfaces.....	303
Generics and Interface Constraints	305
Implementing Multiple Interfaces That Have the Same Method Name and Signature.....	307
Improving Compile-Time Type Safety with Explicit Interface Method Implementations	308
Be Careful with Explicit Interface Method Implementations	310
Design: Base Class or Interface?	312

PART III ESSENTIAL TYPES

Chapter 14 Chars, Strings, and Working with Text	317
Characters.....	317
The System.String Type	320
Constructing Strings	320
Strings Are Immutable.....	323
Comparing Strings.....	323
String Interning.....	329
String Pooling.....	332
Examining a String's Characters and Text Elements.....	333
Other String Operations	335
Constructing a String Efficiently.....	336
Constructing a StringBuilder Object	336
StringBuilder Members	337

Obtaining a String Representation of an Object: <code>ToString</code>	339
Specific Formats and Cultures	340
Formatting Multiple Objects into a Single String	344
Providing Your Own Custom Formatter	345
Parsing a String to Obtain an Object: <code>Parse</code>	348
Encodings: Converting Between Characters and Bytes	350
Encoding and Decoding Streams of Characters and Bytes	355
Base-64 String Encoding and Decoding	356
Secure Strings	357
Chapter 15 Enumerated Types and Bit Flags	361
Enumerated Types	361
Bit Flags	367
Adding Methods to Enumerated Types	371
Chapter 16 Arrays	373
Initializing Array Elements	376
Casting Arrays	378
All Arrays Are Implicitly Derived from <code>System.Array</code>	380
All Arrays Implicitly Implement <code>IEnumerable</code> , <code>ICollection</code> , and <code>IList</code>	381
Passing and Returning Arrays	382
Creating Non-Zero Lower Bound Arrays	383
Array Internals	384
Unsafe Array Access and Fixed-Size Array	388
Chapter 17 Delegates	391
A First Look at Delegates	391
Using Delegates to Call Back Static Methods	394
Using Delegates to Call Back Instance Methods	395

Demystifying Delegates.....	396
Using Delegates to Call Back Many Methods (Chaining).....	400
C#'s Support for Delegate Chains.....	404
Having More Control over Delegate Chain Invocation.....	404
Enough with the Delegate Definitions Already (Generic Delegates)....	407
C#'s Syntactical Sugar for Delegates.....	408
Syntactical Shortcut #1: No Need to Construct a Delegate Object.....	409
Syntactical Shortcut #2: No Need to Define a Callback Method (Lambda Expressions).....	410
Syntactical Shortcut #3: No Need to Wrap Local Variables in a Class Manually to Pass Them to a Callback Method	413
Delegates and Reflection.....	416
Chapter 18 Custom Attributes	421
Using Custom Attributes	421
Defining Your Own Attribute Class	425
Attribute Constructor and Field/Property Data Types.....	428
Detecting the Use of a Custom Attribute.....	430
Matching Two Attribute Instances Against Each Other	434
Detecting the Use of a Custom Attribute Without Creating Attribute-Derived Objects	437
Conditional Attribute Classes	440
Chapter 19 Nullable Value Types	441
C#'s Support for Nullable Value Types	443
C#'s Null-Coalescing Operator.....	446
The CLR Has Special Support for Nullable Value Types	447
Boxing Nullable Value Types	447
Unboxing Nullable Value Types.....	448
Calling GetType via a Nullable Value Type.....	448
Calling Interface Methods via a Nullable Value Type	448

PART IV CORE FACILITIES

Chapter 20 Exceptions and State Management	451
Defining “Exception”	452
Exception-Handling Mechanics	453
The <code>try</code> Block.....	454
The <code>catch</code> Block	455
The <code>finally</code> Block.....	456
The <code>System.Exception</code> Class.....	460
FCL-Defined Exception Classes.....	463
Throwing an Exception	466
Defining Your Own Exception Class.....	467
Trading Reliability for Productivity	469
Guidelines and Best Practices.....	478
Use <code>finally</code> Blocks Liberally.....	478
Don’t Catch Everything.....	480
Recovering Gracefully from an Exception	481
Backing Out of a Partially Completed Operation When an Unrecoverable Exception Occurs—Maintaining State	482
Hiding an Implementation Detail to Maintain a “Contract”.....	483
Unhandled Exceptions.....	485
Debugging Exceptions.....	490
Exception-Handling Performance Considerations	492
Constrained Execution Regions (CERs)	494
Code Contracts	498
Chapter 21 The Managed Heap and Garbage Collection	505
Managed Heap Basics	505
Allocating Resources from the Managed Heap	506
The Garbage Collection Algorithm.....	507
Garbage Collections and Debugging.....	510

Generations: Improving Performance	513
Garbage Collection Triggers	519
Large Objects	519
Garbage Collection Modes.	520
Forcing Garbage Collections	522
Monitoring Your Application's Memory Usage	524
Working with Types Requiring Special Cleanup	525
Using a Type That Wraps a Native Resource	532
An Interesting Dependency Issue	537
Other GC Features for Use with Native Resources.	538
Finalization Internals	542
Monitoring and Controlling the Lifetime of Objects Manually	545
Chapter 22 CLR Hosting and AppDomains	553
CLR Hosting	554
AppDomains	556
Accessing Objects Across AppDomain Boundaries	559
AppDomain Unloading	570
AppDomain Monitoring	571
AppDomain First-Chance Exception Notifications	573
How Hosts Use AppDomains	574
Executable Applications	574
Microsoft Silverlight Rich Internet Applications	574
Microsoft ASP.NET and XML Web Services Applications	575
Microsoft SQL Server.	575
Your Own Imagination	576
Advanced Host Control	576
Managing the CLR by Using Managed Code	576
Writing a Robust Host Application	577
How a Host Gets Its Thread Back	578

Chapter 23 Assembly Loading and Reflection	583
Assembly Loading	584
Using Reflection to Build a Dynamically Extensible Application	588
Reflection Performance	589
Discovering Types Defined in an Assembly.....	590
What Exactly Is a Type Object?	591
Building a Hierarchy of Exception-Derived Types	593
Constructing an Instance of a Type	594
Designing an Application That Supports Add-Ins.....	596
Using Reflection to Discover a Type's Members	599
Discovering a Type's Members	599
Invoking a Type's Members	603
Using Binding Handles to Reduce Your Process's Memory Consumption	608
Chapter 24 Runtime Serialization	611
Serialization/Deserialization Quick Start.....	613
Making a Type Serializable	617
Controlling Serialization and Deserialization.....	619
How Formatters Serialize Type Instances	623
Controlling the Serialized/Deserialized Data.....	624
How to Define a Type That Implements ISerializable	
When the Base Type Doesn't Implement This Interface	630
Streaming Contexts	631
Serializing a Type As a Different Type and Deserializing an Object As a Different Object.....	633
Serialization Surrogates.....	636
Surrogate Selector Chains	639
Overriding the Assembly and/or Type When Deserializing an Object.....	640

Chapter 25 Interoperating with WinRT Components	643
CLR Projections and WinRT Component Type System Rules.....	645
WinRT Type System Core Concepts	645
Framework Projections.....	649
Calling Asynchronous WinRT APIs from .NET Code	649
Interoperating Between WinRT Streams and .NET Streams.....	654
Passing Blocks of Data Between the CLR and WinRT	656
Defining WinRT Components in C#.....	658

PART V THREADING

Chapter 26 Thread Basics	669
Why Does Windows Support Threads?.....	669
Thread Overhead	670
Stop the Madness.....	674
CPU Trends.....	677
CLR Threads and Windows Threads.....	678
Using a Dedicated Thread to Perform an Asynchronous Compute-Bound Operation	678
Reasons to Use Threads.....	681
Thread Scheduling and Priorities.....	683
Foreground Threads vs. Background Threads.....	688
What Now?.....	689
Chapter 27 Compute-Bound Asynchronous Operations	691
Introducing the CLR's Thread Pool	692
Performing a Simple Compute-Bound Operation	693
Execution Contexts.....	694
Cooperative Cancellation and Timeout	696
Tasks	700
Waiting for a Task to Complete and Getting Its Result.....	702
Canceling a Task.....	704

Starting a New Task Automatically When Another Task Completes	705
A Task May Start Child Tasks	707
Inside a Task	707
Task Factories	709
Task Schedulers.....	711
Parallel's Static For, ForEach, and Invoke Methods.....	713
Parallel Language Integrated Query.....	717
Performing a Periodic Compute-Bound Operation	720
So Many Timers, So Little Time	723
How the Thread Pool Manages Its Threads.....	723
Setting Thread Pool Limits.....	724
How Worker Threads Are Managed.....	724
Chapter 28 I/O-Bound Asynchronous Operations	727
How Windows Performs I/O Operations	727
C#'s Asynchronous Functions.....	732
How the Compiler Transforms an Async Function into a State Machine.....	734
Async Function Extensibility.....	738
Async Functions and Event Handlers.....	741
Async Functions in the Framework Class Library	742
Async Functions and Exception Handling	744
Other Async Function Features	745
Applications and Their Threading Models.....	748
Implementing a Server Asynchronously.....	751
Canceling I/O Operations	751
Some I/O Operations Must Be Done Synchronously	752
FileStream-Specific Issues	753
I/O Request Priorities	754

Chapter 29 Primitive Thread Synchronization Constructs	757
Class Libraries and Thread Safety.....	759
Primitive User-Mode and Kernel-Mode Constructs.....	760
User-Mode Constructs.....	762
Volatile Constructs.....	762
Interlocked Constructs	768
Implementing a Simple Spin Lock.....	773
The Interlocked Anything Pattern.....	776
Kernel-Mode Constructs	778
Event Constructs.....	782
Semaphore Constructs	784
Mutex Constructs.....	785
Chapter 30 Hybrid Thread Synchronization Constructs	789
A Simple Hybrid Lock.....	790
Spinning, Thread Ownership, and Recursion.....	791
Hybrid Constructs in the Framework Class Library.....	793
The ManualResetEventSlim and SemaphoreSlim Classes.....	794
The Monitor Class and Sync Blocks	794
The ReaderWriterLockSlim Class	800
The OneManyLock Class	802
The CountdownEvent Class.....	804
The Barrier Class	805
Thread Synchronization Construct Summary	805
The Famous Double-Check Locking Technique	807
The Condition Variable Pattern	811
Asynchronous Synchronization	814
The Concurrent Collection Classes.....	818
<i>Index</i>	823

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Foreword

Well, here we are again. Who would've thought? Oh, I know—I would've thought! When you sign up for marriage, you really are living *Groundhog Day*. If you haven't seen that movie, watch it, because you will suddenly see why you have to make the same mistakes over and over again. In this case, when Jeff said he wouldn't write another book, I knew it was the empty promise of an addict. Jeff cannot not write another book. Just today, we were discussing another book he is absolutely not going to write (except that there is already a chapter in progress). It is coded in his DNA. A thoroughbred is born to run and Jeff is born to write.

Jeff is as predictable as the seasons. He cannot stay away from the little 0s and 1s locked inside his hard drive. They cannot be ignored. And while the rest of you are all snug in your beds, Jeff's internal alarm starts ringing around 3:00 a.m. (coincidentally, when our four-year-old climbs into bed with us, another pattern I seem to have no control over) some mysterious force compels Jeff's brain to unlock little solutions, big brainstorms, and frightening bugs that control him. It forces him into his office to work them out of his head. The rest of us can roll over and go back to sleep, safe, knowing that Jeff is out there somewhere solving these problems for us—like a cyber-super hero, saving the thread from becoming just another loose end.

But accruing this knowledge just for himself is not enough for Jeff. He feels selfish hoarding his insights in his little space in the universe. So he must broadcast them; he must write them down. They're like radio waves hurling outward wondering if a listener will pick them up. This he does for you, dear reader; a testament to his passion for Microsoft technologies.

This book is actually adding a new layer of wisdom. Jeff is getting older each time he flies around the sun, and with the accumulation of years, he is starting to look back. Thinking about things in a more mature manner, he has rewritten the chapter covering Reflection. Maybe you too will join him as he waxes poetic on this subject. This is a place where we can learn how to have the code ask about the code and really encouraging some deeper insights as to why Reflection works the way it does. Put on your smoking jackets, sink into a leather chair, and spend some time thinking about your own code and its greater purpose in life.

On a more lively note, there is stuff about `async/await` in here. Apparently, this is the progression of the `AsyncEnumerator` my love has been going on about for some time. Whew, I didn't think we would ever move on from that! The thing is, as many times as he has talked about his `AsyncEnumerator`, it hasn't stuck in my mind at all. I decided if

I knew what enumerator meant, maybe this would help me. According to Wikipedia, it is a census taker. Is this chapter about census takers working out of synchronicity? That seems like a waste of taxpayer dollars. Whatever it means in the computer world must be better than that. Jeff worked with the team at The Big M to perfect `async/await`, and now it is here in this book laid out for your reading pleasure; I suggest you read it sequentially.

Another major addition to the book is the one I'm most excited about. It is my expectation of you all to read and internalize this WinRT stuff. This is a nerd word that somehow means: "Make Me Really Cool Apps for Some Awesome Slate Device NOW!" That's right; the new Windows Runtime is all about awesome touch screens. My kids would like some birds that fly into pigs. I would like maybe something with flowers, and definitely you could use it for some educational stuff. Just let your imagination go! Come up with Wonderful Innovative Nifty Really Touchy stuff. Use this chapter for my benefit, please. Otherwise, I may run out of patience with Jeff and his continuous book writing and lock him in a room with knitting needles and no electricity. You programmers decide: write cool apps with WinRT or no new books from Jeff!

In summary, with your continued patronage, Jeff has delivered yet another masterpiece. Our family can return to a more normal state. Which is what, really? I think maybe normal is the book writing state.

Ever patiently awaiting the siren call of yet another book,

Kristin Trace (Jeff's wife)

October 2012



Help! Please save Jeff from the knitting!

Introduction

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

It is October 2012 as I write this text, making it 13 years now that I've worked with the .NET Framework and C#. Over the 13 years, I have built all kinds of applications and, as a consultant to Microsoft, have contributed quite a bit to the .NET Framework itself. As a partner in my own company, Wintellect (<http://Wintellect.com>), I have worked with numerous customers to help them design software, debug software, performance-tune software, and solve issues they have with the .NET Framework. All these experiences have really helped me learn the spots that people have trouble with when trying to be productive with the .NET Framework. I have tried to sprinkle knowledge from these experiences through all the topics presented in this book.

Who This Book Is For

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

The book addresses Microsoft Visual Studio 2012, .NET Framework 4.5, and version 5.0 of the C# programming language. Because Microsoft tries to maintain a large degree of backward compatibility when releasing a new version of these technologies, many of the things I discuss in this book apply to earlier versions as well. All the code samples use the C# programming language as a way to demonstrate the behavior of the various facilities. But, because the CLR is usable by many programming languages, the book's content is still quite applicable for the non-C# programmer.



Note You can download the code shown in the book from Wintellect's website (<http://Wintellect.com/Books>).

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

Acknowledgments

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

For this book revision, I truly had some fantastic people helping me. Several people on the .NET Framework team (many of whom I consider friends) reviewed chapters and participated in stimulating conversations with me. Christophe Nasarre, who I've worked with on several book projects, has done just a phenomenal job of verifying my work and making sure that I'd said everything the best way it could possibly be said. He has truly had a significant impact on the quality of this book. As always, the Microsoft Press team is a pleasure to work with. I'd like to extend a special thank you to Ben Ryan, Devon Musgrave, and Carol Dillingham. Also, thanks to Susie Carr and Candace Sinclair for their editing and production support.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

<http://go.microsoft.com/fwlink/?LinkId=266601>

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

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

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>*.

The Managed Heap and Garbage Collection

In this chapter:

Managed Heap Basics	505
Generations: Improving Performance.....	513
Working with Types Requiring Special Cleanup	525
Monitoring and Controlling the Lifetime of Objects	
Manually.....	545

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 garbage collector in the common language runtime (CLR) works, and I'll explain various performance issues related to it. I'll also discuss how to design applications so that they use memory most efficiently.

Managed Heap Basics

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 memory to 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 (usually accomplished by using C#'s new operator).
2. Initialize the memory to set the initial state of the resource and to make the resource usable. The type's instance 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.
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 problems for programmers that must manually manage their memory; for example, native C++ developers. Programmers responsible for managing their own memory routinely forget to free memory, which causes a memory leak. In addition, these programmers frequently use memory after having released it, causing their program to experience memory corruption resulting in bugs and security holes. Furthermore, these two bugs are worse than most others because you can't predict the consequences or the timing of them. For other bugs, when you see your application misbehaving, you just fix the line of code that is not working.

As long as you are writing verifiably type-safe code (avoiding C#'s `unsafe` keyword), then it is impossible for your application to experience memory corruption. It is still possible for your application to leak memory but it is not the default behavior. Memory leaks typically occur because your application is storing objects in a collection and never removes objects when they are no longer needed.

To simplify things even more, most types that developers use quite regularly do not require Step 4 (tear down the state of the resource to clean up). And so, the managed heap, in addition to abolishing the bugs I mentioned, also provides developers with a simple programming model: allocate and initialize a resource and use it as desired. For most types, there is no need to clean up the resource and the garbage collector will free the memory.

When consuming instances of types that require special cleanup, the programming model remains as simple as I've just described. However, sometimes, you want to clean up a resource as soon as possible, rather than waiting for a GC to kick in. In these classes, you can call one additional method (called `Dispose`) in order to clean up the resource on your schedule. On the other hand, implementing a type that requires special cleanup is quite involved. I describe the details of all this in the "Working with Types Requiring Special Cleanup" section later in this chapter. Typically, types that require special cleanup are those that wrap native resources like files, sockets, or database connections.

Allocating Resources from the Managed Heap

The CLR requires that all objects be allocated from the *managed heap*. When a process is initialized, the CLR allocates a region of address space for the managed heap. The CLR 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 address space region.

As the region fills with non-garbage objects, the CLR allocates more regions and continues to do this until the whole process's address space is full. So, your application's memory is limited by the process's virtual address space. In a 32-bit process, you can allocate close to 1.5 gigabytes (GB) and in a 64-bit process, you can allocate close to 8 terabytes.

C#'s `new` operator causes the CLR to perform the following steps:

1. Calculate the number of bytes required for the type's fields (and all the fields it inherits from its base types).
2. Add the bytes required for an object's overhead. Each object has two overhead fields: a type object pointer and a sync block index. For a 32-bit application, each of these fields requires

32 bits, adding 8 bytes to each object. For a 64-bit application, each field 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 region. If there is enough free space in the managed heap, the object will fit, starting at the address pointed to by `NextObjPtr`, and these bytes are zeroed out. The type's constructor is called (passing `NextObjPtr` for the `this` parameter), and the new operator returns a reference to the object. Just before the reference is returned, `NextObjPtr` is advanced past the object and now points to the address where the next object will be placed in the heap.

Figure 21-1 shows a managed heap consisting of three objects: A, B, and C. If another object were to be allocated, it would be placed where `NextObjPtr` points to (immediately after object C).

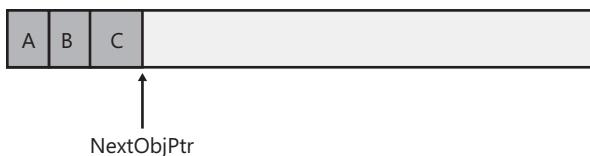


FIGURE 21-1 Newly initialized managed heap with three objects constructed in it.

For the managed heap, allocating an object simply means adding a value to a pointer—this is blazingly fast. 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 before a `BinaryWriter` object is created. Then the application would use the `BinaryWriter` object, which internally uses the `FileStream` object. Because the managed heap allocates these objects next to each other in memory, you get excellent performance when accessing these objects due to locality of reference. Specifically, this means that your process's working set is small, which means your application runs fast with less memory. It's also likely that the objects your code is accessing can all reside in the CPU's cache. The result is that 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 that would force slower access to RAM.

So far, it sounds like the managed heap provides excellent performance characteristics. However, what I have just described is assuming that memory is infinite and that the CLR can always allocate new objects at the end. However, memory is not infinite and so the CLR employs a technique known as garbage collection (GC) to “delete” objects in the heap that your application no longer requires access to.

The Garbage Collection Algorithm

When an application calls the `new` operator to create an object, there might not be enough address space left in the region to allocate the object. If insufficient space exists, then the CLR performs a GC.



Important What I've just said is an oversimplification. In reality, a GC occurs when generation 0 is full. I'll explain generations later in this chapter. Until then, it's easiest for you to think that a garbage collection occurs when the heap is full.

For managing the lifetime of objects, some systems use a reference counting algorithm. In fact, Microsoft's own Component Object Model (COM) uses reference counting. With a reference counting system, each object on the heap maintains an internal field indicating how many "parts" of the program are currently using that object. As each "part" gets to a place in the code where it no longer requires access to an object, it decrements that object's count field. When the count field reaches 0, the object deletes itself from memory. The big problem with many reference counting systems is that they do not handle circular references well. For example, in a GUI application, a window will hold a reference to a child UI element. And the child UI element will hold a reference to its parent window. These references prevent the two objects' counters from reaching 0, so both objects will never be deleted even if the application itself no longer has a need for the window.

Due to this problem with reference counting garbage collector algorithms, the CLR uses a referencing tracking algorithm instead. The reference tracking algorithm cares only about reference type variables, because only these variables can refer to an object on the heap; value type variables contain the value type instance directly. Reference type variables can be used in many contexts: static and instance fields within a class or a method's arguments or local variables. We refer to all reference type variables as *roots*.

When the CLR starts a GC, the CLR first suspends all threads in the process. This prevents threads from accessing objects and changing their state while the CLR examines them. Then, the CLR performs what is called the *marking* phase of the GC. First, it walks through all the objects in the heap setting a bit (contained in the sync block index field) to 0. This indicates that all objects should be deleted. Then, the CLR looks at all active roots to see which objects they refer to. This is what makes the CLR's GC a reference tracking GC. If a root contains null, the CLR ignores the root and moves on to examine the next root.

Any root referring to an object on the heap causes the CLR to mark that object. Marking an object means that the CLR sets the bit in the object's sync block index to 1. When an object is marked, the CLR examines the roots inside that object and marks the objects they refer to. If the CLR is about to mark an already-marked object, then it does not examine the object's fields again. This prevents an infinite loop from occurring in the case where you have a circular reference.

Figure 21-2 shows a heap containing several objects. In this example, the application roots refer directly to objects A, C, D, and F. All of these objects are marked. When marking object D, the garbage collector notices that this object contains a field that refers to object H, causing object H to be marked as well. The marking phase continues until all the application roots have been examined.

Once complete, the heap contains some marked and some unmarked objects. The marked objects must survive the collection because there is at least one root that refers to the object; we say that the object is reachable because application code can reach (or access) the object by way of the variable that still refers to it. Unmarked objects are unreachable because there is no root existing in the application that would allow for the object to ever be accessed again.

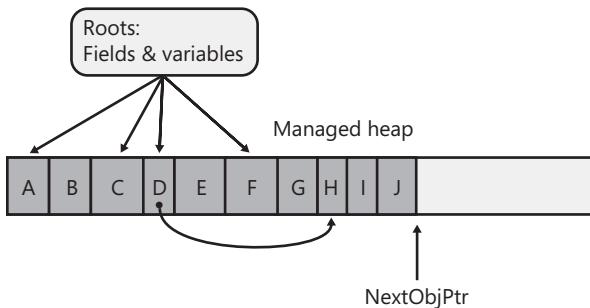


FIGURE 21-2 Managed heap before a collection.

Now that the CLR knows which objects must survive and which objects can be deleted, it begins the GC's compacting phase. During the compacting phase, the CLR shifts the memory consumed by the marked objects down in the heap, compacting all the surviving objects together so that they are contiguous in memory. This serves many benefits. First, all the surviving objects will be next to each other in memory; this restores locality of reference reducing your application's working set size, thereby improving the performance of accessing these objects in the future. Second, the free space is all contiguous as well, so this region of address space can be freed, allowing other things to use it. Finally, compaction means that there are no address space fragmentation issues with the managed heap as is known to happen with native heaps.¹

When compacting memory, the CLR is moving objects around in memory. This is a problem because any root that referred to a surviving object now refers to where that object was in memory; not where the object has been relocated to. When the application's threads eventually get resumed, they would access the old memory locations and corrupt memory. Clearly, this can't be allowed and so, as part of the compacting phase, the CLR subtracts from each root the number of bytes that the object it referred to was shifted down in memory. This ensures that every root refers to the same object it did before; it's just that the object is at a different location in memory.

After the heap memory is compacted, the managed heap's `NextObjPtr` pointer is set to point to a location just after the last surviving object. This is where the next allocated object will be placed in memory. Figure 21-3 shows the managed heap after the compaction phase. After the compaction phase is complete, the CLR resumes all the application's threads and they continue to access the objects as if the GC never happened at all.

¹ Objects in the large object heap (discussed later in this chapter) do not get compacted, and therefore address space fragmentation is possible with the large object heap.

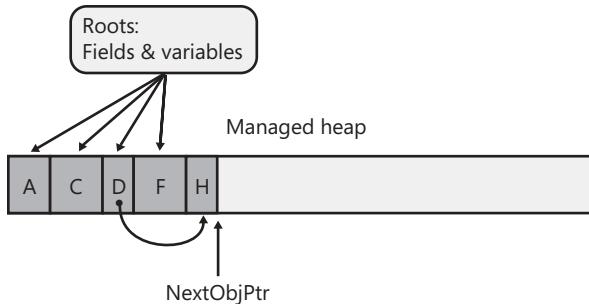


FIGURE 21-3 Managed heap after a collection.

If the CLR is unable to reclaim any memory after a GC and if there is no address space left in the processes to allocate a new GC segment, then there is just no more memory available for this process. In this case, the new operator that attempted to allocate more memory ends up throwing an `OutOfMemoryException`. Your application can catch this and recover from it but most applications do not attempt to do so; instead, the exception becomes an unhandled exception, Windows terminates the process, and then Windows reclaims all the memory that the process was using.

As a programmer, 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 will be collected at some point. Second, it's not possible to corrupt memory by accessing an object that was freed because references can only refer to living objects, because this is what keeps the objects alive anyway.



Important A static field keeps whatever object it refers to forever or until the AppDomain that the types are loaded into is unloaded. A common way to leak memory is to have a static field refer to a collection object and then to keep adding items to the collection object. The static field keeps the collection object alive and the collection object keeps all its items alive. For this reason, it is best to avoid static fields whenever possible.

Garbage Collections and Debugging

As soon as a root goes out of scope, the object it refers to is unreachable and subject to having its memory reclaimed by a GC; objects aren't guaranteed to live throughout a method's lifetime. This can have an interesting impact on your application. For example, examine the following code.

```
using System;
using System.Threading;

public static class Program {
    public static void Main() {
        // Create a Timer object that knows to call our TimerCallback
        // method once every 2000 milliseconds.
        Timer t = new Timer(TimerCallback, null, 0, 2000);
```

```

    // Wait for the user to hit <Enter>.
    Console.ReadLine();
}

private static void TimerCallback(Object o) {
    // Display the date/time when this method got called.
    Console.WriteLine("In TimerCallback: " + DateTime.Now);

    // Force a garbage collection to occur for this demo.
    GC.Collect();
}
}

```

Compile this code from the command prompt without using any special compiler switches. When you run the resulting executable file, you'll see that the `TimerCallback` method is called just once!

From examining the preceding code, you'd think that the `TimerCallback` method would get called once every 2,000 milliseconds. After all, a `Timer` object is created, and the variable `t` refers to this object. As long as the timer object exists, the timer should keep firing. But you'll notice in the `TimerCallback` method that I force a garbage collection to occur by calling `GC.Collect()`.

When the collection starts, it first assumes that all objects in the heap are unreachable (garbage); this includes the `Timer` object. Then, the collector examines the application's roots and sees that `Main` doesn't use the `t` variable after the initial assignment to it. Therefore, the application has no variable referring to the `Timer` object, and the garbage collection reclaims the memory for it; this stops the timer and explains why the `TimerCallback` method is called just once.

Let's say that you're using a debugger to step through `Main`, and a garbage collection just happens to occur just after `t` is assigned the address of the new `Timer` object. Then, let's say that you try to view the object that `t` refers to by using the debugger's Quick Watch window. What do you think will happen? The debugger can't show you the object because it was just garbage collected. This behavior would be considered very unexpected and undesirable by most developers, so Microsoft has come up with a solution.

When you compile your assembly by using the C# compiler's `/debug` switch, the compiler applies a `System.Diagnostics.DebuggableAttribute` with its `DebuggingModes`' `DisableOptimizations` flag set into the resulting assembly. At run time, when compiling a method, the JIT compiler sees this flag set, and artificially extends the lifetime of all roots to the end of the method. For my example, the JIT compiler tricks itself into believing that the `t` variable in `Main` must live until the end of the method. So, if a garbage collection were to occur, the garbage collector now thinks that `t` is still a root and that the `Timer` object that `t` refers to will continue to be reachable. The `Timer` object will survive the collection, and the `TimerCallback` method will get called repeatedly until `Console.ReadLine` returns and `Main` exits.

To see this, just recompile the program from a command prompt, but this time, specify the C# compiler's `/debug` switch. When you run the resulting executable file, you'll now see that the `TimerCallback` method is called repeatedly! Note, the C# compiler's `/optimize+` compiler switch turns optimizations back on, so this compiler switch should not be specified when performing this experiment.

The JIT compiler does this to help you with JIT debugging. You may now start your application normally (without a debugger), and if the method is called, the JIT compiler will artificially extend the lifetime of the variables to the end of the method. Later, if you decide to attach a debugger to the process, you can put a breakpoint in a previously compiled method and examine the root variables.

So now you know how to build a program that works in a debug build but doesn't work correctly when you make a release build! Because no developer wants a program that works only when debugging it, there should be something we can do to the program so that it works all of the time regardless of the type of build.

You could try modifying the Main method to the following.

```
public static void Main() {
    // Create a Timer object that knows to call our TimerCallback
    // method once every 2000 milliseconds.
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Wait for the user to hit <Enter>.
    Console.ReadLine();

    // Refer to t after ReadLine (this gets optimized away)
    t = null;
}
```

However, if you compile this (without the /debug+ switch) and run the resulting executable file, you'll see that the TimerCallback method is still called just once. The problem here is that the JIT compiler is an optimizing compiler, and setting a local variable or parameter variable to null is the same as not referencing the variable at all. In other words, the JIT compiler optimizes the t = null; line out of the code completely, and therefore, the program still does not work as we desire. The correct way to modify the Main method is as follows.

```
public static void Main() {
    // Create a Timer object that knows to call our TimerCallback
    // method once every 2000 milliseconds.
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Wait for the user to hit <Enter>.
    Console.ReadLine();

    // Refer to t after ReadLine (t will survive GCs until Dispose returns)
    t.Dispose();
}
```

Now, if you compile this code (without the /debug+ switch) and run the resulting executable file, you'll see that the TimerCallback method is called multiple times, and the program is fixed. What's happening here is that the object t is required to stay alive so that the Dispose instance method can be called on it. (The value in t needs to be passed as the this argument to Dispose.) It's ironic: by explicitly indicating where you want the timer to be disposed, it must remain alive up to that point.



Note Please don't read this whole discussion and then worry about your own objects being garbage collected prematurely. I use the `Timer` class in this discussion because it has special behavior that no other class exhibits. The "problem/feature" of `Timer` is that the existence of a `Timer` object in the heap causes something else to happen: A thread pool thread invokes a method periodically. No other type exhibits this behavior. For example, the existence of a `String` object in memory doesn't cause anything else to happen; the string just sits there. So, I use `Timer` to show how roots work and how object-lifetime works as related to the debugger, but the discussion is not really about how to keep objects alive. All non-`Timer` objects will live as needed by the application automatically.

Generations: Improving Performance

The CLR's GC is a *generational garbage collector* (also known as an *ephemeral garbage collector*, although I don't use the latter term in this book). A generational GC makes the following assumptions about your code:

- 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 21-4 shows a newly started application with five objects allocated (A through E). After a while, objects C and E become unreachable.

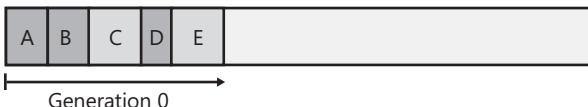


FIGURE 21-4 A newly initialized heap containing some objects, all in generation 0. No collections have occurred yet.

When the CLR initializes, it selects a budget size (in kilobytes) for generation 0. So if allocating a new object causes generation 0 to surpass its budget, a garbage collection must start. Let's say that objects A through E fill all of generation 0. 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, causing it to be 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 21-5.

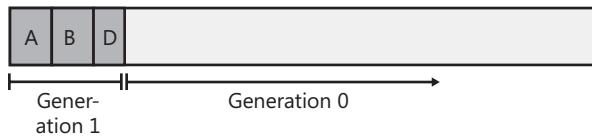


FIGURE 21-5 After one collection, generation 0 survivors are promoted to generation 1; generation 0 is empty.

After a garbage collection, generation 0 contains no objects. As always, new objects will be allocated in generation 0. Figure 21-6 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.

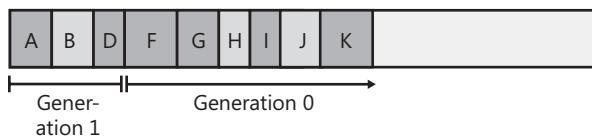


FIGURE 21-6 New objects are allocated in generation 0; generation 1 has some garbage.

Now let's say that attempting to allocate object L would put generation 0 over its budget. Because generation 0 has reached its budget, 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 budget for generation 0. Well, it also selects a budget for generation 1.

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 its budget, so the garbage collector examines only the objects in generation 0. Look again at the assumptions that the 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's field refers to a new object. To ensure that the updated fields of 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 object in generation 0.²



Note Microsoft's performance tests show that it takes less than 1 millisecond 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. As a result, it wouldn't be able to reclaim much memory. So it is likely that collecting generation 1 is a waste of time. If any garbage happens to be in generation 1, it just stays there. The heap now looks like Figure 21-7.

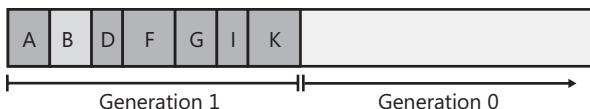


FIGURE 21-7 After two collections, generation 0 survivors are promoted to generation 1 (growing the size of generation 1); generation 0 is empty.

As you can see, all of 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 21-8.

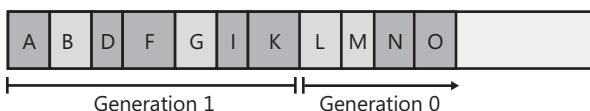


FIGURE 21-8 New objects are allocated in generation 0; generation 1 has more garbage.

Let's say that allocating object P causes generation 0 to exceed its budget, causing a garbage collection to occur. Because the memory occupied by all of the objects in generation 1 is less than its

² For the curious, here are some more details about this. When the JIT compiler produces native code that modifies a reference field inside an object, the native code includes a call to a `write_barrier` method. This `write_barrier` method checks whether the object whose field is being modified is in generation 1 or 2 and if it is, the `write_barrier` code sets a bit in what is called the card table. The card table has 1 bit for every 128-byte range of data in the heap. When the next GC starts, it scans the card table to know which objects in generations 1 and 2 have had their fields changed because the last GC. If any of these modified objects refer to an object in generation 0, then the generation 0 objects survive the collection. After the GC, the card table is reset to all zeroes. The `write_barrier` code causes a slight performance hit when writing to a reference field in an object (as opposed to a local variable or static field) and that performance hit is slightly worse if that object is in generation 1 or 2.

budget, 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 21-9.

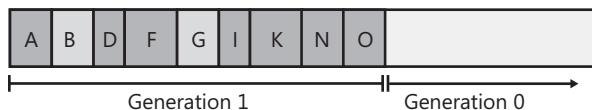


FIGURE 21-9 After three collections, generation 0 survivors are promoted to generation 1 (growing the size of generation 1 again); generation 0 is empty.

In Figure 21-9, you see that generation 1 keeps growing slowly. In fact, let's say that generation 1 has now grown to the point in which all of the objects in it occupy its full budget. 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 budget. The heap now looks like Figure 21-10.

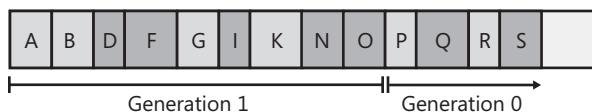


FIGURE 21-10 New objects are allocated in generation 0; generation 1 has more garbage.

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 budget 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 of the objects in generation 1 and generation 0. After both generations have been garbage collected, the heap now looks like Figure 21-11.

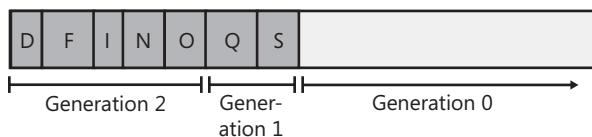


FIGURE 21-11 After four collections: generation 1 survivors are promoted to generation 2, generation 0 survivors are promoted to generation 1, and generation 0 is empty.

As 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 two or more times. There might have been several collections, but the objects in generation 1 are examined only when generation 1 reaches its budget, 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 3.³ When the CLR initializes, it selects budgets for all three generations. However, 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 budget of generation 0. 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 will be small. In fact, if all objects in generation 0 are garbage, a garbage collection doesn't have to compact any memory; it can simply set `NextObjPtr` back to the beginning of generation 0, and then the garbage collection is performed. Wow, this is a fast way to reclaim memory!



Note The garbage collector works extremely well for applications with threads that sit idle at the top of their stack most of the time. Then, when the thread has something to do, it wakes up, creates a bunch of short-lived objects, returns, and then goes back to sleep. Many applications follow this architecture. For example, GUI applications tend to have the GUI thread sitting in a message loop most of its life. Occasionally, the user generates some input (like a touch, mouse, or keyboard event), the thread wakes up, processes the input and returns back to the message pump. Most objects created to process the input are probably garbage now.

Similarly, server applications tend to have thread pool threads sitting in the pool waiting for client requests to come in. When a client request comes in, new objects are created to perform work on behalf of the client request. When the result is sent back to the client, the thread returns to the thread pool and all the objects it created are garbage now.

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 budget. Now, fewer collections will occur, but when they do, a lot more memory should be reclaimed. By the way, if insufficient memory has been reclaimed after a collection, the garbage collector will perform a full collection before throwing an `OutOfMemoryException`.

Throughout this discussion, I've been talking about how the garbage collector dynamically modifies generation 0's budget after every collection. But the garbage collector also modifies the budgets of generation 1 and generation 2 by using similar heuristics. When these generations are garbage collected, the garbage collector again sees how much memory is reclaimed and how many objects survived. Based on the garbage collector's findings, it might grow or shrink the thresholds of these

³ The `System.GC` class's static `MaxGeneration` method returns 2.

generations as well to improve the overall performance of the application. The end result is that the garbage collector fine-tunes itself automatically based on the memory load required by your application—this is very cool!

The following GCNotification class raises an event whenever a generation 0 or generation 2 collection occurs. With these events, you could have the computer beep whenever a collection occurs or you could calculate how much time passes between collections, how much memory is allocated between collections, and more. With this class, you could easily instrument your application to get a better understanding of how your application uses memory.

```
public static class GCNotification {
    private static Action<Int32> s_gcDone = null; // The event's field

    public static event Action<Int32> GCDone {
        add {
            // If there were no registered delegates before, start reporting notifications now
            if (s_gcDone == null) { new GenObject(0); new GenObject(2); }
            s_gcDone += value;
        }
        remove { s_gcDone -= value; }
    }

    private sealed class GenObject {
        private Int32 m_generation;
        public GenObject(Int32 generation) { m_generation = generation; }
        ~GenObject() { // This is the Finalize method
            // If this object is in the generation we want (or higher),
            // notify the delegates that a GC just completed
            if (GC.GetGeneration(this) >= m_generation) {
                Action<Int32> temp = Volatile.Read(ref s_gcDone);
                if (temp != null) temp(m_generation);
            }
        }

        // Keep reporting notifications if there is at least one delegate registered,
        // the AppDomain isn't unloading, and the process isn't shutting down
        if ((s_gcDone != null)
            && !AppDomain.CurrentDomain.IsFinalizingForUnload()
            && !Environment.HasShutdownStarted) {
            // For Gen 0, create a new object; for Gen 2, resurrect the object
            // & let the GC call Finalize again the next time Gen 2 is GC'd
            if (m_generation == 0) new GenObject(0);
            else GC.ReRegisterForFinalize(this);
        } else { /* Let the objects go away */ }
    }
}
```

Garbage Collection Triggers

As you know, the CLR triggers a GC when it detects that generation 0 has filled its budget. This is the most common trigger of a GC; however, there are additional GC triggers as listed here:

- **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 discuss this more in the "Forcing Garbage Collections" section later in this chapter.
- **Windows is reporting low memory conditions** The CLR internally uses the Win32 CreateMemoryResourceNotification and QueryMemoryResourceNotification functions to monitor system memory overall. If Windows reports low memory, the CLR will force a garbage collection in an effort to free up dead objects to reduce the size of a process's working set.
- **The CLR is unloading an AppDomain** When an AppDomain unloads, the CLR considers nothing in the AppDomain to be a root, and a garbage collection consisting of all generations is performed. I'll discuss AppDomains in Chapter 22, "CLR Hosting and AppDomains."
- **The CLR is shutting down** The CLR shuts down when a process terminates normally (as opposed to an external shutdown via Task Manager, for example). During this shutdown, the CLR considers nothing in the process to be a root; it allows objects a chance to clean up but the CLR does not attempt to compact or free memory because the whole process is terminating, and Windows will reclaim all of the processes' memory.

Large Objects

There is one more performance improvement you might want to be aware of. The CLR considers each single object to be either a small object or a large object. So far, in this chapter, I've been focusing on small objects. Today, a large object is 85,000 bytes or more in size.⁴ The CLR treats large objects slightly differently than how it treats small objects:

- Large objects are not allocated within the same address space as small objects; they are allocated elsewhere within the process' address space.
- Today, the GC doesn't compact large objects because of the time it would require to move them in memory. For this reason, address space fragmentation can occur between large objects within the process leading to an OutOfMemoryException being thrown. In a future version of the CLR, large objects may participate in compaction.

⁴ In the future, the CLR could change the number of bytes required to consider an object to be a large object. Do not count 85,000 being a constant.

- Large objects are immediately considered to be part of generation 2; they are never in generation 0 or 1. 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, hurting performance. Usually large objects are large strings (like XML or JSON) or byte arrays that you use for I/O operations, such as reading bytes from a file or network into a buffer so you can process it.

For the most part, large objects are transparent to you; you can simply ignore that they exist and that they get special treatment until you run into some unexplained situation in your program (like why you're getting address space fragmentation).

Garbage Collection Modes

When the CLR starts, it selects a GC mode, and this mode cannot change during the lifetime of the process. There are two basic GC modes:

- **Workstation** This mode fine-tunes the garbage collector for client-side applications. It is optimized to provide for low-latency GCs in order to minimize the time an application's threads are suspended so as not to frustrate the end user. In this mode, the GC assumes that other applications are running on the machine and does not hog CPU resources.
- **Server** This mode fine-tunes the garbage collector for server-side applications. It is optimized for throughput and resource utilization. In this mode, the GC assumes no other applications (client or server) are running on the machine, and it assumes that all the CPUs on the machine are available to assist with completing the GC. This GC mode causes the managed heap to be split into several sections, one per CPU. When a garbage collection is initiated, the garbage collector dedicates one special thread per CPU; each thread collects its own section in parallel with the other threads. Parallel collections work well for server applications in which the worker threads tend to exhibit uniform behavior. This feature requires the application to be running on a computer with multiple CPUs so that the threads can truly be working simultaneously to attain a performance improvement.

By default, applications run with the Workstation GC mode. A server application (such as ASP.NET or Microsoft SQL Server) that hosts the CLR can request the CLR to load the Server GC. However, if the server application is running on a uniprocessor machine, then the CLR will always use Workstation GC mode. A stand-alone application can tell the CLR to use the Server GC mode by creating a configuration file (as discussed in Chapter 2, "Building, Packaging, Deploying, and Administering Applications and Types," and Chapter 3, "Shared Assemblies and Strongly Named Assemblies") that contains a `gcServer` element for the application. Here's an example of a configuration file.

```
<configuration>
  <runtime>
    <gcServer enabled="true"/>
  </runtime>
</configuration>
```

When an application is running, it can ask the CLR if it is running in the Server GC mode by querying the `GCSettings` class's `IsServerGC` read-only Boolean property.

```
using System;
using System.Runtime; // GCSettings is in this namespace

public static class Program {
    public static void Main() {
        Console.WriteLine("Application is running with server GC=" + GCSettings.IsServerGC);
    }
}
```

In addition to the two modes, the GC can run in two sub-modes: concurrent (the default) or non-concurrent. In concurrent mode, the GC has an additional background thread that marks objects concurrently while the application runs. When a thread allocates an object that pushes generation 0 over its budget, the GC first suspends all threads and then determines which generations to collect. If the garbage collector needs to collect generation 0 or 1, it proceeds as normal. However, if generation 2 needs collecting, the size of generation 0 will be increased beyond its budget to allocate the new object, and then the application's threads are resumed.

While the application's threads are running, the garbage collector has a normal priority background thread that finds unreachable objects. Once found, the garbage collector suspends all threads again and decides whether to compact memory. If the garbage collector decides to compact memory, 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 set 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 with the non-concurrent garbage collector.

You can tell the CLR not to use the concurrent collector by creating a configuration file for the application that contains a `gcConcurrent` element. Here's an example of a configuration file.

```
<configuration>
    <runtime>
        <gcConcurrent enabled="false"/>
    </runtime>
</configuration>
```

The GC mode is configured for a process and it cannot change while the process runs. However, your application can have some control over the garbage collection by using the `GCSettings` class's `GCLatencyMode` property. This read/write property can be set to any of the values in the `GCLatencyMode` enumerated type, as shown in Table 21-1.

The `LowLatency` mode requires some additional explanation. Typically, you would set this mode, perform a short-term, time-sensitive operation, and then set the mode back to either `Batch` or `Interactive`. While the mode is set to `LowLatency`, the GC will really avoid doing any generation 2

collections because these could take a long time. Of course, if you call `GC.Collect()`, then generation 2 still gets collected. Also, the GC will perform a generation 2 collection if Windows tells the CLR that system memory is low (see the “Garbage Collection Triggers” section earlier in this chapter).

TABLE 21-1 Symbols Defined by the `GCLatencyMode` Enumerated Type

Symbol Name	Description
<code>Batch</code> (default for the Server GC mode)	Turns off the concurrent GC.
<code>Interactive</code> (default for the Workstation GC mode)	Turns on the concurrent GC.
<code>LowLatency</code>	Use this latency mode during short-term, time-sensitive operations (like drawing animations) where a generation 2 collection might be disruptive.
<code>SustainedLowLatency</code>	Use this latency mode to avoid long GC pauses for the bulk of your application’s execution. This setting prevents all blocking generation 2 collections from occurring as long as memory is available. In fact, users of these applications would prefer to install more RAM in the machine in order to avoid GC pauses. A stock market application that must respond immediately to price changes is an example of this kind of application.

Under `LowLatency` mode, it is more likely that your application could get an `OutOfMemoryException` thrown. Therefore, stay in this mode for as short a time as possible, avoid allocating many objects, avoid allocating large objects, and set the mode back to `Batch` or `Interactive` by using a constrained execution region (CER), as discussed in Chapter 20, “Exceptions and State Management.” Also, remember that the latency mode is a process-wide setting and threads may be running concurrently. These other threads could even change this setting while another thread is using it, so you may want to update some kind of counter (manipulated via `Interlocked` methods) when you have multiple threads manipulating this setting. Here is some code showing how to use the `LowLatency` mode.

```
private static void LowLatencyDemo() {
    GCLatencyMode oldMode = GCSettings.LatencyMode;
    System.Runtime.CompilerServices.RuntimeHelpers.PrepareConstrainedRegions();
    try {
        GCSettings.LatencyMode = GCLatencyMode.LowLatency;
        // Run your code here...
    }
    finally {
        GCSettings.LatencyMode = oldMode;
    }
}
```

Forcing Garbage Collections

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; this property always returns 2.

You can also force the garbage collector to perform a collection by calling `GC` class’s `Collect` method, optionally passing in a generation to collect up to, a `GCCollectionMode`, and a `Boolean`

indicating whether you want to perform a blocking (non-current) or background (concurrent) collection. Here is the signature of the most complex overload of the `Collect` method.

```
void Collect(Int32 generation, GCCollectionMode mode, Boolean blocking);
```

The `GCCollectionMode` type is an enum whose values are described in Table 21-2.

TABLE 21-2 Symbols Defined by the `GCCollectionMode` Enumerated Type

Symbol Name	Description
Default	The same as calling <code>GC.Collect</code> with no flag. Today, this is the same as passing <code>Forced</code> , but this may change in a future version of the CLR.
Forced	Forces a collection to occur immediately for all generations up to and including the specified generation.
Optimized	The garbage collector will only perform a collection if the collection would be productive either by freeing a lot of memory or by reducing fragmentation. If the garbage collection would not be productive, then the call has no effect.

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 budgets based on actual application behavior. However, if you're writing a console user interface (CUI) or GUI application, your application code owns the process and the CLR in that process. For these application types, you *might* want to suggest a garbage collection to occur at certain times using a `GCCollectionMode` of `Optimized`. Normally, modes of `Default` and `Forced` are used for debugging, testing, and looking for memory leaks.

For example, you might consider calling the `Collect` method if some non-recurring event has just occurred that has likely caused a lot of old objects to die. The reason that calling `Collect` in such a circumstance may not be so bad is that the GC's predictions of the future based on the past are not likely to be accurate for non-recurring events. For example, it might make sense for your application to force a full GC of all generations after your application initializes or after the user saves a data file. Because calling `Collect` causes the generation budgets to adjust, do not call `Collect` to try to improve your application's response time; call it to reduce your process's working set.

For some applications (especially server applications that tend to keep a lot of objects in memory), the time required for the GC to do a full collection that includes generation 2 can be excessive. In fact, if the collection takes a very long time to complete, then client requests might time out. To help these kinds of applications, the `GC` class offers a `RegisterForFullGCNotification` method. Using this method and some additional helper methods (`WaitForFullGCAccomplished`, `WaitForFullGCComplete`, and `CancelFullGCNotification`), an application can now be notified when the garbage collector is getting close to performing a full collection. The application can then call `GC.Collect` to force a collection at a more opportune time, or the application could communicate with another server to better load balance the client requests. For more information, examine these methods and the "Garbage Collection Notifications" topic in the Microsoft .NET Framework SDK documentation. Note that you should always call the `WaitForFullGCAccomplished` and `WaitForFullGCComplete` methods in pairs because the CLR handles them as pairs internally.

Monitoring Your Application's Memory Usage

Within a process, there are a few methods that you can call to monitor the garbage collector. Specifically, the GC class offers the following static methods, which you can call to see how many collections have occurred of a specific generation or how much memory is currently being used by objects in the managed heap.

```
Int32 CollectionCount(Int32 generation);  
Int64 GetTotalMemory(Boolean forceFullCollection);
```

To profile a particular code block, I have frequently written code to call these methods before and after the code block and then calculate the difference. This gives me a very good indication of how my code block has affected my process's working set and indicates how many garbage collections occurred while executing the code block. If the numbers are high, I know to spend more time tuning the algorithms in my code block.

You can also see how much memory is being used by individual AppDomains as opposed to the whole process. For more information about this, see the "AppDomain Monitoring" section in Chapter 22.

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 click the + toolbar button, which causes the Add Counters dialog box shown in Figure 21-12 to appear.

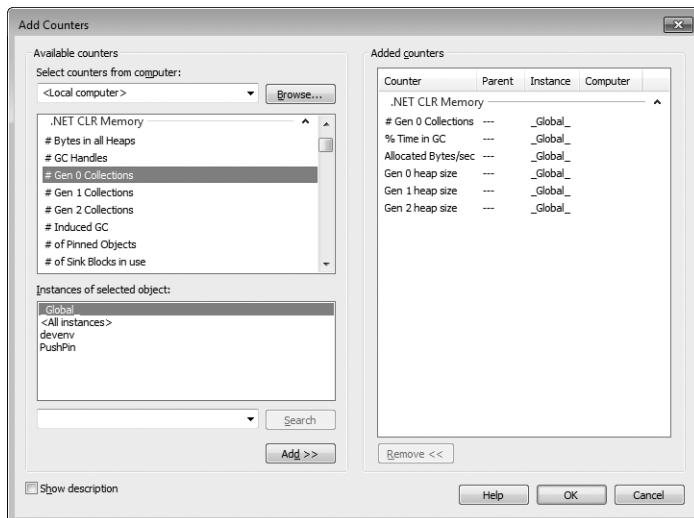


FIGURE 21-12 PerfMon.exe showing the .NET CLR Memory counters.

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, click Add, and then click OK. 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 then select the Show Description check box.

Another great tool for analyzing the memory and performance of your application is PerfView. This tool can collect Event Tracing for Windows (ETW) logs and process them. The best way to acquire this tool is for you to search the web for PerfView. Finally, you should look into using the SOS Debugging Extension (SOS.dll), which can often offer great assistance when debugging memory problems and other CLR problems. For memory-related actions, the SOS Debugging Extension allows you to see how much memory is allocated within the process to the managed heap, displays all objects registered for finalization in the finalization queue, displays the entries in the GCHandle table per AppDomain or for the entire process, shows the roots that are keeping an object alive in the heap, and more.

Working with Types Requiring Special Cleanup

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 need only memory to operate. However, some types require more than just memory to be useful; some types require the use of a native resource in addition to memory.

The `System.IO.FileStream` type, for example, needs to open a file (a native resource) and store 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 (a native resource) and stores its handle, using it when the `Mutex`'s methods are called.

If a type wrapping a native resource gets GC'd, the GC will reclaim the memory used by the object in the managed heap; but the native resource, which the GC doesn't know anything about, will be leaked. This is clearly not desirable, so the CLR offers a mechanism called *finalization*. Finalization allows an object to execute some code after the object has been determined to be garbage but before the object's memory is reclaimed from the managed heap. All types that wrap a native resource—such as a file, network connection, socket, or mutex—support finalization. When the CLR determines that one of these objects is no longer reachable, the object gets to finalize itself, releasing the native resource it wraps, and then, later, the GC will reclaim the object from the managed heap.

`System.Object`, the base class of everything, defines a protected and virtual method called `Finalize`. When the garbage collector determines that an object is garbage, it calls the object's `Finalize` method (if it is overridden). Microsoft's C# team felt that `Finalize` methods were a special kind of method requiring special syntax in the programming language (similar to how C# requires special syntax to define a constructor). So, in C#, you must define a `Finalize` method by placing a tilde symbol (~) in front of the class name, as shown in the following code sample.

```
internal sealed class SomeType {
    // This is the Finalize method
    ~SomeType() {
        // The code here is inside the Finalize method
    }
}
```

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 `protected override` method named `Finalize` into the module's metadata. If you examined the `Finalize` method's IL code, you'd also see that the code inside the method's body is emitted into a `try` block, and that a call to `base.Finalize` is emitted into a `finally` block.



Important If you're familiar with C++, you'll notice that the special syntax 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 calls this method a *destructor*. However, a `Finalize` method doesn't work like a C++ destructor at all, and this has caused a great deal of confusion for developers migrating from one language to another.

The problem is that developers mistakenly believe that using the C# destructor syntax means that the type's objects will be deterministically destructed when they go out of lexical scope, just as they would be in C++. However, the CLR doesn't support deterministic destruction, preventing C# from providing this mechanism.

`Finalize` methods are called at the completion of a garbage collection on objects that the GC has determined to be garbage. This means that the memory for these objects cannot be reclaimed right away because the `Finalize` method might execute code that accesses a field. Because a finalizable object must survive the collection, it gets promoted to another generation, forcing the object to live much longer than it should. This is not ideal in terms of memory consumption and is why you should avoid finalization when possible. To make matters worse, when finalizable objects get promoted, any object referred to by its fields also get promoted because they must continue to live too. So, try to avoid defining finalizable objects with reference type fields.

Furthermore, be aware of the fact that you have no control over when the `Finalize` method will execute. `Finalize` methods run when a garbage collection occurs, which may happen when your application requests more memory. Also, the CLR doesn't make any guarantees as to the order in which `Finalize` methods are called. So, you should avoid writing a `Finalize` method that accesses other objects whose type defines a `Finalize` method; those other objects could have been finalized already. However, it is perfectly OK to access value type instances or reference type objects that do not define a `Finalize` method. You also need to be careful when calling static methods because these methods can internally access objects that have been finalized, causing the behavior of the static method to become unpredictable.

The CLR uses a special, high-priority dedicated thread to call `Finalize` methods to avoid some deadlock scenarios that could occur otherwise.⁵ If a `Finalize` method blocks (for example, enters an infinite loop or waits for an object that is never signaled), this special thread can't call any more `Finalize` methods. This is a very bad situation because the application will never be able to reclaim the memory occupied by the finalizable objects—the application will leak memory as long as it runs.

⁵ A future version of the CLR might use multiple finalizer threads to improve performance.

If a `Finalize` method throws an unhandled exception, then the process terminates; there is no way to catch this exception.

So, as you can see, there are a lot of caveats related to `Finalize` methods and they must be used with caution. Specifically, they are designed for releasing native resources. To simplify working with them, it is highly recommended that developers avoid overriding `Object`'s `Finalize` method; instead, use helper classes that Microsoft now provides in the Framework Class Library (FCL). The helper classes override `Finalize` and add some special CLR magic I'll talk about as we go on. You will then derive your own classes from the helper classes and inherit the CLR magic.

If you are creating a managed type that wraps a native resource, you should first derive a class from a special base class called `System.Runtime.InteropServices.SafeHandle`, which looks like the following (I've added comments in the methods to indicate what they do).

```
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {
    // This is the handle to the native resource
    protected IntPtr handle;

    protected SafeHandle(IntPtr invalidHandleValue, Boolean ownsHandle) {
        this.handle = invalidHandleValue;
        // If ownsHandle is true, then the native resource is closed when
        // this SafeHandle-derived object is collected
    }

    protected void SetHandle(IntPtr handle) {
        this.handle = handle;
    }

    // You can explicitly release the resource by calling Dispose
    // This is the IDisposable interface's Dispose method
    public void Dispose() { Dispose(true); }

    // The default Dispose implementation (shown here) is exactly what you want.
    // Overriding this method is strongly discouraged.
    protected virtual void Dispose(Boolean disposing) {
        // The default implementation ignores the disposing argument.
        // If resource already released, return
        // If ownsHandle is false, return
        // Set flag indicating that this resource has been released
        // Call virtual ReleaseHandle method
        // Call GC.SuppressFinalize(this) to prevent Finalize from being called
        // If ReleaseHandle returned true, return
        // If we get here, fire ReleaseHandleFailed Managed Debugging Assistant (MDA)
    }

    // The default Finalize implementation (shown here) is exactly what you want.
    // Overriding this method is very strongly discouraged.
    ~SafeHandle() { Dispose(false); }

    // A derived class overrides this method to implement the code that releases the resource
    protected abstract Boolean ReleaseHandle();

    public void SetHandleAsInvalid() {
```

```

        // Set flag indicating that this resource has been released
        // Call GC.SuppressFinalize(this) to prevent Finalize from being called
    }

    public Boolean IsClosed {
        get {
            // Returns flag indicating whether resource was released
        }
    }

    public abstract Boolean IsValid {
        // A derived class overrides this property.
        // The implementation should return true if the handle's value doesn't
        // represent a resource (this usually means that the handle is 0 or -1)
        get;
    }

    // These three methods have to do with security and reference counting;
    // I'll talk about them at the end of this section
    public void DangerousAddRef(ref Boolean success) {...}
    public IntPtr DangerousGetHandle() {...}
    public void DangerousRelease() {...}
}

```

The first thing to notice about the `SafeHandle` class is that it is derived from `CriticalFinalizerObject`, which is defined in the `System.Runtime.ConstrainedExecution` namespace. The CLR treats this class and classes derived from it in a very special manner. In particular, the CLR endows this class with three cool features:

- The first time an object of any `CriticalFinalizerObject`-derived type is constructed, the CLR immediately JIT-compiles all of the `Finalize` methods in the inheritance hierarchy. Compiling these methods upon object construction guarantees that the native resource will be released when the object is determined to be garbage. Without this eager compiling of the `Finalize` method, it would be possible to allocate the native resource and use it, but not to get rid of it. Under low memory conditions, the CLR might not be able to find enough memory to compile the `Finalize` method, which would prevent it from executing, causing the native resource to leak. Or the resource might not be freed if the `Finalize` method contained code that referred to a type in another assembly, and the CLR failed to locate this other assembly.
- The CLR calls the `Finalize` method of `CriticalFinalizerObject`-derived types after calling the `Finalize` methods of non-`CriticalFinalizerObject`-derived types. This ensures that managed resource classes that have a `Finalize` method can access `CriticalFinalizerObject`-derived objects within their `Finalize` methods successfully. For example, the `FileStream` class's `Finalize` method can flush data from a memory buffer to an underlying disk with confidence that the disk file has not been closed yet.
- The CLR calls the `Finalize` method of `CriticalFinalizerObject`-derived types if an `AppDomain` is rudely aborted by a host application (such as SQL Server or ASP.NET). This also is part of ensuring that the native resource is released even in a case in which a host application no longer trusts the managed code running inside of it.

The second thing to notice about `SafeHandle` is that the class is abstract; it is expected that another class will be derived from `SafeHandle`, and this class will provide a constructor that invokes the protected constructor, the abstract method `ReleaseHandle`, and the abstract `IsInvalid` property get accessor method.

Most native resources are manipulated with handles (32-bit values on a 32-bit system and 64-bit values on a 64-bit system). So the `SafeHandle` class defines a protected `IntPtr` field called `handle`. In Windows, most handles are invalid if they have a value of 0 or -1. The `Microsoft.Win32.SafeHandles` namespace contains another helper class called `SafeHandleZeroOrMinusOneIsInvalid`, which looks like this.

```
public abstract class SafeHandleZeroOrMinusOneIsInvalid : SafeHandle {
    protected SafeHandleZeroOrMinusOneIsInvalid(Boolean ownsHandle)
        : base(IntPtr.Zero, ownsHandle) {
    }

    public override Boolean IsInvalid {
        get {
            if (base.handle == IntPtr.Zero) return true;
            if (base.handle == (IntPtr) (-1)) return true;
            return false;
        }
    }
}
```

Again, you'll notice that the `SafeHandleZeroOrMinusOneIsInvalid` class is abstract, and therefore, another class must be derived from this one to override the protected constructor and the abstract method `ReleaseHandle`. The .NET Framework provides just a few public classes derived from `SafeHandleZeroOrMinusOneIsInvalid`, including `SafeFileHandle`, `SafeRegistryHandle`, `SafeWaitHandle`, and `SafeMemoryMappedViewHandle`. Here is what the `SafeFileHandle` class looks like.

```
public sealed class SafeFileHandle : SafeHandleZeroOrMinusOneIsInvalid {
    public SafeFileHandle(IntPtr preexistingHandle, Boolean ownsHandle)
        : base(ownsHandle) {
        base.SetHandle(preexistingHandle);
    }

    protected override Boolean ReleaseHandle() {
        // Tell Windows that we want the native resource closed.
        return Win32Native.CloseHandle(base.handle);
    }
}
```

The `SafeWaitHandle` class is implemented similarly to the `SafeFileHandle` class just shown. The only reason why there are different classes with similar implementations is to achieve type safety; the compiler won't let you use a file handle as an argument to a method that expects a wait handle, and vice versa. The `SafeRegistryHandle` class's `ReleaseHandle` method calls the Win32 `RegCloseKey` function.

It would be nice if the .NET Framework included additional classes that wrap various native resources. For example, one could imagine classes such as `SafeProcessHandle`, `SafeThreadHandle`, `SafeTokenHandle`, `SafeLibraryHandle` (its `ReleaseHandle` method would call the Win32 `FreeLibrary` function), `SafeLocalAllocHandle` (its `ReleaseHandle` method would call the Win32 `LocalFree` function), and so on.

All of the classes just listed (and more) actually do ship with the Framework Class Library (FCL). However, these classes are not publicly exposed; they are all internal to the assemblies that define them. Microsoft didn't expose these classes publicly because they didn't want to document them and do full testing of them. However, if you need any of these classes for your own work, I'd recommend that you use a tool such as `ILDasm.exe` or some IL decompiler tool to extract the code for these classes and integrate that code into your own project's source code. All of these classes are trivial to implement, and writing them yourself from scratch would also be quite easy.

The `SafeHandle`-derived classes are extremely useful because they ensure that the native resource is freed when a GC occurs. In addition to what we've already discussed, `SafeHandle` offers two more capabilities. First, the CLR gives `SafeHandle`-derived types special treatment when used in scenarios in which you are interoperating with native code. For example, let's examine the following code.

```
using System;
using System.Runtime.InteropServices;
using Microsoft.Win32.SafeHandles;

internal static class SomeType {
    [DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]
    // This prototype is not robust
    private static extern IntPtr CreateEventBad(
        IntPtr pSecurityAttributes, Boolean manualReset, Boolean initialState, String name);

    // This prototype is robust
    [DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]
    private static extern SafeWaitHandle CreateEventGood(
        IntPtr pSecurityAttributes, Boolean manualReset, Boolean initialState, String name);

    public static void SomeMethod() {
        IntPtr handle = CreateEventBad(IntPtr.Zero, false, false, null);
        SafeWaitHandle swh = CreateEventGood(IntPtr.Zero, false, false, null);
    }
}
```

You'll notice that the `CreateEventBad` method is prototyped as returning an `IntPtr`, which will return the handle back to managed code; however, interoperating with native code this way is not robust. You see, after `CreateEventBad` is called (which creates the native event resource), it is possible that a `ThreadAbortException` could be thrown prior to the handle being assigned to the `handle` variable. In the rare cases when this would happen, the managed code would leak the native resource. The only way to get the event closed is to terminate the whole process.

The `SafeHandle` class fixes this potential resource leak. Notice that the `CreateEventGood` method is prototyped as returning a `SafeWaitHandle` (instead of an `IntPtr`). When `CreateEventGood` is called, the CLR calls the Win32 `CreateEvent` function. As the `CreateEvent` function returns to managed code, the CLR knows that `SafeWaitHandle` is derived from `SafeHandle`, causing the CLR to automatically construct an instance of the `SafeWaitHandle` class on the managed heap, passing in the handle value returned from `CreateEvent`. The constructing of the `SafeWaitHandle` object and the assignment of the handle happen in native code now, which cannot be interrupted by a `ThreadAbortException`. Now, it is impossible for managed code to leak this native resource. Eventually, the `SafeWaitHandle` object will be garbage collected and its `Finalize` method will be called, ensuring that the resource is released.

One last feature of `SafeHandle`-derived classes is that they prevent someone from trying to exploit a potential security hole. The problem is that one thread could be trying to use a native resource while another thread tries to free the resource. This could manifest itself as a handle-recycling exploit. The `SafeHandle` class prevents this security vulnerability by using reference counting. Internally, the `SafeHandle` class defines a private field that maintains a count. When a `SafeHandle`-derived object is set to a valid handle, the count is set to 1. Whenever a `SafeHandle`-derived object is passed as an argument to a native method, the CLR knows to automatically increment the counter. Likewise, when the native method returns to managed code, the CLR knows to decrement the counter. For example, you would prototype the Win32 `SetEvent` function as follows.

```
[DllImport("Kernel32", ExactSpelling=true)]
private static extern Boolean SetEvent(SafeWaitHandle swh);
```

Now when you call this method passing in a reference to a `SafeWaitHandle` object, the CLR will increment the counter just before the call and decrement the counter just after the call. Of course, the manipulation of the counter is performed in a thread-safe fashion. How does this improve security? Well, if another thread tries to release the native resource wrapped by the `SafeHandle` object, the CLR knows that it cannot actually release it because the resource is being used by a native function. When the native function returns, the counter is decremented to 0, and the resource will be released.

If you are writing or calling code to manipulate a handle as an `IntPtr`, you can access it out of a `SafeHandle` object, but you should manipulate the reference counting explicitly. You accomplish this via `SafeHandle`'s `DangerousAddRef` and `DangerousRelease` methods. You gain access to the raw handle via the `DangerousGetHandle` method.

I would be remiss if I didn't mention that the `System.Runtime.InteropServices` namespace also defines a `CriticalHandle` class. This class works exactly as the `SafeHandle` class in all ways except that it does not offer the reference-counting feature. The `CriticalHandle` class and the classes derived from it sacrifice security for better performance when you use it (because counters don't get manipulated). As does `SafeHandle`, the `CriticalHandle` class has two types derived from it: `CriticalHandleMinusOneIsInvalid` and `CriticalHandleZeroOrMinusOneIsInvalid`. Because Microsoft favors a more secure system over a faster system, the class library includes no types derived from either of these two classes. For your own work, I would recommend that you use `CriticalHandle`-derived types only if performance is an issue. If you can justify reducing security, you can switch to a `CriticalHandle`-derived type.

Using a Type That Wraps a Native Resource

Now that you know how to define a `SafeHandle`-derived class that wraps a native resource, let's take a look at how a developer uses it. Let's start by talking about the 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. When a `FileStream` object is constructed, the Win32 `CreateFile` function is called, the returned handle is saved in a `SafeFileHandle` object, and a reference to this object is maintained via a private field in the `FileStream` object. The `FileStream` class also offers several additional properties (such as `Length`, `Position`, `CanRead`) and methods (such as `Read`, `Write`, `Flush`).

Let's say that 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;

public static class Program {
    public 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 while it is still open. 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 be deleted! If another thread somehow caused a garbage collection to start after the call to `Write` and before the call to `Delete`, the `FileStream`'s `SafeFileHandle` field would have its `Finalize` method called, which would close the file and allow `Delete` to work. The likelihood of this situation is extremely rare, however, and therefore the previous code will fail more than 99 percent of the time.

Classes that allow the consumer to control the lifetime of native resources it wraps implement the `IDisposable` interface, which looks like this.

```
public interface IDisposable {  
    void Dispose();  
}
```



Important If a class defines a field in which the field's type implements the dispose pattern, the class itself should also implement the dispose pattern. The `Dispose` method should dispose of the object referred to by the field. This allows someone using the class to call `Dispose` on it, which in turn releases the resources used by the object itself.

Fortunately, the `FileStream` class implements the `IDisposable` interface and its implementation internally calls `Dispose` on the `FileStream` object's private `SafeFileHandle` field. Now, we can modify our code to explicitly close the file when we want to as opposed to waiting for some GC to happen in the future. Here's the corrected source code.

```
using System;  
using System.IO;  
  
public static class Program {  
    public 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 finished writing to it.  
        fs.Dispose();  
  
        // Delete the temporary file.  
        File.Delete("Temp.dat"); // This always works now.  
    }  
}
```

Now, when `File`'s `Delete` method is called, Windows sees that the file isn't open and successfully deletes it.

Keep in mind that calling `Dispose` is not required to guarantee native resource cleanup. Native resource cleanup will always happen eventually; calling `Dispose` lets you control when that cleanup happens. Also, calling `Dispose` does not delete the managed object from the managed heap. The only way to reclaim memory in the managed heap is for a garbage collection to kick in. This means you can still call methods on the managed object even after you dispose of any native resources it may have been using.

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;

public static class Program {
    public 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 finished writing to it.
        fs.Dispose();

        // Try to write to the file after closing it.
        fs.Write(bytesToWrite, 0, bytesToWrite.Length); // Throws ObjectDisposedException

        // Delete the temporary file.
        File.Delete("Temp.dat");
    }
}
```

Note that no memory corruption occurs here because the memory for the `FileStream` object still exists in the managed heap; 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 `IDisposable` interface, be sure to write code in all of your methods and properties to throw a `System.ObjectDisposedException` if the object has been explicitly cleaned up. A `Dispose` method should never throw an exception; if it's called multiple times, it should just return.



Important In general, I strongly discourage explicitly calling `Dispose` in your code. 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.⁶ When application code calls `Dispose`, 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`, but later, some other code might try to access the object, causing an `ObjectDisposedException` to be thrown. I recommend that you call `Dispose` only at places in your code where you know you must clean up the resource (as in the case of attempting to delete an open file).

Along the same lines, it is possible to have multiple threads call `Dispose` on a single object simultaneously. However, the design guidelines state that `Dispose` does not have to be thread-safe. The reason is because code should be calling `Dispose` only if the code knows for a fact that no other thread is using the object.

The previous code examples show how to explicitly call a type's `Dispose` method. If you decide to call `Dispose` 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;

public static class Program {
    public 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);
        try {
            // Write the bytes to the temporary file.
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }
        finally {
            // Explicitly close the file when finished writing to it.
        }
    }
}
```

⁶ There are many nice features about a garbage collected system: no memory leaks, no memory corruption, no address space fragmentation, and a reduced working set. And now, a new one: synchronization. That's right; you can use the GC as a thread synchronization mechanism. Question: How can you know when all threads are done using an object? Answer: the GC finalizes the object. There is nothing wrong with taking advantage of all the GC features as you architect your software.

```

        if (fs != null) fs.Dispose();
    }

    // Delete the temporary file.
    File.Delete("Temp.dat");
}
}

```

Adding the exception-handling code is the right thing to do, and you must have the diligence to do it. Fortunately, the C# language provides a `using` statement, which offers a simplified syntax that produces code identical to the code just shown. Here's how the preceding code would be rewritten using C#'s `using` statement.

```

using System;
using System.IO;

public static class Program {
    public 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");
    }
}

```

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` and `finally` blocks. Inside the `finally` block, the compiler emits code to cast the object to an `IDisposable` and calls the `Dispose` method. Obviously, the compiler allows the `using` statement to be used 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 use just an already initialized variable. For more information about this topic, refer to the "Using Statements" topic in the C# Programmer's Reference.

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 characters and strings, you can use a `System.IO.StreamWriter`, as is demonstrated in the following code.

```
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
sw.WriteLine("Hi there");

// The following call to Dispose is what you should do.
sw.Dispose();
// NOTE: StreamWriter.Dispose closes the FileStream;
// the FileStream doesn't have to be explicitly closed.
```

Notice that the `StreamWriter`'s constructor takes a reference to a `Stream` object as a parameter, allowing a reference to a `FileStream` object to be passed as an argument. Internally, the `StreamWriter` object saves the `Stream`'s reference. When you write to a `StreamWriter` object, it internally buffers the data in its own memory buffer. When the buffer is full, the `StreamWriter` object writes the data to the `Stream`.

When you're finished writing data via the `StreamWriter` object, you should call `Dispose`. (Because the `StreamWriter` type implements the `IDisposable` interface, you can also use it with C#'s `using` statement.) This causes the `StreamWriter` object to flush its data to the `Stream` object and close the `Stream` object.⁷



Note You don't have to explicitly call `Dispose` on the `FileStream` object because the `StreamWriter` calls it for you. However, if you do call `Dispose` explicitly, the `FileStream` will see that the object has already been cleaned up—the method does nothing and just returns.

What do you think would happen if there were no code to explicitly call `Dispose`? 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 objects are finalized. So if the `FileStream` object were finalized first, it would close the file. Then when the `StreamWriter` object was finalized, it would attempt to write data to the closed file, throwing an exception. If, on the other hand, the `StreamWriter` object were finalized first, the data would be safely written to the file.

⁷ You can override this behavior by calling `StreamWriter`'s constructor that accepts a Boolean `leaveOpen` parameter.

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 `StreamWriter` type does not support finalization, and therefore it never flushes data in its buffer to the underlying `FileStream` object. This means that if you forget to explicitly call `Dispose` on the `StreamWriter` 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 `Dispose`.



Note The .NET Framework offers a feature called Managed Debugging Assistants (MDAs). When an MDA is enabled, the .NET Framework looks for certain common programmer errors and fires a corresponding MDA. In the debugger, it looks like an exception has been thrown. There is an MDA available to detect when a `StreamWriter` object is garbage collected without previously having been explicitly disposed. To enable this MDA in Microsoft Visual Studio, open your project and select the `Debug.Exceptions` menu item. In the Exceptions dialog box, expand the Managed Debugging Assistants node and scroll to the bottom. There you will see the `StreamWriterBufferedDataLost` MDA. Select the Thrown check box to have the Visual Studio debugger stop whenever a `StreamWriter` object's data is lost.

Other GC Features for Use with Native Resources

Sometimes, a native resource consumes a lot of memory, but the managed object wrapping that resource occupies very little memory. The quintessential example of this is the bitmap. A bitmap can occupy several megabytes of native memory, but the managed object is tiny because it contains only an `HBITMAP` (a 4-byte or 8-byte value). From the CLR's perspective, a process could allocate hundreds of bitmaps (using little managed memory) before performing a collection. But if the process is manipulating many bitmaps, the process's memory consumption will grow at a phenomenal rate. To fix this situation, the `GC` class offers the following two static methods.

```
public static void AddMemoryPressure(Int64 bytesAllocated);  
public static void RemoveMemoryPressure(Int64 bytesAllocated);
```

A class that wraps a potentially large native resource should use these methods to give the garbage collector a hint as to how much memory is really being consumed. Internally, the garbage collector monitors this pressure, and when it gets high, a garbage collection is forced.

There are some native resources that are fixed in number. For example, Windows formerly had a restriction that it could create only five device contexts. There had also been a restriction on the number of files that an application could open. Again, from the CLR's perspective, a process could allocate hundreds of objects (that use little memory) before performing a collection. But if the number of these native resources is limited, attempting to use more than are available will typically result in exceptions being thrown.

To fix this situation, the `System.Runtime.InteropServices` namespace offers the `HandleCollector` class.

```
public sealed class HandleCollector {
    public HandleCollector(String name, Int32 initialThreshold);
    public HandleCollector(String name, Int32 initialThreshold, Int32 maximumThreshold);
    public void Add();
    public void Remove();

    public Int32 Count { get; }
    public Int32 InitialThreshold { get; }
    public Int32 MaximumThreshold { get; }
    public String Name { get; }
}
```

A class that wraps a native resource that has a limited quantity available should use an instance of this class to give the garbage collector a hint as to how many instances of the resource are really being consumed. Internally, this class object monitors the count, and when it gets high, a garbage collection is forced.



Note Internally, the `GC.AddMemoryPressure` and `HandleCollector.Add` methods call `GC.Collect`, forcing a garbage collection to start prior to generation 0 reaching its budget. Normally, forcing a garbage collection to start is strongly discouraged, because it usually has an adverse effect on your application's performance. However, classes that call these methods are doing so in an effort to keep limited native resources available for the application. If the native resources run out, the application will fail. For most applications, it is better to work with reduced performance than to not be working at all.

Here is some code that demonstrates the use and effect of the memory pressure methods and the `HandleCollector` class.

```
using System;
using System.Runtime.InteropServices;

public static class Program {
    public static void Main() {
        MemoryPressureDemo(0); // 0 causes infrequent GCs
        MemoryPressureDemo(10 * 1024 * 1024); // 10MB causes frequent GCs

        HandleCollectorDemo();
    }

    private static void MemoryPressureDemo(Int32 size) {
        Console.WriteLine();
        Console.WriteLine("MemoryPressureDemo, size={0}", size);
        // Create a bunch of objects specifying their logical size
        for (Int32 count = 0; count < 15; count++) {
            new BigNativeResource(size);
        }
    }
}
```

```

// For demo purposes, force everything to be cleaned-up
GC.Collect();
}

private sealed class BigNativeResource {
    private readonly Int32 m_size;

    public BigNativeResource(Int32 size) {
        m_size = size;
        // Make the GC think the object is physically bigger
        if (m_size > 0) GC.AddMemoryPressure(m_size);
        Console.WriteLine("BigNativeResource create.");
    }

    ~BigNativeResource() {
        // Make the GC think the object released more memory
        if (m_size > 0) GC.RemoveMemoryPressure(m_size);
        Console.WriteLine("BigNativeResource destroy.");
    }
}

private static void HandleCollectorDemo() {
    Console.WriteLine();
    Console.WriteLine("HandleCollectorDemo");
    for (Int32 count = 0; count < 10; count++) new LimitedResource();

    // For demo purposes, force everything to be cleaned-up
    GC.Collect();
}

private sealed class LimitedResource {
    // Create a HandleCollector telling it that collections should
    // occur when two or more of these objects exist in the heap
    private static readonly HandleCollector s_hc = new HandleCollector("LimitedResource", 2);

    public LimitedResource() {
        // Tell the HandleCollector a LimitedResource has been added to the heap
        s_hc.Add();
        Console.WriteLine("LimitedResource create. Count={0}", s_hc.Count);
    }

    ~LimitedResource() {
        // Tell the HandleCollector a LimitedResource has been removed from the heap
        s_hc.Remove();
        Console.WriteLine("LimitedResource destroy. Count={0}", s_hc.Count);
    }
}
}

```

If you compile and run the preceding code, your output will be similar to the following output.

Finalization Internals

On the surface, finalization seems pretty straightforward: you create an object and its `Finalize` method is called when it is collected. But after 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 21-13 shows a heap containing several objects. Some of these objects are reachable from application roots, and some are not. When objects C, E, F, I, and J were created, the system detected

that these objects' types defined a `Finalize` method and so added references to these objects to the finalization list.

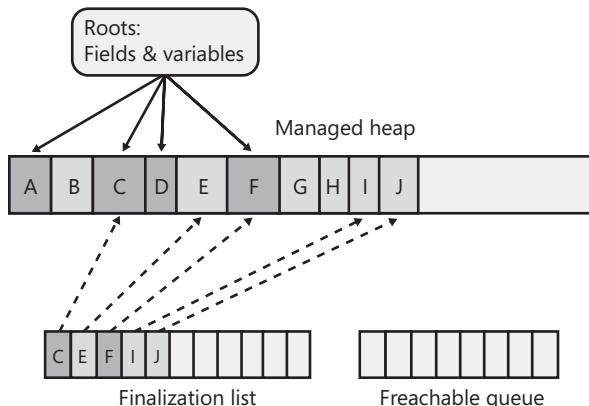


FIGURE 21-13 The managed heap showing pointers to its finalization list.



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 references to these objects. When a reference is found, the reference 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 reference 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 21-14.

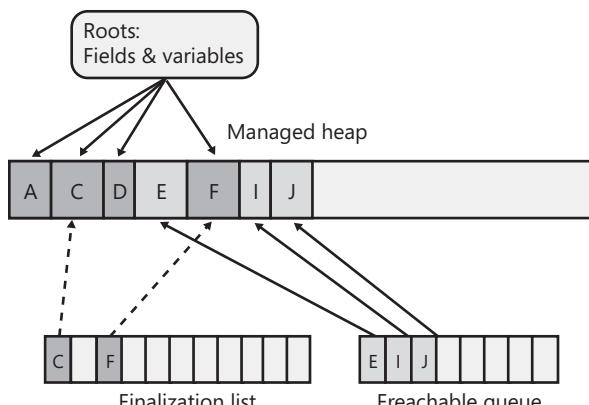


FIGURE 21-14 The managed heap showing pointers that moved from the finalization list to the reachable queue.

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. 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 normal-priority threads were 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 then 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.

In the future, the CLR may use multiple finalizer threads. So you should avoid writing any code that assumes that `Finalize` methods will be called serially. With just one finalizer thread, there could be performance and scalability issues in the scenario in which you have multiple CPUs allocating finalizable objects but only one thread executing `Finalize` methods—the one thread might not be able to keep up with the allocations.

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 is a reference to an object in the managed heap that 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 static fields are roots. So a reference in the freachable queue keeps the object it refers to reachable and is *not* garbage.

In short, when an object isn't reachable, the garbage collector considers the object to be garbage. Then when the garbage collector moves an object's reference from the finalization list to the freachable queue, the object is no longer considered garbage and its memory can't be reclaimed. When an object is garbage and then not garbage, we say that the object has been *resurrected*.

As freachable objects are marked, objects referred to by their reference type fields are also marked recursively; all these objects must get resurrected in order to survive the collection. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been resurrected. The garbage collector compacts the reclaimable memory, which promotes the resurrected object to an older generation (not ideal). And now, the special finalization thread empties the freachable queue, executing each object's `Finalize` method.

The next time the garbage collector is invoked on the older generation, it will see 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 either. 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 will be necessary because the objects get promoted to another generation. Figure 21-15 shows what the managed heap looks like after the second garbage collection.

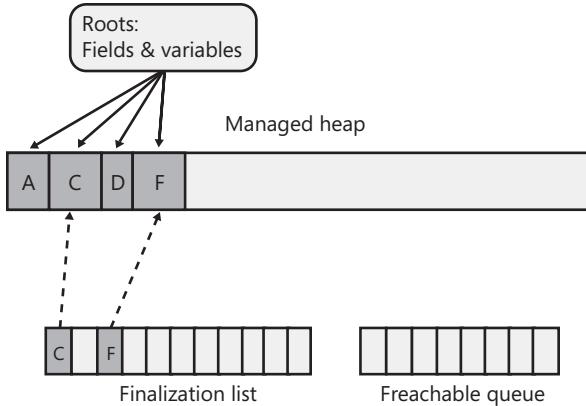


FIGURE 21-15 Status of the managed heap after second garbage collection.

Monitoring and Controlling the Lifetime of Objects Manually

The CLR provides each AppDomain with a GC handle table. This table allows an application to monitor the lifetime of an object or manually control the lifetime of an object. When an AppDomain is created, the table is empty. Each entry on the table consists of a reference to an object on the managed heap and a flag indicating how you want to monitor or control the object. An application adds and removes entries from the table via the `System.Runtime.InteropServices.GCHandle` type, as follows.

```
// This type is defined in the System.Runtime.InteropServices namespace
public struct GCHandle {
    // Static methods that create an entry in the table
    public static GCHandle Alloc(object value);
    public static GCHandle Alloc(object value, GCHandleType type);

    // Static methods that convert a GCHandle to an IntPtr
    public static explicit operator IntPtr(GCHandle value);
    public static IntPtr ToIntPtr(GCHandle value);

    // Static methods that convert an IntPtr to a GCHandle
    public static explicit operator GCHandle(IntPtr value);
    public static GCHandle FromIntPtr(IntPtr value);

    // Static methods that compare two GCHandles
    public static Boolean operator ==(GCHandle a, GCHandle b);
    public static Boolean operator !=(GCHandle a, GCHandle b);

    // Instance method to free the entry in the table (index is set to 0)
    public void Free();

    // Instance property to get/set the entry's object reference
    public object Target { get; set; }
}
```

```

// Instance property that returns true if index is not 0
public Boolean IsAllocated { get; }

// For a pinned entry, this returns the address of the object
public IntPtr AddrOfPinnedObject();

public override Int32 GetHashCode();
public override Boolean Equals(object o);
}

```

Basically, to control or monitor an object's lifetime, you call `GCHandle`'s static `Alloc` method, passing a reference to the object that you want to monitor/control, and a `GCHandleType`, which is a flag indicating how you want to monitor/control the object. The `GCHandleType` type is an enumerated type defined as follows.

```

public enum GCHandleType {
    Weak = 0,                      // Used for monitoring an object's existence
    WeakTrackResurrection = 1,      // Used for monitoring an object's existence
    Normal = 2,                     // Used for controlling an object's lifetime
    Pinned = 3                      // Used for controlling an object's lifetime
}

```

Now, here's what each flag means:

- **Weak** This flag allows you to *monitor* the lifetime of an object. Specifically, you can detect when the garbage collector has determined this object to be unreachable from application code. Note that the object's `Finalize` method may or may not have executed yet and therefore, the object may still be in memory.
- **WeakTrackResurrection** This flag allows you to *monitor* the lifetime of an object. Specifically, you can detect when the garbage collector has determined that this object is unreachable from application code. Note that the object's `Finalize` method (if it exists) has definitely executed, and the object's memory has been reclaimed.
- **Normal** This flag allows you to *control* the lifetime of an object. Specifically, you are telling the garbage collector that this object must remain in memory even though there may be no roots in the application that refer to this object. When a garbage collection runs, the memory for this object can be compacted (moved). The `Alloc` method that doesn't take a `GCHandleType` flag assumes that `GCHandleType.Normal` is specified.
- **Pinned** This flag allows you to *control* the lifetime of an object. Specifically, you are telling the garbage collector that this object must remain in memory even though there might be no roots in the application that refer to this object. When a garbage collection runs, the memory for this object cannot be compacted. This is typically useful when you want to hand the address of the memory out to native code. The native code can write to this memory in the managed heap knowing that a GC will not move the object.

When you call `GCHandle`'s static `Alloc` method, it scans the `AppDomain`'s GC handle table, looking for an available entry where the reference of the object you passed to `Alloc` is stored, and a flag is set to whatever you passed for the `GCHandleType` argument. Then, `Alloc` returns a `GCHandle`

instance back to you. A `GCHandle` is a lightweight value type that contains a single instance field, an `IntPtr`, which refers to the index of the entry in the table. When you want to free this entry in the GC handle table, you take the `GCHandle` instance and call the `Free` method (which also invalidates the `GCHandle` instance by setting its `IntPtr` field to zero).

Here's how the garbage collector uses the GC handle table. When a garbage collection occurs:

1. The garbage collector marks all of the reachable objects (as described at the beginning of this chapter). Then, the garbage collector scans the GC handle table; all `Normal` or `Pinned` objects are considered roots, and these objects are marked as well (including any objects that these objects refer to via their fields).
2. The garbage collector scans the GC handle table looking for all of the `Weak` entries. If a `Weak` entry refers to an object that isn't marked, the reference identifies an unreachable object (garbage), and the entry has its reference value changed to `null`.
3. The garbage collector scans the finalization list. If a reference in the list refers to an unmarked object, the reference identifies an unreachable object, and the reference is moved from the finalization list to the reachable queue. At this point, the object is marked because the object is now considered reachable.
4. The garbage collector scans the GC handle table looking for all of the `WeakTrackResurrection` entries. If a `WeakTrackResurrection` entry refers to an object that isn't marked (which now is an object referenced by an entry in the reachable queue), the reference identifies an unreachable object (garbage), and the entry has its reference value changed to `null`.
5. The garbage collector compacts the memory, squeezing out the holes left by the unreachable objects. `Pinned` objects are not compacted (moved); the garbage collector will move other objects around them.

Now that you have an understanding of the mechanism, let's take a look at when you'd use them. The easiest flags to understand are the `Normal` and `Pinned` flags, so let's start with these two. Both of these flags are typically used when interoperating with native code.

The `Normal` flag is used when you need to hand a pointer to a managed object to native code because, at some point in the future, the native code is going to call back into managed code, passing it the pointer. You can't actually pass a pointer to a managed object out to native code, because if a garbage collection occurs, the object could move in memory, invalidating the pointer. So to work around this, you would call `GCHandle`'s `Alloc` method, passing in a reference to the object and the `Normal` flag. Then you'd cast the returned `GCHandle` instance to an `IntPtr` and pass the `IntPtr` into the native code. When the native code calls back into managed code, the managed code would cast the passed `IntPtr` back to a `GCHandle` and then query the `Target` property to get the reference (or current address) of the managed object. When the native code no longer needs the reference, you'd call `GCHandle`'s `Free` method, which allows a future garbage collection to free the object (assuming no other root exists to this object).

Notice that in this scenario, the native code is not actually using the managed object itself; the native code wants a way just to reference the object. In some scenarios, the native code needs to actually

use the managed object. In these scenarios, the managed object must be pinned. Pinning prevents the garbage collector from moving/compacting the object. A common example is when you want to pass a managed `String` object to a Win32 function. In this case, the `String` object must be pinned because you can't pass the reference of a managed object to native code and then have the garbage collector move the object in memory. If the `String` object were moved, the native code would either be reading or writing to memory that no longer contained the `String` object's characters—this will surely cause the application to run unpredictably.

When you use the CLR's P/Invoke mechanism to call a method, the CLR pins the arguments for you automatically and unpins them when the native method returns. So, in most cases, you never have to use the `GCHandle` type to explicitly pin any managed objects yourself. You do have to use the `GCHandle` type explicitly when you need to pass the pointer to a managed object to native code; then the native function returns, but native code might still need to use the object later. The most common example of this is when performing asynchronous I/O operations.

Let's say that you allocate a byte array that should be filled as data comes in from a socket. Then, you would call `GCHandle`'s `Alloc` method, passing in a reference to the array object and the `Pinned` flag. Then, using the returned `GCHandle` instance, you call the `AddrOfPinnedObject` method. This returns an `IntPtr` that is the actual address of the pinned object in the managed heap; you'd then pass this address into the native function, which will return back to managed code immediately. While the data is coming from the socket, this byte array buffer should not move in memory; preventing this buffer from moving is accomplished by using the `Pinned` flag. When the asynchronous I/O operation has completed, you'd call `GCHandle`'s `Free` method, which will allow a future garbage collection to move the buffer. Your managed code should still have a reference to the buffer so that you can access the data, and this reference will prevent a garbage collection from freeing the buffer from memory completely.

It is also worth mentioning that C# offers a `fixed` statement that effectively pins an object over a block of code. Here is some code that demonstrates its use.

```
unsafe public static void Go() {
    // Allocate a bunch of objects that immediately become garbage
    for (Int32 x = 0; x < 10000; x++) new Object();

    IntPtr originalMemoryAddress;
    Byte[] bytes = new Byte[1000];    // Allocate this array after the garbage objects

    // Get the address in memory of the Byte[]
    fixed (Byte* pbytes = bytes) { originalMemoryAddress = (IntPtr) pbytes; }

    // Force a collection; the garbage objects will go away & the Byte[] might be compacted
    GC.Collect();

    // Get the address in memory of the Byte[] now & compare it to the first address
    fixed (Byte* pbytes = bytes) {
        Console.WriteLine("The Byte[] did{0} move during the GC",
            (originalMemoryAddress == (IntPtr) pbytes) ? " not" : null);
    }
}
```

Using C#'s `fixed` statement is more efficient than allocating a pinned GC handle. What happens is that the C# compiler emits a special "pinned" flag on the `pbytes` local variable. During a garbage collection, the GC examines the contents of this root, and if the root is not `null`, it knows not to move the object referred to by the variable during the compaction phase. The C# compiler emits IL to initialize the `pbytes` local variable to the address of the object at the start of a `fixed` block, and the compiler emits an IL instruction to set the `pbytes` local variable back to `null` at the end of the `fixed` block so that the variable doesn't refer to any object, allowing the object to move when the next garbage collection occurs.

Now, let's talk about the next two flags, `Weak` and `WeakTrackResurrection`. These two flags can be used in scenarios when interoperating with native code, but they can also be used in scenarios that use only managed code. The `Weak` flag lets you know when an object has been determined to be garbage but the object's memory is not guaranteed to be reclaimed yet. The `WeakTrackResurrection` flag lets you know when an object's memory has been reclaimed. Of the two flags, the `Weak` flag is much more commonly used than the `WeakTrackResurrection` flag. In fact, I've never seen anyone use the `WeakTrackResurrection` flag in a real application.

Let's say that `Object-A` periodically calls a method on `Object-B`. However, the fact that `Object-A` has a reference to `Object-B` forbids `Object-B` from being garbage collected, and in some rare scenarios, this may not be desired; instead, we might want `Object-A` to call `Object-B`'s method if `Object-B` is still alive in the managed heap. To accomplish this scenario, `Object-A` would call `GCHandle`'s `Alloc` method, passing in the reference to `Object-B` and the `Weak` flag. `Object-A` would now just save the returned `GCHandle` instance instead of the reference to `Object-B`.

At this point, `Object-B` can be garbage collected if no other roots are keeping it alive. When `Object-A` wants to call `Object-B`'s method, it would query `GCHandle`'s read-only `Target` property. If this property returns a non-`null` value, then `Object-B` is still alive. `Object-A`'s code would then cast the returned reference to `Object-B`'s type and call the method. If the `Target` property returns `null`, then `Object-B` has been collected (but not necessarily finalized) and `Object-A` would not attempt to call the method. At this point, `Object-A`'s code would probably also call `GCHandle`'s `Free` method to relinquish the `GCHandle` instance.

Because working with the `GCHandle` type can be a bit cumbersome and because it requires elevated security to keep or pin an object in memory, the `System` namespace includes a `WeakReference<T>` class to help you.

```
public sealed class WeakReference<T> : ISerializable where T : class {
    public WeakReference(T target);
    public WeakReference(T target, Boolean trackResurrection);
    public void SetTarget(T target);
    public Boolean TryGetTarget(out T target);
}
```

This class is really just an object-oriented wrapper around a `GCHandle` instance: logically, its constructor calls `GCHandle`'s `Alloc`, its `TryGetTarget` method queries `GCHandle`'s `Target` property, its `SetTarget` method sets `GCHandle`'s `Target` property, and its `Finalize` method (not shown in the preceding code, because it's protected) calls `GCHandle`'s `Free` method. In addition, no special permissions are required for code to use the `WeakReference<T>` class because the class supports only weak

references; it doesn't support the behavior provided by `GCHandle` instances allocated with a `GCHandleType` of `Normal` or `Pinned`. The downside of the `WeakReference<T>` class is that an instance of it must be allocated on the heap. So the `WeakReference<T>` class is a heavier-weight object than a `GCHandle` instance.



Important When developers start learning about weak references, they immediately start thinking that they are useful in caching scenarios. For example, they think it would be cool to construct a bunch of objects that contain a lot of data and then to create weak references to these objects. When the program needs the data, the program checks the weak reference to see if the object that contains the data is still around, and if it is, the program just uses it; the program experiences high performance. However, if a garbage collection occurred, the objects that contained the data would be destroyed, and when the program has to re-create the data, the program experiences lower performance.

The problem with this technique is the following: garbage collections do not only occur when memory is full or close to full. Instead, garbage collections occur whenever generation 0 is full. So objects are being tossed out of memory much more frequently than desired, and your application's performance suffers greatly.

Weak references can be used quite effectively in caching scenarios, but building a good cache algorithm that finds the right balance between memory consumption and speed is very complex. Basically, you want your cache to keep strong references to all of your objects and then, when you see that memory is getting tight, you start turning strong references into weak references. Currently, the CLR offers no mechanism to notify an application that memory is getting tight. But some people have had much success by periodically calling the Win32 `GlobalMemoryStatusEx` function and checking the returned `MEMORYSTATUSEX` structure's `dwMemoryLoad` member. If this member reports a value above 80, memory is getting tight, and you can start converting strong references to weak references based on whether you want a least-recently used algorithm, a most-frequently used algorithm, a time-base algorithm, or whatever.

Developers frequently want to associate a piece of data with another entity. For example, you can associate data with a thread or with an `AppDomain`. It is also possible to associate data with an individual object by using the `System.Runtime.CompilerServices.ConditionalWeakTable< TKey , TValue >` class, which looks like this.

```
public sealed class ConditionalWeakTable< TKey , TValue >
    where TKey : class where TValue : class {
    public ConditionalWeakTable();
    public void Add(TKey key, TValue value);
    public TValue GetValue(TKey key, CreateValueCallback< TKey , TValue > createValueCallback);
    public Boolean TryGetValue(TKey key, out TValue value);
    public TValue GetOrCreateValue(TKey key);
    public Boolean Remove(TKey key);

    public delegate TValue CreateValueCallback(TKey key); // Nested delegate definition
}
```

If you want to associate some arbitrary data with one or more objects, you would first create an instance of this class. Then, call the Add method, passing in a reference to some object for the key parameter and the data you want to associate with the object in the value parameter. If you attempt to add a reference to the same object more than once, the Add method throws an ArgumentException; to change the value associated with an object, you must remove the key and then add it back in with the new value. Note that this class is thread-safe so multiple threads can use it concurrently, although this means that the performance of the class is not stellar; you should test the performance of this class to see how well it works for your scenario.

Of course, a table object internally stores a WeakReference to the object passed in as the key; this ensures that the table doesn't forcibly keep the object alive. But what makes the ConditionalWeakTable class so special is that it guarantees that the value remains in memory as long as the object identified by the key is in memory. So this is more than a normal WeakReference because if it were, the value could be garbage collected even though the key object continued to live. The ConditionalWeakTable class could be used to implement the dependency property mechanism used by XAML. It can also be used internally by dynamic languages to dynamically associate data with objects.

Here is some code that demonstrates the use of the ConditionalWeakTable class. It allows you to call the GCWatch extension method on any object passing in some String tag. Then it notifies you via the console window whenever that particular object gets garbage collected.

```
internal static class ConditionalWeakTableDemo {
    public static void Main() {
        Object o = new Object().GCWatch("My Object created at " + DateTime.Now);
        GC.Collect(); // We will not see the GC notification here
        GC.KeepAlive(o); // Make sure the object o refers to lives up to here
        o = null; // The object that o refers to can die now

        GC.Collect(); // We'll see the GC notification sometime after this line
        Console.ReadLine();
    }
}

internal static class GCWatcher {
    // NOTE: Be careful with Strings due to interning and MarshalByRefObject proxy objects
    private readonly static ConditionalWeakTable<Object, NotifyWhenGCd<String>> s_cwt =
        new ConditionalWeakTable<Object, NotifyWhenGCd<String>>();

    private sealed class NotifyWhenGCd<T> {
        private readonly T m_value;

        internal NotifyWhenGCd(T value) { m_value = value; }
        public override string ToString() { return m_value.ToString(); }
        ~NotifyWhenGCd() { Console.WriteLine("GC'd: " + m_value); }
    }

    public static T GCWatch<T>(this T @object, String tag) where T : class {
        s_cwt.Add(@object, new NotifyWhenGCd<String>(tag));
        return @object;
    }
}
```

Assembly Loading and Reflection

In this chapter:

Assembly Loading	584
Using Reflection to Build a Dynamically Extensible Application.....	588
Reflection Performance.....	589
Designing an Application That Supports Add-Ins	596
Using Reflection to Discover a Type's Members.....	599

This chapter is all about discovering information about types, creating instances of them, and accessing their members when you didn't know anything about them at compile time. The information in this chapter is typically used to create a dynamically extensible application. This is the kind of application for which one company builds a host application and other companies create add-ins to extend the host application. The host can't be built or tested against the add-ins because the add-ins are created by different companies and are likely to be created after the host application has already shipped. This is why the host needs to discover the add-ins at run time.

A dynamically extensible application could take advantage of common language runtime (CLR) hosting and AppDomains as discussed in Chapter 22, "CLR Hosting and AppDomains." The host could run the add-in code in an AppDomain with its own security and configuration settings. The host could also unload the add-in code by unloading the AppDomain. At the end of this chapter, I'll talk a little about how to put all of this stuff together—CLR hosting, AppDomains, assembly loading, type discovery, type instance construction, and reflection—in order to build a robust, secure, and dynamically extensible application.



Important For version 4.5 of the .NET Framework, Microsoft has introduced a new reflection API. The old API had many shortcomings. For example, it did not support LINQ well, it had policies embedded in it that were not correct for some languages, it would sometimes force the loading of assemblies unnecessarily, and it was an overly complex API that offered solutions for problems rarely encountered. The new API fixes all of these issues. However, as of .NET 4.5, the new reflection API is not as complete as the old API. With the new API and some extension methods (in the `System.Reflection.RuntimeReflectionExtensions` class), you can accomplish all you need. Expect additional methods to be added to the new API in future versions of the .NET Framework.

Of course, for desktop applications, the old API still exists so that it doesn't break existing code when re-compiling it. However, the new API is the recommended API going forward, and that is why I explain the new API exclusively in this chapter. For Windows Store Apps (where backward compatibility is not an issue), using the new API is mandatory.

Assembly Loading

As you know, when the just-in-time (JIT) compiler compiles the Intermediate Language (IL) for a method, it sees what types are referenced in the IL code. Then at run time, the JIT compiler uses the assembly's TypeRef and AssemblyRef metadata tables to determine what assembly defines the type being referenced. The AssemblyRef metadata table entry contains all of the parts that make up the strong name of the assembly. The JIT compiler grabs all of these parts—name (without extension or path), version, culture, and public key token—concatenates them into a string, and then attempts to load an assembly matching this identity into the AppDomain (assuming that it's not already loaded). If the assembly being loaded is weakly named, the identity is just the name of the assembly (no version, culture, or public key token information).

Internally, the CLR attempts to load this assembly by using the `System.Reflection.Assembly` class's static `Load` method. This method is publicly documented, and you can call it to explicitly load an assembly into your AppDomain. This method is the CLR equivalent of Win32's `LoadLibrary` function. There are actually several overloaded versions of `Assembly`'s `Load` method. Here are the prototypes of the more commonly used overloads.

```
public class Assembly {  
    public static Assembly Load(AssemblyName assemblyRef);  
    public static Assembly Load(String assemblyString);  
    // Less commonly used overloads of Load are not shown  
}
```

Internally, `Load` causes the CLR to apply a version-binding redirection policy to the assembly and looks for the assembly in the global assembly cache (GAC), followed by the application's base directory, private path subdirectories, and codebase locations. If you call `Load` by passing a weakly named assembly, `Load` doesn't apply a version-binding redirection policy to the assembly, and the CLR won't look in the GAC for the assembly. If `Load` finds the specified assembly, it returns a reference to an `Assembly` object that represents the loaded assembly. If `Load` fails to find the specified assembly, it throws a `System.IO.FileNotFoundException`.



Note In some extremely rare situations, you may want to load an assembly that was built for a specific CPU architecture. In this case, when specifying an assembly's identity, you can also include a process architecture part. For example, if my GAC happened to have an IL-neutral and an x86-specific version of an assembly, the CLR would favor the CPU-specific version of the assembly (as discussed in Chapter 3, "Shared Assemblies and Strongly Named Assemblies"). However, I can force the CLR to load the IL-neutral version by passing the following string to `Assembly`'s `Load` method.

```
"SomeAssembly, Version=2.0.0.0, Culture=neutral,  
PublicKeyToken=01234567890abcde, ProcessorArchitecture=MSIL"
```

Today, the CLR supports five possible values for `ProcessorArchitecture`: MSIL (Microsoft IL), x86, IA64, AMD64, and Arm.



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 because 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 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 by using the calling `AppDomain`'s policy and search locations, a `FileNotFoundException` is thrown. This behavior is usually undesirable and is the reason that you should avoid `AppDomain`'s `Load` method.

In most dynamically extensible applications, `Assembly`'s `Load` method is the preferred way of loading an assembly into an `AppDomain`. However, it does require that you have all of the pieces that make up an assembly's identity. Frequently, developers write tools or utilities (such as `ILDasm.exe`, `PE-Verify.exe`, `CorFlags.exe`, `GACUtil.exe`, `SGen.exe`, `SN.exe`, `XSD.exe`) that perform some kind of processing on an assembly. All of these tools take a command-line argument that refers to the path name of an assembly file (including file extension).

To load an assembly specifying a path name, you call `Assembly`'s `LoadFrom` method.

```
public class Assembly {  
    public static Assembly LoadFrom(String path);  
    // Less commonly used overloads of LoadFrom are not shown  
}
```

Internally, `LoadFrom` first calls `System.Reflection.AssemblyName`'s static `GetAssemblyName` method, which opens the specified file, finds the `AssemblyDef` metadata table's entry, and extracts the assembly identity information and returns it in a `System.Reflection.AssemblyName` object (the file is also closed). Then, `LoadFrom` internally calls `Assembly`'s `Load` method, passing it the `AssemblyName` object. At this point, the CLR applies a version-binding redirection policy and searches the various locations looking for a matching assembly. If `Load` finds the assembly, it will load it, and an `Assembly` object that represents the loaded assembly will be returned; `LoadFrom` returns this value. If `Load` fails to find an assembly, `LoadFrom` loads the assembly at the path name specified in `LoadFrom`'s argument. Of course, if an assembly with the same identity is already loaded, `LoadFrom` simply returns an `Assembly` object that represents the already loaded assembly.

By the way, the `LoadFrom` method allows you to pass a URL as the argument. Here is an example.

```
Assembly a = Assembly.LoadFrom(@"http://Wintellect.com/SomeAssembly.dll");
```

When you pass an Internet location, the CLR downloads the file, installs it into the user's download cache, and loads the file from there. Note that you must be online or an exception will be thrown. However, if the file has been downloaded previously, and if Windows Internet Explorer has been set to work offline (see Internet Explorer's Work Offline menu item in its File menu), the previously downloaded file will be used, and no exception will be thrown. You can also call `UnsafeLoadFrom`, which can load a web-downloaded assembly, bypassing some security checks.



Important It is possible to have different assemblies on a single machine all with the same identity. Because `LoadFrom` calls `Load` internally, it is possible that the CLR will not load the specified file and instead will load a different file, giving you unexpected behavior. It is highly recommended that each build of your assembly change the version number; this ensures that each version has its own identity, and because of this, `LoadFrom` will now work as expected.

Microsoft Visual Studio's UI designers and other tools typically use `Assembly`'s `LoadFile` method. This method can load an assembly from any path and can be used to load an assembly with the same identity multiple times into a single `AppDomain`. This can happen as changes to an application's UI are made in the designer/tool and the user rebuilds the assembly. When loading an assembly via `LoadFile`, the CLR will not resolve any dependencies automatically; your code must register with `AppDomain`'s `AssemblyResolve` event and have your event callback method explicitly load any dependent assemblies.

If you are building a tool that simply analyzes an assembly's metadata via reflection (as discussed later in this chapter), and you want to ensure that none of the code contained inside the assembly

executes, the best way for you to load an assembly is to use `Assembly`'s `ReflectionOnlyLoadFrom` method, or in some rarer cases, `Assembly`'s `ReflectionOnlyLoad` method. Here are the prototypes of both methods.

```
public class Assembly {  
    public static Assembly ReflectionOnlyLoadFrom(String assemblyFile);  
    public static Assembly ReflectionOnlyLoad(String assemblyString);  
    // Less commonly used overload of ReflectionOnlyLoad is not shown  
}
```

The `ReflectionOnlyLoadFrom` method will load the file specified by the path; the strong-name identity of the file is not obtained, and the file is not searched for in the GAC or elsewhere. The `ReflectionOnlyLoad` method will search for the specified assembly looking in the GAC, application base directory, private paths, and codebases. However, unlike the `Load` method, the `ReflectionOnlyLoad` method does not apply versioning policies, so you will get the exact version that you specify. If you want to apply versioning policy yourself to an assembly identity, you can pass the string into `AppDomain`'s `ApplyPolicy` method.

When an assembly is loaded with `ReflectionOnlyLoadFrom` or `ReflectionOnlyLoad`, the CLR forbids any code in the assembly from executing; any attempt to execute code in an assembly loaded with either of these methods causes the CLR to throw an `InvalidOperationException`. These methods allow a tool to load an assembly that was delay-signed, would normally require security permissions that prevent it from loading, or was created for a different CPU architecture.

Frequently, when using reflection to analyze an assembly loaded with one of these two methods, the code will have to register a callback method with `AppDomain`'s `ReflectionOnlyAssemblyResolve` event to manually load any referenced assemblies (calling `AppDomain`'s `ApplyPolicy` method, if desired); the CLR doesn't do it automatically for you. When the callback method is invoked, it must call `Assembly`'s `ReflectionOnlyLoadFrom` or `ReflectionOnlyLoad` method to explicitly load a referenced assembly and return a reference to this assembly.



Note People often ask about assembly unloading. Unfortunately, the CLR doesn't support the ability to unload individual assemblies. If the CLR allowed it, your application would crash if a thread returned back from a method to code in the unloaded assembly. The CLR is all about robustness, security, and allowing an application to crash in this way would be counterproductive to its goals. If you want to unload an assembly, you must unload the entire `AppDomain` that contains it. This was discussed in great detail in Chapter 22.

It would seem that assemblies loaded with either the `ReflectionOnlyLoadFrom` or the `ReflectionOnlyLoad` method could be unloaded. After all, code in these assemblies is not allowed to execute. However, the CLR also doesn't allow assemblies loaded via either of these two methods to be unloaded. The reason is that after an assembly is loaded this way, you can still use reflection to create objects that refer to the metadata defined inside these assemblies. Unloading the assembly would require the objects to be invalidated somehow. Keeping track of this would be too expensive in terms of implementation and execution speed.

Many applications consist of an EXE file that depends on many DLL files. When deploying this application, all the files must be deployed. However, there is a technique that you can use to deploy just a single EXE file. First, identify all the DLL files that your EXE file depends on that do not ship as part of the Microsoft .NET Framework itself. Then add these DLLs to your Visual Studio project. For each DLL file you add, display its properties and change its Build Action to Embedded Resource. This causes the C# compiler to embed the DLL file(s) into your EXE file, and you can deploy this one EXE file.

At run time, the CLR won't be able to find the dependent DLL assemblies, which is a problem. To fix this, when your application initializes, register a callback method with the AppDomain's `ResolveAssembly` event. The callback method's code should look something like the following.

```
private static Assembly ResolveEventHandler(Object sender, ResolveEventArgs args) {
    String dllName = new AssemblyName(args.Name).Name + ".dll";

    var assem = Assembly.GetExecutingAssembly();
    String resourceName = assem.GetManifestResourceNames().FirstOrDefault(rn =>
        rn.EndsWith(dllName));
    if (resourceName == null) return null; // Not found, maybe another handler will find it
    using (var stream = assem.GetManifestResourceStream(resourceName)) {
        Byte[] assemblyData = new Byte[stream.Length];
        stream.Read(assemblyData, 0, assemblyData.Length);
        return Assembly.Load(assemblyData);
    }
}
```

Now, the first time a thread calls a method that references a type in a dependent DLL file, the `AssemblyResolve` event will be raised and the preceding callback code will find the embedded DLL resource desired and load it by calling an overload of `Assembly`'s `Load` method that takes a `Byte[]` as an argument. Although I love the technique of embedding dependent DLLs inside another assembly, you should be aware that this does increase the memory used by your application at run time.

Using Reflection to Build a Dynamically Extensible Application

As you know, metadata is stored in a bunch of tables. When you build an assembly or a module, the compiler that you're using creates a type definition table, a field definition table, a method definition table, and so on. The `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 of the types in a type definition metadata table. Then for each type, you can obtain its base type, the interfaces it implements, and the flags that 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 18, "Custom Attributes") that have been applied to any of the metadata entities. There are even classes that let you determine referenced assemblies and methods that return the IL byte stream for a method. With all of this information, you could easily build a tool very similar to Microsoft's `ILDasm.exe`.



Note 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 Framework Class Library (FCL) 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 by class libraries that need to understand a type's definition in order to provide some rich functionality. For example, the FCL's serialization mechanism (discussed in Chapter 24, "Runtime Serialization") 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 that is used for sending across the Internet, saving to a file, or copying to the clipboard. 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 defined in 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 used by an application are determined at compile time.)

Reflection Performance

Reflection is an extremely powerful mechanism because it allows you to discover and use types and members at run time that you did not know about at compile time. This power does come with two main drawbacks:

- Reflection prevents type safety at compile time. Because reflection uses strings heavily, you lose type safety at compile time. For example, if you call `Type.GetType("int")`; to ask reflection to find a type called "int", the code compiles but returns `null` at run time because the CLR knows the "int" type as "System.Int32".
- Reflection is slow. When using reflection, the names of types and their members are not known at compile time; you discover them at run time by using a string name to identify each type and member. This means that reflection is constantly performing string searches as the types in the `System.Reflection` namespace scan through an assembly's metadata. Often, the string searches are case-insensitive comparisons, which can slow this down even more.

Invoking a member by using reflection will also hurt performance. When using reflection to invoke a method, you must first package the arguments into an array; internally, reflection must unpack these on to the thread's stack. Also, the CLR must check that the arguments are of the correct data type before invoking a method. Finally, the CLR ensures that the caller has the proper security permission to access the member being invoked.

For all of these reasons, it's best to avoid using reflection to access a field or invoke a method/property. If you're writing an application that will dynamically discover and construct type instances, 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 derived type, place the reference in a variable that is of the base type (by way of a cast), 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 (by way of a cast), and call the methods defined by the interface.

I tend to prefer using the interface 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. Although the base type technique works better in versioning scenarios, because you could always add a member to the base type and the derived types just inherit it; you can't add a member to an interface without forcing all types that implement the interface to modify their code and recompile it.

When you use either of these two techniques, I strongly suggest that the interface or base type be defined in its own assembly. This will reduce versioning issues. For more information about how to do this, see the section titled "Designing an Application That Supports Add-Ins" later in this chapter.

Discovering Types Defined in an Assembly

Reflection is frequently used to determine what types an assembly defines. The FCL offers many APIs to get this information. By far, the most commonly used API is `Assembly`'s `ExportedTypes` property. Here is an example of code that loads an assembly and shows the names of all of the publicly exported types defined in it.

```
using System;
using System.Reflection;

public static class Program {
    public static void Main() {
        String dataAssembly = "System.Data, version=4.0.0.0, " +
            "culture=neutral, PublicKeyToken=b77a5c561934e089";
        LoadAssemAndShowPublicTypes(dataAssembly);
    }

    private static void LoadAssemAndShowPublicTypes(String assemId) {
        // Explicitly load an assembly in to this AppDomain
        Assembly a = Assembly.Load(assemId);
```

```

// Execute this loop once for each Type
// publicly-exported from the loaded assembly
foreach (Type t in a.ExportedTypes) {
    // Display the full name of the type
    Console.WriteLine(t.FullName);
}
}
}

```

What Exactly Is a Type Object?

Notice that the previous code iterates over a sequence of `System.Type` objects. The `System.Type` type is your starting point for doing type and object manipulations. A `System.Type` object represents a type reference (as opposed to a type definition).

Recall that `System.Object` defines a public, nonvirtual instance method named `GetType`. When you call this method, the CLR determines the specified object's type and returns a reference to its `Type` object. Because there is only one `Type` object per type in an `AppDomain`, you can use equality and inequality operators to see whether two objects are of the same type.

```

private static 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 the static `GetType` method. All versions of this method take a `String`. The string must specify the full name of the type (including its namespace). Note that the primitive type names supported by the compiler (such as C#'s `int`, `string`, `bool`, and so on) aren't allowed because these names mean nothing to the CLR. 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 `Type` 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` is thrown, depending on which overload of the `GetType` method you called and what parameters you passed to it. The FCL documentation fully explains this method.

You can pass an assembly-qualified type string, such as "System.Int32, mscorelib, Version=4.0.0.0, 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 a static `ReflectionOnlyGetType` method. This method behaves similarly to the `GetType` method mentioned in the previous bullet, except that the type is loaded so that it can be reflected over but cannot be executed.

- The `System.TypeInfo` type offers the following instance members: `DeclaredNestedTypes` and `GetDeclaredNestedType`.
- The `System.Reflection.Assembly` type offers the following instance members: `GetType`, `DefinedTypes`, and `ExportedTypes`.



Note Microsoft has defined a Backus-Naur Form grammar for type names and assembly-qualified type names that is used for constructing strings that will be passed to reflection methods. Knowledge of the grammar can come in quite handy when you are using reflection, specifically if you are working with nested types, generic types, generic methods, reference parameters, or arrays. For the complete grammar, see the FCL documentation or do a web search for “Backus-Naur Form Grammar for Type Names.” You can also look at `Type`’s and `TypeInfo`’s `MakeArrayType`, `MakeByRefType`, `MakeGenericType`, and `MakePointerType` methods.

Many programming languages also offer an operator that allows you to obtain a `Type` object from a type name that is known at compile time. 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`, and you use this operator typically to compare late-bound type information with early-bound (known at compile time) type information. The following code demonstrates an example of its use.

```
private static void SomeMethod(Object o) {  
    // GetType returns the type of the object at runtime (late-bound)  
    // typeof returns the type of the specified class (early-bound)  
    if (o.GetType() == typeof(FileInfo)) { ... }  
    if (o.GetType() == typeof(DirectoryInfo)) { ... }  
}
```



Note The first `if` statement in the code checks whether the variable `o` refers to an object of the `FileInfo` type; it does not check whether `o` refers to an object that is derived from the `FileInfo` type. In other words, the preceding code tests for an exact match, not a compatible match, which is what you would get if you use a cast or C#’s `is` or `as` operators.

As mentioned earlier, a `Type` object represents a type reference that is a lightweight object. If you want to learn more about the type itself, then you must acquire a `TypeInfo` object, which represents a type definition. You can convert a `Type` object to a `TypeInfo` object by calling `System.Reflection.IntrospectionExtensions`’ `GetTypeInfo` extension method.

```
Type typeReference = ...; // For example: o.GetType() or typeof(Object)  
TypeInfo typeDefinition = typeReference.GetTypeInfo();
```

And, although less useful, you can convert a `TypeInfo` object to a `Type` object by calling `TypeInfo`'s `AsType` method.

```
TypeInfo typeDefinition = ...;
Type typeReference = typeDefinition.AsType();
```

Obtaining a `TypeInfo` object forces the CLR to resolve the type by ensuring that the assembly that defines the type is loaded. This can be an expensive operation that can be avoided if all you need are type references (`Type` objects). However, after you have a `TypeInfo` 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 a reference to the type's base type, and a slew of members will give you even more information about the type. The FCL documentation describes all of the methods and properties that `TypeInfo` exposes.

Building a Hierarchy of Exception-Derived Types

The following code uses many of the concepts discussed already in this chapter to load a bunch of assemblies into the `AppDomain` and display all of the classes that are ultimately derived from `System.Exception`. By the way, this is the program I wrote to build the exception hierarchy displayed in the "FCL-Defined Exception Classes" section in Chapter 20, "Exceptions and State Management."

```
public static void Go() {
    // Explicitly load the assemblies that we want to reflect over
    LoadAssemblies();

    // Filter & sort all the types
    var allTypes =
        (from a in AppDomain.CurrentDomain.GetAssemblies()
         from t in a.ExportedTypes
         where typeof(Exception).GetTypeInfo().IsAssignableFrom(t.GetTypeInfo())
         orderby t.Name
         select t).ToArray();

    // Build the inheritance hierarchy tree and show it
    Console.WriteLine(WalkInheritanceHierarchy(new StringBuilder(), 0, typeof(Exception),
allTypes));
}

private static StringBuilder WalkInheritanceHierarchy(
    StringBuilder sb, Int32 indent, Type BaseType, IEnumerable<Type> allTypes) {
    String spaces = new String(' ', indent * 3);
    sb.AppendLine(spaces + BaseType.FullName);
    foreach (var t in allTypes) {
        if (t.GetTypeInfo().BaseType != BaseType) continue;
        WalkInheritanceHierarchy(sb, indent + 1, t, allTypes);
    }
}
```

```

    }
    return sb;
}

private static void LoadAssemblies() {
    String[] assemblies = {
        "System",                      PublicKeyToken={0}",
        "System.Core",                  PublicKeyToken={0}",
        "System.Data",                  PublicKeyToken={0}",
        "System.Design",                PublicKeyToken={1}",
        "System.DirectoryServices",     PublicKeyToken={1}",
        "System.Drawing",               PublicKeyToken={1}",
        "System.Drawing.Design",        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.Xml",                   PublicKeyToken={0}",
    };
}

String EcmaPublicKeyToken = "b77a5c561934e089";
String MSPublicKeyToken = "b03f5f7f11d50a3a";

// Get the version of the assembly containing System.Object
// We'll assume the same version for all the other assemblies
Version version = typeof(System.Object).Assembly.GetName().Version;

// Explicitly load the assemblies that we want to reflect over
foreach (String a in assemblies) {
    String AssemblyIdentity =
        String.Format(a, EcmaPublicKeyToken, MSPublicKeyToken) +
        ", Culture=neutral, Version=" + version;
    Assembly.Load(AssemblyIdentity);
}
}
}

```

Constructing an Instance of a Type

After you have a reference to a Type-derived object, you might want to construct an instance of this type. The FCL offers several mechanisms to accomplish this:

- **System.Activator's CreateInstance methods** The Activator class offers several overloads of its 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 for 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 by using 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 object to materialize. When you're ready to materialize the object, you call `ObjectHandle`'s `Unwrap` method. This method loads the assembly that defines the type being materialized in the AppDomain where `Unwrap` is called. If the object is being marshaled by reference, the proxy type and object are created. If the object is being marshaled by value, the copy is deserialized.

- **`System.Activator`'s `CreateInstanceFrom` methods** The `Activator` class also offers a set of static `CreateInstanceFrom` methods. These methods behave just as 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 by using `Assembly`'s `LoadFrom` method (instead of `Load`). Because none of these methods takes a `Type` parameter, all of the `CreateInstanceFrom` methods return a reference to an `ObjectHandle`, which must be unwrapped.
- **`System.AppDomain`'s methods** The `AppDomain` type offers four instance methods (each with several overloads) that construct an instance of a type: `CreateInstance`, `CreateInstanceAndUnwrap`, `CreateInstanceFrom`, and `CreateInstanceFromAndUnwrap`. These methods work just as `Activator`'s methods except that these 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.Reflection.ConstructorInfo`'s `Invoke` instance method** Using a reference to a `TypeInfo` 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 of the mechanisms in the preceding list construct an object by calling its constructor. However, `Activator`'s `CreateInstance` methods will allow you to 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 `MethodInfo`'s `CreateDelegate` method. The first parameter to all versions of `CreateDelegate` is a reference to the `Type` of delegate you want to create. `CreateDelegate`'s other parameter allows you to specify which object should be passed as the `this` parameter when calling an instance method.

To construct an instance of a generic type, first get a reference to the open type, and then call `Type`'s `MakeGenericType` method, passing in an array of types that you want to use as the type arguments. Then, take the returned `Type` object and pass it into one of the various methods previously listed. Here is an example.

```
using System;
using System.Reflection;

internal sealed class Dictionary<TKey, TValue> { }

public static class Program {
    public static void Main() {
        // Get a reference to the generic type's type object
        Type openType = typeof(Dictionary<,>);

        // Close the generic type by using TKey=String, TValue=Int32
        Type closedType = openType.MakeGenericType(typeof(String), typeof(Int32));

        // Construct an instance of the closed type
        Object o = Activator.CreateInstance(closedType);

        // Prove it worked
        Console.WriteLine(o.GetType());
    }
}
```

If you compile the preceding code and run it, you get the following output.

```
Dictionary`2[System.String, System.Int32]
```

Designing an Application That Supports Add-Ins

When you're building extensible applications, interfaces should be the centerpiece. You could use a base class instead of an interface, but in general, an interface is preferred because it allows add-in developers to choose their own base class. 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 a Host SDK assembly that defines an interface whose methods are used as the communication mechanism between the host application and the add-in components. When defining the parameters and return types 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 Host SDK assembly, too. After 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. Once published, you should really avoid making any kind of breaking changes to the types in this assembly. For example, do not change the interface in any way. However, if you define any data types, it is OK to add new members. If you make any modifications to the assembly, you'll probably want to deploy it with a publisher policy file (also discussed in Chapter 3).



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 CLR instance. 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.

- The add-in developers will, of course, define their own types in their own Add-In assembly. Their Add-In assembly will reference the types in your Host SDK assembly. The add-in developers are able to put out a new version of their assembly as often as they'd like, and the host application will be able to consume the add-in types without any problem whatsoever.
- Create a separate Host Application assembly containing your application's types. This assembly will obviously reference the Host SDK assembly and use the types defined in it. Feel free to modify the code in the Host Application assembly to your heart's desire. Because the add-in developers don't reference the Host Application assembly, you can put out a new version of it every hour if you want to and not affect any of the add-in developers.

This section contains some very important information. When using types across assemblies, you need to be concerned with assembly-versioning issues. Take your time to architect this cleanly by isolating the types that you use for communication across assembly boundaries into their own assembly. Avoid mutating or changing these type definitions. However, if you really need to modify the type definitions, make sure that you change the assembly's version number and create a publisher policy file for the new version.

I'll now walk through a very simple scenario that puts all of this together. First, here is the code for the HostSDK.dll assembly.

```
using System;

namespace Wintellect.HostSDK {
    public interface IAddIn {
        String DoSomething(Int32 x);
```

```
    }
}
```

Second, here is the code for an AddInTypes.dll assembly defining two public types that implement the HostSDK's interface. To build this assembly, the HostSDK.dll assembly must be referenced.

```
using System;
using Wintellect.HostSDK;

public sealed class AddIn_A : IAddIn {
    public AddIn_A() {
    }
    public String DoSomething(Int32 x) {
        return "AddIn_A: " + x.ToString();
    }
}

public sealed class AddIn_B : IAddIn {
    public AddIn_B() {
    }
    public String DoSomething(Int32 x) {
        return "AddIn_B: " + (x * 2).ToString();
    }
}
```

Third, here is the code for a simple Host.exe assembly (a console application). To build this assembly, the HostSDK.dll assembly must be referenced. To discover usable add-in types, this host code assumes that the types are defined in assemblies ending with a .dll file extension and that these assemblies are deployed into the same directory as the host's EXE file. Microsoft's Managed Extensibility Framework (MEF) is built on top of the various mechanisms that I show here, and it also offers add-in registration and discovery mechanisms. I urge you to check MEF out if you are building a dynamically extensible application, because it can simplify some of the material in this chapter.

```
using System;
using System.IO;
using System.Reflection;
using System.Collections.Generic;
using Wintellect.HostSDK;

public static class Program {
    public static void Main() {
        // Find the directory that contains the Host exe
        String AddInDir = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);

        // Assume AddIn assemblies are in same directory as host's EXE file
        var AddInAssemblies = Directory.EnumerateFiles(AddInDir, "*.dll");

        // Create a collection of Add-In Types usable by the host
        var AddInTypes =
            from file in AddInAssemblies
            let assembly = Assembly.Load(file)
            from t in assembly.ExportedTypes // Publicly-exported types
            // Type is usable if it is a class that implements IAddIn
            where t.IsClass && typeof(IAddIn).GetTypeInfo().IsAssignableFrom(t.GetTypeInfo())
            select t;
    }
}
```

```

    select t;
    // Initialization complete: the host has discovered the usable Add-Ins

    // Here's how the host can construct Add-In objects and use them
    foreach (Type t in AddInTypes) {
        IAddIn ai = (IAddIn) Activator.CreateInstance(t);
        Console.WriteLine(ai.DoSomething(5));
    }
}
}

```

The simple host/add-in scenario just shown doesn't use AppDomains. However, in a real-life scenario, you will likely create each add-in in its own AppDomain with its own security and configuration settings. And of course, each AppDomain could be unloaded if you wanted to remove an add-in from memory. To communicate across the AppDomain boundary, you'd either tell the add-in developers to derive their add-in types from `MarshalByRefObject` or, more likely, have the host application define its own internal type that is derived from `MarshalByRefObject`. As each AppDomain is created, the host would create an instance of its own `MarshalByRefObject`-derived type in the new AppDomain. The host's code (in the default AppDomain) would communicate with its own type (in the other AppDomains) to have it load add-in assemblies and create and use instances of the add-in types.

Using Reflection to Discover a Type's Members

So far, this chapter has focused on the parts of reflection—assembly loading, type discovery, and object construction—necessary to build a dynamically extensible application. In order to have good performance and compile-time type safety, you want to avoid using reflection as much as possible. In the dynamically extensible application scenario, after an object is constructed, the host code typically casts the object to an interface type or a base class that is known at compile time; this allows the object's members to be accessed in a high-performance and compile-time type-safe way.

In the remainder of this chapter, I'm going to focus on some other aspects of reflection that you can use to discover and then invoke a type's members. The ability to discover and invoke a type's members is typically used to create developer tools and utilities that analyze an assembly by looking for certain programming patterns or uses of certain members. Examples of tools/utilities that do this are ILDasm.exe, FxCopCmd.exe, and Visual Studio's Windows Forms, Windows Presentation Foundation, and Web Forms designers. In addition, some class libraries use the ability to discover and invoke a type's members in order to offer rich functionality as a convenience to developers. Examples of class libraries that do so are serialization/deserialization and simple data binding.

Discovering 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`. This class is an abstract base class that encapsulates a bunch of properties common to all type members. Derived from `MemberInfo` are a bunch of classes; each class encapsulates some more properties related to a specific type member. Figure 23-1 shows the hierarchy of these types.

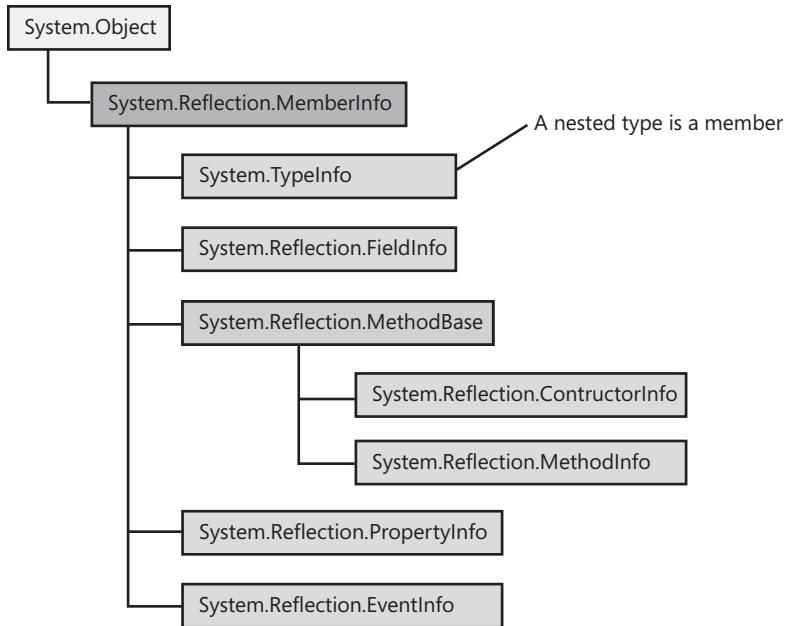


FIGURE 23-1 Hierarchy of the reflection types that encapsulate information about a type's member.

The following program demonstrates how to query a type's members and display some information about them. This code processes all of the public types defined in all assemblies loaded in the calling AppDomain. For each type, the `DeclaredMembers` property is called and returns a collection of `MemberInfo`-derived objects; each object refers to a single member defined within the type. Then, for each member, its kind (field, constructor, method, property, etc.) and its string value (obtained by calling `ToString`) is shown.

```

using System;
using System.Reflection;

public static class Program {
    public static void Main() {
        // Loop through all assemblies loaded in this AppDomain
        Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
        foreach (Assembly a in assemblies) {
            Show(0, "Assembly: {0}", a);

            // Find Types in the assembly
            foreach (Type t in a.ExportedTypes) {
                Show(1, "Type: {0}", t);

                // Discover the type's members
                foreach (MemberInfo mi in t.GetTypeInfo().DeclaredMembers) {
                    String typeName = String.Empty;
                    if (mi is Type) typeName = "(Nested) Type";
                    if (mi is FieldInfo) typeName = "FieldInfo";
                    if (mi is MethodInfo) typeName = "MethodInfo";
                    if (mi is ConstructorInfo) typeName = "ConstructorInfo";

```

```

        if (mi is PropertyInfo) typeName = " PropertyInfo";
        if (mi is EventInfo) typeName = " EventInfo";
        Show(2, "{0}: {1}", typeName, mi);
    }
}
}

private static void Show(Int32 indent, String format, params Object[] args) {
    Console.WriteLine(new String(' ', 3 * indent) + format, args);
}
}

```

When you compile and run this code, a ton of output is produced. Here is a small sampling of what it looks like.

```

Assembly: mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
Type: System.Object
MethodInfo: System.String ToString()
MethodInfo: Boolean Equals(System.Object)
MethodInfo: Boolean Equals(System.Object, System.Object)
MethodInfo: Boolean ReferenceEquals(System.Object, System.Object)
MethodInfo: Int32 GetHashCode()
MethodInfo: System.Type GetType()
MethodInfo: Void Finalize()
MethodInfo: System.Object MemberwiseClone()
MethodInfo: Void FieldSetter(System.String, System.String, System.Object)
MethodInfo: Void FieldGetter(System.String, System.String, System.Object ByRef)
MethodInfo: System.Reflection.FieldInfo GetFieldInfo(System.String, System.String)
ConstructInfo: Void .ctor()
Type: System.Collections.Generic.IComparer`1[T]
MethodInfo: Int32 Compare(T, T)
Type: System.Collections.IEnumerator
MethodInfo: Boolean MoveNext()
MethodInfo: System.Object get_Current()
MethodInfo: Void Reset()
 PropertyInfo: System.Object Current
Type: System.IDisposable
MethodInfo: Void Dispose()
Type: System.Collections.Generic.IEnumerator`1[T]
MethodInfo: T get_Current()
 PropertyInfo: T Current
Type: System.ArraySegment`1[T]
MethodInfo: T[] get_Array()
MethodInfo: Int32 get_Offset()
MethodInfo: Int32 get_Count()
MethodInfo: Int32 GetHashCode()
MethodInfo: Boolean Equals(System.Object)
MethodInfo: Boolean Equals(System.ArraySegment`1[T])
MethodInfo: Boolean op_Equality(System.ArraySegment`1[T], System.ArraySegment`1[T])
MethodInfo: Boolean op_Inequality(System.ArraySegment`1[T], System.ArraySegment`1[T])
ConstructInfo: Void .ctor(T[])
ConstructInfo: Void .ctor(T[], Int32, Int32)
 PropertyInfo: T[] Array
 PropertyInfo: Int32 Offset
 PropertyInfo: Int32 Count
 FieldInfo: T[] _array
 FieldInfo: Int32 _offset

```

Because `MemberInfo` is the root of the member hierarchy, it makes sense for us to discuss it a bit more. Table 23-1 shows several read-only properties and methods offered by the `MemberInfo` class. These properties and methods are common to all members of a type. Don't forget that `System.TypeInfo` is derived from `MemberInfo`, and therefore, `TypeInfo` also offers all of the properties shown in Table 23-1.

TABLE 23-1 Properties and Methods Common to All `MemberInfo`-Derived Types

Member Name	Member Type	Description
Name	String property	Returns the name of the member.
DeclaringType	Type property	Returns the <code>Type</code> that declares the member.
Module	Module property	Returns the <code>Module</code> that declares the member.
CustomAttributes	Property returning <code>IEnumerable<CustomAttributeData></code>	Returns a collection in which each element identifies an instance of a custom attribute applied to this member. Custom attributes can be applied to any member. Even though <code>Assembly</code> does not derive from <code>MemberInfo</code> , it provides the same property that can be used with assemblies.

Each element of the collection returned by querying `DeclaredMembers` is a reference to one of the concrete types in the hierarchy. Although `TypeInfo`'s `DeclaredMembers` property returns all of the type's members, `TypeInfo` also offers methods that return specific member types for a specified string name. For example, `TypeInfo` offers `GetDeclaredNestedType`, `GetDeclaredField`, `GetDeclaredMethod`, `GetDeclaredProperty`, and `GetDeclaredEvent`. These methods all return a reference to a `TypeInfo` object, `FieldInfo` object, `MethodInfo` object, `PropertyInfo` object, or `EventInfo` object, respectively. There is also a `GetDeclaredMethods` method that returns a collection of `MethodInfo` objects describing the methods matching the specified string name.

Figure 23-2 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. Namespaces are not part of this hierarchy because they are simply syntactical gatherings of types. If you want to list all of the namespaces defined in an assembly, you need to enumerate all of the types in this assembly and take a look at their `Namespace` property.

From a type, it is also possible to discover the interfaces it implements. And from a constructor, method, property accessor method, or event add/remove method, you can call the `GetParameters` method to obtain an array of `ParameterInfo` objects, which tells you the types of the member's parameters. You can also query the read-only `ReturnParameter` property to get a `ParameterInfo` object for detailed information about a member's return type. For a generic type or method, you can call the `GetGenericArguments` method to get the set of type parameters. Finally, for any of these items, you can query the `CustomAttributes` property to obtain the set of custom attributes applied to them.

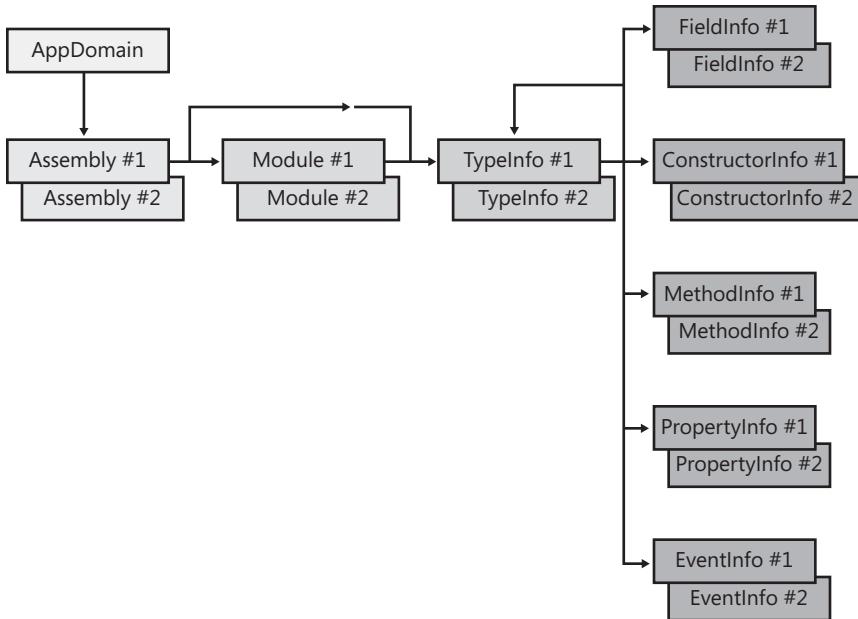


FIGURE 23-2 Types an application uses to walk reflection's object model.

Invoking a Type's Members

Now that you know how to discover the members defined by a type, you may want to invoke one of these members. What invoke means depends on the kind of member being invoked. Table 23-2 shows which method to call for each kind of member to invoke that member.

TABLE 23-2 How to Invoke a Member

Type of Member	Method to Invoke Member
FieldInfo	Call <code>GetValue</code> to get a field's value. Call <code>SetValue</code> to set a field's value.
ConstructorInfo	Call <code>Invoke</code> to construct an instance of the type and call a constructor.
MethodInfo	Call <code>Invoke</code> to call a method of the type.
PropertyInfo	Call <code>GetValue</code> to call a property's get accessor method. Call <code>SetValue</code> to call a property's set accessor method.
EventInfo	Call <code>AddEventHandler</code> to call an event's add accessor method. Call <code>RemoveEventHandler</code> to call an event's remove accessor method.

The `PropertyInfo` type represents metadata information about a property (as discussed in Chapter 10, “Properties”); 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` also has read-only `GetMethod` and `SetMethod` properties, which return `MethodInfo` objects representing the methods that get and set a property's value.

`PropertyInfo`'s `GetValue` and `SetValue` methods exist for convenience; internally, they invoke the appropriate `MethodInfo` object. To support parameterful properties (C# indexers), the `GetValue` and `SetValue` methods offer an `index` parameter of `Object[]` type.

The `EventInfo` type represents metadata information about an event (as discussed in Chapter 11, "Events"). The `EventInfo` type offers a read-only `EventHandlerType` property that returns the Type of the event's underlying delegate. The `EventInfo` type also has read-only `AddMethod` and `RemoveMethod` properties, which return the `MethodInfo` objects corresponding to the methods that add or remove a delegate to/from the event. To add or remove a delegate, you can invoke these `MethodInfo` objects, or you can call `EventInfo`'s more convenient `AddEventHandler` and `RemoveEventHandler` methods.

The following sample application demonstrates the various ways to use reflection to access a type's members. The `SomeType` class represents a type that has various members: a private field (`m_someField`), a public constructor (`SomeType`) that takes an `Int32` argument passed by reference, a public method (`ToString`), a public property (`SomeProp`), and a public event (`SomeEvent`). Having defined the `SomeType` type, I offer three different methods that use reflection to access `SomeType`'s members. Each method uses reflection in a different way to accomplish the same thing.

- The `BindToMemberThenInvokeTheMember` method demonstrates how to bind to a member and invoke it later.
- The `BindToMemberCreateDelegateToMemberThenInvokeTheMember` method demonstrates how to bind to an object or member, and then it creates a delegate that refers to that object or member. Calling through the delegate is very fast, and this technique yields faster performance if you intend to invoke the same member on the same object multiple times.
- The `UseDynamicToBindAndInvokeTheMember` method demonstrates how to use C# dynamic primitive type (discussed at the end of Chapter 5, "Primitive, Reference, and Value Types") to simplify the syntax for accessing members. In addition, this technique can give reasonably good performance if you intend to invoke the same member on different objects that are all of the same type because the binding will happen once per type and be cached so that it can be invoked multiple times quickly. You can also use this technique to invoke a member on objects of different types.

```
using System;
using System.Reflection;
using Microsoft.CSharp.RuntimeBinder;
using System.Linq;

// This class is used to demonstrate reflection
// It has a field, constructor, method, property, and an event
internal sealed class SomeType {
    private Int32 m_someField;
    public SomeType(ref Int32 x) { x *= 2; }
    public override String ToString() { return m_someField.ToString(); }
    public Int32 SomeProp {
```

```

        get { return m_someField; }
        set {
            if (value < 1)
                throw new ArgumentOutOfRangeException("value");
            m_someField = value;
        }
    public event EventHandler SomeEvent;
    private void NoCompilerWarnings() { SomeEvent.ToString(); }
}

public static class Program {
    public static void Main() {
        Type t = typeof(SomeType);
        BindToMemberThenInvokeTheMember(t);
        Console.WriteLine();

        BindToMemberCreateDelegateToMemberThenInvokeTheMember(t);
        Console.WriteLine();

        UseDynamicToBindAndInvokeTheMember(t);
        Console.WriteLine();
    }

    private static void BindToMemberThenInvokeTheMember(Type t) {
        Console.WriteLine("BindToMemberThenInvokeTheMember");

        // Construct an instance
        Type ctorArgument = Type.GetType("System.Int32&"); // or typeof(Int32).MakeByRefType();
        ConstructorInfo ctor = t.GetTypeInfo().DeclaredConstructors.First(
            c => c.GetParameters()[0].ParameterType == ctorArgument);
        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());
        Console.WriteLine("x after constructor returns: " + args[0]);

        // Read and write to a field
        FieldInfo fi = obj.GetType().GetTypeInfo().GetDeclaredField("m_someField");
        fi.SetValue(obj, 33);
        Console.WriteLine("someField: " + fi.GetValue(obj));

        // Call a method
        MethodInfo mi = obj.GetType().GetTypeInfo().GetDeclaredMethod("ToString");
        String s = (String)mi.Invoke(obj, null);
        Console.WriteLine("ToString: " + s);

        // Read and write a property
        PropertyInfo pi = obj.GetType().GetTypeInfo().GetDeclaredProperty("SomeProp");
        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().GetTypeInfo().GetDeclaredEvent("SomeEvent");
    EventHandler eh = new EventHandler(EventCallback); // See ei.EventHandlerType
    ei.AddEventHandler(obj, eh);
    ei.RemoveEventHandler(obj, eh);
}

// Callback method added to the event
private static void EventCallback(Object sender, EventArgs e) { }

private static void BindToMemberCreateDelegateToMemberThenInvokeTheMember(Type t) {
    Console.WriteLine("BindToMemberCreateDelegateToMemberThenInvokeTheMember");

    // Construct an instance (You can't create a delegate to a constructor)
    Object[] args = new Object[] { 12 }; // Constructor arguments
    Console.WriteLine("x before constructor called: " + args[0]);
    Object obj = Activator.CreateInstance(t, args);
    Console.WriteLine("Type: " + obj.GetType().ToString());
    Console.WriteLine("x after constructor returns: " + args[0]);

    // NOTE: You can't create a delegate to a field

    // Call a method
    MethodInfo mi = obj.GetType().GetTypeInfo().GetDeclaredMethod("ToString");
    var toString = mi.CreateDelegate<Func<String>>(obj);
    String s = toString();
    Console.WriteLine("ToString: " + s);

    // Read and write a property
    PropertyInfo pi = obj.GetType().GetTypeInfo().GetDeclaredProperty("SomeProp");
    var setSomeProp = pi.SetMethod.CreateDelegate<Action<Int32>>(obj);
    try {
        setSomeProp(0);
    }
    catch (ArgumentOutOfRangeException) {
        Console.WriteLine("Property set catch.");
    }
    setSomeProp(2);
    var getSomeProp = pi.GetMethod.CreateDelegate<Func<Int32>>(obj);
    Console.WriteLine("SomeProp: " + getSomeProp());

    // Add and remove a delegate from the event
    EventInfo ei = obj.GetType().GetTypeInfo().GetDeclaredEvent("SomeEvent");
    var addSomeEvent = ei.AddMethod.CreateDelegate<Action<EventHandler>>(obj);
    addSomeEvent(EventCallback);
    var removeSomeEvent = ei.RemoveMethod.CreateDelegate<Action<EventHandler>>(obj);
    removeSomeEvent(EventCallback);
}

```

```

private static void UseDynamicToBindAndInvokeTheMember(Type t) {
    Console.WriteLine("UseDynamicToBindAndInvokeTheMember");

    // Construct an instance (You can't use dynamic to call a constructor)
    Object[] args = new Object[] { 12 }; // Constructor arguments
    Console.WriteLine("x before constructor called: " + args[0]);
    dynamic obj = Activator.CreateInstance(t, args);
    Console.WriteLine("Type: " + obj.GetType().ToString());
    Console.WriteLine("x after constructor returns: " + args[0]);

    // Read and write to a field
    try {
        obj.m_someField = 5;
        Int32 v = (Int32)obj.m_someField;
        Console.WriteLine("someField: " + v);
    }
    catch (RuntimeBinderException e) {
        // We get here because the field is private
        Console.WriteLine("Failed to access field: " + e.Message);
    }

    // Call a method
    String s = (String)obj.ToString();
    Console.WriteLine("ToString: " + s);

    // Read and write a property
    try {
        obj.SomeProp = 0;
    }
    catch (ArgumentOutOfRangeException) {
        Console.WriteLine("Property set catch.");
    }
    obj.SomeProp = 2;
    Int32 val = (Int32)obj.SomeProp;
    Console.WriteLine("SomeProp: " + val);

    // Add and remove a delegate from the event
    obj.SomeEvent += new EventHandler(EventCallback);
    obj.SomeEvent -= new EventHandler(EventCallback);
}

internal static class ReflectionExtensions {
    // Helper extension method to simplify syntax to create a delegate
    public static TDelegate CreateDelegate<TDelegate>(this MethodInfo mi, Object target = null) {
        return (TDelegate)(Object)mi.CreateDelegate(typeof(TDelegate), target);
    }
}

```

If you build and run this code, you'll see the following output.

```

BindToMemberThenInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24

```

```

someField: 33
ToString: 33
Property set catch.
SomeProp: 2

BindToMemberCreateDelegateToMemberThenInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
ToString: 0
Property set catch.
SomeProp: 2

UseDynamicToBindAndInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
Failed to access field: 'SomeType.m_someField' is inaccessible due to its protection level
ToString: 0
Property set catch.
SomeProp: 2

```

Notice that `SomeType`'s constructor takes an `Int32` by 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. Near the top of the `BindToMemberThenInvokeTheMember` method, I show how to accomplish this by calling `Type`'s `GetType` method passing in a string of "`System.Int32&`". The ampersand (`&`) in the string allows me to identify a parameter passed by reference. This ampersand is part of the Backus-Naur Form grammar for type names, which you can look up in the FCL documentation. The code also shows how to accomplish the same thing using `Type`'s `MakeByRefType` method.

Using Binding Handles to Reduce Your Process's Memory Consumption

Many applications bind to a bunch of types (`Type` objects) or type members (`MethodInfo`-derived objects) and save these objects in a collection of some sort. Then later, the application searches the collection for a particular object and then invokes this object. This is a fine way of doing things except for one small issue: `Type` and `MethodInfo`-derived objects require a lot of memory. So if an application holds on to too many of these objects and invokes them occasionally, the application's memory consumption increases dramatically, having an adverse effect on the application's performance.

Internally, the CLR has a more compact way of representing this information. The CLR creates these objects for our applications only to make things easier for developers. The CLR doesn't need these big objects itself in order to run. Developers who are saving/caching a lot of `Type` and `MethodInfo`-derived objects can reduce their working set by using run-time handles instead of objects. The FCL defines three runtime handle types (all defined in the `System` namespace): `RuntimeTypeHandle`, `RuntimeFieldHandle`, and `RuntimeMethodHandle`. All of these types are value types that contain just one field, an `IntPtr`; this makes instances of these types cheap (memory-wise). The `IntPtr` field is a handle that refers to a type, field, or method in an `AppDomain`'s loader heap. So what you need now is an easy and efficient way to convert a heavyweight `Type/MethodInfo` object to a lightweight

run-time handle instance and vice versa. Fortunately, this is easy using the following conversion methods and properties:

- To convert a `Type` object to a `RuntimeTypeHandle`, call `Type`'s static `GetTypeHandle` method passing in the reference to the `Type` object.
- To convert a `RuntimeTypeHandle` to a `Type` object, call `Type`'s static `GetTypeFromHandle` method passing in the `RuntimeTypeHandle`.
- To convert a `FieldInfo` object to a `RuntimeFieldHandle`, query `FieldInfo`'s instance read-only `FieldHandle` property.
- To convert a `RuntimeFieldHandle` to a `FieldInfo` object, call `FieldInfo`'s static `GetFieldFromHandle` method.
- To convert a `MethodInfo` object to a `RuntimeMethodHandle`, query `MethodInfo`'s instance read-only `MethodHandle` property.
- To convert a `RuntimeMethodHandle` to a `MethodInfo` object, call `MethodInfo`'s static `GetMethodFromHandle` method.

The following program sample acquires a lot of `MethodInfo` objects, converts them to `RuntimeMethodHandle` instances, and shows the working set difference.

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

public sealed class Program {
    private const BindingFlags c_bf = BindingFlags.FlattenHierarchy | BindingFlags.Instance |
        BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic;

    public static void Main() {
        // Show size of heap before doing any reflection stuff
        Show("Before doing anything");

        // Build cache of MethodInfo objects for all methods in MSCorlib.dll
        List<MethodBase> methodInfos = new List<MethodBase>();
        foreach (Type t in typeof(Object).Assembly.GetExportedTypes()) {
            // Skip over any generic types
            if (t.IsGenericTypeDefinition) continue;

            MethodBase[] mb = t.GetMethods(c_bf);
            methodInfos.AddRange(mb);
        }

        // Show number of methods and size of heap after binding to all methods
        Console.WriteLine("# of methods={0:N0}", methodInfos.Count);
        Show("After building cache of MethodInfo objects");
    }
}
```

```

// Build cache of RuntimeMethodHandles for all MethodInfo objects
List<RuntimeMethodHandle> methodHandles =
    methodInfos.ConvertAll<RuntimeMethodHandle>(mb => mb.MethodHandle);

Show("Holding MethodInfo and RuntimeMethodHandle cache");
GC.KeepAlive(methodInfos); // Prevent cache from being GC'd early

methodInfos = null;           // Allow cache to be GC'd now
Show("After freeing MethodInfo objects");

methodInfos = methodHandles.ConvertAll<MethodBase>(
    rmh=> MethodBase.GetMethodFromHandle(rmh));
Show("Size of heap after re-creating MethodInfo objects");
GC.KeepAlive(methodHandles); // Prevent cache from being GC'd early
GC.KeepAlive(methodInfos); // Prevent cache from being GC'd early

methodHandles = null;         // Allow cache to be GC'd now
methodInfos = null;           // Allow cache to be GC'd now
Show("After freeing MethodInfos and RuntimeMethodHandles");
}
}

```

When I compiled and executed this program, I got the following output.

```

Heap size= 85,000 - Before doing anything
# of methods=48,467
Heap size= 7,065,632 - After building cache of MethodInfo objects
Heap size= 7,453,496 - Holding MethodInfo and RuntimeMethodHandle cache
Heap size= 6,732,704 - After freeing MethodInfo objects
Heap size= 7,372,704 - Size of heap after re-creating MethodInfo objects
Heap size= 192,232 - After freeing MethodInfos and RuntimeMethodHandles

```

Index

Symbols

.cctor methods. *See* type constructors
.NET Framework
 checking for installation of, 8
 extension methods for casting to WinRT, 655
 installing, 8
 migrating older code, 459
 namespace documentation, 101
 reflection API, 583
 stability improvements with, 34
 standardization of, 24
 stream projections, 655
 version, checking, 8
 WinRT interoperability with, 654, 655
.winmd file extension, 644
.winmdobj file extension, 659

A

AbandonedMutexException exception, 785

Abort method, 579, 581

 Thread class, 679

aborting threads, 578

accessibility, 166

accessibility modifiers, 156

accessing resources, 505

accessor methods, 228, 231

 accessibility of, 248

 generic, 248

 for indexers, 244

 inlining, 247

 naming, 245

 performance of, 247

AcquireReaderLock method, 577

AcquireWriterLock method, 577

Action Center. *See* Windows Action Center
Action delegate, 279
Activator class, 594
add-in support, 596
Add methods, arguments for, 236
AddEventHandler method, 603
AddItemHelper method, 501
AddMemoryPressure method, 538
AddParticipant method, 805
AddrOfPinnedObject method, 548
AddToAny methods, 822
AddValue method, 627, 631
administrative control over applications, 62
AggregateException objects, 699
AggregateException type, 702
AIPs (automatically implemented properties), 231
AL.exe, 52, 53
 command-line switches for, 87
 publisher policy assemblies, creating, 86
AllBegun method, 770
AllDone method, 771
Alloc method, 546, 548
ampersands in strings, 608
anonymous function, 410, 411
 body of, 413
 modifiers, unnecessary, 410
anonymous method, referencing instance
 members, 412
anonymous methods, vs. lambda expressions, 413
anonymous types, 237
 declaring properties in, 238
 defining, 239
 implicitly typed arrays and, 377
AnotherHybridLock method, 793

API for .NET Framework

API for .NET Framework, 583
APIs, WinRT, calling, 650
AppDomain type, instance methods, 595
AppDomainManager class, 576
AppDomainMonitorDelta class, 573
AppDomains, 17
 aborting threads, 578
 accessing objects across boundaries, 559
 application output, 562
 ASP.NET applications and, 575
 assemblies in, 557
 assemblies in, discovering, 602
 assemblies using multiple, 558
 callback methods, 573
 confirming thread transitions between, 565
CreateDomain method overloads, 563
creating, 563
default, 556
definition of, 556
demonstrations of, 563
executable applications and, 574
features of, 556
first-chance exception notifications, 573
foreground/background threads and, 688
friendly names, 563
garbage collection, 571
garbage collection handle tables, 545
graceful aborts, 577
instance fields and, 566, 567
isolation provided by, 557, 558, 564, 566
loading assemblies for specific, 585
loading assemblies into, 564
loading assemblies into multiple times, 586
managed code and, 576
marshaling objects by value, 568
marshaling objects by references, 564
memory usage, monitoring, 524
method calls across boundaries, 566
Microsoft Silverlight applications and, 574
monitoring, 571
passing objects across boundaries, 569
permission set inheritance, 563
proxy types, defining, 564
purpose of, 553
read-only properties in, 572
robust host applications, 577
singleton types and, 634
SQL Server and, 575
static members and, 567
thread pools. *See* thread pools
threads and, 562
unloading, 477, 494, 566, 570
unloading, as garbage collection trigger, 519
XML web services applications and, 575
AppDomainSetup object, 563
Append method, 338
AppendChar method, 358
AppendFormat method, 338, 345, 346, 347
AppendLine method, 338
application developers, exception handling
 guidelines, 478, 485
application models, threading models for, 748
application state, serialization/deserialization
 of, 616
application types, 36
ApplicationException type, 466
applications
 add-in support, 596
 administrative control, 62
 AppDomains and, 574
 architecting for events, 250
 compiling, 35
 configuration files, 62
 configuring for assembly download, 45
 dynamically extensible, 583, 588
 installation complexity, 34
 optimizing, 19
 single instance, 781
ApplyPolicy method, 587
.appx files, 60
architecting applications for events, 250
ArgumentNullException exception, 647
arguments
 commas in, 212
 order of evaluation, 210, 211
 passing variable number of, 220
arity, 273
 overloading based on, 286
Array class, 373
array covariance, 379
array initializers, 376
Array type, 380

ArrayEnumerator objects, 278
arrays, 373. *See also* vectors
accessing, 386
accessing elements of, 383
accessing with unsafe code, 386
allocating, 373, 374
allocating on thread stack, 388
casting, 378
casting between types, 379
collection implementation, 381
compiler inferences of, 376
converting, 378
copying, 378, 382
copying elements, 380
covariance, 379
creating, 596
of Decimal values, 383
derivation of, 373, 380
dynamically creating, 383
embedding in structures, 390
implicitly typed, 239, 376, 377
indexes into, 375
initializing elements, 376
inline, 390
internals, 384
jagged, 375
in managed heap, 374
multi-dimensional, 385, 386
non-zero lower bounds, 383
overhead information, 374
passing and returning, 382
of polygons, 375
of reference types, 373
referencing, 376
single-dimensional, 375
static methods for, 380
of tasks, 710
type names, 385
types of, 384
unsafe access, 388
WinRT, 648, 660
zero-based, 374
zero-length, 382
as operator, 96
AsBuffer method, 658
ASCII encoding, 351

ASCIIEncoding object, 353
AsOrdered method, 718
ASP.NET
asynchronous objects, 751
threading model, 748
ASP.NET applications
AppDomains and, 575
processes run by, 687
assemblies
32-bit and 64-bit, 8
administrative control, 62
Assembly Linker and, 52
benefits of, 7
characteristics of, 45
combining modules into, 45
compressing, 74
configuration files, 84
configuring download of, 45
creating from managed modules, 7
culture, 59
data files in, 46
default path to, 38
defined, 6, 45
delayed signing, 76
deploying and packaging, 60
different types in, 46
directories for, 63
discovering types in, 590
double copies of, 75
downloading from web, 586
executing code, 11
files required for, 50
friend, 154
global assembly cache, 72
globally deployed, 67
loading into AppDomains, 564
locating, 63, 83
manifest tables, 47
multifile, 46, 48
namespaces and, 101
platform, specifying, 8
preventing execution of, 587
private deployment of, 67
privately deployed, 61
publisher policy controls, 86
referencing, 37, 51, 70, 74, 98

Assembly Linker

assemblies (*continued*)
registering, 74
resource files, adding, 53
satellite, 59
self-describing nature of, 7
serialization and, 616
signing files, 70
statistics, viewing, 44
strongly named, 66, 67, 68, 72
uninstalling from GAC, 73
unloading, 587
used by multiple AppDomains, 558
verification, 76
verification, skipping, 77
version numbers, 58
version resource attributes, 56
version resources in, 54
versioning issues, 597
versioning policies, 587
Visual Studio IDE and, 51
weakly named, 66, 68
Win32 resources, embedding, 54
Windows installer and, 73
Assembly Linker, 52, 53
assembly loading, 584
callback methods, 587, 588
embedding dependent DLLs, 588
Load method, 585
for metadata analysis, 586
multiple times into AppDomain, 586
for specific AppDomain, 585
for specific architecture, 585
with specific path name, 586
AssemblyDef table, 47
AssemblyRef table, 39
AssemblyResolve event, 588
assembly type, 25, 592
AsSequential method, 718
Assert method, 502, 503
associating data, 550
AsStream method, 658
buffering, 656
Assume method, 502, 503
AsTask method, 653, 654
async, 651
AsyncCoordinator class, 770, 771
asynchronous compute-bound operations, 693. *See* compute-bound asynchronous operations
calling, 693
initiating, 700
Windows scheduler and, 694
asynchronous exceptions, 495
asynchronous flag, 729
asynchronous functions, 732
await operator and, 746
awaiters and, 738
compiler warnings, turning off, 746
debugging, 745
event handlers and, 741
examples of, 735
exception handling and, 744
extensibility of, 738
in Framework Class Library, 742
identifying, 733
initiating from non-caller thread, 745
restrictions on, 734
as state machines, 734
stepping into and out of, 745
Task object and, 733
thread synchronization and, 757
unhandled exceptions and, 745
WebClient and, 744
asynchronous I/O operations, 729
asynchronous functions and, 732
canceling, 751
concurrent function of, 746
debuggers and, 731
event handlers and, 742
FileStream objects and, 753
garbage collection and, 731
GUI responsiveness and, 732
ignoring SynchronizationContext object, 750
incomplete, showing, 738
performance benefits, 731
Task type for, 738
thread priorities, 754
asynchronous lambda expressions, 746
asynchronous operations, 650, 691
asynchronous server implementation, 751
asynchronous thread synchronization, 814
asynchronous WinRT APIs, 649

AsyncOneManyLock class, 816
 Attribute class, 424
 attributes. *See also* custom attributes
 setting, 368
 AttributeUsage attribute, 428
 automatically implemented properties, 231
 auto-reset events, 782, 784
 AutoResetEvent method, 786
 await, 651
 await operator, 652
 awaiters, 738, 740

B

background threads vs. foreground, 688
 backing fields, 230
 Backus-Naur Form grammar for type
 names, 592
 Barrier class, 805
 base-64 encoding and decoding, 356
 Base class, 296, 312
 in WinRT, 645
 base types, vs. interfaces, 312
 BeginBackgroundProcessing method, 755
 BeginWaitForConnection method, 743
 BinaryFormatter object, 614
 binary namespace, 614
 binary operators, 192
 nullable types and, 444
 BinaryReader type, 350
 BinaryWriter type, 350
 binding
 early vs. late, 589
 handles for, 608
 BindToName method, 641
 BindToType method, 641
 Bing maps, 659
 bit flags, 367
 converting numeric values into, 370
 enumerated types for, 368
 numeric equivalents, 369
 setting, 368
 bitmaps, garbage collection of, 538
 BlockCopy method, 380
 blocking collections, 820

blocking threads
 avoiding, 705, 805
 countdown for, 804
 reasons for, 806
 BlockingCollection class, 820
 blocks
 catch, 455, 457, 480, 482
 finally, 455, 456, 478, 497, 535
 try, 454, 497
 bool type, 112
 boxing, nullable types, 447
 boxing value methods, 135
 boxing value types, 124
 changing fields, 136
 efficiency of, 130
 example code, 128
 Break method (Parallel class), 716
 breakpoints, AIPs and, 232
 buffering
 AsStream methods and, 656
 WinRT, 658
 buffering LINQ query results, 720
 byte array method, garbage collection
 and, 548
 byte arrays, 656
 byte order mark (BOM), 353
 byte type, 112

C

C#
 asynchronous functions, 732
 is operator, 95
 optimizing code, 14
 question marks in, 443
 using directive, 98
 WinRT component definitions in, 658
 C# compiler
 event implementation, 256
 language translation done by, 763
 output of, 9
 platform command-line switch, 8, 10
 primitive types and, 114
 referencing types, 35
 response files, 36
 unsafe code, 17

C++ compiler, managed/unmanaged code and

C++ compiler, managed/unmanaged code
and, 6
caching
 threads, 673
 weak references and, 550
callback functions, 391. *See also* delegates
 unmanaged, 391
 uses for, 391
callback methods, 249. *See also* lambda
 method
 for AppDomains, 573
 assembly loading and, 587, 588
 calling, 393
 disposing of, 699
 exceptions thrown by, 698
 invoking, 399
 lambda expressions, 410
 referencing local parameters, 413
 registering after cancellation, 698
 returning values with, 408
 syntax for, 409
calling back static methods, 394
calling methods, 163
 across AppDomain boundaries, 566
 calling type constructors, 188
 callvirt instruction, 163
CAN-DO relationships, 312
Cancel method, 760
canceling operations, 696
 after set time, 700
 cooperatively, 696
 invoking methods after, 698
 preventing, 698
canceling tasks, 704
canceling WinRT APIs, 653
CancellationToken objects
 associating with tasks, 705
 passing as closure variables, 705
CancellationTokenSource objects, 696
 creating, 699
CancellationTokenSource type, 704
CancellationToken value type, 697
CannotUnloadAppDomainException
 exception, 571
carriage returns, 322
casting arrays, 378
casting, Char type to numeric value, 319
casting types
 between objects, 93
 dynamic, 145
 implicit vs. explicit, 114
 with is/as operators, 95
 primitive, 114
catch blocks, 455, 457, 482. *See also* finally
 blocks; try blocks
constrained execution regions and, 475
exceptions thrown by, 457
overuse of, 480
redundant, warning for, 459
stack trace, obtaining, 461
catch types, 455
 variable names after, 456
CCCheck.exe (Code Contract Checker), 503
CCDocGen.exe (Code Contract Document
 Generator), 504
CCRefGen.exe (Code Contract Reference
 Assembly Generator), 504
CCWs (COM Callable Wrappers), 644
CERs. *See* constrained execution regions (CERs)
 chaining delegates, 400, 404
 C# support for, 404
ChainSelector method, 639
char type, 112, 317
 casting to numeric value, 319
 converting numeric types to, 319
 instance methods, 318
 static methods, 318
character arrays, 337
 methods for, 338
characters, 317. *See also* text elements
 converting to bytes, 353
 examining in strings, 333
 expanding, 327
 lowercase/uppercase, changing, 318
 numeric equivalent, returning, 318
checked operators, 116
child tasks, 707
circuit boards, 727
class constraint, 287
class constructors. *See* type constructors
class inheritance. *See* inheritance
class libraries, thread synchronization and, 759

class library developers, exception handling guidelines, 478, 485

classes

- defining, 168, 276
- generic collection, 125
- inheritance, 296
- partial, 159
- as reference types, 119
- sealed, 168
- security, 168
- static, 158
- WinRT, 646

cleanup code, 478. *See also* catch blocks; finally blocks; try blocks; garbage collection

construction of, 479

Dispose method, 535, 537

exceptions thrown by, 457

guaranteeing, 497

for native resources, 533

Clear method, 337

CLI (Common Language Infrastructure), 24

Clone method, 335

cloneable variables, 301

cloning strings, 335

cloning, via serialization, 615

Close method, 648

closed types, 272

CloseHandle method, 780

CLR (Common Language Runtime)

- code language vs. behavior, 25
- code location for, 554
- common language aspect of, 152
- compilers for, 4
- defined, 3
- demonstration of, 104
- field layout, 123
- functions performed by, 471
- headers, 5
- hosting, 554
- interoperability with unmanaged code, 30
- loading, 8, 556
- namespaces and, 98
- nullable type support, 447
- passing blocks of data to WinRT, 656
- setting host managers, 555

shutting down, as garbage collection trigger, 519

type references, resolving, 80

type safety, 95

version, checking, 8

versioning in, 161

WinRT, mapping types to, 659

CLR projections, 645

WinRT component type system rules and, 645

CLRCreatelnstance function, 554

CLRVer.exe, 8

CLS (Common Language Specification), 26

- example code using, 27
- exceptions, 458
- rules overview, 28

Code Contract Checker (CCCheck.exe), 503

Code Contract Document Generator Tool (CCDocGen.exe), 504

Code Contract Reference Assembly Generator (CCRefGen.exe), 504

code contracts, 475, 498

- checking, 503, 504
- Contract class, 498
- defining, 501
- exceptions, 500
- runtime function, 502
- tools for maximizing, 499
- turning on features for, 500
- violation of, 500

code explosion, 183, 277

code optimization, 14, 763

- Volatile class and, 765

code pages, specifying, 352

code, program order, 765

code reuse. *See* generics

codeBase elements, 45, 84

Collect method, 519, 522, 523

collection classes

- generic, 125, 270
- thread-safe, 818

collection initializers, 235

collection objects, serializing/deserializing, 622

collection types, WinRT, 648

CollectionCount method, 524

collections

collections, 260
 adding items to, 236
 array implementation of, 381
 maximum number in, 821
 non-blocking, into blocking, 820
 producer/consumer scenario, 821
 WinRT, 648
ColorMatrix class, 244
COM Callable Wrappers (CCWs), 644
COM components, WinRT components as, 644
Combine method, 400
command-line switches, response files, 36
Common Language Specification. *See* CLS
 (Common Language Specification)
Common Type System. *See* CTS (Common Type System)
compacting memory in garbage collection, 509, 521
Compare method, comparing strings, 325
CompareExchange method, 769, 776, 810
CompareInfo
 class, methods in, 329
 object, 326, 327
 property, 327
CompareOptions type, 324
CompareTo method, 135, 311, 318
comparing strings, 323
comparisonType argument, 324
CompilationRelaxations attribute, 330
CompilerGenerated attribute, 410
compilers
 code optimization by, 763
 functions performed by, 470
 managed, exception handling by, 492
 compiling source code, 3
CompleteAdding method, 821
Component Software Processing (CSP), 160
components
 overview of, 160
 servicing, 161
composability of object-oriented
 programming, 470
compressing, assemblies, 74
compute-bound asynchronous operations, with
 threads, 678
compute-bound operations, 691, 693
canceling, 696
canceling after set time, 700
cooperative cancellation, 696
initiating, 700
periodic, 720
preventing cancellation of, 698
ComputeResult method, 770
Concat method, 129, 222
concatenating strings, 322
ConcurrentBag class, 818
concurrent classes, 818
ConcurrentDictionary class, 818
ConcurrentExclusiveSchedulerPair class, 815
ConcurrentQueue class, 818
ConcurrentStack class, 818
condition variable pattern, 811
conditional attribute classes, 440
ConditionalWeakTable class, 551
configuration files, 62, 84
 components of, 84
 locating, 64
 publisher policy controls and, 86
 as separate assembly files, 87
ConfigureAwait method, 750
Console class, 345, 759
constants
 constraints on, 175
 defined, 151, 175
 metadata and, 175
 for non-primitive types, 175
const-ness, unsupported, 224
ConstrainedCopy method, 380
ConstrainedExecution namespace, 528
constrained execution regions (CERs), 475, 494
constraints
 constructor, 289
 generics and, 285
 primary, 287
 reference types as, 287
 secondary, 288
 on type parameters, 287
constructing type instances, 594
constructor constraints, 289

constructor method, 43, 181, 184, 187, 396
calling default, 186
creating types without, 182
default, 181
defining multiple, 182
defining on value types, 185
for delegates, 398
inheritance, 181
parameterless, 186
virtual methods and, 182
`ConstructorInfo` object, 595
constructors
defining, 595
for custom attribute classes, 428
named parameters, 424
parameterless, 235
positional parameters, 424
for serialization/deserialization, 628
Task objects and, 701
thread, 679
context structures, 683
context switching, 672, 683
in asynchronous synchronization, 814
avoiding, 673
I/O operations and, 731
performance issues, 672, 673
contexts, thread, 671
`ContinueWhenAll` method, 710
`ContinueWith` method, 705, 707, 729, 733
`Contract` class, 475
contract reference assemblies, 504
`ContractFailed` event, 500
`ContractInvariantMethod` attribute, 502
`ContractReferenceAssembly` attribute, 504
contra-variant reference types, 395
contra-variant type parameters, 279
 in and out keywords, 280
convenient syntax, 183
conversion constructors, 195
conversion operator methods, 195
 constructing, 197
 implicit and explicit, 196, 197
 metadata for, 197
 parameters for, 197
 support for, 198
conversion operators, defined, 152
Convert class, 628
Convert method, 355
Convert type, 319, 356
`ConvertToList` method, 289
cooperative cancellation, 696
 unhandled exceptions and, 698
`Copy` method, 335
 `Array` class, 378, 382
 vs. `BlockCopy` method, 380
copy operation, and unboxing types, 127
copying
 arrays, 378, 382
 strings, 335
`CopyTo` method, 335, 338, 658
`CoreClr.dll`, 574
cores, multiple. *See* multiple cores
`CorFlags.exe`, 9
corrupted
 memory, 506
 object state, avoiding, 228
 state exceptions (CSEs), 489
corruption. *See* state corruption
`CountdownEvent` class, 804
`Counter` method, 393, 394
 delegate chaining and, 403
country identifiers, 326
covariant
 in and out keywords, 280
 reference types, 395
 type parameters, 279
CPU virtualization, 670. *See also* threads
CPUs. *See also* multiple cores
 aggressive utilization of, 682
 analyzing usage of, 682
 heat dissipation and, 677, 681
 high usage, as good thing, 688
 hyperthreaded, 677
 low usage of, 691
 multiple cores, 677
 power utilization, optimal, 681
 trends for, 677
`CreateDelegate` method, 417
`CreateDomain` method, `AppDomain` overloads
 of, 563
`CreateEvent` method, 780

CreateFile method, synchronous execution of

CreateFile method, synchronous execution of, 752
CreateInstanceAndUnwrap method, 564
CreateInstanceFrom method, 595
CreateInstance method, 383
 Activator class, 594
 AppDomain class, 595
 Array class, 596
CreateMemoryResource notification, 519
CreateMutex method, 780
CreateProcess function, 557
CreateSemaphore method, 780
critical regions, 577
CriticalFinalizer object, 528
CriticalHandle class, 531
CSC.exe, 36
CSC.rsp, 36
CSP containers for public/private keys, 78
CTS (Common Type System), 24
CUI (Console User Interface) applications, 36
culture setting in assemblies, 59
CultureInfo class, 318
CultureInfo type, 326, 342
CurrentCulture property, 318, 326
CurrentUICulture property, 326
custom attributes, 421. *See also* attributes
 applying multiple times, 427
 attribute objects, constructing, 431
 AttributeUsage attribute, 428
 class inheritance, 425, 428
 conditional attribute classes, 440
 constructors, 428
 creating, 425
 C# syntax for, 422
 data types for, 428
 defining attribute class, 425
 defining classes for, 428
 definition of, 424
 derivation of, 424
 detecting in code, 430
 detecting in code, without executing, 437
 detecting specific, 432
 discovering, 602
 enumerated types, 427
 examples of, 422
 as logical state containers, 425
 matching against each other, 434
 multiple, applying, 424
 named parameters, 424
 positional parameters, 424
 prefixes, 423
 public constructors, 427
 security concerns, 437
 serialization and, 429
 suffix, omitting, 424, 425
 syntax for, 424
 targets for, 423
 Visual Basic .NET syntax for, 423
CustomAttributeData class, 437
CustomAttributeExtensions class, 430

D

data collections, LINQ queries on, 717
data encapsulation, 228
data types, for custom attribute classes, 428.
 See types
DataContractSerializer class, 612
datatips, 339
date formatting, 341
dates and times, in WinRT, 647
DateTime
 structure, 647
 type, 349
DateTimeFormatInfo type, 343
DateTimeOffset structure, 647
deadlocks, 750, 761
debuggable attribute, 511
debuggers
 generics and, 271
 asynchronous I/O operations and, 731
 threads and, 673
debugging
 and garbage collections, 510
 asynchronous functions, 745
 exceptions, 490
decimal type, 112, 118
declaring delegates, 392
declaring methods, for variable number of
 arguments, 220
declaring properties, 231
 in anonymous types, 238

Decoder class, 356
 Decoding
 bytes, 356
 strings, 356
 Decrement method, 791
 decrypting SecureStrings, 360
 dedicated threads, 679
 default AppDomain, 556
 default properties. *See parameterful properties*
 DefaultMember attribute, 246
 DefaultValue attribute, 212
 defining
 classes, 168
 fields, without initialization, 184
 properties, 230
 types, 251
 definition tables in metadata, 39
 Delay method, 722
 delayed signing, 76
 Delegate class, 397
 delegate fields, 257
 delegate types
 extension methods for, 202
 serializing, 618
 WinRT, 647
 delegates, 249. *See also* callback functions
 adding/removing, 604
 arrays, returning each reference in, 405
 callback method, invoking, 399
 calling back instance methods, 395
 calling back static methods with, 394
 calling private members with, 394
 chaining, 400, 404
 chaining, C# support for, 404
 chaining, result variable and, 404
 class defined by, 396, 397
 constructors, 398
 contravariant type parameters, 279
 covariant type parameters, 279
 creating, 392
 defining, 408
 definitions for, 407
 extension methods and, 203
 garbage collection, 401
 generic, 278, 407
 initialization of, 398
 invariant type parameters, 279
 invoking dynamically, 417
 MulticastDelegate class, 397
 reflection and, 416
 removing from chains, 403
 syntax, 408
 syntax shortcuts for, 409, 410, 413
 for tasks, passing, 715
 unregistering, 260
 Delete method, exceptions thrown by, 532
 delete operator, 93
 DeleteAsync method, 742
 deploying assemblies, 60
 privately, 79
 derived class, 311, 312
 derived types, preconditions and, 502
 deserialization
 of application state, 616
 of collection objects, 622
 controlling, 619
 controlling data, 624
 definition of, 611
 formatter consistency, 615
 ISerializable interface, 625
 matching serialization method, 628
 into object graphs, 615
 of objects, as different objects, 633
 overriding assembly/type, 640
 quick start for, 613
 setting up, 628
 of singleton types, 633
 streaming contexts for, 631
 surrogates for, 636
 type bindings, 641
 of type instances, 624
 Deserialize method, 635
 DeserializeFromStream method, 615
 deserializing objects, 182
 diagnostics namespace, 688
 dictionary type, serialization, 626
 direct memory accesses, 386
 DisableOptimizations flag, 511
 disambiguating types, 99
 DispatcherTimer classes, 723
 DispatcherUnhandledException event, 488
 Display method, 283

Dispose method

Dispose method, 260, 358, 512, 533, 648, 699
 avoiding, 535
 calling explicitly, 537
 in finally block, 535
 native resource cleanup and, 534
 on Task objects, 708
 WaitHandle class, 780
distributed applications, unhandled exceptions
 in, 488
DIIIMain method, 671
DLLs
 interoperability of, 34
 performance issues and, 672
 thread-attach and thread-detach
 notifications, 671
double-check locking, 807
double type, 112
downloading assemblies from web, 586
DumpBin.exe, 9
dynamic expressions, as object types, 147
dynamic keyword, 214
dynamic type, 112, 144
 example of, 145
 vs. var keyword, 146
dynamically extensible applications, 583
 building, 588
 Managed Extensibility Framework (MEF), 598
DynamicAttribute method, 145
DynamicInvoke method, 417
dynamification, 147

EnsureCapacity method, 337
EnsureInitialized method, 811
EnsureSufficientExecutionStack method, 489
Enter method, 790
 overloading, discouraged, 476
 Monitor class, 795, 797
Enum class, 362
Enum type, 364
enumerated type, 361
 bit flags, defining with, 368
 classes derived from, 362
 converting symbols into, 366
 converting values into flags, 370
 declaring, 363
 defined symbols in, 370
 defining, 367
 displaying all values for, 365
 extension methods for, 202
 formatting, 364
 IsDefined method and, 371
 mapping to strings, 364
 methods, adding, 371
 with multiple symbols, 364
 parameter validation, 366
 as primitive types, 363
 serializing, 618
 structure of, 362
 symbols as constant fields, 362, 363
 underlying types for, 363
 user interface display of, 365
enumerating types, 588
enumeration types, in WinRT, 646
ephemeral garbage collector. *See garbage collection*
epilogue code, 102
equality operators, nullable types and, 444
Equals method, 139, 238, 332, 338, 355
 overriding, 140, 141
 properties of, 141
 System.Object, 92
error messages, 157
errors. *See exceptions; exception handling*
escalation policy, 577
event handlers
 asynchronous functions and, 741
 class naming convention, 251

E

early binding, 589
encoding
 explicitly constructing types, 353
 methods, 355
 stateless, 356
Encoding class, 352
 properties, 352
encodings, 350
 property values, 354
EndBackgroundProcessing, 755
EndsWith method, comparing strings, 325
EndWaitForConnection method, 743
EndXxx method, 481

code contract violations and, 500
method signatures, 741
void return type, 253
event log, viewing unhandled exceptions in, 485
EventArgs class, 251
EventAwaiter class, 741
EventDef table, 39
EventHandler delegate type (WinRT), 647
EventHandlersStore type, 262
EventInfo type, 604
events, 249, 782
 _add prefix, 257
 architecting applications for, 250
 auto-reset, 784
 compiler implementation of, 256
 defined, 24, 152
 delegates, 249
 delegates, adding/removing, 604
 designing types to expose, 250
 explicitly implementing, 260
 members, defining, 252
 metadata on, 604
 notification for, 253
 notifications, unregistering, 260
 Object type, 252
 as primitive kernel-mode thread synchronization constructs, 778
 raising, in thread-safe way, 254
 types for listening, 258
 typing the return parameter, 253
 typing the sender parameter, 252
 unregistering, 260
 unregistering interest from objects, 257
 WinRT, 647
EventSet class, 262
EventWaitHandle class, 780
Exception class, 460
exception-derived types, hierarchy of, 593
exception handling
 assuming exceptions will occur, 470
 asynchronous functions and, 744
 backing out of partial operations, 482
 best practices, 478
 catch blocks, 455, 480
for class library developers vs. application developers, 478, 485
CLS objects, 458
defined, 453
differing implementations of, 492
example code for, 453
exceptions thrown by, 457
finally blocks, 456, 478
guidelines, 478, 485
for high error rate methods, 493
language for, 457
localizing string messages, 467
managed compilers and, 492
mechanics of, 453
migrating code for, 459
non-CLS objects, 458
notifications for, 456
performance issues, 492
problems with, 477
programming culture and, 473
reflection and, 485
Reliability Monitor problem signatures, 487
reliability vs. productivity, 469
resetting exception starting point, 462
stack tracing, 461, 462, 463
state corruption and, 474
state corruption, mitigating, 475
System.Exception type, 460
testing with Performance Monitor, 493
thread synchronization locks and, 476
try blocks, 454
TryParse method and, 494
viewing exception objects, 455
Windows Error Reporting, 488
Exception objects, avoiding throwing, 466
exceptions
 adding types to Visual Studio debugger, 491
 asynchronous, 495
 catching all, when to, 482
 catching specific, when to, 483
 class derivation of, 469
 for code contract violations, 500
 in critical regions, 577
 debugging, 490
 defined, 452
 defining custom classes for, 467

Exceptions dialog box (Visual Studio)

exceptions (*continued*)
designing for, 453
FCL-defined classes, 463
locating source of, 484
maintaining method contracts after, 483
potential sources of, 471
processing, 573
recovering gracefully from, 481
re-throwing, 480, 482
re-throwing different exception, 483
serializable, 467
System.Exception type properties, 460
throwing, 466
types, 463, 466
unhandled, 474, 480, 485, 488
versioning issues, 466
WinRT and, 647
Exceptions dialog box (Visual Studio), 490
Exchange method, 769
executable applications, AppDomains and, 574
executable files. *See also* PE (portable executable) files
32-bit vs. 64-bit, 9
compiling, 35
ExecuteCodeWithGuaranteedCleanup
method, 497
ExecuteDbDataReaderAsync method, 742
ExecuteNonQueryAsync method, 742
ExecuteReaderAsync method, 742
ExecuteScalarAsync method, 742
ExecuteXmlReaderAsync method, 742
executing assembly code, 11
execution context, 694
controlling, 695
ExecutionContext class, 695
helper threads and, 695
performance issues and, 695
suppressing, 695
Exit method, 574
Monitor class, 795, 813
expanding characters, 327
ExpandoObject class, 242
explicit interface method implementations
(EIMIs), 303, 308, 310
explicit methods, state corruption and, 476
explicit type casts, 114
explicitly implementing events, 260
ExportedTypesDef table, 47
exposing events, designing types for, 250
exposing state, 168
expressions, casting to dynamic, 145
Extensible Storage Engine, 477
extension attribute, 203
extension methods, 198, 201, 362
advantages of, 199
declaring, 200
defining, 255
for delegate types, 202
displaying list of, 200
for enumerated types, 202
for interface types, 202
LINQ technology, 202
multiple, avoiding, 201
for passing data between CLR and
WinRT, 657
performance issues, 201
rules and guidelines for, 200
versioning and, 201
ExtensionAttribute class, 203

F

factories, task, 709
FailFast method, 477, 480
Family and Assembly accessibility, 157
FaxMsg method, 259
FCL (Framework Class Library), 22
exceptions defined by, 463
generics in, 270
hybrid constructs in, 793
thread safety, 759
unhandled exception documentation, 488
WinRT supported types, 645
FCL types
vs. primitive types, 113
with corresponding primitive types, 112
FeedbackToConsole method, 394
FeedbackToFile method, 396
FeedbackToMessageBox method, 394
field layout for types, 123
FieldDef table, 39
FieldGetter method, 566

FieldInfo objects, converting to
 RunTimeFieldHandle, 609

fields

- backing, 230
- in boxed value types, changing, 136
- defined, 24, 151, 177
- defining, 178
- defining, without initialization, 184
- encapsulating, 228
- initializing, 182, 187
- initializing, with convenient syntax, 183
- inline initialization, 179
- modifiers for, 177
- private, 168, 228, 231

FieldSetter method, 566

file extensions

- for WinRT components, 644
- .winmdobj, 659

file names, WinRT, 645

file versioning, difficulties with, 66

FileAttributes type, 367

FileDef table, 47, 81

FileNotFoundException exception, 584

FileStream

- class, 532
- I/O operations and, 728
- objects, asynchronous and synchronous operations with, 753
- type, 525, 537

finalization, 525, 542

- garbage collection and, 544
- list, 542
- freachable queue and, 544

Finalize method, 525, 526, 542

- CLR thread dedicated to, 544
- inherited, 543
- memory leaks and, 526
- overriding, 479
- similarity to destructor, 526
- System.Object, 92

finally blocks, 455, 456. *See also* catch blocks; try blocks

as callback methods, 497

constrained execution regions and, 475

Dispose method in, 535

exceptions thrown by, 457

liberal use of, 478

try blocks and, 457

FirstChanceException event, 573

fixed statement, 548

Flags attribute, 368

FlagsAttribute class, 430

Flatten method, 703

float type, 112

FlushAsync method, 742

For method (Parallel class), 713, 714

ForAll method, 718

forcing garbage collection, 522, 539

ForEach method (Parallel class), 713, 714

foreground threads

- forcing, 679
- vs. background, 688

Format method, 344, 364

- overloading, 344
- vs. ToString method, 364

FormatterConverter type, 628

FormatterServices type, 623, 628

Formatting, strings, 346

Forms namespace, Timer class, 723

Framework Class Library. *See* FCL (Framework Class Library)

- asynchronous functions in, 742
- types. *See* FCL types

framework projections, 645, 649

freachable queue, 543

- finalization list and, 544
- naming of, 544

Free method, 548

friend assemblies, 154

- modules in, 156

friendly names, 563

FromAsync method, 743

FromBase64String method, 356

FromCurrentSynchronizationContext

- method, 711

Func delegate, 279, 408

functions, callback. *See* callback functions

FxCopCmd.exe, 480

GAC (global assembly cache)

G

GAC (global assembly cache), 72
installing into, 72
permissions required for, 73
uninstalling assemblies from, 73
GACUtil.exe, 72
Windows install switch, 73
garbage collection, 507
algorithm, 507
asynchronous I/O operations and, 731
for bitmaps, 538
byte array method and, 548
Collect method, 523
compacting phase, 509
concurrent vs. non-concurrent mode, 521
debugging and, 510
dependency issues, 537
Dispose method, 512
vs. Dispose method, 535
finalized objects and, 544
Finalize method and, 526
forcing, 522, 539
generational, 513
generational, assumptions made by, 514
generations and, 513
handle tables, 545, 547
idle threads and, 517
large objects and, 519
latency modes, 522
lease managers, 567
marking phase, 508
memory budgeting and, 517
memory compacting, 521
memory usage, monitoring, 524
monitoring, 524
native resources and, 538
notifications for, 518, 523
object lifetime and, 510
object lifetime, controlling manually, 545
OutOfMemory exceptions, 510
performance goals, 515
pinning objects, 548

reference counting algorithms, 508
resurrected objects, 544
roots and, 509
server mode, 520
symbols for, 522
threads and, 673
Timer objects and, 721
triggers for, 519
for types wrapping native resources, 525
on unloaded AppDomains, 571
weak references and, 550
workstation mode, 520
GCMode enumerated type, 523
GCHandle type, 545
explicit, 548
GCHandleType type, 546
GCLatencyMode property, 521
GCNotification class, 518
GCSettings class, 521
GCWatch method, 551
generation, 1, 2, 513, 514, 516
growth of, 516
promotions to, 515
generic
arguments, limiting types, 285
collection classes, 125, 270
contravariant and covariant type
parameters, 281
delegates, 278, 407
interfaces, 277, 278, 297, 303
list algorithm, 266
namespace, 313
vs. non-generic versions, 304
type identity, 275
type parameters, 305
generic methods, 265, 281
example of, 282
overriding, 286
static, 270
type inference and, 283
generic type variables
as operands, 292
casting, 290

comparing to each other, 292
comparing with null, 291
setting to default value, 291
generic types, 265
 constructing, 596
 defining, 273
 inheritance and, 274
 out and ref parameters, 282
parameters, 266
reference, 269
type arguments, 266
type objects for, 272
using directive and, 276
generics
 backwards compatibility, 270
 benefits of, 267
 constraints, 285
 explicit declarations, 281
 in and out keywords, 281
 in FCL, 270
 infrastructure of, 271
 interface constraints and, 305
 members and, 284
 overview of, 265
 performance gain with, 267, 268
 verifiability, 284, 290
 variance and, 280
GetAssemblyName method, 586
GetAsync method, 742
GetAvailableThreads method, 724
GetAwaiter method, 652, 738, 740
GetBaseException method, 703
GetByteArrayAsync method, 742, 770
GetBytes method, 353, 356, 658
GetCharCount method, 353, 356
GetChars method, 356
GetCustomAttribute method, 430, 431, 437
GetDecoder method, 356
GetDomain method, 562
GetEncoder method, 356
GetEncoding method, 352
GetEnumerator method, 820
GetEnumUnderlyingType method, 363
GetEnumValues method, 364
GetFileAsync method, 650
GetFormat method, 342, 347
GetGenericArguments method, 602
GetHashCode method, 142, 164, 238, 355
 algorithms for, 143
 overriding, 142
 System.Object, 92
GetHttp method, 750
GetInvocationList method, 405
GetLowerBound method, 383
GetMaxByteCount method, 353
GetMaxCharCount method, 353
GetMaxThreads method, 724
GetMetaObject method, 148
GetMinThreads method, 724
GetNumericValue method, 318
GetObjectData method, 623, 626, 630, 635
 ISerializationSurrogate interface, 637
GetParameters method, 602
GetPreamble method, 355
GetRealObject method, 635
GetRequestStreamAsync method, 742
GetResponseAsync method, 742
GetResult method, 738
GetRuntime function, 554
GetSingleton method, 809
GetStreamAsync method, 742
GetString method, 355
GetTextElementEnumerator method, 333
GetTotalMemory method, 524
GetType method, 93, 135, 591, 608
 calling via nullable type, 448
 overloads, 591
 System.Object, 92
GetTypeInfo method, 592
GetUnderlyingType method, 363
GetUnicodeCategory method, 317
GetUninitializedObject method, 624, 628, 637
GetUpperBound method, 383
GetValue method, 364, 603
GetVersionInfo method, 54
GetWindowsRuntimeBuffer method, 658

global assembly cache (GAC)

global assembly cache (GAC). *See* GAC (global assembly cache)
globally deployed assemblies, 67, 72
GlobalMemoryStatusEx function, 550
graceful aborts, 577
GUI (graphical user interface) applications, 36

H

handle-recycling exploits, 531
handle tables, 545, 547
HandleCollector class, 539
handles, 529, 531
HasFlag method, 368
hash codes, 142
 algorithms for, 143
 persisting values, avoiding, 143
 read-only property for, 238
hashes, 70
helper types, 169
hidden files
 determining, 368
 setting attribute for, 368
High priority class, 685
high surrogate code value, 333
host applications, retrieving threads, 578
Host SDK assemblies, 597
hosting, 553
 CLR (Common Language Runtime), 554
hybrid constructs, 761, 789, 805
 Barrier class, 805
 CountdownEvent class, 804
 in Framework Class Library (FCL), 793
 hybrid locks, 790
 ManualResetEventSlim class, 794
 Monitor class, 794
 OneManyLock class, 802
 ReaderWriterLockSlim class, 800
 SemaphoreSlim class, 794
 sync blocks, 794
hybrid locks, 790
 performance issues, 790
 performance of, 793
 with spinning, ownership, and recursion, 791
hyperthreaded CPUs, 677

I

IAsyncAction interfaces, 651
IAsyncOperation interface, 651, 652
IAsyncXxx method, conversion into Task method, 654
IAsyncXxxWithProgress interfaces, 651
IBuffer interface, 656
IBufferByteAccess interface, 656, 658
IClosable interface, 648
ICLRLMetaHost interface, 554
ICollection interface, arrays and, 381
IComparable interface, 298, 308
IConvertible interface, 319
IConvertible method, 310
IDisposable interface, 358, 533, 534, 648, 708
 BlockingCollection class and, 821
Idle priority class, 685
Idle thread priority level, 687
IEnumerable interface, arrays and, 381
IFormatProvider
 interface, 340, 341, 343, 344
 parameter, 348
IFormattable interface, 340, 344
IFormatter interface, 614, 632
ignoring response files, 38
IInputStream interface, 655, 657
IL
 code defined, 5
 converting into CPU instructions, 104
 intellectual property and, 19
 verification and, 16
ILDasm.exe, 40, 259, 659
IList interface, arrays and, 381
implicit type casts, 114
implicitly typed arrays, 239, 376
 with anonymous types, 377
implicitly typed local variables, 237
 arrays and, 377
incomplete asynchronous operations,
 displaying, 738
Increment method, 791
IndexerName attribute, 245
indexers. *See* parameterful properties
 accessing, 245
 accessor methods, 244

compiling, 244
 naming, 245
 parameters for, 244
 IndexOf method, 199, 329
 INEFFICIENT symbol, 132
 inherit parameter, 432
 inheritance, 295

- for interfaces, 297
- of interfaces, 298
- multiple, 295
- Object class and, 296
- of preconditions, 502

 inherited member conflicts, 170
 InitializeLifetimeServices method, 567
 initializers, 235
 initializing

- collections, 235
- objects, 106, 235

 initializing fields, 182, 187

- inline, 179
- with convenient syntax, 183

 inline arrays, 390
 inline field initialization, 179
 inlining code, 247
 InnerExceptions collection, 699
 Insert method, 338
 InsertAt method, 358
 instance constructors, 181, 184

- calling explicitly, 185
- creating types without, 182
- default, 181
- defined, 151
- defining multiple, 182
- inheritance, 181
- virtual methods and, 182

 instance fields

- AppDomains and, 566
- performance and, 567
- serializing, 619

 instance methods, 163, 595

- calling, 199
- calling back, with delegates, 395
- for Char type, 318
- thread safety, 760
- volatile keyword, 767

 int type, 112

 Int32 field, in Task objects, 708
 Int32 variables, static methods for, 768
 intellectual property

- IL code and, 19
- NGen.exe and, 20

 IntelliSense, 200
 interface keyword, 297
 interface methods

- calling, 302
- calling with nullable types, 448
- strongly typed, 304
- for value types, 319

 interface types, constraints for, 288
 interfaces

- vs. base types, 312
- boxing and, 306, 308
- collection-related, 313
- constraining generic type parameters to, 305
- defined, 296
- examples of, 297
- generic, 297, 303
- generic vs. non-generic versions, 304
- inheritance, 297, 298
- items comprising, 296
- methods, calling, 300
- multiple implementations of, 305
- naming conventions, 297
- non-generic, 308
- as parameter types, 306
- public methods, 299
- with same methods, 307
- static members and, 297
- String type and, 300
- type parameters, 305
- type safety and, 309
- value types and, 301
- variables of, 300
- versioning, 313
- virtual methods and, 299
- visibility/accessibility, 297
- WinRT, 646

 Interlocked class, 768, 769, 773, 776, 803, 805, 810
 interlocked constructs, 762, 768
 Intermediate Disassembler utility.
See ILDasm.exe

InternalsVisibleTo attribute

InternalsVisibleTo attribute, 155
InteropServices namespace, 424, 531
InvalidOperationException exception, 587
InvariantCulture property, 342
invariant type parameters, 279
Invoke method, 396, 403, 595, 603
 Parallel class, 714
invoking type members, 603
I/O Completion Ports, 731
I/O operations, 727
 asynchronous, 729, 731, 746
 canceling, 751
 context switching and, 731
 event handlers and, 742
 ignoring SynchronizationContext object, 750
IRPs in, 728
synchronous, 752
thread blocking and, 731
thread priorities, 754
wasted resources and, 728
I/O Request Packets (IRPs), 728
IOException exception, 532
IOTaskScheduler, 712
IOutputStream interface, 655
IProducerConsumerCollection interface, 820
IRandomAccessStream interface, 655, 657
is operator, 95
IS-A relationship, 312
IsCancellationRequested property, 698
IsDefined method, 366, 367, 430, 431
 for bit flag-enumerated types, 371
IsEnumDefined method, 366
ISerializable interface, 625, 629, 630
 defining type for, 630
 special constructor, 629
ISerializationSurrogate
 interface, 640
 type, 637
IsInterned method, 330
IsPrefix method, 329
IsServerGC property, 521
IssueClientRequestAsync method, 733, 734, 747
IsSuffix method, 329
IsTransparentProxy method, 565, 569
ISurrogateSelector interface, 639

J

jagged arrays, 375
JIT compiler, 12
 calling methods, 106
 optimizing, 15
 performance hit from, 14, 16
 unsafe code and, 18
JITCompiler function, 12
Join method, 680
JustEnded method, 771

K

kernel constructs, when to use, 806
kernel-mode constructs, 760, 761, 778, 784
 vs. user-mode constructs, 778, 784
kernel-mode stacks, 671
kernel objects, 671
key/value pairs, hash codes, 142
keywords, vs. type names, 113

L

lambda expression, 410, 411
 vs. anonymous methods, 413
 appropriate use of, 416
 asynchronous, 746
 benefit of, 413
 in separate method, 414
language identifiers, 326
language settings for assemblies, 59
large objects, 519
LastIndexOf method, 329
late binding, 589
Lazy class, 810
lazy initialization, 807
LazyInitializer class, 811
LazyThreadSafetyMode flags, 811
Idstr instruction, 321
lease managers, 567
Leave method, 790
letter casing, 318
LimitedConcurrencyLevelTaskScheduler, 712

linked lists
 data types in, 274
 inheritance and, 274

LINQ queries
 buffering and merging, 720
 parallel processing, 719
 preserving operation order, 718
 processing, 719
 processing data from, 718
 sequential and parallel, 717
 stopping prematurely, 719

LINQ technology, 202
 namespace, 717
 anonymous types and, 239
 Parallel language integrated query, 717
 querying data collections with, 717
 standard operators, 717

LINQ to Objects queries, 717

List class (generic), 266

Listeners, adding delegates to list, 257

literal
 strings. *See* strings
 types, 114

livelocks, 761

Load method (AppDomain class), 585

Load method (Assembly class), 585, 616
 overloads of, 584
 serialization and, 617

LoadFile method, 586

LoadFrom method, 586
 serialization and, 617

loading assemblies, from URLs, 586.
See also assembly loading

LoadLibrary function, 10

local parameters, callback method references
 to, 413

local variables, implicitly typed, 212

locating assemblies, 63

lock, 799
 scalability issues, 814

locking threads, double-check locking, 807

lockTaken variable, 799

long type, 112, 113

low surrogate code value, 333

lowercase/uppercase, converting between, 318

LowLatency garbage collection mode, 521

M

Machine.config file, 64

Main method, 43, 764

maintaining state, 482

MakeByRefType method, 608

MakeGenericType method, 596

managed applications, prioritization of, 687

managed code. *See* IL code
 AppDomains and, 576
 C++ compiler and, 6
 escalation policy, 577
 performance concerns, 15
 performance of, vs. unmanaged, 15
 reducing processes with, 17
 state corruption and, 477
 WinRT components as, 660

Managed Debugging Assistants (MDAs), 538

Managed Extensibility Framework (MEF), 598

managed heap
 allocating resources from, 506
 arrays in, 374
 example of, 507
 garbage collection and, 507

managed modules, 4
 combining into assemblies, 7
 components of, 5
 metadata in, 5

manifest tables, 47, 49

ManifestResourceDef table, 47

manual-reset events, 782

ManualResetEventSlim class, 794, 804

marking objects in garbage collection, 508

Marshal class, 360

MarshalByRefObject class, 564, 567, 599, 798

MarshalByValType type, 568

marshaling objects by value, 568

Match method, 434

Math class, 759

Maximum method, 776

Max method, 759

MaxValue field, 317

MBCS (Multi-Byte Character Set), converting
 Unicode to, 350

MDAs (Managed Debugging Assistants), 538

MEF (Managed Extensibility Framework), 598

member accessibility

member accessibility, 156, 166
MemberInfo
 class, 602
 type, 599
MemberRef table, 40
MemberwiseClone method, 182
 System.Object, 92
memcpy function, 378
memory
 addresses, passing between CLR and
 WinRT, 656
 allocating, 506, 507, 513
 compacting, in garbage collection, 509, 521
corruption, 506
garbage collection and, 507
inefficient allocation, 676
leaks, 506, 510, 526
limitations of, 506
monitoring, 524
OutOfMemory exception, 510
pressure methods, 539
reclaiming, with garbage collection, 517
MemoryStream
 class, 656
 object, 614
merging LINQ query results, 720
metadata, 5
 assembly, 586
 components of, 38
 constants and, 175
 for conversion operator methods, 197
 definition tables in, 39
 for events, 604
 examining, 431
 in human-readable form, 40
 identical format of, 152
 manifest tables, 47, 49
 on properties, 603
 property definitions in, 231
 reference tables, 39
 strings in, 332
 tokens, 50
 uses for, 5
 WinRT, 644, 659
method parameters, passing, 214
MethodDef table, 39
MethodInfo
 method, 417
 objects, converting to
 RuntimeMethodHandle, 609
methods, 162
 accessor, 228, 231, 247, 248
 anonymous types and, 240
 callback, 249
 calling, 102, 103, 163
 calling for the first time, 11
 calling for the second time, 14
 compiling once only, 277
 conflict resolution, 170, 171
 contract, maintaining when exception
 thrown, 483
 conversion operator, 195
 declaring as constants, 224
 declaring return types, 223
 defined, 24, 152
 for encoding classes, 355
 for enumerated types, 371
 examples of, 163
 explicit, and state corruption, 476
 explicit interface method implementations
 (EIMIs), 303, 308, 310
 extension, 198, 255
 generic, 265, 270, 281, 283, 286
 generic and non-generic, 284
 instance constructors, 181, 184
 interface, calling, 302, 448
 invoking after removal from delegate
 chain, 255
 marking as async, 733
 nonvirtual vs. virtual, 165
 operator overload, 191
 out and ref parameters, 282
 overloading, arity-based, 286
 overrides, 172
 overriding, 299
 parameters. *See* parameters
 parameter types, 214
 partial, 204
 passing variable number of arguments
 to, 220
 performance issues, 167
 polymorphic, 166

public, 299
referencing, 35
reliability contract documentation, 496
reliable, 497
return types, 395
splitting specific and generic, 283
static, 106
tables, 301
thread-safe, 759
translating input into events, 256
tuple types and, 241
type constructors, 187
type parameters, defining, 282
virtual, 133, 163, 169
virtual, changing to override, 172
virtual instance, 108
WinRT, 646, 650
MethodWithReturn method, 568
Microsoft Silverlight applications, AppDomains and, 574
Microsoft Spy++, 683
Microsoft SQL Server, AppDomains and, 575
Microsoft Visual Studio, Checked setting, 117
Microsoft Windows NT kernel, 670
migrating code from .NET Framework versions, 459
MinValue field, 317
ModuleDef table, 39, 81
ModuleRef table, 39, 81
modules building types into, 35 combining into assemblies, 45 in friend assemblies, 156
Monitor class, 476, 794 commonly used methods, 795 example code, 796 passing reference to type object into, 798 problems with, 798 as static class, avoiding, 798
monitoring AppDomains, 571 memory usage, 524 object lifetimes, 546
MPGO.exe, 21
MSCorEE.dll file, 554
MSCorLib.dll, 35, 558
MSCorWks.dll, 554
MSIExec.exe, 61
MulticastDelegate class, 397 non-public fields, 398
multi-dimensional arrays, 385, 386
multifile assemblies, 46, 48
multiple cores, 677. *See also* CPUs processing power of, 682 taking advantage of, 682 thread starvation on, 684 UI element removal and, 682 user productivity and, 682
multiple inheritance, 295. *See also* interfaces
multithreaded operating system, 684
MultiWebRequest class, 770
Mutex class, 780, 785 lock, 792 type, 525
mutexes, 785

N

naked constraints, 288
named parameters, 209, 424 arguments, order of evaluation with, 210 rules and guidelines, 211
NamedPipeServerStream class, 743
namespaces, 97 aliases, creating, 99 assemblies and, 101 creating, 100 WinRT, 645
native C++, 659
native resources cleanup code for, 534 fixed in number, 538 garbage collection and, 525, 538 wrapping, 530, 532
NetworkStream object, 355
new operator, 92, 506 delete operator and, 93
newline characters, 322
NewLine property, 322
NewMailEventArgs object, 253
newobj instruction, 321

NGen.exe

 NGen.exe, 16, 19
 intellectual property and, 20
 nonstatic events, 152
 Normal flag, 546, 547
 notifications
 for exceptions, 456
 for garbage collection, 518, 523
 for thread attachment/detachment, 671
 unregistering interest in, 260
 null
 comparing generic type variables with, 291
 initializing locally typed variables with, 213
 in WinRT, 647
 nullable types, 288, 441
 boxing, 447
 C# support for, 443
 calling `GetType` via, 448
 calling interface methods with, 448
 CLR support for, 447
 null-coalescing operator and, 446
 operators and, 444
 overloaded operators and, 445
 performance and, 445
 unboxing, 448
 nullable value types, 646
 number types, parsing strings into, 348
 `NumberFormatInfo` type, 343
 Numbers, vs. enumerated types, 361
 `NumberStyles` parameter, 348
 numeric types, 117. *See also* types
 recognized formats, 340
 `NumTimesWordAppearsEquals` method, 331
 `NumTimesWordAppearsIntern` method, 332

O

 obfuscator utilities, 19
 `Object` class, 296, 362
 inheritance, 296
 object equality, 139
 checking, 140
 properties of, 141
 object graphs, serializing, 614
 serializing multiple, 615
 object handles, wrapping, 778

 object identity, checking, 140
 object lifetime, controlling manually, 545
 object-oriented programming (OOP), 160
 `ObjectDisposedException` exception, 534
 `ObjectHandle` type, 595
 `ObjectInvariant` method, 502
 objects
 allocating and initializing, 106
 casting between types, 93
 converting between types, 195
 corrupted state, avoiding, 228
 creating, 92, 106, 596
 `Equals` method, 139
 formatting, 344
 `GetHashCode` method, 142
 hash codes, 142
 initializers, 235
 invariants, 498, 501
 marshaling by reference, 564
 type, 23, 112
 `OnDeserialization` method, 629
 `OneManyLock` class, 802
 performance results, 804
 `OneManyResourceLock` class, 804
 `OnNewMail` method, 254
 `OnThreadException` virtual method, 488
 `op>Addition` method, 193
 open types, 272
 `OpenAsync` method, 753
 `OpenExisting` methods, 780
 operands, generic type variables as, 292
 operating system instability, reasons for, 34
 `OperationCanceledException` exception, 652
 operations
 canceling, 696
 canceling after set time, 700
 canceling, invoking methods after, 698
 cooperative cancellation, 696
 preventing cancellation of, 698
 operator overload methods, 191
 calling internally, 194
 CLR handling of, 191
 for core numeric types, 193
 example of, 192
 naming rules, 194

operators, list of, 193
 overloads defined, 152
 parameters for, 191
 support for, 191
 unsupported, 193

operators
 binary, 192
 checked and unchecked, 116
 null-coalescing, 446
 and nullable types, 444
 unary, 192

op_Explicit method, 197
 op_Implicit method, 197

OptimizedAway method, 763
 optimizing code, 14, 277, 763
 Volatile class and, 765

optional attribute, 212

optional parameters, 209
 arguments, order of evaluation with, 210
 rules and guidelines, 211

OrderedTaskScheduler, 713
 ordinal equality check, 329

out keyword
 overloading methods based on, 216
 vs. ref keyword, 215

out parameter, 282

OutOfMemory exception, 510
 overflows, checked and unchecked, 115
 overhead fields, 795

overloading
 arity-based, 286
 nullable types and, 445

overloading operators. *See* operator overload
 methods

override methods, 172
 overriding methods, 299
 generic, 286

ownership, of thread locks, 791

P

packaging assemblies, 60
 Parallel class, 713, 714
 performance issues, 714
 when to use, 714

Parallel LINQ, 717
 buffering and merging, 720
 parallel processing queries, 719
 partitions across CPU cores, 720
 preserving operation order, 718
 sequential and parallel queries, 717
 stopping queries prematurely, 719

parallel queries, 717

ParallelEnumerable class, 717, 719
 ParallelLoopResult instance, 716
 ParallelLoopState object, 716
 ParallelOptions object, 714
 ParamArray attribute, 221
 ParamDef table, 39
 parameterful properties, 227, 242
 accessing from unsupported languages, 247
 selecting, 246

ParameterInfo object, 602
 parameterless constructors, 186, 235
 parameterless properties, 227

parameters
 declaring as constants, 224
 declaring types, 214, 223
 default values, 210, 212
 default values, changing, 211
 for indexers, 244
 named, 209
 optional, 209
 out keyword vs. ref keyword, 215
 params keyword, 221
 passing by reference, 215
 passing via method, 214
 types, in methods, 214
 types, interfaces as, 306
 validation, 366
 variables, 211
 weak vs. strong, 223

params keyword, 221

Parse method, 348, 366, 370
 exception handling for, 493
 overloading, 349
 parameters, 349
 performance and, 350
 skipping leading space, 348

ParseCombiningCharacters method, 333
 ParseExact method, 349

parsing

parsing
dates and times, 349
strings, 348
partial keyword, 159
partial methods, 204
efficiency of, 205
overriding types with, 205
private keyword and, 207
return type for, 207
rules and guidelines, 207
runtime performance and, 206
sealed classes and, 205
in Visual Studio, 206
passing arrays, 382
passing parameters, via reference, 215
path names, loading assemblies with, 586
pbytes local variable, 549
PDB (Program Database) files, 15
PE (portable executable) files
manifest, 47
command-line switches for compiling, 47
components of, 38
embedding resource files into, 53
PE32
(+) files, 6
(+) headers, 38
PE32 headers, 5
platform version and, 9
PerfMon.exe, 524
performance concerns, 15
performance counters, 524
performance issues
binding handles and, 608
context switching and, 672, 673
DLLs and, 672
double-check locking and, 808
exception handling and, 492
execution contexts and, 695
hyperthreading and, 677
instance fields and, 567
I/O operations and, 731
of methods with variable number of
arguments, 222
monitoring with Task Manager, 675
Parallel class and, 714
reflection and, 589
tasks and, 708
thread pools and, 692
threads and, 670, 674, 681
thread synchronization and, 758
thread synchronization locks and, 790
Performance Monitor, testing exception
handling with, 493
PerfView, 525
periodic compute-bound operations, 720
PEVerify.exe, 18
Pinned flag, 546
pinning objects, 548
points of failure, sources of, 471. *See also*
exceptions; exception handling
policy controls, 86
polygons, arrays of, 375
polymorphism, WinRT components and, 646
pooling strings, 332
positional parameters, 424
PostAsync method, 742
postconditions, 498, 501
execution of, 502
side-effect-free members, 501
Power Threading library, 804
preamble, 353
preconditions, 498, 501
inheritance of, 502
side-effect-free members, 501
preemptive multithreaded operating
system, 684
PrepareConstrainedRegions method, 496
primary constraints, 287
primitive thread synchronization constructs, 760
kernel-mode, 761, 778, 784
interlocked, 768
user-mode, 762
volatile, 766
primitive types, 111
checked and unchecked, 115
comparing to each other, 292
defining constants with, 175
dynamic, 144
as enumerated types, 363
vs. FCL types, 113
with FCL types, 112
parameter default values, 211

primitives, overflows and, 115
 prioritizing threads, 684, 754
 by application, 687
 classes for, 685
 High priority, 685
 Idle priority class, 685
 Idle priority level, 687
 importance of, 688
 mapping priority levels, 686
 priority boosting, 774
 process priority class, 686
 raising and lowering priority, 687
 Realtime class, 685
 relative priority classes, 685
 Time-Critical priority level, 687
 for Windows Store apps, 687
 zero page thread, 684, 686
 PrioritizingTaskScheduler, 713
 priority
 inversion, 756
 thread, 679
 private assemblies, 67
 strongly named, 79
 private fields, 168, 228, 231
 private keys. *See* public and private keys
 privately deployed assemblies, 61
 private methods
 partial methods as, 207
 serialization and, 622
 type constructors, 188
 private thread locks, 797
 problem signatures, 487
 Process class, 688
 processes
 creating, expensive nature of, 675
 creating, vs. creating threads, 675
 defined, 670
 example of, 558
 isolation, 557
 process priority class, 686
 reducing, with managed code, 17
 terminating, 477, 574
 threads per, 674
 wasteful, identifying, 675

processor architecture, loading assemblies
 for, 585
 ProcessThread class, 688
 productivity vs. reliability in error handling, 469
 program order, 765
 Program type, 43
 prologue code, 102
 properties
 accessor accessibility, 248
 accessor methods, 231
 automatically implemented, 231
 backing fields, 230
 behavior and, 248
 compiler results, 244
 confusion caused by, 232
 C# support for, 231
 declaring, 231
 declaring, in anonymous types, 238
 defined, 24, 152
 defining, 230
 defining intelligently, 232
 example of, 229
 generic accessor methods, 248
 in metadata, 231
 metadata on, 603
 parameterful, 227, 242, 246
 parameterless, 227
 vs. public fields, 425
 returning different values, 233
 thread synchronization and, 233
 in Tuple types, 241
 Visual Studio debugger and, 234
 when to use, 233
 PropertyDef table, 39
 PropertyInfo type, 603
 public and private keys
 for assemblies, 67. *See also* strongly named
 assemblies
 creating, 69
 in CSP containers, 78
 delayed signing, 76
 in hardware devices, securing, 78
 hashing to reduce space, 71
 for publisher policy assemblies, 87

public

public
 constructors, defining, 195
 fields, vs. properties, 425
 methods, interfaces and, 299
 types, 25, 154
publisher policy controls, 86
publisherPolicy elements, 85
publishing values, 808
Pulse method, 813
PulseAll method, 813
purpose of book, xxiii
PutAsync method, 742

Q

qsort function, 391
quanta, 672
 ending early, 673
queries, LINQ. *See* LINQ queries
QueryMemoryResource notification, 519
question marks, in C# code, 443
queues, thread-safe, 813
QueueUserWorkItem method, 409, 693,
 700, 724

R

raising events, in thread-safe way, 254
Rational type, 195
RCWs (Runtime Callable Wrappers), 644
Read method, 753
 in I/O operations, 728
Read method (Volatile class), 765, 767, 768
read-only files, setting attribute for, 368
ReadAsync method, 729, 733, 742, 753
ReadBlockAsync method, 742
ReaderWriterLock class, 802
reader-writer locks, 801, 804
ReaderWriterLockSlim class, 800
ReadLineAsync method, 742
readonly fields, 177
 defining, 178
ReadToEndAsync method, 742
real-time operating systems, 684

Realtime priority class, 685
Rectangles, constructing, 185
recursive locks, avoiding, 806
recursive thread locks, 786
ref keyword
 overloading methods based on, 216
 vs. out keyword, 215
ref parameter, 282
reference counting, 508
ReferenceEquals method, 140, 331
References, checking, 157
reference tables in metadata, 39
reference tracking, 508
reference type variables, setting to null, 288
reference types, 118, 320
 arrays of, 373
 classes as, 119
 converting value types into, 125
 example code, 121
 examples of, 119
 field layout, 123
 generic, 269
 null-coalescing operator and, 446
 overlapping with value types, 124
 performance issues, 119
 as primary constraints, 287
 ref keyword vs. out keyword, 215
 serializing, 618
 sharing code, 277
 swapping, 218, 219
 with type constructors, 188
 vs. value types, 122
referencing
 assemblies, 70, 98
 strongly named assemblies, 74
 types, 80
reflection, 141, 148, 430
 discovering type members, 599
 exception handling and, 485
 object model, discovering, 602
 performance issues, 589
 serialization and, 614
 type discovery with, 590, 604
 type safety and, 589

reflection API for .NET Framework, 583
 reflection members, generics and, 271
 reflection methods, 432
Reflection namespace, 431, 588
 reflection types, 588, 589
 hierarchy of, 600
ReflectionOnlyGetType method, 591
ReflectionOnlyLoadFrom method, 587
ReflectionOnlyLoad method, 587
 regional settings, 326
Register method
 CancellationToken class, 699
 CancellationTokenSource class, 698
RegisterForFullGCNotification method, 523
 registering assemblies, 74
 relational operators, nullable types and, 444
 relative thread priority classes, 685
ReleaseMutex method, 785
ReliabilityContract attribute, 496
Reliability Monitor
 problem signatures, 487
 terminated applications, viewing in, 487
 reliability vs. productivity in error handling, 469
Remove method, 260, 338, 403
RemoveAt method, 358
RemoveEventHandler method, 603
RemoveMemoryPressure method, 538
RemoveParticipant method, 805
Replace method, 338
ReportStatus method, 772
ResetAbort method, 579, 580
 resolving types, 593
 resource files, adding to assemblies, 53
 resource leaks, 531
 resources, accessing, 505
 response files, 36
 ignoring, 38
 resurrected objects, 544
 return types, discovering information on, 602
 returning arrays, 382
 reverse engineering, protecting against, 19
 roots, 508
 in garbage collection, 509
Run method, 745

Runtime
 binders, 148
 handles, 608
Runtime Callable Wrappers (RCWs), 644
RuntimeWrappedException class, 458

S

safe places, 580
SafeHandle class, 527
 derivation of, 528
SafeHandle namespace, 529, 530, 531
SafeWaitHandle class, 529
 satellite assemblies, 59
sbyte type, 112
 scalability, threads and, 674
 scheduling tasks, 711
 reader/writer semantics, 815
 scheduling threads, 683, 684
 sealed classes, 168
 partial methods and, 205
 secondary constraints, 288
SecureString class, 357, 358
 decrypting objects, 360
 FCL support for, 358
 passing objects as a password, 358
 secure strings, 357
 security, class, 168
SecurityCritical attribute, 489, 695
 security settings, thread pools and, 694
Semaphore class, 780, 784
 semaphores, 784
 vs. auto-reset events, 784
 as primitive kernel-mode thread
 synchronization constructs, 778
SemaphoreSlim class, 794, 815
 sequential queries, 717
 serialization, 589
 AddValue method and, 627
 of application state, 616
 assemblies and, 616
 cloning with, 615
 of collection objects, 622

SerializationBinder class

serialization (*continued*)
 controlling, 619
 controlling data, 624
 converting types, 628
 defining type for, 630
 definition of, 611
 formatter consistency, 615
 of instance types, 619
 ISerializable interface, 625, 630
 matching deserialization method, 628
 of multiple object graphs, 615
 namespace, 623
.NET Framework support for, 612
 of object graph, 614
 of object set, 622
 overriding assembly/type, 641
 quick start for, 613
 reflection and, 614
 security and, 619
 of singleton types, 633
 special constructor type, 628
 streaming contexts for, 631
 surrogate selector chains, 639
 surrogates for, 636
 of type instances, 623
 of types, 617, 618
 of types, as different types, 633
 to XML stream, 612, 614
 when to avoid, 619
SerializationBinder class, 640
SerializationException exception, 615, 616,
 618, 622
SerializationInfo object, 625, 628
Serialize method, 614
SerializeToMemory method, 614
Server garbage collection mode, 520
Servers, asynchronous implementation, 751
servicings, 161
SetAt method, 358
SetMaxThreads method, 724
SetMinThreads method, 724, 726
SetType method, 626, 635
SetValue method, 603
shared data, Parallel class and, 714
shim DLL, 554
short type, 112
SignalAndWait method, 805
signed data types, 117
signing files, 70
Silverlight applications
 AppDomains and, 574
 processes run by, 687
Silverlight, threading model, 748
SimpleHybridLock method, 793
SimpleSpinLock method, 773
SimpleWaitLock method, 782, 784
simplifying applications, 682
SimulateNewMail method, 256
single-dimensional arrays, 375
single-instance applications, 781
Singleton class, 809
singleton types, serialization of, 633
Sleep method, 774
SN.exe, 68
 delayed signing with, 77
Soap namespace, 614
SoapFormatter
 class, 614
 object, 614
Software Publisher's Digital ID for
 Authenticode, 488
SomeMethod method, 565, 566
SOS Debugging Extension, 525
source code
 compiling, 3
 splitting types between, 159
 protection, 267
source control, 159
spin locks, 773. *See also* thread synchronization
spin loops, thread yielding and, 775
SpinLock method, 775
SpinLocks, 818
spinning threads, with hybrid locks, 791
SpinWait method, 774
Spy++, 683
SQL Server, AppDomains and, 575
stackalloc statement, 388
StackOverflowExceptions, 497

stack space, 102
 checking for, 489
stack trace, 461, 462
 complete, 462
 constructing, 462
 interpreting, 463
StackTrace property, 461
Start method (Thread class), 679
StartNew method, 710
StartsWith method, comparing strings, 325
startup time, improving, 19, 21
starvation, thread, 684
state corruption
 destroying, 476
 exception handling and, 474
 FailFast method, 480
 managed code and, 477
 mitigating, 475
 terminating process for, 477
state machines, returning threading model, 749
stateless encoding and decoding, 356
statements, checked and unchecked, 116
static classes, 158
static constructors. *See* type constructors
 defining on generic types, 273
static events, 152
static fields, 177
 defining, 178
 initializing, 189
 memory leaks and, 510
static members
 interfaces and, 297
 MarshalByRefObject class and, 567
static methods
 for arrays, 380
 calling, 106
 calling back, with delegates, 394
 calling operator overload methods with, 194
 for Char type, 318
 extension attribute, 203
 Finalize method and, 526
 for Int32 variables, 768
 thread safety, 759, 760
 volatile keyword, 767
StaticDelegateDemo method, 394
Stop method (Parallel class), 716
streaming contexts for serialization, 631
StreamReader type, 350
streams, 656
StreamWriter type, 350, 537
 finalization and, 538
String class
 hash table, accessing, 330
 methods, 338
 sealed, 323
string interning, 329
String objects. *See* strings
 immutability of, 569
String type, 112, 320
 constructing, 320
 interfaces and, 300
 method overloads defined by, 325
 as primitive type, 320
StringBuilder class, 198, 345
 creating objects, 336
 members, 337
 methods, 338
StringBuilder type, 336
StringComparer class, 329
StringComparison type, 324
StringInfo, 333
strings
 base-64 encoding/decoding, 356
 changing case of characters in, 325
 character arrays, 337
 characters, examining, 333
 cloning, 335
 comparing, 323
 concatenating, 322
 constructing, 320
 constructing efficiently, 336
 converting to bytes, 353
 converting Unicode to MBCS, 350
 copying, 335
 custom formatter, 345
 date formatting, 341
 deleting securely, 358
 encodings, 350

Strong Name utility (SN.exe)

strings (*continued*)
 formatting, 344, 346
 formatting multiple objects into, 344
 immutability of, 323
 interning, 329
 mapping enumerated types to, 364
 in metadata, 332
 number formatting, 341
 as object representation, 339
 parsing, 348
 pooling, 332
 secure, 357
 in source code, 321
 uses for, 324
 verbatim, 322
 WinRT, 647

Strong Name utility (SN.exe), 68

strongly named assemblies
 characteristics of, 66
 components of, 68
 compressing, 74
 creating, 67
 deploying globally, 72
 deploying privately, 79
 obtaining key for, 68
 referencing, 74
 security of, 76
 tamper resistant nature of, 75
 unique identification of, 67
 verification, 76

struct constraint, 287

Structured Exception Handling (SEH), 453.
 See also exceptions; exception handling structures. *See* value types

Substring method, 335

SubstringByTextElements method, 333

SUPPRESSFLOW method, 695

SUPPRESSMESSAGE attribute, 440

SurrogateSelector objects, 639

surrogate types for serialization, 636

surrogates for Unicode elements, 333

suspending threads, 774

sync blocks, 794
 array, 795
 index, 105
 initialization of, 795

optimizing, 795

public indexes, 797

relationships, 795

type objects and, 795

synchronization context task schedulers, 711

SynchronizationContext class, 749

synchronizing threads. *See* thread synchronization

synchronous I/O operations, 752. *See* I/O operations

FileStream objects and, 753

System Monitor, 524

System namespace, 22, 23

System.Collections.Generic namespace, 270

System.Decimal type, 118, 198

System.Diagnostics.ConditionalAttribute attribute, 206

System.Exception
 class, catching, state corruption and, 474
 type, 460

SystemException type, 466

System.InvalidCastException type, 95

System.Object type, 26, 91
 delete operator, 93
 new operator, 92
 protected methods, 92
 public methods, 92

System.Reflection.AssemblyName class, 68

System.Runtime.ProfileOptimization class, 16

System.Text namespace, 97

System.Type object, type object pointer, 110

SZ arrays, 375

T

TakeFromAny methods, 822

Task class
 vs. Barrier class, 814
 vs. thread synchronization, 814

Task Manager, 670, 675
 monitoring performance with, 675

Task objects, 701
 arrays of, 710
 Boolean properties for, 709
 fields in, 707
 ContinueWith task collections, 705

creation flags for, 701
 CurrentId property, 708
 disposing of, 708
 factory for, 709
 Int32 fields in, 708
 statuses of, 708
 Status property, querying, 708
 supplementary state, 707
 WaitingForActivation state, 709
TaskFactory type, 709, 710
TaskLogger class, 738
tasks, 700
 arrays of, 710
 canceling, 704
 CancellationToken objects, associating with, 705
 child tasks, 707
 composition of, 707
 constructors and, 701
 creation options, 701
 deadlock situations, 702
 delaying, 722
 delegates, passing, 715
 exceptions, detecting, 703
 exceptions thrown by, 704
 performance issues and, 708
 reader/writer semantics, 815
 returning void with, 709
 scheduling to run, 701
 starting automatically on other task completion, 705
 state, 707
 supplementary state, 707
 synchronization context task schedulers, 711
 task factories, 709
 task schedulers, 711, 712
 vs. thread synchronization, 807
 unhandled exceptions thrown by, 702
 unique IDs for, 708
 waiting for completion, 702
Tasks namespace, 651, 700, 709
TaskScheduler objects, 711
TaskScheduler type, defining classes derived from, 712
TEB (thread environment block), 671
 temporary variables, 284
 terminated applications, viewing in Reliability Monitor, 487
 terminating process, 477, 574
 text elements. *See also* characters examining in strings, 333
TextElementEnumerator method, 333
 thread blocking, I/O operations and, 731
Thread class, 678
 thread environment block (TEB), 671
 thread kernel objects, 671
 thread locks, asynchronous functions and, 734
 thread pools, 692
 calling methods asynchronously, 693
 child tasks, 707
 continuing tasks in, 705, 707
 deadlock situations, 724
 delaying tasks, 722
 execution contexts, 695
 function of, 692
 heuristic nature of, 692
 limiting number of threads in, 724
 loop state management, 716
 parallel performance of, 713
 parallel processing, exceptions thrown when, 716
 performance and, 692
 processing order, 725
 queuing in, 693
 scheduling method calls, 720
 scheduling tasks in, 712
 security settings and, 694
 sequential method execution, 713
 thread management in, 723
 thread synchronization locks and, 725
 worker thread management, 724
 work items in, 693
 thread prioritization, 754
 thread-safe
 collection classes, 818
 queues, 813
 thread safety, 759
 thread synchronization
 async functions and, 757
 asynchronous, 814
 auto-reset events and, 782
 avoiding, 759

ThreadAbortException exception

thread synchronization (*continued*)
 blocking, avoiding, 805
 blocking countdown, 804
 class libraries and, 759
 condition variable pattern, 811
 constructs, 805
 double-check locking, 807
 drawbacks of, 757
 exception handling and, 476
 hybrid, 790
 hybrid constructs. *See hybrid constructs*
 livelocks and deadlocks, 761
 lock, 189, 799
 locks, 476, 725
 lock scalability issues, 814
 m_opCount field, 772
 Mutex objects, 786
 parallel execution, 805
 performance issues, 758
 primitive constructs, 760
 priority boosting, disabling, 774
 private locks, 797
 properties and, 233
 reader-writer locks, 801, 804
 recursive locks, 786
 recursive locks, avoiding, 806
 specifying synchronization object, 811
 spin lock addition logic, 774
 spin locks, 773
 tasks vs., 807
 yielding threads when, 775
ThreadAbortException exception, 580
ThreadException event, 488
threading models, 748
Threading namespace, 695, 720, 778
 DispatcherTimer class, 723
 Timer class, 723
ThreadPerTaskScheduler, 713
ThreadPool class, 693
 for Windows Store apps, 694
threads, 102
 in 16-bit Windows, 669
 aborting, 578, 679
 allocating local variables, 105
asynchronous compute-bound operations
 with, 678
 avoiding, 674
 blocking, avoiding, 705
 caching of, 673
 CLR and Windows interoperability, 678
 constructors for, 679
 contexts, 671
 context structures, 683
 context switching, 672, 673, 683
 creating, vs. creating processes, 675
 debuggers and, 673
 dedicated, 679
 defined, 670
 DLL thread-attach/thread-detach
 notifications, 671
 early implementation of, 669
 ending early, 673
 execution contexts, 694, 695
 forcibly killing, 670
 foreground, 679
 foreground vs. background, 688
 garbage collection and, 673
 gracefully aborting, 577
 High priority class, 685
 Idle priority class, 685
 Idle priority level, 687
 inefficient memory allocation and, 676
 kernel-mode stacks, 671
 labeling, avoiding, 806
 lowering priority, 687
 minimizing number of, 676
 number per process, 674
 overhead, 670
 performance issues, 674, 681
 prioritizing, 684, 685, 688
 priority levels, mapping, 686
 priority, non-normal, 679
 properties, viewing, 683
 Realtime priority class, 685
 reasons for using, 681
 relative priorities, 685
 responsiveness of, 681
 safe places, 580

scalability, 674
scheduling, 683, 684
starting, 679
starvation, 684
suspending, 774
Time-Critical priority level, 687
time-slices, 672
user-mode stacks, 671
Windows Store apps, APIs removed for, 678
yielding, 775
zero page, 684, 686
ThreadsSharingData class, 766, 767
ThrowIfCancellationRequested method, 704
throwing exceptions, 466. *See also* exceptions;
exception types
Time-Critical thread priority level, 687
time-slices, 672
 ending early, 673
Timer class, 720, 723
Timer objects, garbage collection of, 721
Timers namespace, Timer class, 723
ToArray method, 658
ToLowerInvariant method, 318
ToObject method, 367
torn reads, 762
ToSingle method, 310
ToString method, 134, 238, 318, 335, 337, 339,
 364, 365, 569
 number formatting, 341
 output formatting, 369
 output of, 364
 overriding, 339
 parameterless, 340
 parameters, 340
 vs. Format method, 364
 System.Object, 92
ToString, null for format string, 341
ToUpperInvariant method, 318
Transaction class, 797
TransactionScope class, 475
triggering garbage collections, 519
try blocks, 454. *See also* catch blocks; finally
 blocks
 as callback methods, 497
 constrained execution regions and, 475
finally blocks and, 457
threads and, 671
TryParse method, 366, 370, 493
 exception handling and, 494
Tuple
 class, 241
 types, 237, 240
 properties and, 241
type arguments, 266
type constructors, 187
 calling, 188
 calling in order, avoiding, 189
 default, 187
 defined, 151
 defining, 188
 defining inside value types, avoiding, 188
 initializing singleton objects with, 189
 mutual references in, 189
 naming convention for, 190
 private, 188
 thread synchronization lock, 189
 unhandled exceptions, 189
type definitions, example of, 153
type inference
 generic methods and, 283
 overview of, 283
type object pointer, 105
Type objects, 591
 converting to **RuntimeTypeHandle**, 609
 converting to **TypeInfo** objects, 592
 for generic types, 272
 obtaining, 591
 obtaining from type names at compile
 time, 592
 open, creating, 272
 sync blocks and, 795
type parameters, 266
 constraining to interfaces, 305
 constraints, 287, 288
 constructor constraints for, 289
 contra-variant and covariant, 279
 generic, 281
 naming conventions, 266
 overriding methods and, 286
 variance and, 280

type-safe languages

type-safe languages, 144
type safety, 219, 267
 interfaces and, 308, 309
 protecting, 277
 reflection and, 589
type variables, generic, 290, 291, 292
TypeDef table, 39
TypeInfo class, 602
TypeInfo object, 592
 querying, 593
TypeInfo objects, 592
typeless instructions, 17
typeof operator, 429, 592
TypeRef table, 40
types, 24. *See also* CTS (Common Type System)
 across assemblies, 597
 as add-ins, 596
 aliases, creating, 99
 ambiguous references, 99
 anonymous, 237, 238, 239
 arity, 273
 assembly, 25
 Backus-Naur Form grammar for, 592
 building into modules, 35
 casting between, 93
 casting with is/as operators, 95
 catch, 456
 common properties and methods, 602
 constructing instance of, 594
 converting, 628
 converting between, 195
 creating objects for, 596
 data fields in, 105
 declaring, 214
 defining, 35, 105, 251
 disambiguating, 99
 discovering members, 599, 604
 dynamically constructing, 590
 enumerated, 361
 enumerating, 588
 example definition of, 28
 exception, 463, 466
 exception-derived hierarchy of, 593
 exposing events with, 250
 external, 35
 field layout, 123
 general format, 341
 generic, 265, 266, 269, 274, 276, 596
 generic identity, 275
 generic interfaces and, 278
 help documentation for, 310
 helper, 169
 inferring from initializing expression, 212
 initializing static fields, 189
 invoking members, 603
 vs. keywords, 113
 limiting, for generics, 285
 linked lists, 274
 listening for events, 258
 literal, 114
 locations of, 81
 member accessibility, 156
 members of, 24, 151
 method tables, 301
 moving between assemblies, 83
 namespaces, 97, 99
 naming conventions, 273
 nullable, 441, 443, 445
 overloading, arity-based, 286
 partial, 159
 performance issues, 167
 primitive, 111
 properties, obtaining, 593
 public, 25, 154
 querying, 600
 recognized formats, 340
 reference, 118
 referenced assemblies, 98
 referencing, 35
 reflection, 588, 589, 600
 resolving, 593
 resolving references, 80
 restricting access to, 25
 returning name of, 339
 for runtime handles, 608
 safety, 95
 serializing, 617, 618, 620, 623
 signed, 117
 singleton, 633
 surrogate, for serialization, 636

System.Object derivation, 91
 Tuple, 237, 240
 unique names, 99
 unloaded, 191
 value, 118, 277
 visibility, 154, 156, 166
 Visual Studio tooltips for, 214
 WinRT, 646
 interoperability, 649
 without instance constructors, 182

U

UI elements, removing, 682
 uint type, 112
 ulong type, 112
 unary operators, 192
 nullable types and, 444
 unboxing
 nullable types, 448
 value methods, 135
 unboxing value types, 124, 126
 efficiency of, 130
 example code, 128
 field copying and, 127
 unchecked operators, 116
 unhandled exceptions, 474, 480, 485
 asynchronous functions and, 745
 in distributed applications, 488
 FCL documentation, 488
 from type constructors, 189
 UnhandledException event, 488
 UnhandledExceptionFilter event, 488
 Unicode characters, 333
 converting to MBCS, 350
 Unicode code values, 317
 UnicodeCategory type, 317
 Unload method, 566, 570
 unloaded types, 191
 unloading
 AppDomains, 477, 494, 566, 570
 assemblies, 587
 as garbage collection trigger, 519
 unmanaged C++, 224

unmanaged callback functions, 391
 unmanaged code
 C++ compiler and, 6
 CLR interoperability with, 30
 performance of, vs. managed, 15
 UnobservedTaskException event, 703
 Unregister method, 260
 unsafe code, 17
 unsafe modifier, 388
 UnsafeLoadFrom method, 586
 Unwrap method, 595
 uppercase/lowercase, converting between, 318
 Uri type, 648
 URIs, WinRT, 648
 user-defined attributes. *See custom attributes*
 user-mode constructs, 760, 762
 vs. kernel-mode constructs, 784
 vs. primitive-mode constructs, 778
 user-mode stacks, 671
 ushort, 112
 using directive, generic types and, 276
 using statement, 536
 UTF-7 encoding, 351
 UTF-8 encoding, 351
 UTF-16 encoding, 351
 UTF-32 encoding, 351

V

value methods, boxing and unboxing, 135
 value types, 118, 441
 boxing and unboxing, 124, 126, 128
 calling virtual methods, 133
 changing fields in, 136
 complexity of, 138
 constructors, 184
 converting to reference types, 125
 default constructors, 186
 defining, 132
 defining constructor methods on, 185
 defining type constructors in, avoiding, 188
 example code, 121
 examples of, 119
 field layout, 123
 immutability of, 138

ValueType class

value types (*continued*)
initializing fields, 187
interface methods for, 319
interfaces and, 301
native code for, 277
nullable, 441, 443, 646
out keyword vs. ref keyword, 215
overlapping with reference types, 124
vs. reference types, 122
serializing, 618
with type constructors, 188
unboxing, and field copying, 127
verifiability, 186
virtual methods, calling, 165
when to use, 121
WinRT, 646
ValueType class, 362
var keyword, 237
 vs. dynamic, 146
 vs. dynamic keyword, 214
variables
 cloneable, 301
 implicitly typed, 212, 237
 initializing with null, 213
 of interface types, 300
 temporary, 284
variance, restrictions on, 280
vectors, 375. *See also* arrays
verbatim strings, 322
verification
 of generics, 284
 generics and, 290
 IL and, 16
 skipping, 18
verifying assemblies, 76
 skipping, 77
VeriSign IDs, 488
version
 numbers, 58
 resources, 54
 attributes, 56
versioning, 161, 167
 assemblies and, 587, 597
 extension methods and, 201
 interfaces and, 313
 virtual methods and, 169

virtual instance methods, 108
virtual methods, 163, 165
 calling, 133
 call instruction for, 165
 constructor methods and, 182
 interfaces and, 299
 minimizing, 166
 performance of, 167
 security of, 168
 versioning and, 169
visibility, type, 154, 166
Visual Studio
 debugger, and properties, 234
 exception debugging, 490
 exceptions, adding to debugger, 491
 Exceptions dialog box, 490
 IDE, adding assemblies with, 51
 Intellisense, 200
 partial methods in, 206
 publishing applications with, 61
 speed of, due to multiple cores, 682
 target platform, setting, 8
 tooltips, viewing, 214
Windows Runtime Component project, 659
volatile
 class, 765, 767, 768, 773, 805, 808
 fields, 177
 keyword, 767
volatile constructs, 762, 766
C# support for, 767
drawbacks of, 767
Volatile.Read method, 254

W

Wait method, 702
WaitAll method, 703, 779
WaitAny method, 703, 779
WaitAsync method, 815
WaitCallBack delegate type, 693
WaitForConnectionAsync method, 743
WaitHandle class, 778
WaitOne method, 779
Weak flag, 546, 549
weak references, 550, 551

weakly named assemblies, 66, 68
private-only deployment, 67
WeakReference class, 549
WeakTrackResurrection flag, 546, 549
WebClients, asynchronous functions and, 744
web-downloading assemblies, 586
Web Forms, asynchronous, 751
WhenAll method, 747
WhenAny method, 747
Windows, as preemptive multithreaded
 operating system, 684
Windows Action Center, viewing unhandled
 exceptions in, 486
Windows Error Reporting, 488
Windows Forms, threading model, 748
Windows Installer service, 61
Windows NT kernel, 670
Windows Presentation Foundation (WPF),
 threading models, 748
Windows Runtime Component project in Visual
 Studio, 659
Windows scheduler, asynchronous operations
 and, 694
Windows Store apps, 36
 C#-implemented WinRT components, 659
 threading APIs removed, 678
 threading model, 748
 thread prioritization and, 687
 WinRT components, 643
WindowsRuntimeBuffer class, 658
WindowsRuntimeBufferExtensions class,
 657, 658
WindowsRuntime directive, 657
WindowsRuntimeStorageExtensions class, 654
WindowsRuntimeSystemExtensions class, 653
Windows RunTime (WinRT), 31. *See* WinRT
WinMDExp.exe, 659
WinRT
 arrays, 648, 660
 asynchronous APIs, calling, 649, 653
 base type, 645
 canceling APIs, 653
 classes, 646
 collections, 648
 core concepts, 645
 dates and times, 647
 delegate types, 647
 enumeration types, 646
 events, 647
 exceptions, 647
 explicitly calling methods, 648
 features exposed by, 643
 file names and namespaces, 645
 framework projections, 649
 IAsyncXxx interfaces, 651
 IAsyncXxx method conversion into .NET
 Framework Task method, 654
 IBuffer interface, 656
 IClosable/IDisposable interfaces, 648
 interfaces, 646
 JavaScript example code, 664
 mapping CLR types to, 659
 memory addresses, passing to CLR, 656
 metadata, 659
 methods, 646, 650
 .NET interoperability, 654
 nullable value types, 646
 passing blocks of data to CLR, 656
 properties, 647
 StorageFile, converting to .NET XElement, 655
 strings, 647
 structures, 646
 type system, 646
 type system interoperability, 649
 URLs, 648
 value types, 646
WinRT components
 CLR projections and, 645
 as COM components, 644
 defining in C#, 658
 FCL types and, 645
 file extensions, 644
 as managed components, 660
 metadata used by, 644
 polymorphism and, 646
 type system, 645
 for Windows Store apps, 659
WinRTAsynIntro method, 650, 652
Wintellect Power Threading Library, 690

WithCancellation method

WithCancellation method, 719, 751
WithMergeOptions method, 720
work items in thread pools, 693
Worker method, 764
Workstation garbage collection mode, 520
WoW64 technology, 9
wrapping native resources, 530, 532
Write method, 131, 345
 exceptions thrown by, 534
Write method (Volatile class), 765, 767, 773, 808
WriteAsync method, 742
WriteLineAsync method, 742
WriteLine method, 130, 131, 138, 164, 345, 385

X

Xaml namespace, DispatcherTimer class, 723
XML configuration files. *See* configuration files
XML elements, converting, 195
XML web wervices applications, AppDomains
 and, 575
XmlSerializer class, 612

Y

Yield method, 775

Z

zero-based arrays, 374
zero-length arrays, 382
zero page threads, 684, 686