

THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES



THIRD EDITION

CONVENTIONS, IDIOMS, AND
PATTERNS FOR REUSABLE .NET LIBRARIES

FRAMEWORK DESIGN

GUIDELINES

KRZYSZTOF CWALINA
JEREMY BARTON
BRAD ABRAMS

Forewords by SCOTT GUTHRIE,
MIGUEL DE ICAZA, and ANDERS HEJLSBERG

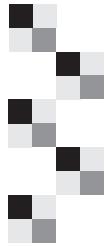


From the Library of ABE BARBERENA

Framework Design Guidelines

Third Edition

This page intentionally left blank



Framework Design Guidelines

*Conventions, Idioms, and Patterns
for Reusable .NET Libraries*

Third Edition

- Krzysztof Cwalina
- Jeremy Barton
- Brad Abrams

◆ Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informat.com/aw

Library of Congress Control Number: 20200935344

Copyright © 2020 Pearson Education, Inc.

Cover image: Jakub Krechowicz/Shutterstock

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions/.

ISBN-13: 978-0-13-589646-4

ISBN-10: 0-13-589646-0

ScoutAutomatedPrintCode

*To my wife, Ela,
for her support throughout the long process of
writing this book, and to my parents, Jadwiga and Janusz,
for their encouragement.*

—Krzysztof Cwalina



*To my lovely wife, Janine.
I didn't fully appreciate before why authors always dedicate
books to their spouse, and now I do.
So, thank you. I'm sorry. I probably have time now for whatever those
things were that you wanted me to do while I was writing.*

—Jeremy Barton



*To my wife, Tamara:
Your love and patience strengthen me.*

—Brad Abrams



This page intentionally left blank



Contents

<i>Figures</i>	<i>xvii</i>
<i>Tables</i>	<i>xix</i>
<i>Foreword</i>	<i>xxi</i>
<i>Foreword to the Second Edition</i>	<i>xxiii</i>
<i>Foreword to the First Edition</i>	<i>xxv</i>
<i>Preface</i>	<i>xxvii</i>
<i>Acknowledgments</i>	<i>xxxiii</i>
<i>About the Authors</i>	<i>xxxv</i>
<i>About the Annotators</i>	<i>xxxvii</i>

1 Introduction 1

1.1 Qualities of a Well-Designed Framework	3
1.1.1 <i>Well-Designed Frameworks Are Simple</i>	3
1.1.2 <i>Well-Designed Frameworks Are Expensive to Design</i>	4
1.1.3 <i>Well-Designed Frameworks Are Full of Trade-Offs</i>	6
1.1.4 <i>Well-Designed Frameworks Borrow from the Past</i>	6
1.1.5 <i>Well-Designed Frameworks Are Designed to Evolve</i>	7
1.1.6 <i>Well-Designed Frameworks Are Integrated</i>	7
1.1.7 <i>Well-Designed Frameworks Are Consistent</i>	7

2 Framework Design Fundamentals 9

2.1 Progressive Frameworks	12
2.2 Fundamental Principles of Framework Design	15

- 2.2.1 *The Principle of Scenario-Driven Design* 16
- 2.2.2 *The Principle of Low Barrier to Entry* 23
- 2.2.3 *The Principle of Self-Documenting Object Models* 29
- 2.2.4 *The Principle of Layered Architecture* 36

3 Naming Guidelines 41

- 3.1 Capitalization Conventions 42**
 - 3.1.1 *Capitalization Rules for Identifiers* 42
 - 3.1.2 *Capitalizing Acronyms* 45
 - 3.1.3 *Capitalizing Compound Words and Common Terms* 48
 - 3.1.4 *Case Sensitivity* 51
 - 3.2 General Naming Conventions 52**
 - 3.2.1 *Word Choice* 52
 - 3.2.2 *Using Abbreviations and Acronyms* 55
 - 3.2.3 *Avoiding Language-Specific Names* 56
 - 3.2.4 *Naming New Versions of Existing APIs* 58
 - 3.3 Names of Assemblies, DLLs, and Packages 61**
 - 3.4 Names of Namespaces 63**
 - 3.4.1 *Namespaces and Type Name Conflicts* 65
 - 3.5 Names of Classes, Structs, and Interfaces 67**
 - 3.5.1 *Names of Generic Type Parameters* 70
 - 3.5.2 *Names of Common Types* 71
 - 3.5.3 *Naming Enumerations* 72
 - 3.6 Names of Type Members 74**
 - 3.6.1 *Names of Methods* 74
 - 3.6.2 *Names of Properties* 75
 - 3.6.3 *Names of Events* 77
 - 3.6.4 *Naming Fields* 78
 - 3.7 Naming Parameters 79**
 - 3.7.1 *Naming Operator Overload Parameters* 80
 - 3.8 Naming Resources 81**
-
- 4 Type Design Guidelines 83**
 - 4.1 Types and Namespaces 85**
 - 4.2 Choosing Between Class and Struct 89**
 - 4.3 Choosing Between Class and Interface 92**

4.4 Abstract Class Design	100
4.5 Static Class Design	102
4.6 Interface Design	104
4.7 Struct Design	106
4.8 Enum Design	111
4.8.1 <i>Designing Flag Enums</i>	119
4.8.2 <i>Adding Values to Enums</i>	123
4.9 Nested Types	124
4.10 Types and Assembly Metadata	127
4.11 Strongly Typed Strings	129
5 Member Design	135
5.1 General Member Design Guidelines	135
5.1.1 <i>Member Overloading</i>	136
5.1.2 <i>Implementing Interface Members Explicitly</i>	148
5.1.3 <i>Choosing Between Properties and Methods</i>	152
5.2 Property Design	158
5.2.1 <i>Indexed Property Design</i>	161
5.2.2 <i>Property Change Notification Events</i>	163
5.3 Constructor Design	165
5.3.1 <i>Type Constructor Guidelines</i>	172
5.4 Event Design	175
5.5 Field Design	180
5.6 Extension Methods	184
5.7 Operator Overloads	192
5.7.1 <i>Overloading Operator ==</i>	198
5.7.2 <i>Conversion Operators</i>	198
5.7.3 <i>Inequality Operators</i>	200
5.8 Parameter Design	202
5.8.1 <i>Choosing Between Enum and Boolean Parameters</i>	205
5.8.2 <i>Validating Arguments</i>	207
5.8.3 <i>Parameter Passing</i>	210
5.8.4 <i>Members with Variable Number of Parameters</i>	214
5.8.5 <i>Pointer Parameters</i>	218
5.9 Using Tuples in Member Signatures	220

6 Designing for Extensibility 227

6.1 Extensibility Mechanisms 227

6.1.1 *Unsealed Classes* 228

6.1.2 *Protected Members* 230

6.1.3 *Events and Callbacks* 231

6.1.4 *Virtual Members* 237

6.1.5 *Abstractions (Abstract Types and Interfaces)* 239

6.2 Base Classes 242

6.3 Sealing 244

7 Exceptions 249

7.1 Exception Throwing 254

7.2 Choosing the Right Type of Exception to Throw 260

7.2.1 *Error Message Design* 264

7.2.2 *Exception Handling* 265

7.2.3 *Wrapping Exceptions* 271

7.3 Using Standard Exception Types 273

7.3.1 *Exception and SystemException* 274

7.3.2 *ApplicationException* 274

7.3.3 *InvalidOperationException* 274

7.3.4 *ArgumentException, ArgumentNullException, and*

ArgumentOutOfRangeException 275

7.3.5 *NullReferenceException, IndexOutOfRangeException, and*
AccessViolationException 276

7.3.6 *StackOverflowException* 276

7.3.7 *OutOfMemoryException* 277

7.3.8 *ComException, SEHException, and*
ExecutionEngineException 278

7.3.9 *OperationCanceledException and*
TaskCanceledException 278

7.3.10 *FormatException* 278

7.3.11 *PlatformNotSupportedException* 279

7.4 Designing Custom Exceptions 279

7.5 Exceptions and Performance 281

7.5.1 *The Tester–Doer Pattern* 281

7.5.2 *The Try Pattern* 282

8 Usage Guidelines 287	
8.1 Arrays 287	
8.2 Attributes 291	
8.3 Collections 294	
8.3.1 <i>Collection Parameters</i> 296	
8.3.2 <i>Collection Properties and Return Values</i> 298	
8.3.3 <i>Choosing Between Arrays and Collections</i> 302	
8.3.4 <i>Implementing Custom Collections</i> 303	
8.4 DateTime and DateTimeOffset 306	
8.5 ICloneable 308	
8.6 IComparable<T> and IEquatable<T> 309	
8.7 IDisposable 311	
8.8 Nullable<T> 311	
8.9 Object 312	
8.9.1 <i>Object.Equals</i> 312	
8.9.2 <i>Object.GetHashCode</i> 315	
8.9.3 <i>Object.ToString</i> 316	
8.10 Serialization 319	
8.11 Uri 321	
8.11.1 <i>System.Uri Implementation Guidelines</i> 322	
8.12 System.Xml Usage 323	
8.13 Equality Operators 324	
8.13.1 <i>Equality Operators on Value Types</i> 327	
8.13.2 <i>Equality Operators on Reference Types</i> 328	
9 Common Design Patterns 329	
9.1 Aggregate Components 329	
9.1.1 <i>Component-Oriented Design</i> 331	
9.1.2 <i>Factored Types</i> 334	
9.1.3 <i>Aggregate Component Guidelines</i> 335	
9.2 The Async Patterns 339	
9.2.1 <i>Choosing Between the Async Patterns</i> 339	
9.2.2 <i>Task-Based Async Pattern</i> 341	
9.2.3 <i>Async Method Return Types</i> 348	

9.2.4	<i>Making an Async Variant of an Existing Synchronous Method</i>	351
9.2.5	<i>Implementation Guidelines for Async Pattern Consistency</i>	355
9.2.7	<i>Classic Async Pattern</i>	361
9.2.8	<i>Event-Based Async Pattern</i>	361
9.2.9	<i>IAsyncDisposable</i>	362
9.2.10	<i>IAsyncEnumerable</i> <T>	362
9.3	Dependency Properties	365
9.3.1	<i>Dependency Property Design</i>	366
9.3.2	<i>Attached Dependency Property Design</i>	369
9.3.3	<i>Dependency Property Validation</i>	370
9.3.4	<i>Dependency Property Change Notifications</i>	371
9.3.5	<i>Dependency Property Value Coercion</i>	371
9.4	Dispose Pattern	372
9.4.1	<i>Basic Dispose Pattern</i>	375
9.4.2	<i>Finalizable Types</i>	383
9.4.3	<i>Scoped Operations</i>	387
9.4.4	<i>IAsyncDisposable</i>	391
9.5	Factories	394
9.6	LINQ Support	400
9.6.1	<i>Overview of LINQ</i>	400
9.6.2	<i>Ways of Implementing LINQ Support</i>	402
9.6.3	<i>Supporting LINQ through IEnumerable</i> <T>	402
9.6.4	<i>Supporting LINQ through IQueryable</i> <T>	403
9.6.5	<i>Supporting LINQ through the Query Pattern</i>	404
9.7	Optional Feature Pattern	408
9.8	Covariance and Contravariance	412
9.8.1	<i>Contravariance</i>	415
9.8.2	<i>Covariance</i>	417
9.8.3	<i>Simulated Covariance Pattern</i>	420
9.9	Template Method	423
9.10	Timeouts	426
9.11	XAML Readable Types	427

9.12 Operating on Buffers	430
9.12.1 <i>Data Transformation Operations</i>	445
9.12.2 <i>Writing Fixed or Predetermined Sizes to a Buffer</i>	451
9.12.3 <i>Writing Values to Buffers with the Try-Write Pattern</i>	452
9.12.4 <i>Partial Writes to Buffers and OperationStatus</i>	458
9.13 And in the End...	464

A C# Coding Style Conventions 465

A.1 General Style Conventions	466
A.1.1 <i>Brace Usage</i>	466
A.1.2 <i>Space Usage</i>	469
A.1.3 <i>Indent Usage</i>	470
A.1.4 <i>Vertical Whitespace</i>	472
A.1.5 <i>Member Modifiers</i>	473
A.1.6 <i>Other</i>	475
A.2 Naming Conventions	480

A.3 Comments	482
A.4 File Organization	483

B Obsolete Guidance 487

B.3 Obsolete Guidance from Naming Guidelines	488
B.3.8 <i>Naming Resources</i>	488
B.4 Obsolete Guidance from Type Design Guidelines	489
B.4.1 <i>Types and Namespaces</i>	489
B.5 Obsolete Guidance from Member Design	491
B.5.4 <i>Event Design</i>	491
B.7 Obsolete Guidance from Exceptions	492
B.7.4 <i>Designing Custom Exceptions</i>	492
B.8 Obsolete Guidance from Usage Guidelines	493
B.8.10 <i>Serialization</i>	493
B.9 Obsolete Guidance from Common Design Patterns	502
B.9.2 <i>The Async Patterns</i>	502
B.9.4 <i>Dispose Pattern</i>	517

C Sample API Specification	523
D Breaking Changes	529
D.1	Modifying Assemblies 530
D.1.1	<i>Changing the Name of an Assembly</i> 530
D.2	Adding Namespaces 531
D.2.1	<i>Adding a Namespace That Conflicts with an Existing Type</i> 531
D.3	Modifying Namespaces 532
D.3.1	<i>Changing the Name or Casing of a Namespace</i> 532
D.4	Moving Types 532
D.4.1	<i>Moving a Type via [TypeForwardedTo]</i> 532
D.4.2	<i>Moving a Type Without [TypeForwardedTo]</i> 533
D.5	Removing Types 533
D.5.1	<i>Removing Types</i> 533
D.6	Modifying Types 534
D.6.1	<i>Sealing an Unsealed Type</i> 534
D.6.2	<i>Unsealing a Sealed Type</i> 534
D.6.3	<i>Changing the Case of a Type Name</i> 534
D.6.4	<i>Changing a Type Name</i> 535
D.6.5	<i>Changing the Namespace for a Type</i> 535
D.6.6	<i>Adding readonly on a struct</i> 535
D.6.7	<i>Removing readonly from a struct</i> 535
D.6.8	<i>Adding a Base Interface to an Existing Interface</i> 536
D.6.9	<i>Adding the Second Declaration of a Generic Interface</i> 536
D.6.10	<i>Changing a class to a struct</i> 537
D.6.11	<i>Changing a struct to a class</i> 537
D.6.12	<i>Changing a struct to a ref struct</i> 538
D.6.13	<i>Changing a ref struct to a (Non-ref) struct</i> 538
D.7	Adding Members 539
D.7.1	<i>Masking Base Members with new</i> 539
D.7.2	<i>Adding abstract Members</i> 539
D.7.3	<i>Adding Members to an Unsealed Type</i> 539
D.7.4	<i>Adding an override Member to an Unsealed Type</i> 540
D.7.5	<i>Adding the First Reference Type Field to a struct</i> 540
D.7.6	<i>Adding a Member to an Interface</i> 541

D.8	Moving Members	541
D.8.1	<i>Moving Members to a Base Class</i>	541
D.8.2	<i>Moving Members to a Base Interface</i>	541
D.8.3	<i>Moving Members to a Derived Type</i>	542
D.9	Removing Members	542
D.9.1	<i>Removing a Finalizer from an Unsealed Type</i>	542
D.9.2	<i>Removing a Finalizer from a Sealed Type</i>	542
D.9.3	<i>Removing a Non-override Member</i>	543
D.9.4	<i>Removing an override of a virtual Member</i>	543
D.9.5	<i>Removing an override of an abstract Member</i>	543
D.9.6	<i>Removing or Renaming Private Fields on Serializable Types</i>	544
D.10	Overloading Members	544
D.10.1	<i>Adding the First Overload of a Member</i>	545
D.10.2	<i>Adding Alternative-Parameter Overloads for a Reference Type Parameter</i>	545
D.11	Changing Member Signatures	545
D.11.1	<i>Renaming a Method Parameter</i>	545
D.11.2	<i>Adding or Removing a Method Parameter</i>	546
D.11.3	<i>Changing a Method Parameter Type</i>	546
D.11.4	<i>Reordering Method Parameters of Differing Types</i>	547
D.11.5	<i>Reordering Method Parameters of The Same Type</i>	547
D.11.6	<i>Changing a Method Return Type</i>	547
D.11.7	<i>Changing the Type of a Property</i>	548
D.11.8	<i>Changing Member Visibility from public to Any Other Visibility</i>	548
D.11.9	<i>Changing Member Visibility from protected to public</i>	548
D.11.10	<i>Changing a virtual (or abstract) Member from protected to public</i>	549
D.11.11	<i>Adding or Removing the static Modifier</i>	549
D.11.12	<i>Changing to or from Passing a Parameter by Reference</i>	549
D.11.13	<i>Changing By-Reference Parameter Styles</i>	550
D.11.14	<i>Adding the readonly Modifier to a struct Method</i>	550

D.11.15 <i>Removing the readonly Modifier from a struct Method</i>	551
D.11.16 <i>Changing a Parameter from Required to Optional</i>	551
D.11.17 <i>Changing a Parameter from Optional to Required</i>	551
D.11.18 <i>Changing the Default Value for an Optional Parameter</i>	552
D.11.19 <i>Changing the Value of a const Field</i>	552
D.11.20 <i>Changing an abstract Member to virtual</i>	553
D.11.21 <i>Changing a virtual Member to abstract</i>	553
D.11.22 <i>Changing a Non-virtual Member to virtual</i>	553
D.12 Changing Behavior	553
D.12.1 <i>Changing Runtime Error Exceptions to Usage Error Exceptions</i>	553
D.12.2 <i>Changing Usage Error Exceptions to Functioning Behavior</i>	554
D.12.3 <i>Changing the Type of Values Returned from a Method</i>	554
D.12.4 <i>Throwing a New Type of Error Exception</i>	555
D.12.5 <i>Throwing a New Type of Exception, Derived from an Existing Thrown Type</i>	555
D.13 A Final Note	556
<i>Glossary</i>	557
<i>Index</i>	563



Figures

FIGURE 2-1: *Learning curve of a multiframework platform* 13

FIGURE 2-2: *Learning curve of a progressive framework platform* 14

FIGURE 4-1: *The logical grouping of types* 83

FIGURE 9-1: *Query pattern method signatures* 404

This page intentionally left blank



Tables

TABLE 3-1: *Capitalization Rules for Different Types of Identifiers* 44

TABLE 3-2: *Capitalization and Spelling for Common Compound Words and Common Terms* 49

TABLE 3-3: *Alternative Spellings to Avoid Diacritical Marks* 55

TABLE 3-4: *CLR Type Names for Language-Specific Type Names* 57

TABLE 3-5: *Name Rules for Types Derived from or Implementing Certain Core Types* 71

TABLE 5-1: *Operators and Corresponding Method Names* 196

TABLE 8-1: *.Net Serialization Technologies* 493

This page intentionally left blank



Foreword

When we designed the .NET platform, we wanted it to be the most productive platform for enterprise application development of the time. Twenty years ago, that meant client-server applications hosted on dedicated hardware.

Today, we find ourselves in the midst of one of the biggest paradigm shifts in the industry: the move to cloud computing. Such transformations bring new opportunities for businesses but can be tricky for existing platforms, as they need to adapt to often different requirements imposed by the new kinds of applications that developers want to write.

The .NET platform has transitioned quite successfully, and I think one of the main reasons is that we designed it carefully and deliberately, focusing not only on productivity, consistency, and simplicity, but also on making sure that it can evolve over time. .NET Core represents such evolution with advances important to cloud application developers: performance, resource utilization, container support, and others.

This third edition of *Framework Design Guidelines* adds guidelines related to changes that the .NET team adopted during transition from the world of client-server application to the world of the Cloud.

—Scott Guthrie
Redmond, WA
January 2020



This page intentionally left blank



Foreword to the Second Edition

When the .NET Framework was first published, I was fascinated by the technology. The benefits of the CLR (Common Language Runtime), its extensive APIs, and the C# language were immediately obvious. But underneath all the technology were a common design for the APIs and a set of conventions that were used everywhere. This was the .NET culture. Once you had learned a part of it, it was easy to translate this knowledge into other areas of the framework.

For the past 16 years, I have been working on open source software. Since contributors span not only multiple backgrounds but also multiple years, adhering to the same style and coding conventions has always been very important. Maintainers routinely rewrite or adapt contributions to software to ensure that code adheres to project coding standards and style. It is always better when contributors and people who join a software project follow conventions used in an existing project. The more information that can be conveyed through practices and standards, the simpler it becomes for future contributors to get up-to-speed on a project. This helps the project converge code, both old and new.

As both the .NET Framework and its developer community have grown, new practices, patterns, and conventions have been identified. Brad and Krzysztof have become the curators who turned all of this new knowledge into the present-day guidelines. They typically blog about a new convention, solicit feedback from the community, and keep track of these



guidelines. In my opinion, their blogs are must-read documents for everyone who is interested in getting the most out of the .NET Framework.

The first edition of *Framework Design Guidelines* became an instant classic in the Mono community for two valuable reasons. First, it provided us a means of understanding why and how the various .NET APIs had been implemented. Second, we appreciated it for its invaluable guidelines that we too strived to follow in our own programs and libraries. This new edition not only builds on the success of the first but has been updated with new lessons that have since been learned. The annotations to the guidelines are provided by some of the lead .NET architects and great programmers who have helped shape these conventions.

In conclusion, this text goes beyond guidelines. It is a book that you will cherish as the “classic” that helped you become a better programmer, and there are only a select few of those in our industry.

—Miguel de Icaza

Boston, MA

October 2008



Foreword to the First Edition

In the early days of development of the .NET Framework, before it was even called that, I spent countless hours with members of the development teams reviewing designs to ensure that the final result would be a coherent platform. I have always felt that a key characteristic of a framework must be consistency. Once you understand one piece of the framework, the other pieces should be immediately familiar.

As you might expect from a large team of smart people, we had many differences of opinion—there is nothing like coding conventions to spark lively and heated debates. However, in the name of consistency, we gradually worked out our differences and codified the result into a common set of guidelines that allow programmers to understand and use the framework easily.

Brad Abrams, and later Krzysztof Cwalina, helped capture these guidelines in a living document that has been continuously updated and refined during the past six years. The book you are holding is the result of their work.

The guidelines have served us well through three versions of the .NET Framework and numerous smaller projects, and they are guiding the development of the next generation of APIs for the Microsoft Windows operating system.



With this book, I hope and expect that you will also be successful in making your frameworks, class libraries, and components easy to understand and use.

Good luck and happy designing.

—Anders Hejlsberg

Redmond, WA

June 2005



Preface

This book, *Framework Design Guidelines*, presents best practices for designing frameworks, which are reusable object-oriented libraries. The guidelines are applicable to frameworks in various sizes and scales of reuse, including the following:

- Large system frameworks, such as the core libraries in .NET, usually consisting of thousands of types and used by millions of developers.
- Medium-size reusable layers of large distributed applications or extensions to system frameworks, such as the Azure SDKs or a game engine.
- Small components shared among several applications, such as a grid control library.

It is worth noting that this book focuses on design issues that directly affect the programmability of a framework (publicly accessible APIs¹). As a result, we generally do not cover much in terms of implementation details. Just as a user interface design book doesn't cover the details of how to implement hit testing, this book does not describe how to implement a binary sort, for example. This scope allows us to provide a definitive guide for framework designers instead of being yet another

1. This includes public types, and the public, protected, and explicitly implemented members of these types.

book about programming. The book assumes the reader has basic familiarity with programming in .NET already.

These guidelines were created in the early days of .NET Framework development. They started as a small set of naming and design conventions but have been enhanced, scrutinized, and refined to a point where they are generally considered the canonical way to design frameworks at Microsoft. They carry the experience and cumulative wisdom of thousands of developer hours over two decades of .NET. We tried to avoid basing the text purely on some idealistic design philosophies, and we think its day-to-day use by development teams at Microsoft has made it an intensely pragmatic book.

The book contains many annotations that explain trade-offs, explain history, amplify, or provide critiquing views on the guidelines. These annotations are written by experienced framework designers, industry experts, and users. They are the stories from the trenches that add color and setting for many of the guidelines presented.

To make them more easily distinguished in text, namespace names, classes, interfaces, methods, properties, and types are set in a monospace font.

Guideline Presentation

The guidelines are organized as simple recommendations using **DO**, **CONSIDER**, **AVOID**, and **DO NOT**. Each guideline describes either a good or bad practice, and all have a consistent presentation. Good practices have a ✓ in front of them, and bad practices have an ✗ in front of them. The wording of each guideline also indicates how strong the recommendation is. For example, a **DO** guideline is one that should always² be followed (all examples are from this book):

✓ **DO** name custom attribute classes with the suffix “Attribute.”

```
public class ObsoleteAttribute : Attribute { ... }
```

2. *Always* might be a bit too strong a word. There are guidelines that should literally be always followed, but they are extremely rare. In contrast, you probably need to have a really unusual case for breaking a **DO** guideline and still have it be beneficial to the users of the framework.

On the other hand, **CONSIDER** guidelines should generally be followed, but if you fully understand the reasoning behind a guideline and have a good reason to not follow it anyway, you should not feel bad about breaking the rules:

- ✓ **CONSIDER** defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects.

Similarly, **DO NOT** guidelines indicate something you should almost never do:

- ✗ **DO NOT** provide set-only properties or properties with the setter having broader accessibility than the getter.

Less strong, **AVOID** guidelines indicate that something is generally not a good idea, but there are known cases where breaking the rule makes sense:

- ✗ **AVOID** using `ICollection<T>` or `ICollection` as a parameter just to access the `Count` property.

Some more complex guidelines are followed by additional background information, illustrative code samples, and rationale:

- ✓ **DO** implement `IEquatable<T>` on value types.

The `Object.Equals` method on value types causes boxing and its default implementation is not very efficient because it uses reflection. `IEquatable<T>.Equals` can offer much better performance and can be implemented so it does not cause boxing.

```
public struct Int32 : IEquatable<Int32> {
    public bool Equals(Int32 other){ ... }
}
```

Language Choice and Code Examples

One of the goals of the Common Language Runtime (CLR) is to support a variety of programming languages: those with implementations provided

by Microsoft, such as C++, VB, C#, F#, IronPython, and PowerShell, as well as third-party languages such as Eiffel, COBOL, Fortran, and others. Therefore, this book was written to be applicable to a broad set of languages that can be used to develop and consume modern frameworks.

To reinforce the message of multilanguage framework design, we considered writing code examples using several different programming languages. However, we decided against this. We felt that using different languages would help to carry the philosophical message, but it could force readers to learn several new languages, which is not the objective of this book.

We decided to choose a single language that is most likely to be readable to the broadest range of developers. We picked C#, because it is a simple language from the C family of languages (C, C++, Java, and C#), a family with a rich history in framework development.

Choice of language is close to the hearts of many developers, and we offer apologies to those who are uncomfortable with our choice.

About This Book

This book offers guidelines for framework design from the top down.

Chapter 1, “Introduction,” is a brief orientation to the book, describing the general philosophy of framework design. This is the only chapter without guidelines.

Chapter 2, “Framework Design Fundamentals,” offers principles and guidelines that are fundamental to overall framework design.

Chapter 3, “Naming Guidelines,” contains common design idioms and naming guidelines for various parts of a framework, such as namespaces, types, and members.

Chapter 4, “Type Design Guidelines,” provides guidelines for the general design of types.

Chapter 5, “Member Design,” takes a further step and presents guidelines for the design of members of types.

Chapter 6, “Designing for Extensibility,” presents issues and guidelines that are important to ensure appropriate extensibility in your framework.

Chapter 7, “Exceptions,” presents guidelines for working with exceptions, the preferred error reporting mechanisms.

Chapter 8, “Usage Guidelines,” contains guidelines for extending and using types that commonly appear in frameworks.

Chapter 9, “Common Design Patterns,” offers guidelines and examples of common framework design patterns.

Appendix A, “C# Coding Style Conventions,” describes coding conventions used by the team that produces and maintains the core libraries in .NET.

Appendix B, “Obsolete Guidance,” contains guidance from previous editions of this book that applies to features or concepts that are no longer recommended.

Appendix C, “Sample API Specification,” is a sample of an API specification that framework designers within Microsoft create when designing APIs.

Appendix D, “Breaking Changes,” explores various kinds of changes that can negatively impact your users from one version to the next.

Register your copy of *Framework Design Guidelines, Third Edition*, on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135896464) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank



Acknowledgments

This book, by its nature, is the collected wisdom of many hundreds of people, and we are deeply grateful to all of them.

Many people within Microsoft have worked long and hard, over a period of years, proposing, debating, and finally writing many of these guidelines. Although it is impossible to name everyone who has been involved, a few deserve special mention: Chris Anderson, Erik Christensen, Jason Clark, Joe Duffy, Patrick Dussud, Anders Hejlsberg, Jim Miller, Michael Murray, Lance Olson, Eric Gunnerson, Dare Obasanjo, Steve Starck, Kit George, Mike Hillberg, Greg Schechter, Mark Boulter, Asad Jawahar, Justin Van Patten, and Mircea Trofin.

We'd also like to thank the annotators: Mark Alcazar, Chris Anderson, Christopher Brumme, Pablo Castro, Jason Clark, Steven Clarke, Joe Duffy, Patrick Dussud, Kit George, Jan Gray, Brian Grunkemeyer, Eric Gunnerson, Phil Haack, Anders Hejlsberg, Jan Kotas, Immo Landwerth, Rico Mariani, Anthony Moore, Vance Morrison, Christophe Nasarre, Dare Obasanjo, Brian Pepin, Jon Pincus, Jeff Prosise, Brent Rector, Jeffrey Richter, Greg Schechter, Chris Sells, Steve Starck, Herb Sutter, Clemens Szyperski, Stephen Toub, Mircea Trofin, and Paul Vick. Their insights provide much needed commentary, color, humor, and history that add tremendous value to this book.



For all of the help, reviews, and support, both technical and moral, we thank Martin Heller and Stephen Toub. And for their insightful and helpful comments, we appreciate Pierre Nallet, George Byrkit, Khristof Falk, Paul Besley, Bill Wagner, and Peter Winkler.

We would also like to give special thanks to Susann Ragsdale, who turned this book from a semi-random collection of disconnected thoughts into seamlessly flowing prose. Her flawless writing, patience, and fabulous sense of humor made the process of writing this book so much easier.



About the Authors

Krzysztof Cwalina is a software architect at Microsoft. He was a founding member of the .NET Framework team and throughout his career has designed many .NET APIs. He is currently working on helping various teams across Microsoft design reusable APIs for many different programming languages. Krzysztof graduated with a B.S. and an M.S. in computer science from the University of Iowa.

Jeremy Barton is an engineer on the .NET Core Libraries team. After a decade of designing and developing small frameworks in C#, he joined the .NET team in 2015 to help get the cryptography types working across all platforms in the then-new .NET Core project. Jeremy graduated with a B.S. in computer science and mathematics from Rose-Hulman Institute of Technology.

Brad Abrams was a founding member of the Common Language Runtime and .NET Framework teams at Microsoft Corporation. He has been designing parts of the .NET Framework since 1998. Brad started his framework design career building the Base Class Library (BCL) that ships as a core part of the .NET Framework. Brad was also the lead editor on the Common Language Specification (CLS), the .NET Framework Design Guidelines, and the libraries in the ECMA\ISO CLI Standard. Brad has authored and coauthored multiple publications, including *Programming in the .NET Environment* and *.NET Framework Standard Library Annotated*.



Reference, Volumes 1 and 2. Brad graduated from North Carolina State University with a B.S. in computer science. You can find his most recent musings on his blog at <http://blogs.msdn.com/BradA>. Brad is currently Group Product Manager at Google where he is incubating new projects for the Google Assistant.



About the Annotators

Mark Alcazar has been at Microsoft for the last 23 years, where he's spent most of his career on UI frameworks and angle-brackets. Mark has worked on Internet Explorer, WPF, Silverlight, and the last several releases of the Windows developer platform. Mark loves snowboarding, sailing, and cooking. He has a B.S. from the University of the West Indies and an M.S. from the University of Pennsylvania, and lives in Seattle with his wife and two kids.

Chris Anderson worked at Microsoft for 22 years, on a variety of projects, but specialized in the design and architecture of .NET technologies used to implement the next generation of applications and services. Chris has written numerous articles and white papers, and he has presented and been a keynote speaker at numerous conferences (e.g., Microsoft Professional Developers Conference, Microsoft TechEd, WinDev, DevCon) worldwide. He has a very popular blog at www.simplegeek.com.

Christopher Brumme joined Microsoft in 1997, when the Common Language Runtime (CLR) team was being formed. Since then, he has contributed to the execution engine portions of the codebase and more broadly to the design. He is currently focused on concurrency issues in managed code. Prior to joining the CLR team, Chris was an architect at Borland and Oracle.

Pablo Castro is a distinguished engineer at Microsoft. He's currently part of the Azure Data group, where he's the director of engineering for Azure Synapse/SQL and Azure Cognitive Search. Prior to his current role



Pablo was the director of engineering and data science for the Applied AI group in Azure, and before that he worked on multiple products within the database systems group, including SQL Server, .NET, Entity Framework/LINQ, and OData.

Jason Clark works as a software architect for Microsoft. His Microsoft software engineering credits include three versions of Windows, three releases of the .NET Framework, and WCF. In 2000, he published his first book on software development, and he continues to contribute to magazines and other publications. He is currently responsible for the Visual Studio Team System Database Edition. Jason's only other passions are his wife and kids, with whom he happily lives in the Seattle area.

Steven Clarke has been a user experience researcher in the Developer Division at Microsoft since 1999. His main interests are observing, understanding, and modeling the experiences that developers have with APIs so as to help design APIs that provide an optimal experience to their users.

Joe Duffy is the founder and CEO of Pulumi, a Seattle start-up giving developers and infrastructure teams cloud superpowers. Prior to founding Pulumi in 2017, Joe held leadership roles at Microsoft in the Developer Division, Operating Systems Group, and Microsoft Research. Most recently Joe was director for engineering and technical strategy for Microsoft's developer tools, leading key technical architecture initiatives, in addition to managing the groups building the C#, C++, Visual Basic, and F# languages, as well as IoT and Visual Studio IDE, compiler, and static analysis services. Joe was instrumental in Microsoft's overall open source transformation and assembled the initial team who took .NET open source and to new platforms. Joe has more than 20 years of professional software experience, has written two books, and still loves to code.

Patrick Dussud is a technical fellow at Microsoft, where he serves as the chief architect of both the CLR and the .NET Framework architecture groups. He works on .NET Framework issues across the company, helping development teams best utilize the CLR. He specifically focuses on taking advantage of the abstractions the CLR provides to optimize program execution.



Kit George is a program manager on the .NET Framework team at Microsoft. He graduated in 1995 with a B.A. in psychology, philosophy, and mathematics from Victoria University of Wellington (New Zealand). Prior to joining Microsoft, he worked as a technical trainer, primarily in Visual Basic. He participated in the design and implementation of the first two releases of .NET Framework for the last two years.

Jan Gray is a software architect at Microsoft who now works on concurrency programming models and infrastructure. He was previously a CLR performance architect, and in the 1990s he helped write the early MS C++ compilers (e.g., semantics, runtime object model, precompiled headers, PDBs, incremental compilation, and linking) and Microsoft Transaction Server. Jan's interests include building custom multiprocessors in FPGAs.

Brian Grunkemeyer has been a software design engineer on the .NET Framework team at Microsoft since 1998. He implemented a large portion of the Framework Class Libraries and contributed to the details of the classes in the ECMA/ISO CLI standard. Brian is currently working on future versions of the .NET Framework, including areas such as generics, managed code reliability, versioning, contracts in code, and improving the developer experience. He has a B.S. in computer science with a double major in cognitive science from Carnegie Mellon University.

Eric Gunnerson found himself at Microsoft in 1994 after working in the aerospace and going-out-of-business industries. He has worked on the C++ compiler team, as a member of the C# language design team, and as an early thought follower on the DevDiv community effort. He worked on the Windows DVD Maker UI during Vista and joined the Microsoft HealthVault team in early 2007. He spends his free time cycling, skiing, cracking ribs, building decks, blogging, and writing about himself in the third person.

Phil Haack is the founder of Haacked LLC, where he coaches software organizations to become the best versions of themselves. To do this, Phil draws upon his more than 20 years of experience in the software industry. Most recently, he was a director of engineering at GitHub and helped make GitHub friendly to developers on the Microsoft platform. Prior to his work

at GitHub, he was a senior program manager at Microsoft responsible for shipping ASP.NET MVC and NuGet, among other projects. These products had permissive open source licenses and ushered in Microsoft's open source era. Phil is a co-author of *GitHub For Dummies* as well as the popular Professional ASP.NET MVC series, and regularly speaks at conferences around the world. He's also made several appearances on technology podcasts such as .NET Rocks, Hanselminutes, Herding Code, and The Official jQuery Podcast. You can find him sharing his thoughts at <https://haacked.com> or on Twitter, <https://twitter.com/haacked>.

Anders Hejlsberg is a technical fellow in the Developer Division at Microsoft. He is the chief designer of the C# programming language and a key participant in the development of the .NET Framework. Before joining Microsoft in 1996, Anders was a principal engineer at Borland International. As one of the first employees of Borland, he was the original author of Turbo Pascal and later worked as the chief architect of the Delphi product line. Anders studied engineering at the Technical University of Denmark.

Jan Kotas has worked on the .NET Runtime at Microsoft since 2001. He has an eye on striking the right balance between productivity, performance, security and reliability for the .NET platform. Over the years, he has touched nearly all areas of the .NET runtime, including ports to new architectures, ahead-of-time compilers, and many optimizations. He graduated in 1998 with a master's degree in computer science from Charles University in Prague (Czech Republic).

Immo Landwerth is a program manager on the .NET Framework team at Microsoft. He specializes in API design, the Base Class Libraries (BCL), and open source in .NET. He tweets in GIFs.

Rico Mariani has been coding professionally since 1980 with experience in everything from real-time controllers to flagship development systems. Rico was at Microsoft from 1988 to 2017 working on language products, online properties, operating systems, web browsers, and more. In 2017, Rico joined Facebook to work on Facebook Messenger, bringing his passion for high quality and performance with him. Rico's interests



include compilers and language theory, databases, 3D art, and good fiction.

Anthony Moore is a development lead for the Connected Systems Division. He was the development lead for the Base Class Libraries of the CLR from 2001 to 2007, spanning FX V1.0 to FX 3.5. Anthony joined Microsoft in 1999 and initially worked on Visual Basic and ASP.NET. Before that, he worked as a corporate developer for eight years in his native Australia, including a three-year period working in the snack food industry.

Vance Morrison is a performance architect for the .NET Runtime at Microsoft. He involves himself with most aspects of runtime performance, with current attention devoted to improving start-up time. He has been involved in designs of components of the .NET runtime since its inception. He previously drove the design of the .NET Intermediate Language (IL) and has been the development lead for the JIT compiler for the runtime.

Christophe Nasarre is a software engineer in the Performance team at Criteo. During his spare time, Christophe writes .NET-related posts on <https://medium.com/@chnasarre> and provides tools and code samples from <https://github.com/chrisnas>. He is also a Microsoft MVP in Developer Technologies.

Dare Obasanjo is a program manager on the MSN Communication Services Platform team at Microsoft. He brings his love of solving problems with XML to building the server infrastructure utilized by the MSN Messenger, MSN Hotmail, and MSN Spaces teams. He was previously a program manager on the XML team responsible for the core XML application programming interfaces and W3C XML Schema-related technologies in the .NET Framework.

Brian Pepin is a software developer at Microsoft and is currently working on the Xbox system software. He's been involved in developer tools and frameworks for 16 years and has provided input on the design of Visual Basic 5, Visual J++, the .NET Framework, WPF, Silverlight, and more than one unfortunate experiment that luckily never made it to market.

Jonathan Pincus was a senior researcher in the Systems and Networking Group at Microsoft Research, where he focused on the security, privacy, and reliability of software and software-based systems. He was previously founder and CTO of Intrinsa and worked in design automation (placement and routing for ICs and CAD frameworks) at GE Calma and EDA Systems.

Jeff Prosise is a co-founder of Wintellect who makes his living writing software and helping others do the same. He has written nine books and hundreds of magazine articles, trained thousands of developers at Microsoft, and spoken at some of the world's largest software conferences. Jeff's passion is teaching software developers how to build cloud-based apps with Microsoft Azure and introducing them to the wonders of AI and machine learning. In his spare time, Jeff builds and flies large radio-controlled jets and travels to development shops, universities, and research institutions around the world educating them about Azure and AI.

Brent Rector is a program manager at Microsoft on a technical strategy incubation effort. He has more than 30 years of experience in the software development industry in the production of programming language compilers, operating systems, ISV applications, and other products. Brent is the author and coauthor of numerous Windows software development books, including *ATL Internals*, *Win32 Programming* (both Addison-Wesley), and *Introducing WinFX* (Microsoft Press). Prior to joining Microsoft, Brent was the president and founder of Wise Owl Consulting, Inc., and chief architect of its premier .NET obfuscator, Demeanor for .NET.

Jeffrey Richter is a Microsoft Azure software architect who is best known as having authored the best-selling *Windows via C/C++* and *CLR via C#* books. Most recently, he's produced the free Architecting Distributed Cloud Applications video series available at <http://aka.ms/RichterCloud> Apps and other videos available at <http://WintellectNOW.com>. Jeffrey was a consultant and co-founder of Wintellect, a software consulting and training company.

Greg Schechter is a Big Tech industry veteran, having worked at Sun Microsystems from 1988 to 1994 and at Microsoft from 1994 to 2010, primarily on API implementation and API design for more than 20 years. His



experience is mostly in the 2D and 3D graphics realm, but also in media, imaging, general user interface systems, and asynchronous programming. In 2011, Greg joined Facebook as one of the first dozen or so Seattle employees, and is currently at Facebook London working as a software engineer in the Ads Infrastructure organization. Beyond all of that, Greg also loves to write about himself in the third person.

Chris Sells, in his past life, was deeply involved in .NET since the beta and has done much writing and speaking on the subject. Currently he is a Google Product Manager on the Flutter development experience. He still enjoys long walks on the beach and various technologies.

Steve Starck is a technical lead on the ADO.NET team at Microsoft, where he has been developing and designing data access technologies, including ODBC, OLE DB, and ADO.NET, for the past ten years.

Herb Sutter is a leading authority on software development. During his career, Herb has been the creator and principal designer of several major commercial technologies, including the PeerDirect peer replication system for heterogeneous distributed databases, the C++/CLI language extensions to C++ for .NET programming, and most recently the Concur concurrent programming model. Currently a software architect at Microsoft, he also serves as chair of the ISO C++ standards committee and is the author of four acclaimed books and hundreds of technical papers and articles on software development topics.

Clemens Szyperski joined Microsoft Research as a software architect in 1999. He focuses on leveraging component software to effectively build new kinds of software. Clemens is cofounder of Oberon Microsystems and its spin-off Esmertec, and he was an associate professor at the School of Computer Science, Queensland University of Technology, Australia, where he retains an adjunct professorship. He is the author of the Jolt award-winning *Component Software* (Addison-Wesley) and the coauthor of *Software Ecosystem* (MIT Press). He has a Ph.D. in computer science from the Swiss Federal Institute of Technology in Zurich and an M.S. in electrical engineering/computer engineering from the Aachen University of Technology.

Stephen Toub is a partner software engineer at Microsoft. He has computer science degrees from Harvard University and New York University, and has spent many years developing .NET, with a focus on its libraries, and in particular with an eye toward performance, parallelism, and asynchrony. He was instrumental in taking .NET open source and cross-platform, and is thrilled by all the possibilities that .NET will enable in the future.

Mircea Trofin is a software engineer at Google, working on compiler optimization problems. He has previously worked at Microsoft as part of the .NET team.

Paul Vick was the language architect for Visual Basic during the transition to .NET and led the language design team for the first several releases of the language. Paul originally began his career working at Microsoft in 1992 on the Microsoft Access team, shipping versions 1.0 through 97 of Access. In 1998, he moved to the Visual Basic team, participating in the design and implementation of the Visual Basic compiler and driving the redesign of the language for the .NET Framework. He is the author of the Visual Basic .NET Language Specification and the Addison-Wesley book *The Visual Basic .NET Language*. His weblog can be found at www.panopticoncentral.net.

 1

Introduction

If you could stand over the shoulder of every developer who is using your framework to write code and explain how it is supposed to be used, guidelines would not be necessary. The guidelines presented in this book give you, as the framework author, a palette of tools that allow you to create a common language between framework authors and the developers who will use the frameworks. For example, exposing an operation as a property instead of exposing it as a method conveys vital information about how that operation is to be used.

In the early days of the PC era, the main tools for developing applications were a programming language compiler, a very small set of standard libraries, and the raw operating system application programming interfaces (APIs)—a very basic set of low-level programming tools.

Even as developers were building applications using such basic tools, they were discovering an increasing amount of code that was repetitive and could be abstracted away through higher-level APIs. Operating system vendors noticed that they could make it cheaper for developers to create applications for their systems if they provided them with such higher-level APIs. The number of applications that could run on the system would increase, which would then make the system more appealing to end users who demanded a variety of applications. Also, independent

tool and component vendors quickly recognized the business opportunities offered by raising the API abstraction level.

In parallel, the industry slowly began to accept object-oriented design and its emphasis on extensibility and reusability.¹ When reusable library vendors adopted object-oriented programming (OOP) for the development of their high-level APIs, the concept of a *framework* was born. That is, application developers were no longer expected to write most of the application from scratch. The framework would provide most of the needed pieces, which would then be customized and connected² to form applications.

As more vendors started to provide components that could be reused by stitching them together into a single application, developers noticed that some of the components did not fit together well. Their applications looked and worked like a house built by different contractors who never talked to each other. Likewise, as a larger percentage of application source code was devoted to API calls rather than standard language constructs, developers started to complain that they now had to read and write multiple languages: one programming language and several “languages” of the components they wanted to reuse. This had a significant negative impact on developer productivity—and productivity is one of the main factors in the success of any framework. It became clear that there was a need for common rules that would ensure consistency and seamless integration of reusable components.

Most of today’s application development platforms spell out some kind of design conventions to be used when designing frameworks for the

-
1. Object-oriented languages are not the only languages well suited for developing extensible and reusable libraries, but they played a key role in popularizing the concepts of reusability and extensibility. Extensibility and reusability are a large part of the philosophy of object-oriented programming (OOP), and the adoption of OOP contributed to increased awareness of their benefits.
 2. There has been a great deal of recent criticism of object-oriented (OO) design, which claims that the promise of reusability never materialized. OO design is not a guarantee of reusability, but we are not sure that it was ever promised. However, OO design provides natural constructs to express units of reusability (types), to communicate and control extensibility points (virtual members), and to facilitate decoupling (abstractions).

platform. Frameworks that do not follow such conventions, and so do not integrate well with the platform, either are a source of constant frustration to those trying to use them, or are at a competitive disadvantage and ultimately fail in the marketplace. The ones that succeed are often described as self-consistent, making sense, and well designed.

1.1 Qualities of a Well-Designed Framework

The question is, then, what defines a well-designed framework, and how do you get there? Many factors—such as performance, reliability, security, dependency management, and so on—affect software quality. Frameworks, of course, must adhere to these same quality standards. The difference between frameworks and other kinds of software is that frameworks are made up of reusable APIs, a factor that presents a set of special considerations in designing quality frameworks.

1.1.1 Well-Designed Frameworks Are Simple

Most frameworks do not lack power, because more features are reasonably easy to add as requirements become clearer. Conversely, simplicity often gets sacrificed when schedule pressure, feature creep, or the desire to satisfy every little corner-case scenario takes over the development process. However, simplicity is a must-have feature of every framework. If you have any second thoughts about the complexity of a design, it is almost always much better to cut the feature from the current release and spend more time getting the design right for the next release. As framework designers often say, “You can always add; you cannot ever remove.” If the design does not feel right, and you ship it anyway, you are likely to regret having done so.

■ **CHRIS SELLS** As a test of whether an API is “simple,” I like to submit it to a test I call “client-first programming.” If you say what your library does and ask a developer to write a program against what he or she expects such a library to look like (without actually looking at your library), does the developer come up with something substantially similar to what you’ve produced? Do this with a few developers. If the majority of them write similar things without matching what you’ve produced, they’re right, you’re wrong, and your library should be updated appropriately.

I find this technique so useful that I often design library APIs by writing the client code I wish I could write and then just implement the library to match that. Of course, you have to balance simplicity with the intrinsic complexity of the functionality you’re trying to provide, but that’s what your computer science degree is for!

Many of the guidelines described in this book are motivated by the desire to strike the right balance between power and simplicity. In particular, Chapter 2 extensively covers some basic techniques used by the most successful framework designers to design the right level of simplicity and power.

1.1.2 Well-Designed Frameworks Are Expensive to Design

Good framework design does not happen magically. It is hard work that consumes lots of time and resources. If you are not willing to invest real money in the design, you should not expect to create a well-designed framework.

■ **STEPHEN TOUB** For those of us who find ourselves participating in more meetings than we’d ideally like, it can be an interesting and humbling hobby to calculate the cost of a meeting: number of people in the room, multiplied by the estimated average hourly income of those in attendance, multiplied by the frequency of the meeting. In that light, one of the most expensive meetings I attend is our .NET API review meeting, where we review planned new API surface area being exposed and decide whether and how to move forward with it. This expense is worth it, however, as inconsistencies and mistakes in API design end up having an even larger negative cost, and well-designed, integrated APIs end up having a worth that’s “greater than the sum of their parts.”

Framework design should be an explicit and distinct part of the development process;³ it must be explicit because it needs to be appropriately planned, staffed, and executed, and it must be distinct because it cannot just be a side effect of the implementation process. It is too often the case that the framework consists of whatever types and members happen to remain public after the implementation process ends.

The best framework designs are done either by people whose explicit responsibility is framework design, or by people who can put the framework designer's hat on at the right time in the development process. Mixing the responsibilities is a mistake and leads to designs that expose implementation details, which should not be visible to the end user of the framework.⁴

JEREMY BARTON It is surprisingly hard to produce a good API as an area expert. You know how complicated the problem space really is, and your proposal has reduced the problem to its essence—it's the pinnacle of simplicity. Except it isn't. Someone unfamiliar with the details will likely still tell you it's too complicated, and they're probably right.

Even API proposals made by members of the .NET API review team get altered during the review process. An API review process, which involves someone who *isn't* an expert in the feature's domain, significantly improves the quality of any API proposal.

-
3. Do not misunderstand this as an endorsement of heavy up-front design processes. In fact, heavy API design processes lead to waste, because APIs often need to be tweaked after they are implemented. However, the API design process has to be separate from the implementation process and has to be incorporated in every part of the product cycle: the planning phase (which APIs do our users need?), the design process (what are the functionality trade-offs we are willing to make to get the right framework APIs?), the development process (have we allocated time to try to use the framework to see how the end result feels?), the beta process (have we allocated time for the costly API redesign?), and maintenance (are we decreasing the design quality as we evolve the framework?).
 4. Prototyping is one of the most important parts of the framework design process, but prototyping is very different from implementation.

1.1.3 Well-Designed Frameworks Are Full of Trade-Offs

There is no such thing as the perfect design. Design is all about making trade-offs, and to make the right decisions, you need to understand the options, their benefits, and their drawbacks. If you find yourself thinking you have a design without trade-offs, you are probably missing something big instead of finding the silver bullet.

The practices described in this book are presented as guidelines, rather than rules, precisely because framework design requires managing trade-offs. Some of the guidelines discuss the trade-offs involved and even provide alternatives that need to be considered in specific situations.

1.1.4 Well-Designed Frameworks Borrow from the Past

Most successful frameworks borrow from and build on top of existing proven designs. It is possible—and often desirable—to introduce completely novel solutions into framework design, but it should be done with the utmost caution. As the number of new concepts increases, the probability that the overall design will be right decreases.

■ **CHRIS SELLS** Please don't innovate in library design. Make the API to your library as boring as possible. You want the functionality to be interesting, not the API.

■ **JEREMY BARTON** Completely novel solutions are best reserved for cross-cutting areas. They help you understand the true value of the novel solution, and your users to understand the broad impact of why they need to learn "just one more thing."

Change is terrible, unless it's great. Being just a little bit better in a small area probably isn't worth the time it took your users to learn how to work with your new approach.

The guidelines contained in this book are based on the experiences we gained while designing the core libraries for .NET; they encourage borrowing from things that worked and withstood the test of time, and warn about ones that did not. We encourage you to use these good practices as a

starting point, and to improve on them. Chapter 9 talks extensively about common design approaches that worked.

1.1.5 Well-Designed Frameworks Are Designed to Evolve

Thinking about how to evolve your framework in the future will require thinking about which trade-offs you will need to make. On the one hand, a framework designer can certainly take more time and put more painstaking effort into the design process, and occasionally additional complexity can be introduced “just in case.” On the other hand, careful consideration can save you from shipping something that will degrade over time or, even worse, something that will not be able to preserve backward compatibility.⁵ As a general rule, it is better to postpone a feature until the next release than to do it halfway in the current release.

Whenever you make a design trade-off, you should determine how the decision will affect your ability to evolve the framework in the future. The guidelines presented in this book take this important concern into account.

1.1.6 Well-Designed Frameworks Are Integrated

Modern frameworks need to be designed to integrate well with a large ecosystem of different development tools, programming languages, application models, and so on. Cloud computing and other server-oriented workloads mean that the era of frameworks designed for specific application models is over. This is also true of frameworks designed without thinking about proper tool support or integration with programming languages used by the developer community.

1.1.7 Well-Designed Frameworks Are Consistent

Consistency is the key characteristic of a well-designed framework. It is one of the most important factors affecting productivity. A consistent framework allows for transfer of knowledge between the parts of the framework that a developer knows to the parts that the developer is trying

5. Backward compatibility is not discussed in detail in this book, but it should be considered one of the basics of framework design, together with reliability, security, and performance.

to learn. Consistency also helps developers quickly recognize which parts of the design are truly unique to the particular feature area and so require special attention, and which are just the same old common design patterns and idioms.

Consistency is probably the main theme of this book. Almost every single guideline is partially motivated by consistency, but Chapters 3 to 5 are probably the most important ones, because they describe the core consistency guidelines. We offer these guidelines to help you make your framework successful. The next chapter presents guidelines for general library design.

2

Framework Design Fundamentals

A SUCCESSFUL GENERAL-PURPOSE framework must be designed for a broad range of developers with different requirements, skills, and backgrounds. One of the biggest challenges facing framework designers is to offer both power and simplicity to this diverse group of users.

Another important goal of a framework designer must be to offer a unified programming model regardless of the kind of application¹ a developer writes or, in case of a multi-language runtime, the programming language the developer uses.

By using widely accepted general software design principles and following the guidelines described in this chapter, you can create a framework that offers consistent functionality that is appropriate for a broad range of developers who are building different kinds of applications using a variety of programming languages.

1. For example, a framework component should have the same programming model whether it is used in a console, Windows Forms, or an ASP.NET application, if at all possible.

✓ **DO** design frameworks that are both powerful and easy to use.

A well-designed framework makes implementing simple scenarios easy. At the same time, it does not prohibit implementing more advanced scenarios, though these might be more difficult. As Alan Kay said, “Simple things should be simple and complex things should be possible.”

This guideline is also related to the Pareto principle (the 80/20 rule), which says that in any situation, 20 percent will be important, and 80 percent will be trivial. When designing a framework, concentrate on the important 20 percent of scenarios and APIs. In other words, invest in the design of the most commonly used parts of the framework.

✓ **DO** understand and explicitly design for a broad range of developers with different programming styles, requirements, and skill levels.

■ **PAUL VICK** There is no magic bullet when designing frameworks for Visual Basic developers. Our users run the gamut from people who are picking up a programming tool for the first time to industry veterans building large-scale commercial applications. The key to designing a framework that appeals to Visual Basic developers is to focus on allowing them to get the job done with the minimum amount of fuss and bother. Designing a framework that uses the minimum number of concepts is a good idea, not because VB developers can't handle concepts, but because having to stop and think about concepts extraneous to the task at hand interrupts workflow. The goal of a VB developer usually is not to learn some interesting or exciting new concept or to be impressed with the intellectual purity and simplicity of your design, but to get the job done and move on.

■ **KRZYSZTOF CWALINA** It is easy to design for users who are like you, and very difficult to design for somebody unlike you. There are too many APIs that are designed by domain experts and, frankly, they are only good for domain experts. The problem is that most developers are not, will never be, and do not need to be experts in all technologies used in modern applications.

■ **BRAD ABRAMS** Although the famous Hewlett-Packard motto “Build for the engineer at the next bench” is useful for driving quality and completeness into software projects, it is misleading for API design. For example, the developers on the Microsoft Word team have a clear understanding that they are not the target customers for Word. My mom is much more the target customer. Therefore the Word team puts in many more features that my mom might find helpful rather than the features the development team finds helpful. Although that is obvious in the case of applications such as Word, we often tend to miss the principle when designing APIs. We tend to design APIs only for ourselves instead of thinking clearly about the customer scenarios.

✓ **DO** understand and design for the broad variety of programming languages.

Many programming language implementations are available that support the Common Language Runtime (CLR) and .NET. Some of these languages might be very different from the language you are using to implement your APIs. Often special care needs to be taken to ensure that your APIs can work well with a variety of languages.

For example, developers using dynamically typed languages with the capability of interacting with .NET (such as PowerShell, IronPython, and others) might have problems using APIs that require them to create a custom type with attributes.

As another example, the F# language does not honor user-defined implicit conversion operators. In consequence, APIs designed with implicit conversions to facilitate calling patterns will not necessarily be easy to use in F#.

■ **JAN KOTAS** Designing for the least common denominator of existing programming languages started to hamper evolution of the .NET platform. It was deemphasized in recent years to enable innovations such as `Span<T>`. C# and F# introduced new language features to enable `Span<T>`. Other traditional .NET-targeting languages (such as Visual Basic) have not enabled it, however, and the new `Span`-based APIs cannot be used from them.

■ **JEREMY BARTON** While it's true that we added `Span<T>` without language support from VB, the guidance for using the `Span` types—found in section 9.12—recommends having array-based alternative methods. That recommendation is based partly on usability, but ultimately comes back to this guideline and the variety of languages that can interact with the .NET CLR.

Other special considerations for various programming languages are described throughout the book.

2.1 Progressive Frameworks

Designing a single framework for a broad range of developers, scenarios, and languages is a difficult and costly enterprise. Historically, framework vendors offered several products targeted at specific developer groups for specific scenarios. For example, Microsoft provided Visual Basic APIs optimized for simplicity and a relatively narrow set of scenarios, and Win32 APIs optimized for power and flexibility, even if it meant sacrificing ease of use. Other frameworks, such as MFC and ATL, were also targeted at specific developer groups and scenarios.

Although this multiframework approach proved to be a successful way to provide APIs that were powerful and easy to use by specific developer groups, it has significant drawbacks. The main drawback² is that the multitude of frameworks makes it difficult for developers using one of the frameworks to transfer their knowledge to the next skill level or scenario (which often requires a different framework). For example, when they need to implement a different application that requires more powerful functionality, developers face a very steep learning curve, because they have to learn a completely different way of programming, as shown in Figure 2-1.

2. Other drawbacks include slower time to market for frameworks that are wrappers on top of other frameworks, duplication of effort, and lack of common tools.

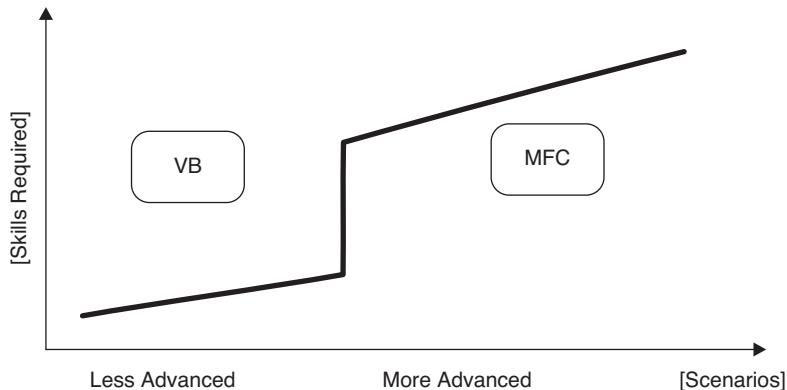


FIGURE 2-1: Learning curve of a multiframework platform

■ **ANDERS HEJLSBERG** In the good old days of early Windows, you had the Windows API. To write apps you fired up your C compiler, #included `windows.h`, created a `winproc`, and handled your windows messages—basically the old Petzold style of Windows programming. Although this worked, it was neither particularly productive nor particularly easy.

Over time, various programming models on top of the Windows API have emerged. VB embraced Rapid Application Development (RAD). With VB you could instantiate a form, drag components onto the form, and write event handlers; through delegation, your code executes.

In the world of C++, we had MFC and ATL taking a different view. The key concept here is subclassing. Developers would subclass from an existing monolithic, object-oriented framework. Although this gives you more power and expressiveness, it doesn't really match the ease or productivity of VB's composition model.

If you look at this picture, one of the problems is that your choice of programming model also necessarily becomes your choice of programming language. This is an unfortunate situation. If you're a skilled MFC developer and you need to write some code in VB, your skills don't translate. Likewise, if you know a lot about VB, not much of that knowledge transfers to MFC.

There is also not a consistent availability of APIs. Each of these models has dreamed up its own solutions to a number of problems that are actually core and common to all of the models—for example, how do I deal with file I/O, how do I do string formatting, how do I do security, threading, and so on?

What the .NET Framework does is unify all of these models. It gives you a consistent API that is available everywhere, regardless of what language you use or what programming model you are targeting.

■ **PAUL VICK** It's also worth noting that this unification comes at a cost. There is an unresolvable tension between writing frameworks that expose a great amount of power and allow a developer a great deal of control over behavior and writing frameworks that expose a more limited functionality in an extremely conceptually simple way. In most cases, there is no silver bullet, and trade-offs inevitably have to be made between power on the one hand and simplicity on the other hand. An enormous amount of effort went into designing the .NET Framework to ensure that it achieved the best possible balance between these two goals, but it is something that I think we continue to work on to this day.

A much better approach is to provide a *progressive framework*, which is a single framework targeted at a broad range of developers that allows for transfer of knowledge from less advanced to more advanced scenarios. The .NET Framework is a progressive framework and provides such a gradual learning curve (see Figure 2-2).

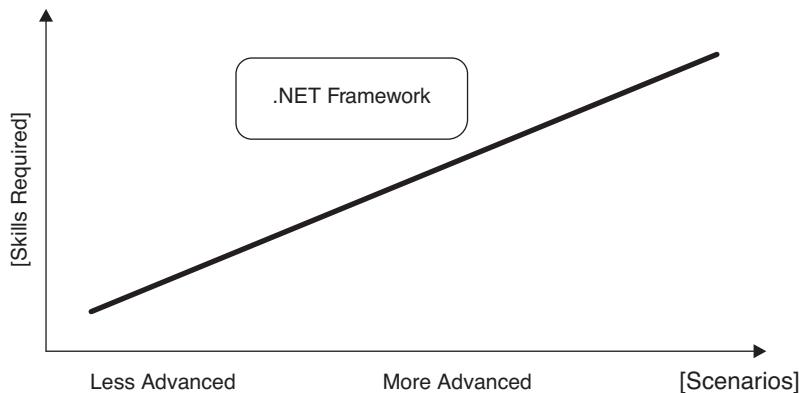


FIGURE 2-2: Learning curve of a progressive framework platform

Achieving a gradual learning curve with a low entry point is difficult but not impossible. It is difficult because it requires a new approach to the process of framework design, demands much greater design discipline, and has a higher design cost. Fortunately, the guidelines described in this chapter and throughout the book are meant to guide you through the difficult design process and ultimately help you design a great progressive framework.

You should also keep in mind that the developer community is vast. It ranges from office workers recording macros to authors of low-level device drivers. Any framework that attempts to serve all of these users could end up being a mess that does not satisfy any of them. The goal of a progressive framework is to scale through a broad range of developers, but not every possible developer. This clearly means that those developers who fall outside this target group will need specialty APIs.

2.2 Fundamental Principles of Framework Design

Providing a development platform that is both powerful and easy to use is one of the main goals of .NET, and it should be one of your goals if you are extending it. The first version of .NET Framework indeed presented a powerful set of APIs, but some developers found parts of the framework too difficult to use.

RICO MARIANI The flip side of this is that it must not only be easy to use the API; it must also be easy to use the API in the best possible way. Think carefully about what patterns you offer and be sure that the most natural way to use your system gives results that are correct, is secure against attacks, and has great performance. Make it hard to do things the wrong way. A few years ago I wrote:

The Pit of Success

In stark contrast to a summit, a peak, or a journey across a desert to find victory through many trials and surprises, we want our customers to simply fall into winning practices by using our platform and frameworks. To the extent that we make it easy to get into trouble, we fail.

True productivity comes from being able to easily create great products—not from being able to easily create junk. Build a pit of success.

User feedback and usability studies have shown that a large segment of the VB developer group had problems learning VB.NET. Part of the problem stemmed from the simple fact that .NET is different from VB 6.0

libraries, but there were also several major usability problems related to API design. Fixing these problems became a high priority for Microsoft in the .NET Framework 2.0 time frame.

The principles described in this section, identified with the label “Framework Design Principle,” were developed to address the problems just mentioned. They are meant to help framework designers avoid the most severe design mistakes reported in many usability studies and in user feedback. We believe that these principles are central to the design of any general-purpose framework. Some of the principles and recommendations overlap, which is probably a testimony to their validity.

2.2.1 The Principle of Scenario-Driven Design

Frameworks often contain a very large set of APIs. This is necessary to enable advanced scenarios that require power and expressiveness. However, most development revolves around a small set of common scenarios that rely on a relatively small subset of the full framework. To optimize the overall productivity of the developers using a framework, it is crucial to invest heavily in the design of the APIs used in the most common scenarios.

To that end, framework design should focus on a set of common scenarios to the point where the whole design process is scenario-driven. We recommend that framework designers first write code that the users of the framework will have to write in main scenarios, and then design the object model to support these code samples.³

Framework Design Principle

Frameworks must be designed starting from a set of usage scenarios and code samples implementing these scenarios.

3. This is similar to processes based on test-driven development (TDD) or on use cases. There are some differences, though. TDD is more heavyweight because it has other objectives beyond driving the design of APIs. Use cases describe scenarios on a higher level than individual API calls.

■ **KRZYSZTOF CWALINA** I would like to add “There is simply no other way to design a great framework” to the principle just spelled out. If I had to choose only one design principle to be included in the book, this would be it. If I could not write a book but only a short article on what’s important in API design, I would choose this principle.

Framework designers often make the mistake of starting with the design of the object model (using various design methodologies) and then writing code samples based on the resulting API. The problem is that most design methodologies (including most commonly used object-oriented design methodologies) are optimized for the maintainability of the resulting implementation, not for the usability of the resulting APIs. They are best suited for internal architecture designs—not for designs of the public API layer of a large framework.

When designing a framework, you should start by producing a scenario-driven API specification (see Appendix C). This specification can be either separate from the functional specification or part of a larger specification document. In the latter case, the API specification should precede the functional one in location and time.

The specification should contain a scenario section listing the top five to ten scenarios for a given technology area and should show code samples that implement these scenarios. When your API or code samples make use of new or otherwise unusual language features, you should consider writing the sample in at least a second language, because sometimes code written using those languages differs significantly.

It is also important to write these scenarios using different coding styles that are common among users of the particular language (using language-specific features). The samples should be written using language-specific casing. For example, VB.NET is case insensitive, so samples should reflect that. C# code should follow the standard casing described in Chapter 3.

✓ **DO** make sure that the API design specification is the central part of the design of any feature that includes a publicly accessible API.

Appendix C contains an example of such a specification.

✓ **DO** define top usage scenarios for each major feature area.

The API specification should include a section that describes the main scenarios and shows code samples implementing these scenarios. This section should appear immediately after the executive overview section. The average feature area (such as file I/O) should have five to ten main scenarios.

- ✓ **DO** ensure that the scenarios correspond to an appropriate abstraction level. They should roughly correspond to the end-user use cases.

For example, reading from a file is a good scenario. Opening a file, reading a line of text from a file, or closing a file are not good scenarios; they are too granular.

- ✓ **DO** design APIs by first writing code samples for the main scenarios and then defining the object model to support the code samples.

For example, when designing an API to measure elapsed time, you might write the following scenario code samples:

```
// scenario #1 : measure time elapsed
Stopwatch watch = Stopwatch.StartNew();
DoSomething();
Console.WriteLine(watch.Elapsed);

// scenario #2 : reuse stopwatch
Dim watch As Stopwatch = Stopwatch.StartNew()
DoSomething();
Console.WriteLine(watch.ElapsedMilliseconds)

watch.Reset()
watch.Start()
DoSomething()
Console.WriteLine(watch.Elapsed)
```

These code samples lead to the following object model:

```
public class Stopwatch {
    public static Stopwatch StartNew();

    public void Start();
    public void Reset();

    public TimeSpan Elapsed { get; }
    public long ElapsedMilliseconds { get; }
    ...
}
```

■ **JOE DUFFY** As software developers, we enjoy creating fun and powerful new capabilities, and sharing them with other developers. That's one of the reasons API design is so enjoyable. But it's also incredibly difficult to step back and objectively assess whether some new capability that you're particularly passionate about has utility in the real world. Using scenarios is the best way I know of to identify the need for and ideal usage of new capabilities. Developing scenarios is in fact incredibly *hard*, for good reason: It requires a unique combination of technical skill and customer understanding. When you're finished, you could make a series of decisions based only on gut feel and intuition, and perhaps deliver some useful APIs, but the risk that you will make a decision you will later regret is far greater. When in doubt, it's best to leave a feature out and decide to add it later when a compelling need is better understood.

■ **STEPHEN TOUB** It's fun to add new APIs. But every new one that's added comes with a cost. Sometimes that cost is "just" the cost of designing, developing, testing, documenting, and maintaining the functionality (plus the associated run-time costs). It's unfortunately common, however, that adding an API actually inhibits one's ability in the future to add other potentially much more desirable or impactful functionality, because such functionality would conflict with the functionality exposed by that API. This is one of the reasons we're so deliberate in choosing what new APIs to expose, because we might be sealing our fate on some future innovation. If asked, I'm sure many of us have a favorite example of an API we wish we'd never added; I know I have several.

✓ **DO** write main scenario code samples in at least two different language families (e.g., C# and F#).

It is best to ensure that the languages chosen have significantly different syntax, style, and capabilities.

■ **PAUL VICK** If you are writing a framework to be used by multiple languages, it is helpful to actually know more than one programming language (and knowing more than one C-style language doesn't count). We've found that sometimes an API works well only in one language, and that's because the person designing the API (and testing the API) only really knew that one language. Learn several .NET languages and really work with them the way they were designed to be used. Expecting the whole world to speak your language does not work well on a multilanguage platform like the .NET Framework.

✓ **CONSIDER** writing main scenario code samples using a dynamically typed language such as PowerShell or IronPython.

It is easy to design APIs that don't work well with dynamically typed languages. Such languages often have problems dealing with some generic methods, as well as APIs that rely on applying attributes or creating strongly typed types.

✗ **DO NOT** rely solely on standard design methodologies when designing the public API layer of a framework.

Standard design methodologies (including object-oriented design methodologies) are optimized for the maintainability of the resulting implementation, not for the usability of the resulting APIs. Scenario-driven design, together with prototyping, usability studies, and some amount of iteration, is a much better approach.

■ **CHRIS ANDERSON** Each developer has his or her own methodology, and although there isn't anything fundamentally wrong with using other modeling approaches, the problem generally is the output. Starting by writing the code you want a developer to write is almost always the best approach—think of it as a form of test-driven development. You write the perfect code and then work backward to figure out the object model that you would want.

2.2.1.1 Usability Studies

Usability studies of a framework prototype conducted with a wide range of developers are the key to scenario-driven design. The APIs for the top scenarios might seem simple to their authors, but they might not actually be simple to other developers.

Understanding the way developers approach each of the main scenarios provides useful insight into the design of the framework and how well it meets the needs of all target developers. For this reason, conducting usability studies—either formally or informally—is a very important part of the framework design process.

If you discover during usability studies that the majority of developers cannot implement one of the scenarios, or if the approach they take is significantly different from what the designer expected, the API should be redesigned.

■ **KRZYSZTOF CWALINA** We did not test the usability of the types in the `System.IO` namespace before shipping version 1.0 of the .NET Framework. Soon after shipping, we received negative customer feedback about `System.IO` usability. We were quite surprised, and we decided to conduct usability studies with eight average developers. Eight out of eight failed to read text from a file in the 30 minutes we allocated for the task. We believe this was due in part to problems with the documentation search engine and insufficient sample coverage; however, it is clear that the API itself had several usability problems. If we had conducted the studies before shipping the product, we could have eliminated a significant source of customer dissatisfaction and avoided the cost of trying to fix the API of a major feature area without introducing breaking changes.

■ **BRAD ABRAMS** There is no more powerful experience to give an API designer a visceral understanding of the usability of his or her API than sitting behind a one-way mirror watching developer after developer get frustrated with an API he or she designed and ultimately fail to complete the task. I personally went through a full range of emotions while watching the usability studies for `System.IO` that we did right after shipping version 1.0. As developer after developer failed to complete the simple task, my emotions moved from arrogance to disbelief, then to frustration, and finally to stringent resolve to fix the problem in the API.

■ **CHRIS SELLS** Usability studies can be formal, if you've got that kind of time and money. But you'll get 80 percent of the feedback you need from just running your proposed API by a few developers close to the target audience for your library. Don't let the term "usability study" scare you into doing nothing: Think of it as a "hey, look at this" study instead.

■ **STEVEN CLARKE** Rather than expending a lot of effort planning, designing, and running one large study with lots of participants and trying to cover as much of the API surface area as possible, we've found that it is much more valuable to run a series of smaller, more focused studies throughout the API development process. In each study, we ask a small number of participants to focus on one design issue or area of the API. We take what we learn from that, iterate on the design, and then a week or two later run another study on the updated design or another area of the API. This continuous learning approach means that there is a constant stream of customer insights that informs the design process, rather than one large dose delivered at one specific point in the process.

The API usability studies should ideally be performed using the actual development environment, code editors, and documentation most widely used by the targeted developer group. However, it is best to run usability studies early rather than late in the product cycle—so don't postpone organizing a study just because the whole product is not ready yet.

■ **STEPHEN TOUB** You don't even need a real implementation to get a sense for the usability of your API. While it's best to have something developers can run with to see the results of their experiments, early design feedback can be obtained just by having an API they can code and compile against, which means all your implementations can be stubbed out to be no-ops or throws; it doesn't matter, since they won't be invoked. Did the developers intuitively find the relevant types involved? Were they able to discern the patterns used to access the functionality? Was IntelliSense able to help guide them through the API in a meaningful way? Did they approach the problems in the way you hypothesized they would? Did they routinely go searching for things named something else?

Often, formal usability studies are impractical for small development teams and frameworks that target a relatively small set of developers. In such cases, an informal study can be conducted. Giving a prototype library to someone not familiar with the design, asking them to spend 30 minutes writing a simple program, and observing how they deal with the design is a great way to find the most troublesome API design issues.

✓ **DO** organize usability studies to test APIs in main scenarios.

These studies should be organized early in the development cycle, because the most severe usability problems often require substantial design changes. Most developers should be able to write code for the main scenarios without major problems; if they cannot, you need to redesign the API. Although redesign is a costly practice, we have found that it actually saves resources in the long run because the cost of fixing an unusable API without introducing changes that break existing code is enormous.

The next section describes the importance of designing APIs so that the initial encounter is not discouraging. This is called the principle of low barrier to entry.

2.2.2 The Principle of Low Barrier to Entry

Today, many developers expect to learn the basics of new frameworks very quickly. They want to do it by experimenting with parts of the framework on an ad hoc basis, and only take the time to fully understand the whole architecture if they find a particular feature interesting or if they need to move beyond the simple scenarios. The initial encounter with a badly designed API can leave a lasting impression of complexity and discourage some developers from using the framework. This is why it is very important for frameworks to provide a very low barrier for developers who just want to experiment with the framework.

Framework Design Principle

Frameworks must offer a low barrier to entry for nonexpert users through ease of experimentation.

Many developers want to experiment with an API to discover what it does and then adjust their code slowly to get their program to do what they really want.

■ **PAUL VICK** Most developers, regardless of the language that they work in, learn by doing. Documentation can help give the initial idea of what's supposed to happen, but we all know that you never really learn how something works until you get down into it and start fiddling around, trying to do something useful. Visual Basic, in particular, encourages this kind of experimental approach to programming. Although we never eschew forethought and advance planning, we try to make the process of learning and programming a continuous flow. Writing APIs that are self-evident and do not require a complex knowledge of the interaction of multiple objects or APIs encourages this flow. (In fact, this seems to apply across most languages, not just Visual Basic.)

Some APIs lend themselves to experimentation and some do not. To be easy to experiment with, an API must do the following:

- Allow easy identification of the right set of types and members for common programming tasks. A namespace intended to hold common scenario APIs that contains 500 types, out of which only a handful are actually important in common scenarios, is not easy to experiment with. The same applies to mainline scenario types with many members that are intended only for very advanced scenarios.

■ **CHRIS ANDERSON** In the early days of the Windows Presentation Foundation (WPF) project, we ran into this exact issue. We had a common base type named `Visual` from which almost all of our elements were derived. The problem was that it introduced members that directly conflicted with the object model of the more derived elements, specifically around children. `Visual` had a single hierarchy of child visuals for rendering, but our elements wanted to introduce domain-specific children (like a `TabControl` only accepting `TabPage`s). Our solution was to create a `VisualOperations` class that had static members that worked on a `Visual` instead of complicating the object model of every element.

- Allow a developer to use the API immediately, whether or not it does what the developer ultimately wants it to do. A framework that requires an extensive initialization or the instantiation of several types, followed by hooking them together, is not easy to experiment with. Similarly, APIs with no convenience overloads (overloaded members with short parameter lists) or bad defaults for properties pose a high barrier for developers who just want to experiment with the APIs.

CHRIS ANDERSON Think of the object model as a map: You have to put up clear signs explaining how to get from one place to another. You want a property to clearly point people to what it does, which values it takes, and what will happen if you set it. Pointing to an abstract base type with no obvious derivations is a bad, bad thing. An example of this being broken is how animations were exposed in WPF: The type for animations was `Timeline`, but nothing in the namespace ended in the word “Timeline.” It turns out that `Animation` derived from `Timeline` and there were lots of types like `DoubleAnimation`, `ColorAnimation`, and so on, but there was no connection between the property type and the valid items with which to fill the property.

- Allow for easy finding and fixing of mistakes caused by incorrect usage of an API. For example, APIs should throw exceptions that clearly describe what needs to be done to fix the problem.

CHRIS SELLS In my own programming, I dearly love error messages that say what I did wrong and how to fix it. All too often, all I get is the former, when all I really care about is the latter.

The following guidelines will help you ensure that your framework is well suited for developers who want to learn by experimenting.

- ✓ DO** ensure that each main feature area namespace contains only types that are used in the most common scenarios. Types used in advanced scenarios should be placed in subnamespaces.

For example, the `System.Net` namespace provides networking mainline-scenario APIs. The more advanced socket APIs are placed in the `System.Net.Sockets` subnamespace.

■ **ANTHONY MOORE** The converse of this is also true, which could be stated as, “Don’t bury a commonly used type in a namespace with much less commonly used types.” `StringBuilder` is an example of something we later wished we had included in the `System` namespace. It lives in `System.Text` but is much more commonly used than the other types in there and is not closely related to them.

That being said, this is the only thing in the `System` namespace that suffers from the converse of this rule. For the most part, we suffered from having too many infrequently used types in there.

✓ **DO** provide simple overloads of constructors and methods. A simple overload has a very small number of parameters, and all parameters are primitives.

✗ **DO NOT** have members intended for advanced scenarios on types intended for mainline scenarios.

■ **BRAD ABRAMS** One of the important principles in designing the .NET Framework was the notion of addition through subtraction. That is, by removing features from (or never adding them to) the framework, we could actually make developers more productive because there would be fewer concepts to deal with. Leaving out multiple inheritance is a classic example of addition through subtraction at the CLR level.

✗ **DO NOT** require users to explicitly instantiate more than one type in the most basic scenarios.

■ **KRZYSZTOF CWALINA** Book publishers say that the number of copies a book will sell is inversely proportional to the number of equations in the book. The framework designer version of this law is this: The number of developers who will use your framework is inversely proportional to the number of explicit constructor invocations required to implement the top ten simple scenarios.

- ✗ **DO NOT** require that users perform any extensive initialization before they can start programming basic scenarios.

Mainline-scenario APIs should be designed to require minimal initialization. Ideally, a default constructor or a constructor with one simple parameter should be sufficient to start working with a type designed for the basic scenarios.

```
var zipCodes = new Dictionary<string,int>();  
zipCodes.Add("Redmond",98052);  
zipCodes.Add("Sammamish",98074);
```

If some initialization is necessary, the exception that results from not performing the initialization should clearly explain what needs to be done.

■ **STEVEN CLARKE** Since the first edition of this book was published, we have done significant usability studies in this area. Time and time again, we observe that types that require extensive initialization significantly raise the barrier to entry. The consequences of this are that some developers will decide not to use those types and will look for something else that might do the job instead, some developers will end up using the type incorrectly, and only a few developers will eventually figure out how to use the type correctly.

ADO.NET is an example of a feature area that our users found difficult to use because of the extensive initialization it requires. Even in the simplest scenarios, users are expected to understand complex interactions and dependencies between several types. To use this feature, even in simple scenarios, users must instantiate and hook up several objects (instances of `DataSet`, `DataAdapter`, `SqlConnection`, and `SqlCommand`). Note that many of these problems were addressed in .NET Framework 2.0 through the addition of helper classes, which greatly simplified basic scenarios.

- ✓ **DO** provide good defaults for all properties and parameters (using convenience overloads), if possible.

`System.Messaging.MessageQueue` is a good illustration of this concept. The component can send messages after passing just a path string to the constructor and calling the `Send` method. The message priority, encryption algorithms, and other message properties can be customized by adding code to the simple scenario.

```
var ordersQueue = new MessageQueue(path);
ordersQueue.Send(order); // uses default priority, encryption, etc.
```

These recommendations cannot be applied blindly. Framework designers should avoid providing defaults if the default can lead the user astray. For example, a default should never result in a security hole or horribly performing code.

■ **STEPHEN TOUB** When deciding on “defaults,” it’s also critical to understand the primary use cases for the API, and—to whatever extent possible—try to predict expected use cases in the future. One of my top ten “I wish I could do that over again” cases is from `System.Threading.Tasks`. We’d initially designed the APIs with a focus on CPU-based parallelism, but over time the primary use cases ended up being much more focused on IO-based asynchrony, and some of our initial defaults lent themselves much better to the former to the detriment of the latter. In time we addressed these issues as part of adding easier-to-consume APIs, but the initial issues and resulting difficulties persist for developers who find themselves consuming the original APIs.

■ **JEREMY BARTON** The counterpoint here is very important, and it can be hard to find the right balance. The .NET Cryptography APIs include a number of types that provide defaults to try to be both friendly and secure. Unfortunately, “friendly” persists, but “secure” is a moving target. Sometimes you’re doing your users a favor by providing defaults, and sometimes you’re doing them harm.

✓ **DO** communicate incorrect usage of APIs using exceptions.

The exceptions should clearly describe their cause and the way the developer should modify the code to get rid of the problem. For example, the `EventLog` component requires the `Source` property to be set

before events can be written. If the `Source` is not set before `WriteEntry` is called, an exception is thrown that states, "Source property was not set before writing to the event log."

■ **STEVEN CLARKE** We've observed many developers in our usability studies who consider exceptions like these to be the best kind of documentation an API can provide. The guidance provided is always in the context of what the developer is trying to achieve, and it really supports the learning-by-doing approach favored by many developers.

The next section describes the importance of making the object model as self-documenting as possible.

2.2.3 The Principle of Self-Documenting Object Models

Many frameworks consist of hundreds, if not thousands, of types and significantly more members and parameters. Developers using such frameworks require a great deal of guidance and frequent reminders of the purpose and proper usage of the APIs. Reference documentation by itself cannot meet the demand. If it is necessary to refer to the documentation to answer the simplest questions, doing so is likely to be time-consuming and break the developer's workflow. Moreover, as noted earlier, many developers prefer to code by trial and error, and they resort to reading documentation only when intuition fails them.

For all these reasons, it is very important to design APIs that do not require developers to read the documentation every time they want to perform a simple task. We have found that following a simple set of guidelines can help developers produce intuitive APIs that are relatively self-documenting.

Framework Design Principle

In simple scenarios, frameworks must be usable without the need for documentation.

■ **CHRIS SELLS** Never underestimate the power of IntelliSense when anticipating how a developer will learn to use your framework. If your API is intuitive, IntelliSense is 80 percent of all a new developer will need to be happy and successful with your library. Optimize for IntelliSense.

■ **KRZYSZTOF CWALINA** Reference documentation is still a very important part of the framework. It is impossible to design a completely self-documenting API. Different people, depending on their skills and past experiences, will find different areas of your framework to be self-explanatory. Also, the documentation remains critically important for many users who do take the time to understand the overall design of the framework up front. For those users, informative, concise, and complete documentation is as crucial as self-explanatory object models.

- ✓ **DO** ensure that APIs are intuitive and can be successfully used in basic scenarios without referring to the reference documentation.
- ✓ **DO** provide great documentation with all APIs.
- ✓ **DO** provide code samples illustrating how to use your most important APIs in common scenarios.

Not all APIs can be self-explanatory, and some developers will want to thoroughly understand the APIs before they start using them.

To make a framework self-documenting, care must be taken when choosing names and types, designing exceptions, and so on. The following sections describe some of the most important considerations related to the design of self-documenting APIs.

2.2.3.1 **Naming**

The simplest, yet most often missed, opportunity for making frameworks self-documenting is to reserve simple and intuitive names for types that developers are expected to use (instantiate) in the most common scenarios. Framework designers often “burn” the best names for less commonly used types, with which most users do not have to be concerned.

For example, naming the abstract base class `File` and then providing a concrete type `NtfsFile` works well if the expectation is that all users will understand the inheritance hierarchy before they start using the APIs. If the users do not understand the hierarchy, the first thing they will try to use, most often unsuccessfully, is the `File` type. Although this naming works well in the object-oriented design sense (after all, `NtfsFile` is a kind of `File`), it fails the usability test, because `File` is the name most developers would intuitively think to program against.

■ **KRZYSZTOF CWALINA** The designers of the .NET Framework spent a lot of time discussing naming alternatives for main types. The majority of the identifiers in .NET have well-chosen names. The cases in which the name choices are not so fortunate resulted from focusing on concepts and abstractions instead of the top scenarios.

Another recommendation is to use descriptive identifier names that clearly state what each method does and what each type and parameter represents. Framework designers should not be afraid to be quite verbose when choosing identifier names. For example, `EventLog.DeleteEventSource(string source, string machineName)` might be seen as rather verbose, but we think it has a positive net usability value.

Descriptive method names are only possible for methods that have simple and clear semantics. This is another reason that avoiding complex semantics is a great general design principle to follow.

The overall point is that you need to choose the names of identifiers extremely carefully. Name choices are one of the most important design choices a framework designer has to make. It is extremely difficult and costly to change identifier names after the API has shipped.

✓ **DO** make the discussion about identifier naming choices a significant part of specification reviews.

What are the types most scenarios start with? What are the names most people will think of first when trying to implement this scenario? Are the names of the common types what users will think of first? For example, because “File” is the name most people think of when dealing with file I/O scenarios, the main type for accessing files should be named `File`.

Also, you should discuss the most commonly used methods of the most commonly used types and all of their parameters. Can anybody who is familiar with your technology, but not this specific design, recognize and call those methods quickly, correctly, and easily?

X DO NOT be afraid to use verbose identifier names when doing so makes the API self-documenting.

Most identifier names should clearly state what each method does and what each type and parameter represents.

■ **BRENT RECTOR** Developers read identifier names hundreds, if not thousands, of times more than they type them. Modern editors even make the typing chore minimal. Longer names allow developers to find the right type or member via IntelliSense more quickly. Additionally, code using types with well-chosen identifier names is more understandable and maintainable over the long term.

A note to C-based language developers especially: Free yourselves from the shackles of reduced productivity induced by cryptic identifier naming habits.

✓ CONSIDER involving technical writers early in the design process. They can be a great resource for spotting designs with bad name choices and designs that would be difficult to explain to users.

✓ CONSIDER reserving the best type names for the most commonly used types.

If you believe you will add more high-level APIs in a future version, don't be afraid to reserve the best name in the first version of your framework for future APIs.

■ **ANTHONY MOORE** There are other reasons to avoid names that are too general, even if you never anticipate using the name later. More-specific names help to make the API more understandable and readable. If someone sees a general name in code, that person is likely to assume that it has a very general application, so it is misleading to use a general name for something more specialized. A more descriptive name can also help identify what scenario or technology a type is associated with.

2.2.3.2 Exceptions

Exceptions play a crucial role in designing self-documenting frameworks. They should communicate the correct usage to the developer through the exception message. For example, the following sample code should throw an exception with a message "Source property was not set before writing to the event log."

```
// C#
var log = new EventLog();
// The log source is not set yet.
log.WriteEntry("Hello World");
```

- ✓ **DO** use exception messages to communicate framework usage mistakes to the developer.

For example, if a user forgets to set the Source property on an Event-Log component, any calls to a method that requires the source to be set should state this clearly in the exception message. Chapter 7 provides more guidance about the design of exceptions and exception messages.

2.2.3.3 Strong Typing

Strong typing is probably the single most important factor in determining how intuitive APIs are. Clearly, calling `Customer.Name` is easier than calling `Customer.Properties["Name"]`. Also, a `Name` property returning the name as a `String` is more usable than if the property returned an `Object`.

There are cases where property bags, late-bound calls, and other loosely typed APIs are necessary, but they should be an exception to the rule rather than common practice. Moreover, designers should consider providing strongly typed helpers for the most common operations that the user would perform on the non-strongly typed API layer. For example, the `Customer` type may have a property bag but also provide strongly typed APIs for most common properties like `Name`, `Address`, and so on.

- ✓ **DO** provide strongly typed APIs if at all possible.

Do not rely exclusively on weakly typed APIs such as property bags. In cases in which a property bag is required, provide strongly typed properties for the most common properties in the bag.

■ **VANCE MORRISON** The strong typing (and thus much better IntelliSense) is a crucial reason why .NET frameworks are easier to “learn by programming” than your typical COM API. From time to time, I still need to use functionality exposed through COM, and as long as it stays strongly typed, I do fine. But all too often APIs return or take a generic object or string parameter or passed DWORD when an enumeration is needed, and it takes me ten times longer to discover what exactly needs to be passed.

2.2.3.4 Consistency

Consistency with existing APIs that are already familiar to the user is yet another powerful technique for designing self-documenting frameworks. This includes consistency with other APIs in .NET as well as some legacy APIs. Having said that, you should not use legacy APIs or badly designed existing framework APIs as an excuse to avoid following any guidelines described in this book—but you also should not change good established patterns and designs arbitrarily without having a reason to do so.

- ✓ **DO** ensure consistency with .NET and other frameworks your users are likely to interact with.

Consistency is great for general usability. If a user is familiar with some part of a framework that your API resembles, he or she will see your design as natural and intuitive. Your API should differ from other .NET APIs only in places where there is something unique about your particular API.

2.2.3.5 Limiting Abstractions

Common scenario APIs should not use many interfaces and abstract classes, but should instead correspond to physical or well-known logical parts of the system.

As noted earlier, standard object-oriented design methodologies are aimed at producing designs that are optimized for maintainability of the code base. This makes sense, because the maintenance cost is the largest chunk of the overall expense of developing a software product. One way of improving maintainability is through the use of abstractions via interfaces, or abstract classes. Because of that, modern design methodologies tend to produce a lot of them.

The problem is that frameworks with many abstractions force users to become experts in the framework architecture before they start to implement even the simplest scenarios. However, most developers don't have the desire or business justification to become experts in all of the APIs that such frameworks provide. For simple scenarios, developers demand that APIs be simple enough to be used without their having to understand how the entire feature areas fit together. This is something that the standard design methodologies are not optimized for and never claimed to be optimized for.

Of course, abstractions have their place in framework design. For example, abstractions can be extremely useful in improving the testability and general extensibility of frameworks. Such extensibility is often possible because of well-designed abstractions. Chapter 6 discusses designing extensible APIs and should help you strike the right balance between too much and too little extensibility.

X AVOID many abstractions in mainline-scenario APIs.

KRZYSZTOF CWALINA Abstractions are almost always necessary, but too many abstractions indicate overengineered systems. Framework designers should be careful to design for customers, not for their own intellectual pleasure.

JEFF PROSIE A design with too many abstractions can impact performance, too. I once worked with a customer that reengineered its product to incorporate a heavily object-oriented design. They modeled *everything* as a class and ended up with some ridiculously deeply nested object hierarchies. Part of the intent of the redesign was to improve performance, but the “improved” software ran four times slower than the original!

■ **VANCE MORRISON** Anyone who has had the “pleasure” of debugging through the C++ STL libraries knows that abstraction is a double-edged sword. Too much abstraction and code gets very difficult to understand, because you have to remember what all the abstract names really mean in your scenario. Going overboard with generics and inheritance are common symptoms that you may have overgeneralized.

■ **CHRIS SELLS** It’s often said that any problem in computer science can be solved by adding a layer of abstraction. Unfortunately, problems of developer education are often caused by them.

2.2.4 The Principle of Layered Architecture

Not all developers are required to solve the same kinds of problems. Different developers often require and expect different levels of abstraction and different amounts of control from the frameworks they use. Some developers who typically use C++ or C# value APIs that are expressive and powerful. We refer to APIs of this type as low-level APIs because they often offer a low level of abstraction. In contrast, some developers who typically use C# or VB.NET value APIs that optimize for productivity and simplicity. We refer to these APIs as high-level APIs because they offer a higher level of abstraction. By using a layered design, it is possible to build a single framework that meets these diverse needs.

Framework Design Principle

Layered design makes it possible to provide both power and ease of use in a single framework.

■ **PAUL VICK** Part of the reason for moving Visual Basic to the .NET platform was the fact that many VB developers ran into problems when they needed to use low-level APIs to access specific functionality that was not available in the high-level APIs that we provided. The fact that VB developers might spend much of their initial time rapidly developing their applications using high-level APIs doesn't change the fact that sooner or later most developers need to tweak or fine-tune their applications, and doing that usually involves working with lower-level APIs to achieve those extra bits of functionality. So the design for low-level APIs should very much take VB developers into consideration.

The general guideline for building a single framework that targets the breadth of developers is to factor your API set into low-level types that expose all the richness and power and into high-level types that wrap the lower layer with convenience APIs.

This is a very powerful simplification technique. In a single-layer API, you are often forced to decide between having a more complex design and not supporting some scenarios. Having a low-level power layer provides the freedom to scope the high-level API to truly mainline scenarios.

In some cases, one of the layers might not be needed. For example, some feature areas might expose only the low-level APIs.

The .NET JSON APIs are an example of such a layered design. For the power and expressiveness crowd, the `Utf8JsonReader` type offers a low-level JSON parser that allows developers to code against the individual tokens in a JSON payload. However, .NET also has the `JsonDocument` and `JsonElement` types, which build on `Utf8JsonReader` and allow developers to code against higher-level concepts, like the document structure, without worrying about counting their object depth or unescaping strings. The types have consistent behaviors and identifiers, but are at different layers that target different scenarios and developer audiences.

There are two main namespace factoring approaches for the API layers:

- Expose layers in separate namespaces.
- Expose layers in the same namespace.

2.2.4.1 Exposing Layers in Separate Namespaces

One way to factor a framework is to put the high-level and low-level types in different but related namespaces. This has the advantage of hiding the low-level types from the mainstream scenarios without putting them too far out of reach when developers need to implement more complex scenarios.

The .NET networking APIs are factored this way. The low-level `System.Net.Sockets.Socket` type, medium-level `System.Net.Security.SslStream` type, and high-level `System.Net.Http.HttpClient` types are all found in different namespaces. `HttpClient` ultimately relies on both `Socket` and `SslStream` in its implementation, but most developers working with HTTP can use `HttpClient` without needing the lower-level types.

The large majority of frameworks should follow this namespace-factoring approach.

2.2.4.2 Exposing Layers in the Same Namespace

The other way to factor a framework is to put the high-level and low-level types in the same namespace. This has the advantage of providing a more automatic fallback to the more complex functionality when it is needed. The downside is that having the complex types in the namespace makes some scenarios more difficult, even if the more complex types are not used.

This factorization works best for simple features. For example, the `System.Text` namespace includes both low-level types, such as the `Encoder` and `Decoder` classes, and the higher-level `Encoding` class hierarchy.

■ **STEVEN CLARKE** Take care to think about the runtime behavior of a layered API. For example, make sure that if the developer is working at one layer, he or she isn't expected to catch exceptions thrown from a different layer. You want to make sure that, in writing, reading, and understanding code, developers only ever need to really concern themselves with what is going on in one layer and that they can safely consider other layers as a black box.

- ✓ **CONSIDER** using a layered framework with high-level APIs optimized for productivity, and using low-level APIs optimized for power and expressiveness.
- ✗ **AVOID** mixing low-level and high-level APIs in a single namespace if the low-level APIs are very complex (i.e., they contain many types).
- ✓ **DO** ensure that layers of a single feature area are well integrated. Developers should be able to start programming using one of the layers and then change their code to use the other layer without rewriting the whole application.

SUMMARY

When designing a framework, it is very important to be aware that the audience is very diverse, in terms of both needs and skill levels. Following the principles described in this chapter will ensure that your framework is usable for a diverse group of developers.

This page intentionally left blank

3

Naming Guidelines

FOllowing a consistent set of naming conventions in the development of a framework can be a major contribution to the framework's usability. It allows the framework to be used by many developers on widely separated projects. Beyond consistency of form, names of framework elements must be easily understood and must convey the function of each element.

The goal of this chapter is to provide a consistent set of naming conventions that results in names that make immediate sense to developers. Most of these naming guidelines are simply conventions that have no technical rationale. However, following them will ensure that the names are understandable and consistent.

Although adopting these naming conventions as general code development guidelines would result in more consistent naming throughout your code, the guidance here only applies to APIs that are publicly exposed (public or protected types and members, and parameters to public or protected members).

KRYSZTOF CWALINA The team that develops the .NET Framework Base Class Library spends an enormous amount of time on naming and considers it to be a crucial part of framework development.

This chapter describes general naming guidelines, including how to use capitalization, mechanics, and certain specific terms. It also provides specific guidelines for naming namespaces, types, members, parameters, assemblies, and resources.

3.1 Capitalization Conventions

Because the Common Language Runtime (CLR) supports many languages that might or might not be case sensitive, case alone should not be used to differentiate names. However, the importance of case in enhancing the readability of names cannot be overemphasized. The guidelines in this chapter lay out a simple method for using case that, when applied consistently, make identifiers for types, members, and parameters easy to read.

3.1.1 Capitalization Rules for Identifiers

To differentiate words in an identifier, capitalize the first letter of each word in the identifier. Do not use underscores to differentiate words, or for that matter, anywhere in identifiers. There are two appropriate ways to capitalize identifiers, depending on the use of the identifier:

- PascalCasing
- camelCasing

■ **BRAD ABRAMS** In the initial design of the Framework, we had hundreds of hours of debate about the naming style. To facilitate these debates, we coined a number of terms. With Anders Hejlsberg, the original designer of Turbo Pascal, as a key member of the design team, it is no wonder that we chose the term PascalCasing for the casing style popularized by the Pascal programming language. We were somewhat cute in using the term camelCasing for the casing style that looks something like the hump on a camel. We used the term SCREAMING_CAPS to indicate an all-uppercase style. Luckily, this style (and name) did not survive in the final guideline.

The PascalCasing convention, used for all identifiers except parameter names, capitalizes the first character of each word (including acronyms more than two letters in length), as shown in the following examples:

```
PropertyDescriptor  
HtmlTag
```

A special case is made for two-letter acronyms in which both letters are capitalized, as shown in the following identifier:

```
IOutputStream
```

The camelCasing convention, used only for parameter names, capitalizes the first character of each word except the first word, as shown in the following examples. As the example also shows, two-letter acronyms that begin a camel-cased identifier are both lowercase.

```
propertyDescriptor  
ioStream  
htmlTag
```

The following are two basic capitalization guidelines for identifiers:

- ✓ **DO** use PascalCasing for the names of namespaces, types, members, and generic type parameters.

For example, use `TextColor` rather than `Textcolor` or `Text_color`. Single words, such as `Button`, simply have initial capitals. Compound words that are always written as a single word, like `endpoint`, are treated as single words and have initial capitals only. More information on compound words is given in section 3.1.3.

- ✓ **DO** use camelCasing for parameter names.

Table 3-1 describes the capitalization rules for different types of identifiers.

BRAD ABRAMS An early version of this table included a convention for instance field names. We later adopted the guideline that you should almost never use publicly exposed instance fields and should use properties instead. Thus, the guideline for publicly exposed instance fields was no longer needed. For the record, the convention was camelCasing.

TABLE 3-1: Capitalization Rules for Different Types of Identifiers

Identifier	Casing	Example
Namespace	Pascal	namespace System.Security { ... }
Type	Pascal	public class StreamReader { ... }
Interface	Pascal	public interface IEnumerable { ... }
Method	Pascal	public class Object { public virtual string ToString(); }
Property	Pascal	public class String { public int Length { get; } }
Event	Pascal	public class Process { public event EventHandler Exited; }
Field	Pascal	public class MessageQueue { public static readonly TimeSpan InfiniteTimeout; } public struct UInt32 { public const minValue = 0; }
Enum value	Pascal	public enum FileMode { Append, ... }
Type parameter, generic method	Pascal	public partial class Enum { public static TEnum Parse<TEnum>(string value) { ... } }

Identifier	Casing	Example
Type parameter, generic type	Pascal	<pre>public class Task<TResult> { ... }</pre>
Tuple element	Pascal	<pre>public partial class Range { public (int Offset, int Length) GetOffsetAndLength(int length) { ... } }</pre>
Parameter	Camel	<pre>public class Convert { public static intToInt32(string value); }</pre>

3.1.2 Capitalizing Acronyms

In general, it is important to avoid using acronyms in identifier names unless they are in common usage and are immediately understandable to anyone who might use the framework. For example, HTML, XML, and IO are all well understood, but less well-known acronyms should definitely be avoided.

■ **KRZYSZTOF CWALINA** Acronyms are distinct from abbreviations, which should never be used in identifiers. An acronym is a word made from the initial letters of a phrase, whereas an abbreviation simply shortens a word.

By definition, an acronym must be at least two characters. Acronyms of three or more characters follow the guidelines of any other word. Only the first letter is capitalized, unless it is the first word in a camel-cased parameter name, which is all lowercase.

As mentioned in the preceding section, two-character acronyms (e.g., IO) are treated differently, primarily to avoid confusion. Both characters should be capitalized unless the two-character acronym is the first

word in a camel-cased parameter name, in which case both characters are lowercase. The following examples illustrate all of these cases:

```
public void StartIO(Stream ioStream, bool closeIOStream);  
public void ProcessHtmlTag(string htmlTag)
```

- ✓ **DO** capitalize both characters of two-character acronyms, except the first word of a camel-cased identifier.

```
System.IO  
System.Threading.IOCompletionCallback  
public void StartIO(Stream ioStream)
```

- ✓ **DO** capitalize only the first character of acronyms with three or more characters, except the first word of a camel-cased identifier.

```
System.Xml  
System.Xml.XmlNode  
public void ProcessHtmlTag(string htmlTag)
```

- ✗ **DO NOT** capitalize any of the characters of any acronyms, whatever their length, at the beginning of a camel-cased identifier.

■ **BRAD ABRAMS** In my time working on the .NET Framework, I have heard every possible excuse for violating these naming guidelines. Many teams feel that they have some special reason to use case differently in their identifiers than in the rest of the Framework. These excuses include consistency with other platforms (MFC, HTML, etc.), avoiding geopolitical issues (casing of some country names), honoring the dead (abbreviation names that came up with some crypto algorithm), and the list goes on and on. For the most part, our customers have seen the places in which we have diverged from these guidelines (for even the best excuse) as warts in the Framework. The only time I think it really makes sense to violate these guidelines is when using a trademark as an identifier. However, I suggest not using trademarks, because they tend to change faster than APIs do.

■ **JEREMY BARTON** There's a mild misconception that because the types RSA, DSA, DES, and HMAC are all caps, there's some sort of exception to the rule when it comes to cryptographic (or other) algorithm names.

There isn't; those names were just mistakes. Newer types are named correctly: Advanced Encryption Standard (AES) is `Aes`, Elliptic Curve Digital Signature Algorithm (ECDSA) is `ECDsa` (two-letter acronym followed by a three-letter acronym), and Elliptic Curve Diffie-Hellman (ECDH) is `ECDiffieHellman` (avoiding the four concurrent capital letters required by a two-letter acronym following a two-letter acronym).

For the record, the following types are named incorrectly and should not serve as examples to follow: RSA, DSA, HMAC, HMACMD5, HMACSHA1, HMACSHA256, HMACSHA384, HMACSHA512, SHA1, SHA256, SHA384, SHA512, RIPEMD160, DES, TripleDES, MACTripleDES. (There may be others, and there are certainly many parameter names in .NET Cryptography that violate different rules.)

For these specific, existing names, their casing is kept for consistency with types that interact with them, such as `RSASignaturePadding` and `HashAlgorithmNames.SHA256`.

■ **BRAD ABRAMS** Here is an example of putting these naming guidelines to the test. We have the class shown here in the Framework today. It successfully follows the guidelines for casing and uses `Argb` rather than `ARGB`. But we have actually gotten bug reports along the lines of "How do you convert a color from an ARGB value—all I see are methods to convert 'from argument b'?"

```
public struct Color {  
    ...  
    public static Color FromArgb(int alpha, Color baseColor);  
    public static Color FromArgb(int alpha, int red, int green,  
        int blue);  
    public static Color FromArgb(int argb);  
    public static Color FromArgb(int red, int green, int blue);  
    ...  
}
```

In retrospect, should this have been a place where we violated the guidelines and used `FromARGB`? I do not think so. It turns out that this is a case of over-abbreviation. RGB is a well-recognized acronym for red-green-blue values. An ARGB value is a relatively uncommon abbreviation that includes the alpha channel. It would have been clearer to name these

AlphaRgb and would have been more consistent in naming with the rest of the Framework.

```
public struct Color {  
    ...  
    public static Color FromAlphaRgb(int alpha, Color baseColor);  
    public static Color FromAlphaRgb(int alpha, int red, int green,  
        int blue);  
    public static Color FromAlphaArgb(int argb);  
    public static Color FromAlphaRgb(int red, int green, int blue);  
    ...  
}
```

■ **STEPHEN TOUB** No matter how hard we try to follow all of these guidelines, sometimes we make mistakes. For example, `System.Text.UTF8Encoding` should really have been `System.Text.Utf8Encoding`. Now when we introduce new UTF8-related functionality, what do we do? Do we continue to use “UTF8” for consistency, or do we use “Utf8” for adherence to the guidelines? In general, we choose to not repeat the mistakes of the past, and newer types have used “Utf8”—for example, `System.Buffers.Text.Utf8Formatter`.

3.1.3 Capitalizing Compound Words and Common Terms

Most compound terms are treated as single words for purposes of capitalization.

X DO NOT capitalize each word in so-called closed-form compound words.

These are compound words written as a single word, such as `endpoint`. For the purpose of casing guidelines, treat a closed-form compound word as a single word. Use a current dictionary to determine if a compound word is written in closed form.

Table 3-2 shows capitalization for some of the most commonly used compound words and common terms.

TABLE 3-2: Capitalization and Spelling for Common Compound Words and Common Terms

Pascal	Camel	Not
BitFlag	bitFlag	Bitflag
Callback	callback	CallBack
Canceled	canceled	Cancelled
DoNot	doNot	Don't
Email	email	EMail
Endpoint	endpoint	EndPoint
Filename	filename	FileName
Gridline	gridline	GridLine
Hashtable	hashtable	HashTable
Id	id	ID
Indexes	indexes	Indices
Logoff ¹	logoff	LogOut
Logon ¹	logon	LogIn ²
Metadata	metadata	MetaData, metaData
Multipanel	multipanel	MultiPanel
Multiview	multiview	MultiView
Namespace	namespace	NameSpace
Ok	ok	OK
Pi	pi	PI
Placeholder	placeholder	PlaceHolder

Continues

TABLE 3-2: Continued

Pascal	Camel	Not
SignIn	signIn	SignOn
SignOut	signOut	SignOff
Timestamp	timestamp	TimeStamp
Username	username	UserName
WhiteSpace	whiteSpace	Whitespace
Writable	writable	Writeable

1. When used as a noun (properties, fields, parameters, types). As a verb (methods) it should be considered two words—for example, LogOff.
2. “Login” is now regarded as a closed-form noun, so a type with members named “Login” and “Password” would be OK, except in .NET we prefer “Logon” to “Login.” As a compound verb, “Log In” should be “Log On,” so “LogIn” is never used, and “LogOn” does not appear in this table.

Two other terms that are in common usage are in a category by themselves, because they are common slang abbreviations. The two words *Ok* and *Id* (cased as shown here) are exceptions to the guideline that no abbreviations should be used in names.

Table 3-2 and the rule about closed-form nouns are secondary to the general guideline of maintaining consistency. As of September 2019, “white space” is the dictionary entry, with some dictionaries having a note that computer science often uses it as a single word. If and when “whitespace” becomes an accepted closed-form noun, you should continue to use “WhiteSpace” in type hierarchies built with the older form of the word.

■ **BRAD ABRAMS** Table 3-2 presents specific examples found in the development of the .NET Framework. You might find it useful to create your own appendix to this table for compound words and other terms commonly used in your domain.

■ **BRAD ABRAMS** One abbreviation commonly used in COM interface names was Ex (for interfaces that were extended versions of previously existing interfaces). This abbreviation should be avoided in reusable libraries. Use instead a meaningful name that describes the new functionality. For example, rather than `IDispatchEx`, consider `IDynamicDispatch`.

■ **JEREMY BARTON** Several changes were made to Table 3-2 for the third edition of this book. The *Oxford English Dictionary*, as well as others, now considers “filename,” “logoff,” “logout,” “login,” “logon,” and “username” as closed-form nouns, enabling identifiers like `LogoffTime`. In the sense of an action, the OED still considers “log in,” “log on,” “log off,” or “log out” to be the only acceptable forms.

That doesn’t mean you should immediately change all of your existing names. However, in types that don’t interact with the older forms, you are free (and encouraged) to accept the evolving nature of language when it comes to these terms.

3.1.4 Case Sensitivity

Languages that can run on the CLR are not required to support case sensitivity, although some do. Even if your language supports it, other languages that might access your framework do not. Any APIs that are externally accessible, therefore, cannot rely on case alone to distinguish between two names in the same context.

■ **PAUL VICK** When it came to the question of case sensitivity, there was no question in the minds of the Visual Basic team that the CLR had to support case insensitivity as well as case sensitivity. Visual Basic has been case insensitive for a very long time, and the shock of trying to move VB developers (including myself) into a case-sensitive world would have made any of the other challenges we faced pale in comparison. Add to that the fact that COM is case insensitive, and the matter seemed pretty clear. The CLR would have to take case insensitivity into account.

■ **JEFFREY RICHTER** To be clear, the CLR is actually case sensitive. Some programming languages, like Visual Basic, are case insensitive. When the VB compiler is trying to resolve a method call to a type defined in a case-sensitive language like C#, the compiler (not the CLR) figures out the actual case of the method's name and embeds it in metadata. The CLR knows nothing about this. Now if you are using reflection to bind to a method, the reflection APIs do offer the ability to do case-insensitive lookups. This is the extent to which the CLR supports case insensitivity.

There is really only one guideline for case sensitivity, albeit an important one.

✗ **DO NOT** assume that all programming languages are case sensitive.
They are not. Names cannot differ by case alone.

3.2 General Naming Conventions

This section describes general naming conventions that relate to word choice, guidelines on using abbreviations and acronyms, and recommendations on how to avoid using language-specific names.

3.2.1 Word Choice

It is important that names of framework identifiers make sense on first reading. Identifier names should clearly state what each member does and what each type and parameter represents. To this end, it is more important that the name be clear than that it be short. Names should correspond to scenarios, logical or physical parts of the system, and well-known concepts, rather than to technologies or architecture.

✓ **DO** choose easily readable identifier names.

For example, a property named `HorizontalAlignment` is more English-readable than `AlignmentHorizontal`.

✓ **DO** favor readability over brevity. The property name `CanScrollHorizontally` is better than `ScrollableX` (an obscure reference to the X-axis).

X DO NOT use underscores, hyphens, or any other non-alphanumeric characters.

■ **JEREMY BARTON** Several sets of naming conventions for unit tests appear to violate this guideline with method names like `TestSubjectName_NoMatch`. But, since most test libraries are not distributed as reusable components, the guidelines in this book don't generally apply to test code.

X DO NOT use Hungarian notation.

■ **BRENT RECTOR** It might be useful here to define what Hungarian notation is. It is the convention of prefixing a variable name with some lowercase encoding of its data type. For example, the variable `uiCount` would be an unsigned integer. Another common convention also adds a prefix indicating the scope of the variable in addition to or in place of the type (see Jeffrey's later example of static and member variable-scope prefixes).

One downside of Hungarian notation is that developers frequently change the type of variables during early coding, which requires the name of the variable to also change. Additionally, while commonly used fundamental data types (integers, characters, etc.) had well-recognized and standard prefixes, developers frequently fail to use a meaningful and consistent prefix for their custom data types.

■ **KRZYSZTOF CWALINA** There have always been both positive and negative effects of using the Hungarian naming convention, and they still exist today. Positives include better readability (if used correctly). Negatives include cost of maintenance, confusion if maintenance was not done properly, and finally, Hungarian makes the API more cryptic (less approachable) to some developers. In the world of procedural languages (e.g., C) and the separation of the System APIs for advanced developers from framework libraries for a much wider developer group, the positives seemed to be greater than the negatives. Today, with System APIs designed to be approachable to more developers, and with object-oriented (OO) languages, the trade-off seems to be pulling in the other direction. OO encapsulation brings variable declaration and usage points closer together, OO style favors short, well-factored methods, and abstractions often make the exact type less important or even meaningless.

✗ **AVOID** using identifiers that conflict with keywords of widely used programming languages.

According to Rule 4 of the Common Language Specification (CLS), all compliant languages must provide a mechanism that allows access to named items that use a keyword of that language as an identifier. C#, for example, uses the @ sign as an escape mechanism in this case. However, it is still a good idea to avoid common keywords because it is much more difficult to use a method with the escape sequence than one without it.

■ **JEFFREY RICHTER** When I was porting my *Applied Microsoft .NET Framework Programming* book from C# to Visual Basic, I ran into this situation a lot. For example, the class library has `Delegate`, `Module`, and `Assembly` classes, and Visual Basic uses these same terms for keywords. This problem is exacerbated by the fact that VB is a case-insensitive language. Visual Basic, like C#, has a way to escape the keywords to disambiguate the situation to the compiler (using square brackets), but I was surprised that the VB team selected keywords that conflict with so many class library names.

✓ **DO** use only ASCII characters in identifier names.

All of the identifiers in .NET use ASCII characters, so anyone working with .NET is capable of typing those characters and working with files or tools that understand those characters. But the developers who want to use your library may not have easy capability to type non-ASCII characters, or may use text editors (or other tools) that do not reliably preserve non-ASCII data.

When basing an identifier on a word that properly requires diacritical marks, either use (or invent) the diacritical-free approximation or choose a different identifier.

Table 3-3 describes some suggested replacement spellings to avoid diacritical marks. It is provided for clarity, but is neither authoritative nor exhaustive.

TABLE 3-3: Alternative Spellings to Avoid Diacritical Marks

Spelling with Diacritics	Spelling for an Identifier
Edelweiß	Edelweiss
Mêlée	Melee
Naïve	Naive
Résumé	Resume
Röntgen	Roentgen

3.2.2 Using Abbreviations and Acronyms

In general, do not use abbreviations or acronyms in identifiers. As stated earlier, it is more important for names to be readable than it is for them to be brief. It is equally important not to use abbreviations and acronyms that are not generally understood—that is, do not use anything that the large majority of people who are not experts in a given field would not know the meaning of immediately.

X DO NOT use abbreviations or contractions as part of identifier names.

For example, use `GetWindow` rather than `GetWin`.

X DO NOT use any acronyms that are not widely accepted, and even if they are, only when necessary.

For example, UI is used for User Interface and HTML is used for HyperText Markup Language. Although many framework designers feel that some recent acronym will soon be widely accepted, it is bad practice to use it in framework identifiers.

For acronym capitalization rules, see section 3.1.2.

■ **BRAD ABRAMS** We continually debate about whether a given acronym is well known or not. A good divining rod is what I call the grep test. Simply use some search engine to grep the Web for the acronym. If the first few results returned are indeed the meaning you intend, it is likely that your acronym qualifies as well known; if you don't get those search results, think harder about the name. If you fail the test, don't just spell out the acronym but consider how you can be descriptive in the name.

3.2.3 Avoiding Language-Specific Names

Programming languages that target the CLR often have their own names (aliases) for the so-called primitive types. For example, `int` is a C# alias for `System.Int32`. To ensure that your framework can take full advantage of the cross-language interoperation that is one of the core features of the CLR, it is important to avoid the use of these language-specific type names in identifiers.

■ **JEFFREY RICHTER** Personally, I take this a step further and never use the language's alias names. I find that the alias adds nothing of value and introduces enormous confusion. For example, I'm frequently asked what the difference is between `String` and `string` in C#. I've even heard people say that strings (lowercase "S") are allocated on the stack while `Strings` (uppercase "S") are allocated on the heap. In my book *CLR via C#*, I give several reasons in addition to the one offered here for avoiding the alias names. Another example of a class library/language mismatch is the `NullReferenceException` class, which can be thrown by VB code. But VB uses `Nothing`, not `null`.

- ✓ **DO** use semantically interesting names rather than language-specific keywords for type names.
For example, `GetLength` is a better name than `GetInt`.
- ✓ **DO** use a generic CLR type name, rather than a language-specific name, in the rare cases when an identifier has no semantic meaning beyond its type.

For example, a method converting to System.Int64 should be named ToInt64, not ToLong (because System.Int64 is a CLR name for the C#-specific alias long). Table 3-4 presents several base data types using the CLR type names (as well as the corresponding type names for C#, Visual Basic, and C++).

TABLE 3-4: CLR Type Names for Language-Specific Type Names

C#	Visual Basic	C++	CLR
Sbyte	SByte	char	SByte
Byte	Byte	unsigned char	Byte
short	Short	short	Int16
ushort	UInt16	unsigned short	UInt16
int	Integer	int	Int32
uint	UInt32	unsigned int	UInt32
long	Long	_int64	Int64
ulong	UInt64	unsigned _int64	UInt64
float	Single	float	Single
double	Double	double	Double
bool	Boolean	bool	Boolean
char	Char	wchar_t	Char
string	String	String	String
object	Object	Object	Object

✓ **DO** use a common name, such as *value* or *item*, rather than repeating the type name, in the rare cases when an identifier has no semantic meaning and the type of the parameter is not important.

The following is a good example of methods of a class that support writing a variety of data types into a stream:

```
void Write(double value);
void Write(float value);
void Write(short value);
```

3.2.4 Naming New Versions of Existing APIs

Sometimes a new feature cannot be added to an existing type, even though the type's name implies that it is the best place for the new feature. In such a case, a new type needs to be added, which often leaves the framework designer with the difficult task of finding a good new name for the new type. Similarly, an existing member often cannot be extended or overloaded to provide additional functionality, so a member with a new name needs to be added. The guidelines that follow describe how to choose names for new types and members that supersede or replace existing types or members.

- ✓ **DO** use a name similar to the old API when creating new versions of an existing API.

This helps to highlight the relationship between the APIs.

```
class AppDomain {
    [Obsolete("AppDomain.SetCachePath has been deprecated. Please use
AppDomainSetup.CachePath instead.")]
    public void SetCachePath(String path) { ... }
}

class AppDomainSetup {
    public string CachePath { get { ... }; set { ... }; }
```

- ✓ **DO** prefer adding a suffix rather than a prefix to indicate a new version of an existing API.

■ **VANCE MORRISON** We did exactly this when we added a faster (but not completely backward-compatible) version of `ReaderWriterLock`. We called it `ReaderWriterLockSlim`. There was debate whether we should call it `SlimReaderWriterLock` (following the guideline that you write it like you say it in English), but decided the discoverability (and the fact that lexical sorting would put them close to each other) was more important.

This naming practice will assist discovery when browsing documentation or using IntelliSense. The old version of the API will be organized close to the new APIs, because most browsers and IntelliSense show identifiers in alphabetical order.

- ✓ **CONSIDER** using a brand-new, but meaningful, identifier, instead of adding a suffix or a prefix.
- ✓ **DO** use a numeric suffix to indicate a new version of an existing API, particularly if the existing name of the API is the only name that makes sense (e.g., if it is an industry standard) and if adding any meaningful suffix (or changing the name) is not an appropriate option.

```
// old API
[Obsolete("This type is obsolete. Please use the new version of the same
class, X509Certificate2.")]
public class X509Certificate { ... }
// new API
public class X509Certificate2 { ... }
```

■ **KRZYSZTOF CWALINA** I would use numeric suffixes as the very last resort. A much better approach is to use a new name or a meaningful suffix.

The BCL team shipped a new type named `TimeZone2` in one of the early prereleases of .NET Framework 3.5. The name immediately became the center of a controversy in the blogging community. After a set of lengthy discussions, the team decided to rename the type to `TimeZoneInfo`, which is not a great name but is much better than `TimeZone2`.

It's interesting to note that nobody dislikes `X509Certificate2`. My interpretation of this fact is that programmers are more willing to accept the ugly numeric suffix on rarely used library types somewhere in the corner of the framework than on a core type in the `System` namespace.

■ **JEREMY BARTON** I think that `X509Certificate2` is successful as the type name of the more usable of the two X.509 certificate classes in part because it is so much more powerful than `X509Certificate` that there's really no reason to use the former class. Compounded with there being very few APIs that provide or accept `X509Certificate`, it's simply that `X509Certificate2` is the name of the certificate type in .NET. (That said, I'll stand contrary to Krzysztof's claim and say that I wish that the functionality had been added to `X509Certificate` without a new type being introduced.)

✗ **DO NOT** use the “Ex” (or a similar) suffix for an identifier to distinguish it from an earlier version of the same API.

```
[Obsolete("This type is obsolete. ...")]
public class Car { ... }

// new API
public class CarEx { ... } // the wrong way
public class CarNew { ... } // the wrong way
public class Car2 { ... } // an acceptable way
public class Automobile { ... } // a better way
```

✓ **DO** use the “64” suffix when introducing versions of APIs that operate on a 64-bit integer (a long integer) instead of a 32-bit integer. You only need to take this approach when the existing 32-bit API exists; don't do it for brand-new APIs with only a 64-bit version.

For example, various APIs on `System.Diagnostics.Process` return `Int32` values representing memory sizes, such as `PagedMemorySize` or `PeakWorkingSet`. To appropriately support these APIs on 64-bit systems, APIs have been added that have the same name but a “64” suffix.

```
public class Process {
    // old APIs
    public int PeakWorkingSet { get; }
    public int PagedMemorySize { get; }
    // ...
```

```
// new APIs
public long PeakWorkingSet64 { get; }
public long PagedMemorySize64 { get; }
}
```

KIT GEORGE Note that this guideline applies only to retrofitting APIs that have already shipped. When designing a brand-new API, use the most appropriate type and name for the API that will work on all platforms, and avoid using both “32” and “64” suffixes. Consider using overloading.

3.3 Names of Assemblies, DLLs, and Packages

An assembly is the unit of deployment and identity for managed code programs. Although assemblies can span one or more files, typically an assembly maps one-to-one with a dynamic-link library (DLL). Additionally, when a DLL is distributed via a package management system, the DLL and the package typically have the same name. Therefore, this section describes only DLL naming conventions, which then can be mapped to assembly naming conventions and package naming conventions.

JEFFREY RICHTER Multifile assemblies are rarely used, and Visual Studio has no built-in support for them.

Keep in mind that namespaces are distinct from DLL and assembly names. Namespaces represent logical groupings for developers, whereas DLLs and assemblies represent packaging and deployment boundaries. DLLs can contain multiple namespaces for product factoring and other reasons. Because namespace factoring is different than DLL factoring, you should design them independently. For example, if you decide to name your DLL `MyCompany.MyTechnology`, it does not mean that the DLL has to contain a namespace named `MyCompany.MyTechnology`, though it can.

■ **JEFFREY RICHTER** Programmers are frequently confused by the fact that the CLR does not enforce a relationship between namespaces and assembly filenames. For example, in .NET Framework, `System.IO.FileStream` is in `mscorlib.dll`, and `System.IO.FileSystemWatcher` is in `System.dll`. As you can see, types in a single namespace can span multiple files. Also notice that the .NET Framework doesn't ship with a `System.IO.dll` file at all.

■ **BRAD ABRAMS** We decided early in the design of the CLR to separate the developer view of the platform (namespaces) from the packaging and deployment view of the platform (assemblies). This separation allows each to be optimized independently based on its own criteria. For example, we are free to factor namespaces to group types that are functionally related (e.g., all the I/O stuff is in `System.IO`), but the assemblies can be factored for performance (load time), deployment, servicing, or versioning reasons.

✓ **DO** choose names for your assembly DLLs that suggest large chunks of functionality, such as `System.Data`.

Assembly, DLL, and package names don't have to correspond to namespace names, but it is reasonable to follow the namespace name when naming assemblies. A good rule of thumb is to name the DLL based on the common prefix of the namespaces contained in the assembly. For example, an assembly with two namespaces, `MyCompany.MyTechnology.FirstFeature` and `MyCompany.MyTechnology.SecondFeature`, could be called `MyCompany.MyTechnology.dll`.

✓ **CONSIDER** naming DLLs according to the following pattern:

<Company>.<Component>.dll

<Component> can contain more than one dot-separated clauses. For example:

`Microsoft.VisualBasic.dll`
`Microsoft.VisualBasic.Vsa.dll`
`Fabrikam.Security.dll`
`Litware.Controls.dll`

3.4 Names of Namespaces

As with other naming guidelines, the goal when naming namespaces is to create sufficient clarity for the programmer using the framework to immediately know what the content of the namespace is likely to be. The following template specifies the general rule for naming namespaces:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]
```

The following are examples:

```
Microsoft.VisualStudio
Microsoft.VisualStudio.Design
Fabrikam.Math
Litware.Security
```

- ✓ **DO** prefix namespace names with a company name to prevent namespaces from different companies from having the same name.

For example, the Microsoft Office automation APIs provided by Microsoft should be in the namespace `Microsoft.Office`.

■ BRAD ABRAMS It is important to use the official name of your company or organization when choosing the first part of your namespace name to avoid possible conflicts. For example, if Microsoft had chosen to use MS as its root namespace, it might have been confusing to developers at other companies that use MS as an abbreviation.

■ BRAD ABRAMS This means staying away from the latest cool and catchy name that the marketing folks have come up with. It is fine to tweak the branding of a product from release to release, but the namespace name will be burned into your client's code forever. Therefore, choose something that is technically sound and not subject to the marketing whims of the day.

- ✓ **DO** use a stable, version-independent product name at the second level of a namespace name.

■ **BRAD ABRAMS** We added a set of controls to ASP.NET late in the ship cycle for V1.0 of the .NET Framework that rendered for mobile devices. Because these controls came from a team in a different division, our immediate reaction was to put them in a different namespace (`System.Web.MobileControls`). Then, after a couple of reorganizations and .NET Framework versions, we realized a better engineering trade-off was to fold that functionality into the existing controls in `System.Web.Controls`. In retrospect, we let internal organizational differences affect the public exposure of the APIs, and we came to regret that later. Avoid this type of mistake in your designs.

X DO NOT use organizational hierarchies as the basis for names in namespace hierarchies, because group names within corporations tend to be short-lived. Organize the hierarchy of namespaces around groups of related technologies.

✓ DO use PascalCasing, and separate namespace components with periods (e.g., `Microsoft.Office.PowerPoint`). If your brand employs nontraditional casing, you should follow the casing defined by your brand, even if it deviates from normal namespace casing.

✓ CONSIDER using plural namespace names where appropriate.

For example, use `System.Collections` instead of `System.Collection`. Brand names and acronyms are exceptions to this rule, however. For example, use `System.IO` instead of `System.IOs`.

X DO NOT use the same name for a namespace and a type in that namespace.

For example, do not use `Debug` as a namespace name and then also provide a class named `Debug` in the same namespace. Several compilers require such types to be fully qualified.

These guidelines cover general namespace naming guidelines, but the next section provides specific guidelines for certain special subnamespaces.

3.4.1 Namespaces and Type Name Conflicts

Namespaces are used to organize types into a logical and easy-to-explore hierarchy. They are also indispensable in resolving type name ambiguities that might arise when importing multiple namespaces. However, that fact should not be used as an excuse to introduce known ambiguities between types in different namespaces that are commonly used together. Developers should not be required to qualify type names in common scenarios.

X DO NOT introduce overly general type names such as Element, Node, Log, and Message.

There is a very high probability that doing so will lead to type name conflicts in common scenarios. You should qualify the general type names (e.g., FormElement, XmlNode, EventLog, SoapMessage).

There are specific guidelines for avoiding type name conflicts for different categories of namespaces. Namespaces can be divided into the following categories:

- Application model namespaces
- Infrastructure namespaces
- Core namespaces
- Technology namespace groups

3.4.1.1 Application Model Namespaces

Namespaces belonging to a single application model are very often used together, but they are almost never used with namespaces of other application models. For example, the System.Windows.Forms namespace is very rarely used together with the System.Web.UI namespace. The following are well-known application model namespace groups:

```
System.Windows*
System.Web.UI*
```

X DO NOT give the same name to multiple types across namespaces within a single application model.

For example, do not add a type named `Page` to the `System.Web.UI.Adapters` namespace, because the `System.Web.UI` namespace already contains a type named `Page`.

3.4.1.2 Infrastructure Namespaces

This group contains namespaces that are rarely imported during development of common applications. For example, `.Design` namespaces are mainly used when developing programming tools. Avoiding conflicts with types in these namespaces is not critical.

```
System.Windows.Forms.Design  
*.Design  
*.Permissions
```

3.4.1.3 Core Namespaces

The core namespaces include all `System` namespaces, excluding namespaces of the application models and the Infrastructure namespaces. Core namespaces include, among others, `System`, `System.IO`, `System.Xml`, and `System.Net`.

X DO NOT give types names that would conflict with any type in the core namespaces.

For example, never use `Stream` as a type name. It would conflict with `System.IO.Stream`, a very commonly used type.

By the same token, do not add a type named `EventLog` to the `System.Diagnostics.Events` namespace, because the `System.Diagnostics` namespace already contains a type named `EventLog`.

3.4.1.4 Technology Namespace Groups

This category includes all namespaces with the same first two namespace nodes (`<Company>.<Technology>*`), such as `Microsoft.Build.Utilities` and `Microsoft.Build.Tasks`. It is important that types belonging to a single technology do not conflict with each other.

X DO NOT assign type names that would conflict with other types within a single technology.

X DO NOT introduce type name conflicts between types in technology namespaces and an application model namespace (unless the technology is not expected to be used with the application model).

For example, you should not add a type named `Binding` to the `Microsoft.VisualBasic` namespace because the `System.Windows.Forms` namespace already contains that type name.

3.5 Names of Classes, Structs, and Interfaces

In general, class and struct names should be nouns or noun phrases, because they represent entities of the system. A good rule of thumb is that if you are not able to come up with a noun or a noun phrase name for a class or a struct, you probably should rethink the general design of the type. Interfaces representing roots of a hierarchy (e.g., `IList<T>`) should also use nouns or noun phrases. Interfaces representing capabilities should use adjectives and adjective phrases (e.g., `IComparable<T>`, `IFormattable`).

Another important consideration is that the most easily recognizable names should be used for the most commonly used types, even if the name fits some other less-used type better in the purely technical sense. For example, a type used in mainline scenarios to submit print jobs to print queues should be named `Printer`, rather than `PrintQueue`. Even though technically the type represents a print queue and not the physical device (`printer`), from the scenario point of view, `Printer` is the ideal name because most people are interested in submitting print jobs and not in other operations related to the physical printer device (e.g., configuring the printer). If you need to provide another type that corresponds, for example, to the physical printer to be used in configuration scenarios, the type could be called `PrinterConfiguration` or `PrinterManager`.

■ **KRZYSZTOF CWALINA** I know this goes against the technical precision that is one of the core character traits of most software engineers, but I really do think it's more important to have better names from the point of view of the most common scenario, even if it results in slightly inconsistent or even wrong type names from a purely technical point of view. Advanced users will be able to understand slightly inconsistent naming. Most users are usually not concerned with technicalities and will not even notice the inconsistency, but they *will* appreciate the names guiding them to the most important APIs.

Similarly, names of the most commonly used types should reflect usage scenarios, not inheritance hierarchy. Most users use the leaves of an inheritance hierarchy almost exclusively; they are rarely concerned with the structure of the hierarchy. Yet API designers often see the inheritance hierarchy as the most important criterion for type name selection. For example, `Stream`, `StreamReader`, `TextReader`, `StringReader`, and `FileStream` all describe the place of each of the types in the inheritance hierarchy quite well, but they obscure the most important information for the majority of users: the type that they need to instantiate to read text from a file.

The naming guidelines that follow apply to general type naming.

✓ **DO** name classes and structs with nouns or noun phrases, using PascalCasing.

This distinguishes type names from methods, which are named with verb phrases.

✓ **DO** name interfaces with adjective phrases, or occasionally with nouns or noun phrases.

Nouns and noun phrases should be used rarely. They might indicate that the type should be an abstract class, rather than an interface. See section 4.3 for details about deciding how to choose between abstract classes and interfaces.

✗ **DO NOT** give class names a prefix (e.g., "C").

■ **KRZYSZTOF CWALINA** One of the few prefixes used is “I” for interfaces (as in `ICollection`), but that is for historical reasons. In retrospect, I think it would have been better to use regular type names. In a majority of the cases, developers don’t care that something is an interface and not an abstract class, for example.

■ **BRAD ABRAMS** The “I” prefix for interfaces is a clear recognition of the influence of COM (and Java) on the .NET Framework. COM popularized—even institutionalized—the notation that interfaces begin with “I.” Although we discussed diverging from this historical pattern, we decided to carry forward the pattern because so many of our users were already familiar with COM.

■ **JEFFREY RICHTER** Personally, I like the “I” prefix, and I wish we had more stuff like this. Little one-character prefixes go a long way toward keeping code terse and yet descriptive. As I said earlier, I use prefixes for my private type fields because I find this very useful.

■ **BRENT RECTOR** Note: This is really another application of Hungarian notation (though one without the disadvantages of the notation’s use in variable names).

✓ **CONSIDER** ending the name of derived classes with the name of the base class.

This is very readable and explains the relationship clearly. Two examples of this in code are `ArgumentOutOfRangeException`, which is a kind of `Exception`, and `SerializableAttribute`, which is a kind of `Attribute`. However, it is important to use reasonable judgment in applying this guideline; for example, the `Button` class is a kind of `Control`, although `Control` doesn’t appear in its name. The following are examples of correctly named classes:

```
public class FileStream : Stream {...}  
public class Button : Control {...}
```

- ✓ **DO** prefix interface names with the letter *I*, to indicate that the type is an interface.

For example, `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and `IPersistable` (adjective) are appropriate interface names. As with other type names, avoid abbreviations.

- ✓ **DO** ensure that the names differ only by the “*I*” prefix on the interface name when you are defining a class–interface pair where the class is a standard implementation of the interface.

The following example illustrates this guideline for the interface `IComponent` and its standard implementation, the class `Component`:

```
public interface IComponent { ... }
public class Component : IComponent { ... }
```

3.5.1 Names of Generic Type Parameters

Generics were added to .NET Framework 2.0. This feature introduced a new kind of identifier called *type parameter*. The following guidelines describe naming conventions related to naming such type parameters:

- ✓ **DO** name generic type parameters with descriptive names unless a single-letter name is completely self-explanatory and a descriptive name would not add value.

```
public interface ISessionChannel<TSession> { ... }
public delegate TOutput Converter<TInput,TOutput>(TInput from);
public struct Nullable<T> { ... }
public class List<T> { ... }
```

- ✓ **CONSIDER** using `T` as the type parameter name for types with one single-letter type parameter.

```
public int IComparer<T> { ... }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T:struct { ... }
```

- ✓ **DO** prefix descriptive type parameter names with `T`.

```
public interface ISessionChannel<TSession> where TSession : ISession{
    TSession Session { get; }
}
```

- ✓ **CONSIDER** indicating constraints placed on a type parameter in the name of the parameter.

For example, a parameter constrained to `ISession` might be called `TSession`.

3.5.2 Names of Common Types

If you are deriving from or implementing types contained in .NET, it is important to follow the guidelines in this section.

- ✓ **DO** follow the guidelines described in Table 3-5 when naming types derived from or implementing certain .NET types.

These suffixing guidelines apply to the whole hierarchy of the specified base type. For example, it is not just types derived directly from `System.Exception` that need the suffixes, but also those derived from `Exception` subclasses.

These suffixes should be reserved for the named types. Types derived from or implementing other types should not use these suffixes. For example, the following represent incorrect naming:

```
public class ElementStream : Object { ... }
public class WindowsAttribute : Control { ... }
```

TABLE 3-5: Name Rules for Types Derived from or Implementing Certain Core Types

Base Type	Derived/Implementing Type Guideline
<code>System.Attribute</code>	✓ DO add the suffix “Attribute” to names of custom attribute classes.
<code>System.Delegate</code>	<p>✓ DO add the suffix “EventHandler” to names of delegates that are used in events.</p> <p>✓ DO add the suffix “Callback” to names of delegates other than those used as event handlers.</p> <p>✗ DO NOT add the suffix “Delegate” to a delegate.</p>
<code>System.EventArgs</code>	✓ DO add the suffix “EventArgs.”

Continues

TABLE 3-5: Continued

Base Type	Derived/Implementing Type Guideline
System.Enum	<p>X DO NOT derive from this class; use the keyword supported by your language instead; for example, in C#, use the enum keyword.</p> <p>X DO NOT add the suffix “Enum” or “Flag.”</p>
System.Exception	✓ DO add the suffix “Exception.”
IDictionary IDictionary< TKey , TValue >	✓ DO add the suffix “Dictionary.” Note that IDictionary is a specific type of collection, but this guideline takes precedence over the more general collections guideline that follows.
IEnumerable ICollection IList IEnumerable< T > ICollection< T > IList< T >	✓ DO add the suffix “Collection,” except for reusable, specialized data types such as “Queue” and “HashSet.”
System.IO.Stream	✓ DO add the suffix “Stream.”

3.5.3 Naming Enumerations

Names of enumeration types (also called enums) in general should follow the standard type-naming rules (PascalCasing, etc.). However, there are additional guidelines that apply specifically to enums.

✓ DO use a singular type name for an enumeration unless its values are bit fields.

```
public enum ConsoleColor {
    Black,
    Blue,
    Cyan,
    ...
}
```

- ✓ **DO** use a plural type name for an enumeration with bit fields as values, also called flags enum.

```
[Flags]
public enum ConsoleModifiers {
    Alt = 1 << 0,
    Control = 1 << 1,
    Shift = 1 << 2,
}
```

- ✗ **DO NOT** use an “Enum” suffix in enum type names.

For example, the following enum is badly named:

```
// Bad naming
public enum ColorEnum {
    ...
}
```

- ✗ **DO NOT** use “Flag” or “Flags” suffixes in enum type names.

For example, the following enum is badly named:

```
// Bad naming
[Flags]
public enum ColorFlags {
    ...
}
```

- ✗ **DO NOT** use a prefix on enumeration value names (e.g., “ad” for ADO enums, “rtf” for rich text enums).

```
public enum ImageMode {
    ImageModeBitmap = 0, // ImageMode prefix is not necessary
    ImageModeGrayscale = 1,
    ImageModeIndexed = 2,
    ImageModeRgb = 3,
}
```

The following naming scheme would be better:

```
public enum ImageMode {
    Bitmap = 0,
    Grayscale = 1,
    Indexed = 2,
    Rgb = 3,
}
```

■ **BRAD ABRAMS** Notice that this guideline is the exact opposite of common usage in C++ programming. It is important in C++ to fully qualify each enum member because they can be accessed outside of the scope of the enum name. However, in the managed world, enum members are only accessed through the scope of the enum name.

3.6 Names of Type Members

Types are made of members: methods, properties, events, constructors, and fields. The following sections describe guidelines for naming type members.

3.6.1 Names of Methods

Because methods are the means of taking action, the design guidelines require that method names be verbs or verb phrases. Following this guideline also serves to distinguish method names from property and type names, which are noun or adjective phrases.

■ **STEVEN CLARKE** Do your best to name methods according to the task that they enable, not according to some implementation detail. In a usability study on the `System.Xml` APIs, participants were asked to write code that would perform a query over an instance of an `XPathDocument`. To do this, participants needed to call the `CreateXPathNavigator` method from `XPathDocument`. This returns an instance of an `XPathNavigator` that is used to iterate over the document data returned by a query. However, no participants expected or realized that they would have to do this. Instead, they expected to be able to call some method named `Query` or `Select` on the document itself. Such a method could just as easily return an instance of `XPathNavigator` in the same way that `CreateXPathNavigator` does. By tying the name of the method more directly to the task it enables, rather than to the implementation details, it is more likely that developers using your API will be able to find the correct method to accomplish a task.

- ✓ **DO** give methods names that are verbs or verb phrases.

```
public class String {
    public int CompareTo(...);
    public string[] Split(...);
    public string Trim();
}
```

3.6.2 Names of Properties

Unlike other members, properties should be given noun phrase or adjective names. That is because a property refers to data, and the name of the property reflects that. PascalCasing is always used for property names.

- ✓ **DO** name properties using a noun, noun phrase, or adjective.

```
public class String {
    public int Length { get; }
}
```

- ✗ **DO NOT** have properties that match the name of “Get” methods, such as a property named `TextWriter` and a method named `GetTextWriter`.

```
public string TextWriter { get {...} set {...} }
public string GetTextWriter(int value) { ... }
```

This pattern typically indicates that the property should really be a method. See section 5.1.3 for additional information.

- ✓ **DO** name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by “List” or “Collection.”

```
public class ListView {
    // good naming
    public ItemCollection Items { get; }

    // bad naming
    public ItemCollection ItemCollection { get; }
}
```

- ✓ **DO** name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has,” but only where it adds value.

For example, CanRead is more understandable than Readable. However, Created is actually more readable than IsCreated. Having the prefix is often too verbose and unnecessary, particularly in the face of IntelliSense in the code editors. It is just as clear to type MyObject.Enabled = and have IntelliSense give you the choice of true or false as it is to have MyObject.IsEnabled =, and the second approach is more verbose.

■ **KRZYSZTOF CWALINA** In selecting names for Boolean properties and functions, consider testing out the common uses of the API in an if-statement. Such a usage test will highlight whether the word choices and grammar of the API name (e.g., active versus passive voice, singular versus plural) make sense as English phrases. For example, both of the following

```
if(collection.Contains(item))  
if(regularExpression.Matches(text))
```

read more naturally than

```
if(collection.IsContained(item))  
if(regularExpression.Match(text))
```

Also, all else being equal, you should prefer the active voice to the passive voice:

```
if(stream.CanSeek) // better than ..  
if(stream.IsSeekable)
```

- ✓ **CONSIDER** giving a property the same name as its type.

For example, the following property correctly gets and sets an enum value named Color, so the property is named Color:

```
public enum Color {...}  
public class Control {  
    public Color Color { get; set; }  
}
```

3.6.3 Names of Events

Events always refer to some action, either one that is happening or one that has occurred. Therefore, as with methods, events are named with verbs, and verb tense is used to indicate the time when the event is raised.

- ✓ **DO** name events with a verb or a verb phrase.

Examples include `Clicked`, `Painting`, and `DroppedDown`.

- ✓ **DO** give events names with a concept of before and after, using the present and past tenses, such as `Closing` and `Closed`.

For example, a close event that is raised before a window is closed would be called `Closing`, and one that is raised after the window is closed would be called `Closed`.

- ✗ **DO NOT** use “Before” or “After” prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.

- ✓ **DO** name event handlers (delegates used as types of events) with the “`EventHandler`” suffix, such as `ClickedEventHandler`.

```
public delegate void ClickedEventHandler(object sender,  
                                         ClickedEventArgs e);
```

Note that you should create custom event handlers very rarely. Instead, most APIs should simply use `EventHandler<T>`. Section 5.4.1 talks about event design in more detail.

JASON CLARK Today, you would rarely need to define your own “`EventHandler`” delegate. Instead, you should use the `EventHandler-
<TEventArgs>` delegate type, where `TEventArgs` is either `EventArgs` or your own `EventArgs` derived class. This reduces type definitions in the system and ensures that your event follows the pattern described in the preceding bullet.

- ✓ **DO** use two parameters named *sender* and *e* in event handlers.

The *sender* parameter represents the object that raised the event. The *sender* parameter is typically of type `object`, even if it is possible to employ a more specific type. The pattern is used consistently across .NET and is described in more detail in section 5.4.

■ **JEREMY BARTON** The `object` *sender* parameter may be my least favorite guideline in .NET. But, for consistency, I have followed it in every event I've ever declared.

```
public delegate void <EventName>EventHandler(object sender,  
                                         <EventName>EventArgs e);
```

- ✓ **DO** name event argument classes with the “`EventArgs`” suffix, as shown in the following example:

```
public class ClickedEventArgs : EventArgs {  
    int x;  
    int y;  
    public ClickedEventArgs (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int X { get { return x; } }  
    public int Y { get { return y; } }  
}
```

3.6.4 Naming Fields

The field-naming guidelines apply to static public and protected fields. Internal and private fields are not covered by guidelines, and public or protected instance fields are not allowed by the member design guidelines, which are described in Chapter 5.

- ✓ **DO** use PascalCasing in field names.

```
public class String {  
    public static readonly string Empty = "";  
}
```

```
public struct UInt32 {
    public const MinValue = 0;
}
```

✓ **DO** name fields using a noun, noun phrase, or adjective.

✗ **DO NOT** use a prefix for public or protected field names.

For example, do not use “g_” or “s_” to indicate static fields. Publicly accessible fields (the subject of this section) are very similar to properties from the API design point of view; therefore, they should follow the same naming conventions as properties.

■ **BRAD ABRAMS** As with just about all the guidelines in this book, this guideline is meant to apply only to publicly exposed fields. In this case, it's important that the names be clean and simple so the masses of consumers can easily understand them. As many have noted, there are very good reasons to use some sort of convention for private fields and local variables.

3.7 Naming Parameters

Beyond the obvious reason of readability, it is important to follow the guidelines for parameter names because parameters are displayed in documentation and in the designer when visual design tools provide IntelliSense and class browsing functionality.

✓ **DO** use camelCasing in parameter names.

```
public class String {
    public bool Contains(string value);
    public string Remove(int startIndex, int count);
}
```

✓ **DO** use descriptive parameter names.

Parameter names should be descriptive enough to use with their types to determine their meaning in most scenarios.

- ✓ **CONSIDER** using names based on a parameter's meaning rather than the parameter's type.

Development tools generally provide useful information about the type, so the parameter name can be put to better use describing semantics rather than the type. Occasional use of type-based parameter names is entirely appropriate—but it is not ever appropriate under these guidelines to revert to the Hungarian naming convention.

3.7.1 Naming Operator Overload Parameters

This section discusses naming parameters of operator overloads.

- ✓ **DO** use *left* and *right* for binary operator overload parameter names if there is no meaning to the parameters.

```
public static TimeSpan operator-(DateTimeOffset left,  
                               DateTimeOffset right)  
public static bool operator==(DateTimeOffset left,  
                               DateTimeOffset right)
```

- ✓ **DO** use *value* for unary operator overload parameter names if there is no meaning to the parameters.

```
public static BigInteger operator-(BigInteger value);
```

- ✓ **CONSIDER** meaningful names for operator overload parameters if doing so adds significant value.

```
public static BigInteger Divide(BigInteger dividend,  
                               BigInteger divisor);
```

- ✗ **DO NOT** use abbreviations or numeric indices for operator overload parameter names.

```
// incorrect parameter naming  
public static bool operator ==(DateTimeOffset d1,  
                               DateTimeOffset d2);
```

3.8 Naming Resources

The guidance from this section in previous editions is obsolete, and has been archived in Appendix B.

X DO NOT directly expose localizable resources as public (or protected) members.

The types and members automatically generated by a resource editor should use the `internal` access modifier.

When it does make sense to expose a resource via public API, use intentional type and member design.

SUMMARY

The naming guidelines described in this chapter, if followed, provide a consistent scheme that will make it easy for users of a framework to identify the function of elements of the framework. The guidelines provide naming consistency across frameworks developed by different organizations or companies.

The next chapter provides general guidelines for implementing types.

This page intentionally left blank

4

Type Design Guidelines

FROM THE CLR perspective, there are only two categories of types—reference types and value types—but for the purpose of discussing framework design, we divide types into more logical groups, each with its own specific design rules. Figure 4-1 shows these logical groups.

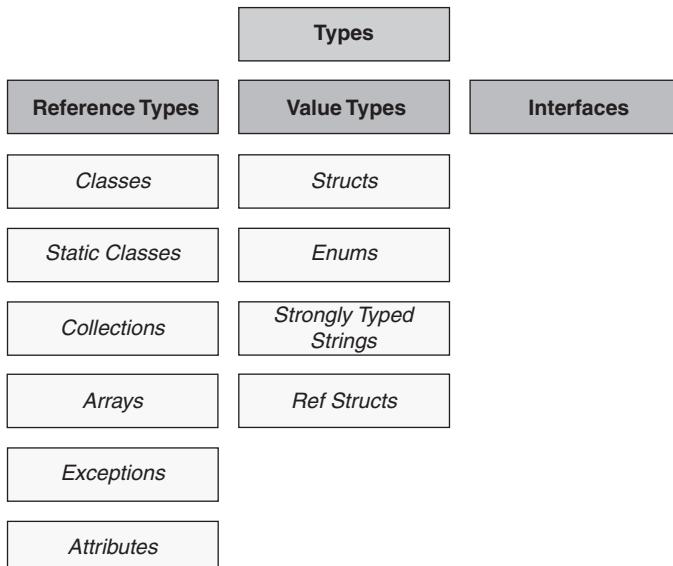


FIGURE 4-1: The logical grouping of types

Classes are the general case of reference types. They make up the bulk of types in the majority of frameworks. Classes owe their popularity to the rich set of object-oriented features they support and to their general applicability. Base classes and abstract classes are special logical groups related to extensibility. Extensibility and base classes are covered in Chapter 6.

Interfaces are types that can be implemented by both reference types and value types. As a consequence, they serve as roots of polymorphic hierarchies of reference types and value types. In addition, interfaces can be used to simulate multiple inheritance, which is not natively supported by the CLR.

Structs are the general case of value types and should be reserved for small, simple types, similar to language primitives.

Enums are a special case of value types used to define short sets of values, such as days of the week, console colors, and so on.

Static classes are types intended to be containers for static members. They are commonly used to provide shortcuts to other operations.

Delegates, exceptions, attributes, arrays, and collections are all special cases of reference types intended for specific uses. Guidelines for their design and usage are discussed elsewhere in this book.

✓ **DO** ensure that each type is a well-defined set of related members, not just a random collection of unrelated functionality.

It is important that a type can be described in one simple sentence. A good definition should also rule out functionality that is only tangentially related.

■ **BRAD ABRAMS** If you have ever managed a team of people, you know that they don't do well without a crisp set of responsibilities. Well, types work the same way. I have noticed that types without a firm and focused scope tend to be magnets for more random functionality, which over time makes a small problem a lot worse. It becomes more difficult to justify why the next member with even more random functionality does not belong in the type. As the focus of the members in a type blurs, the developer's ability to predict where to find a given functionality is impaired, and therefore so is productivity.

■ **RICO MARIANI** Good types are like good diagrams: What has been omitted is as important to clarity and usability as what has been included. Every additional member you add to a type starts at a net negative value; only by proven usefulness does it go from there to positive. If you add too much in an attempt to make the type more useful to some users, you are just as likely to make the type useless to everyone.

■ **JEFFREY RICHTER** When I was learning OOP back in the early 1980s, I was taught a mantra that I still honor today: If things get too complicated, make more types. Sometimes I find that I am thinking really hard trying to define a good set of methods for a type. When I start to feel that I'm spending too much time on this or when things just don't seem to fit together well, I remember my mantra and I define more, smaller types, where each type has well-defined functionality. This has worked extremely well for me over the years.

On the flip side, sometimes types do end up being dumping grounds for various loosely related functions. .NET offers several types like this, such as `Marshal`, `GC`, `Console`, `Math`, and `Application`. You will note that all members of these types are static, so it is not possible to create any instances of these types. Programmers seem to be OK with this. Fortunately, these types' methods are separated a bit into separate types. It would be awful if all these methods were defined in just one type!

4.1 Types and Namespaces

You should decide how to factor your functionality into a set of functional areas represented by namespaces when you design a large framework. This kind of top-down architectural design is important because it ensures a coherent set of namespaces containing types that work well together. The namespace design process is iterative, of course, and it should be expected that the design will have to be tweaked as types are added to the namespaces over the course of several releases. This philosophy leads to the following guidelines.

- ✓ **DO** use namespaces to organize types into a hierarchy of related feature areas.

The hierarchy should be optimized for developers browsing the framework for desired APIs.

■ **KRZYSZTOF CWALINA** This is an important guideline. Contrary to popular belief, the main purpose of namespaces is not to help in resolving naming conflicts between types with the same name. As the guideline states, the main purpose of namespaces is to organize types in a hierarchy that is coherent, easy to navigate, and easy to understand.

I consider type-name conflicts in a single framework to indicate sloppy design. Types with identical names should either be merged to allow for better integration between parts of the library or be renamed to improve code readability and searchability.

✗ **AVOID** very deep namespace hierarchies. Such hierarchies are difficult to browse because the user has to backtrack often.

✗ **AVOID** having too many namespaces.

Users of a framework should not have to import many namespaces in the most common scenarios. Types that are used together in common scenarios should reside in a single namespace if at all possible. This guideline does not mean “have only one namespace,” but instead urges you to seek balance. The types used by developers and the types used by code generators or IDE designers to help the developers are two different concepts and should be in different, but related, namespaces.

■ **JEFFREY RICHTER** As an example of a problem, the runtime serializer types are defined under the `System.Runtime.Serialization` namespace and its subnamespaces. However, the `Serializable` and `NonSerialized` attributes are incorrectly defined in the `System` namespace. Because these types are not in the same namespace, developers don’t realize that they are closely related. In fact, I have run into many developers who apply the `Serializable` attribute to a class that they are serializing with the `System.Xml.Serialization`’s `XmlSerializer` type. However, the `XmlSerializer` completely ignores the `Serializable` attribute; applying the attribute gives no value and just bloats your assembly’s metadata.

✗ **AVOID** having types designed for advanced scenarios in the same namespace as types intended for common programming tasks.

This makes it easier to understand the basics of the framework and to use the framework in the common scenarios.

■ **BRAD ABRAMS** One of the best features of Visual Studio is IntelliSense, which provides a drop-down list for your likely next type or member usage. The benefit of this feature is inversely proportional to the number of options. That is, if there are too many items in the list, it takes longer to find the one you are looking for. Following this guideline to split out advanced functionality into a separate namespace enables developers to see the smallest number of types possible in the common case.

■ **BRIAN PEPIN** One thing we've learned is that most programmers live or die by IntelliSense. If something isn't listed in the drop-down list, most programmers won't believe it exists. But, as Brad says, too much of a good thing can be bad, and having too much stuff in the drop-down list dilutes its value. If you have functionality that should be in the same namespace but you don't think it needs to be shown all the time to users, you can use the `EditorBrowsable` attribute.

■ **RICO MARIANI** Don't go crazy adding members for every exotic thing someone might want to do with your type. You'll make fatter, uglier assemblies that are hard to grasp. Provide good primitives with understandable limitations. A great example of this is the urge people get to duplicate functionality that is already easy to use via Interop to native. Interop is there for a reason—it's not an unwanted stepchild. When wrapping anything, be sure you are adding plenty of value. Otherwise, the value added by not implementing the type and therefore being smaller would make your assembly more helpful to more people.

■ **JEFFREY RICHTER** I agree with this guideline, but I'd like to further add that the more advanced classes should be in a namespace that is under the namespace that contains the simple types. For example, the simple types might be in `System.Mail`, and the more advanced types should be in `System.Mail.Advanced`.

X DO NOT define types without specifying their namespaces.

This practice organizes related types in a hierarchy and can help resolve potential type name collisions. Of course, the fact that namespaces can help resolve name collisions does *not* mean that such collisions should be introduced. See section 3.4.1 for details.

■ **BRAD ABRAMS** It is important to realize that namespaces cannot actually prevent naming collisions; however, they can significantly reduce them. I could define a class called `MyNamespace.MyType` in an assembly called `MyAssembly` and define a class with precisely the same name in another assembly. I could then build an application that uses both of these assemblies and types. The CLR would not get confused because the type identity in the CLR is based on the strong name (which includes the fully qualified assembly name) rather than just the namespace and type name. This fact can be seen by looking at the C# and ILASM of code creating an instance of `MyType`.

```
C#:  
new MyType();
```

```
IL:  
IL_0000: newobj instance void [MyAssembly]MyNamespace.MyType::ctor()
```

Notice that the C# compiler adds a reference to the assembly that defines the type, of the form `[MyAssembly]`, so the runtime always has a disambiguated, fully qualified name to work with.

■ **RICO MARIANI** Namespaces are a language thing. The CLR doesn't know anything about them, really. As far as the CLR is concerned, the name of the class really is something like `MyNameSpace.MyOtherNameSpace.MyAmazingType`. The compilers give you syntax (e.g., "using") so that you don't have to type those long class names all the time. So the CLR is never confused about class names because everything is always fully qualified.

4.2 Choosing Between Class and Struct

One of the basic design decisions every framework designer faces is whether to design a type as a class (a reference type) or as a struct (a value type). Good understanding of the differences in the behavior of reference types and value types is crucial in making this choice.

The first difference between reference types and value types we will consider is that reference types are allocated on the heap and garbage-collected, whereas value types are allocated either on the stack or inline in containing types and deallocated when the stack unwinds or when their containing type gets deallocated. Therefore, allocations and deallocations of value types are generally cheaper than allocations and deallocations of reference types.

Next, arrays of reference types are allocated out-of-line, meaning the array elements are just references to instances of the reference type residing on the heap. Value type arrays are allocated inline, meaning that the array elements are the actual instances of the value type. Therefore, allocations and deallocations of value type arrays are much cheaper than allocations and deallocations of reference type arrays. In addition, in a majority of cases value type arrays exhibit much better locality of reference.

■ **RICO MARIANI** The preceding is often true, but it's a very broad generalization that I would be very careful about. Whether or not you get better locality of reference when value types get boxed when cast to an array of value types will depend on how much of the value type you use, how much searching you have to do, how much data reuse there could have been with equivalent array members (sharing a pointer), the typical array access patterns, and probably other factors I can't think of at the moment. Your mileage might vary, but value type arrays are a great tool for your toolbox.

■ **CHRIS SELLS** I find the restrictions of value types to be painful and therefore prefer reference types. Custom value types are often used for performance improvements, so I would recommend profiling large-scale anticipated usage of your library and changing reference types to value types based on actual data instead of on some anticipated problem that may never manifest itself in real-world conditions.

The next difference is related to memory usage. Value types get boxed when they are cast to a reference type or one of the interfaces they implement. They get unboxed when they are cast back to the value type. Because boxes are objects that are allocated on the heap and are garbage-collected, too much boxing and unboxing can have a negative impact on the heap, the garbage collector, and ultimately the performance of the application. In contrast, no such boxing occurs when reference types are cast.

Next, reference type assignments copy the reference, whereas value type assignments copy the entire value. Therefore, assignments of large reference types are cheaper than assignments of large value types.

Finally, reference types are passed by reference, whereas value types default to being passed by value. Changes to an instance of a reference type affect all references pointing to that instance. Value type instances are copied when they are passed by value. When an instance of a value type is changed, it does not affect any of its copies. Because the copies are not created explicitly by the user but rather are created implicitly when arguments are passed or return values are returned, value types that can be changed can be confusing to many users. Therefore, value types should be immutable.¹

■ **RICO MARIANI** If you make your value type mutable, you will find that you end up having to pass it by reference a lot to get the semantics you want (using, for example, “out” syntax in C#). This might be important in cases in which the value type is expected to be embedded in a variety of other objects that are themselves reference types or embedded in arrays. The biggest trouble from having mutable value types arises when they look like independent entities such as complex numbers. Value types that have a mission in life of being an accumulator of sorts or a piece of a reference type have fewer pitfalls for mutability.

1. Immutable types are types that don't have any public members that can modify this instance. For example, `System.String` is immutable. Its members, such as `ToUpper`, do not modify the string on which they are called, but rather return a new modified string instead and leave the original string unchanged.

The downside to this technique is that you don't get garbage collection on elements in your array. Thus, this technique should be used only when that is not an issue (e.g., when you have a large read-only array of small structures).

As a rule of thumb, the majority of types in a framework should be classes. There are, however, some situations in which the characteristics of a value type make it more appropriate to use structs.

✓ **CONSIDER** defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects, especially arrays.

✗ **AVOID** defining a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (`int`, `double`, etc.).
- It has an instance size less than 24 bytes.
- It is immutable.
- It will not have to be boxed frequently.

In all other cases, you should define your types as classes.

■ **JEFFREY RICHTER** In my opinion, a value type should be defined for types with a size of approximately 16 bytes or less. Value types can be more than 16 bytes if you don't intend to pass them to other methods or copy them to and from a collection class (like an array). I would also define a value type if you expect instances of the type to be used for short periods of time (usually they are created in a method and no longer needed after a method returns). I used to discourage defining value types if you thought that instances of them would be placed in a collection due to all the boxing that would have to be done. But, fortunately, newer versions of the CLR, C#, and other languages support generics so that boxing is no longer necessary when putting value type instances in a collection.

■ **JOE DUFFY** I frequently struggle with the reference versus value type decision. Although these rules are very black-and-white, there is a fairly large cliff you jump off when you decide to implement a reference type. This decision means you will pay the cost of heap allocation for each instance—which can impact scalability on multiprocessor machines due to the shared heap and cost of collections—in addition to at least a pointer of overhead (due to the object header) and a level of indirection to each access (due to the need to go through an object reference). That said, my experience has shown that whenever I attempt to be clever and break any of these rules, it usually comes back to bite me.

■ **JEREMY BARTON** Whether a struct or a class is better for performance depends on usage characteristics, which themselves depend on the split between your library code and the calling code from your user. When a struct is passed into a method by value, the details of memory layout and expense differ by runtime OS and CPU architecture—so a struct with three `Int64` fields isn't always the same as a struct with six `Int32` fields. Since the usage characteristics cannot be readily known, the size guideline is a general approximation.

Private types have the flexibility of using performance analysis data to change from a struct to a class, or vice versa, from build to build. But for public types, you have to make a decision and stick with it.

4.3 Choosing Between Class and Interface

In general, classes are the preferred construct for exposing abstractions.

The main drawback of interfaces is that they are much less flexible than classes when it comes to allowing for evolution of APIs. After you ship an interface, the set of its members is largely fixed forever. Any additions to the interface would break existing types that implement the interface.

■ **JEREMY BARTON** C# 8.0 added a feature, Default Interface Methods, which allows creating new methods on an interface without necessarily resulting in compile failures in downstream assemblies. While this feature can be useful for adding a simplified overload for an existing method, it usually can't introduce a new concept without a compile-time failure or a runtime exception. Because new concepts can't be introduced to an interface and Just Work, we still consider the interface to be effectively fixed once it ships.

There isn't any guidance for Default Interface Methods in the third edition because we haven't yet learned where things really go wrong. The best proto-guideline we have is "DO NOT use Default Interface Methods to provide an implementation for a member from a base interface," so as to avoid the "diamond problem."

A class offers much more flexibility. You can add members to classes that have already shipped. As long as the method is not abstract (i.e., as long as you provide a default implementation of the method), any existing derived classes continue to function unchanged.

Let's illustrate the concept with a real example from .NET. The `System.IO.Stream` abstract class shipped in .NET Framework 1.0 without any support for timing out pending I/O operations. In version 2.0, several members were added to `Stream` to allow subclasses to support timeout-related operations, even when accessed through their base class APIs.

```
public abstract class Stream {  
    public virtual bool CanTimeout {  
        get { return false; }  
    }  
    public virtual int ReadTimeout{  
        get {  
            throw new InvalidOperationException(...);  
        }  
        set {  
            throw new InvalidOperationException(...);  
        }  
    }  
  
    public class FileStream : Stream {  
        public override bool CanTimeout {
```

```
        get { return true; }
    }
    public override int ReadTimeout{
        get {
            ...
        }
        set {
            ...
        }
    }
}
```

The only general way to evolve interface-based APIs without runtime or compile-time errors is to add a new interface with the additional members. This might seem like a good option, but it suffers from several problems. Let's illustrate this with a hypothetical `IStream` interface. Let's assume we had shipped the following APIs in .NET Framework 1.0.

```
public interface IStream {
    ...
}

public class FileStream : IStream {
    ...
}
```

If we wanted to add support for timeouts to streams in version 2.0, we would have to do something like the following:

```
public interface ITimeoutEnabledStream : IStream {
    int ReadTimeout{ get; set; }
}

public class FileStream : ITimeoutEnabledStream {
    public int ReadTimeout{
        get{
            ...
        }
        set {
            ...
        }
    }
}
```

But now we would have a problem with all the existing APIs that consume and return `IStream`. For example, `StreamReader` has several constructor overloads and a property typed as `Stream`.

```
public class StreamReader {
    public StreamReader(IStream stream){ ... }
    public IStream BaseStream { get { ... } }
}
```

How would we add support for `ITimeoutEnabledStream` to `StreamReader`? We would have several options, each with substantial development cost and usability issues:

- Leave the `StreamReader` as is, and ask users who want to access the timeout-related APIs on the instance returned from `BaseStream` property to use a dynamic cast and query for the `ITimeoutEnabledStream` interface.

```
StreamReader reader = GetSomeReader();
var stream = reader.BaseStream as ITimeoutEnabledStream;
if(stream != null){
    stream.ReadTimeout = 100;
}
```

Unfortunately, this option does not perform well in usability studies. The fact that some streams can now support the new operations is not immediately apparent to the users of `StreamReader` APIs. Also, some developers have difficulty understanding and using dynamic casts.

- Add a new property to `StreamReader` that would return `ITimeoutEnabledStream` if one is passed to the constructor or `null` if `IStream` is passed.

```
StreamReader reader = GetSomeReader();
var stream = reader.TimeoutEnabledBaseStream;
if(stream != null){
    stream.ReadTimeout = 100;
}
```

Such APIs are only marginally better in terms of usability. It's really not obvious to the user that the `TimeoutEnabledBaseStream` property getter might return `null`, which results in confusing and often unexpected `NullReferenceExceptions`.

■ **JEREMY BARTON** The C# 8.0 Nullable Reference Types feature allows library authors to indicate that an API can (or won't) return a null value, which would seem to negate the surprising `NullReferenceException` scenario. But, if a developer has never seen `TimeoutEnabledBaseStream` return `null` the developer might decide the metadata is incorrect, only to have their code fail when interacting with a new `IStream` type.

We've learned that developers frequently exhibit the human characteristic of trusting their personal experience more than documentation.

- Add a new type called `TimeoutEnabledStreamReader` that would take `ITimeoutEnabledStream` parameters to the constructor overloads and return `ITimeoutEnabledStream` from the `BaseStream` property.

The problem with this approach is that every additional type in the framework adds complexity for the users. What's worse, the solution usually creates more problems like the one it is trying to solve. `StreamReader` itself is used in other APIs. These other APIs will now need new versions that can operate on the new `TimeoutEnabledStreamReader`.

The .NET streaming APIs are based on an abstract class. This allowed for an addition of timeout functionality in version 2.0 of the Framework. The addition is straightforward, is discoverable, and had little impact on other parts of the framework.

```
StreamReader reader = GetSomeReader();
if(reader.BaseStream.CanTimeout){
    reader.BaseStream.ReadTimeout = 100;
}
```

One of the most common arguments in favor of interfaces is that they allow for separating the contract from the implementation. However, this argument incorrectly assumes that you cannot separate contracts from implementation using classes. Abstract classes residing in a separate

assembly from their concrete implementations are a great way to achieve such separation. For example, the contract of `IList<T>` says that when an item is added to a collection, the `Count` property is incremented by one. Such a simple contract can be expressed—and, more importantly, locked—for all subtypes, using the following abstract class:

```
public abstract class CollectionContract<T> : IList<T> {  
  
    public void Add(T item){  
        AddCore(item);  
        _count++;  
    }  
    public int Count {  
        get { return _count; }  
    }  
    protected abstract void AddCore(T item);  
    private int _count;  
}
```

■ **KRZYSZTOF CWALINA** I often hear people saying that interfaces specify contracts. I believe this is a dangerous myth. Interfaces, by themselves, do not specify much beyond the syntax required to use an object. The interface-as-contract myth causes people to do the wrong thing when trying to separate contracts from implementation, which is indeed a great engineering practice. Interfaces separate syntax from implementation, which is not that useful, and the myth provides a false sense of doing the right engineering. In reality, the contract is both syntax and semantics, and these can be better expressed with abstract classes.

COM exposed APIs exclusively through interfaces, but you should not assume that COM did this because interfaces were superior. COM did it because COM is an interface standard that was intended to be supported on many execution environments. CLR is an execution standard, and it provides a great benefit for libraries that rely on portable implementation.

✓ **DO** favor defining classes over interfaces.

Class-based APIs can be evolved with much greater ease than interface-based APIs because it is possible to add members to a class without breaking existing code.

■ **KRZYSZTOF CWALINA** I have talked about this guideline with quite a few developers on our team. Many of them, including those who initially disagreed with the guideline, have said that they regret having shipped some API as an interface. I have not heard of even one case in which somebody regretted that they shipped a class.

■ **JEFFREY RICHTER** I agree with Krzysztof in general. However, you do need to think about some other things. There are some special base classes, such as `MarshalByRefObject`. If your library type provides an abstract base class that isn't itself derived from `MarshalByRefObject`, then types that derive from your abstract base class cannot live in a different `AppDomain`.

✓ **DO** use abstract classes instead of interfaces to decouple the contract from implementations.

Abstract classes, if designed correctly, allow for the same degree of decoupling between contract and implementation.

■ **CHRIS ANDERSON** Here is one instance in which the design guideline, if followed too strictly, can paint you into a corner. Abstract types do version much better, and allow for future extensibility, but they also burn your one and only one base type. Interfaces are appropriate when you are really defining a contract between two objects that is invariant over time. Abstract base types are better for defining a common base for a family of types. When we did .NET, there was somewhat of a backlash against the complexity and strictness of COM—interfaces, GUIDs, variants, and IDL were all seen as bad things. I believe today that we have a more balanced view of this. All of these COM-isms have their place, and in fact you can see interfaces coming back as a core concept in WCF.

■ **BRIAN PEPIN** One thing I've started doing is to actually bake as much contract into my abstract class as possible. For example, I might want to have four overloads to a method, where each overload offers an increasingly complex set of parameters. The best way to do this is to provide a nonvirtual implementation of these methods on the abstract class and have the implementations all route to a protected abstract method that provides the actual implementation. By doing this, you can write all the boring argument-checking logic once. Developers who want to implement your class will thank you.

■ **JEREMY BARTON** What Brian is describing here is the Template Method Pattern, discussed more in section 9.9.

- ✓ **DO** define an interface if you need to provide a polymorphic hierarchy of value types.

Value types cannot inherit from other types, but they can implement interfaces. For example, `IComparable`, `IFormattable`, and `IConvertible` are all interfaces, so value types such as `Int32`, `Int64`, and other primitives can all be comparable, formattable, and convertible.

```
public struct Int32 : IComparable, IFormattable, IConvertible {  
    ...  
}  
public struct Int64 : IComparable, IFormattable, IConvertible {  
    ...  
}
```

■ **RICO MARIANI** Good interface candidates often have this "mix in" feel to them. All sorts of objects can be `IFormattable`—it isn't restricted to a certain subtype. It's more like a type attribute. Other times we have interfaces that look more like they should be classes—`IFormatProvider` springs to mind. The fact that the interface's name ended in "er" speaks volumes.

■ **BRIAN PEPIN** Another sign that you've got a well-defined interface is that the interface does exactly one thing. If you have an interface that has a grab bag of functionality, that's a warning sign. You'll end up regretting it because in the next version of your product you'll want to add new functionality to this rich interface, but you can't.

- ✓ **CONSIDER** defining interfaces to achieve a similar effect to that of multiple inheritance.

For example, `System.IDisposable` and `System.ICloneable` are both interfaces, so types, like `System.Drawing.Image`, can be both disposable and cloneable yet still inherit from the `System.MarshalByRefObject` class.

```
public class Image : MarshalByRefObject, IDisposable, ICloneable {  
    ...  
}
```

■ **JEFFREY RICHTER** When a class is derived from a base class, I say that the derived class has an “is a” relationship with the base. For example, a `FileStream` “is a” `Stream`. However, when a class implements an interface, I say that the implementing class has a “can-do” relationship with the interface. For example, a `FileStream` “can-do” disposing because it implements the `IDisposable`.

■ **JEREMY BARTON** One hint that Jeffrey’s “can-do” paradigm is being followed is when the interface ends in the suffix “able.” For example, `IDisposable` can be loosely rewritten “a thing that can do `Dispose`.”

If we compare this to `ICryptoTransform`, that interface isn’t describing a capability; it’s instead defining a set of related actions. When we added `System.Span<T>` in .NET Core 2.0, we found that we couldn’t add Span support to symmetric cryptography because we used an interface instead of an abstract base class. It may have taken 15 years to realize that it was a mistake to have that type as an interface, but it *was* nonetheless a mistake.

4.4 Abstract Class Design

- ✗ **DO NOT** define public or protected internal constructors in abstract types.

Constructors should be public only if users will need to create instances of the type. Because you cannot create instances of an abstract type, an

abstract type with a public constructor is incorrectly designed and misleading to the users.²

```
// bad design
public abstract class Claim {
    public Claim() {
    }
}

// good design
public abstract class Claim {
    protected Claim() {
    }
}
```

✓ **DO** define a protected or an internal constructor in abstract classes.

A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created.

```
public abstract class Claim {
    protected Claim() {
        ...
    }
}
```

An internal constructor can be used to limit concrete implementations of the abstract class to the assembly defining the class.

```
public abstract class Claim {
    internal Claim() {
        ...
    }
}
```

■ BRAD ABRAMS Many languages (such as C#) will insert a protected constructor if you do not. It is a good practice to define the constructor explicitly in the source so that it can be more easily documented and maintained over time.

2. This also applies to protected internal constructors.

- ✓ **DO** provide at least one concrete type that inherits from each abstract class that you ship.

Doing this helps to validate the design of the abstract class. For example, `System.IO.FileStream` is an implementation of the `System.IO.Stream` abstract class.

■ **BRAD ABRAMS** I have seen countless examples of a “well-designed” base class or interface where the designers spent hundreds of hours debating and tweaking the design, only to have it be blown out of the water when the first real-world client came to use the design. Far too often these real-world clients come too late in the product cycle to allow time for the correct fix. Forcing yourself to provide at least one concrete implementation reduces the chances of finding a new problem late in the product cycle.

■ **CHRIS SELLS** My rule of thumb for testing a type or set of types is to have three people write three apps against it based on scenarios you hope to support. If the client code is pretty in all three cases, you’ve done a good job. Otherwise, you should consider refactoring until pretty happens.

4.5 Static Class Design

A static class is defined as a class that contains only static members (of course, besides the instance members inherited from `System.Object` and possibly a private constructor). Some languages provide built-in support for static classes. In C# 2.0 and later, when a class is declared to be static, it is sealed, abstract, and no instance members can be overridden or declared.

```
public static class File {  
    ...  
}
```

If your language does not have built-in support for static classes, you can declare such classes manually, as shown in the following C++ example:

```
public class File abstract sealed {  
    ...  
}
```

Static classes are a compromise between pure object-oriented design and simplicity. They are commonly used to provide shortcuts to other operations (such as `System.IO.File`), holders of extension methods, or functionality for which a full object-oriented wrapper is unwarranted (such as `System.Environment`).

✓ **DO** use static classes sparingly.

Static classes should be used only as supporting classes for the object-oriented core of the framework.

✗ **DO NOT** treat static classes as a miscellaneous bucket.

There should be a clear charter for each class. When your description of the class involves “and” or a new sentence, you need another class.

✗ **DO NOT** declare or override instance members in static classes.

This is enforced by the C# compiler.

✓ **DO** declare static classes as sealed, abstract, and add a private instance constructor if your programming language does not have built-in support for static classes.

■ **BRIAN GRUNKEMEYER** In .NET Framework 1.0, I wrote the code for the `System.Environment` class, which is an excellent example of a static class. I messed up and accidentally added a property to this class that wasn't static (`HasShutdownStarted`). Because it was an instance method on a class that could never be instantiated, no one could call it. We didn't discover the problem early enough in the product cycle to fix it before releasing version 1.0.

If I were inventing a new language, I would explicitly add the concept of a static class into the language to help people avoid falling into this trap. And in fact, C# 2.0 did add support for static classes!

■ **JEFFREY RICHTER** Make sure that you do not attempt to define a static structure, because structures (value types) can always be instantiated, no matter what. Only classes can be static.

4.6 Interface Design

Although most APIs are best modeled using classes and structs, there are some cases in which interfaces are more appropriate or are the only option.

The CLR does not support multiple inheritance (i.e., CLR classes cannot inherit from more than one base class), but it does allow types to implement one or more interfaces in addition to inheriting from a base class. Therefore, interfaces are often used to achieve the effect of multiple inheritance. For example, `IDisposable` is an interface that allows types to support disposability independent of any other inheritance hierarchy in which they want to participate.

```
public class Component : MarshalByRefObject, IDisposable, IComponent {  
    ...  
}
```

The other situation in which defining an interface is appropriate is in creating a common interface that can be supported by several types, including some value types. Value types cannot inherit from types other than `System.ValueType`, but they can implement interfaces, so using an interface is the only option to provide a common base type.

```
public struct Boolean : IComparable {  
    ...  
}  
public class String: IComparable {  
    ...  
}
```

✓ **DO** define an interface if you need some common API to be supported by a set of types that includes value types.

✓ **CONSIDER** defining an interface if you need to support its functionality on types that already inherit from some other type.

✗ **AVOID** using marker interfaces (interfaces with no members).

If you need to mark a class as having a specific characteristic (marker), in general, use a custom attribute rather than an interface.

```
// Avoid
public interface IImmutable {} // empty interface
public class Key: IImmutable {
    ...
}
// Consider
[Immutable]
public class Key {
    ...
}
```

Methods can be implemented to reject parameters that are not marked with a specific attribute, as follows:

```
public void Add(Key key, object value){
    if(!key.GetType().IsDefined(typeof(ImmutableAttribute), false)){
        throw new ArgumentException("The argument must be declared
[Immutable]", "key");
    }
    ...
}
```

■ **RICO MARIANI** Of course, any kind of marking like this has a cost. Attribute testing is a lot more costly than type checking. You might find that it's necessary to use the marker interface approach for performance reasons—measure and see. My own experience is that true markers (with no members) don't come up very often. Most of the time, you need a no-kidding-around interface with actual functionality to do the job, in which case there is no choice to make.

The problem with this approach is that the check for the custom attribute can occur only at runtime. Sometimes it is very important that the check for the marker be done at compile-time. For example, a method that can serialize objects of any type might be more concerned with verifying the presence of the marker than with type verification at compile-time. Using marker interfaces might be acceptable in such situations. The following example illustrates this design approach:

```
public interface ITextSerializable {} // empty interface
public void Serialize(ITextSerializable item){
    // use reflection to serialize all public properties
    ...
}
```

✓ **DO** provide at least one type that is an implementation of an interface.

Doing this helps to validate the design of the interface. For example, `System.Collections.Generic.List<T>` is an implementation of the `System.Collections.Generic.IList<T>` interface.

✓ **DO** provide at least one API that consumes each interface you define (a method taking the interface as a parameter or a property typed as the interface).

Doing this helps to validate the interface design. For example, `List<T>.Sort` consumes the `IComparer<T>` interface.

✗ **DO NOT** add members to an interface that has previously shipped.

Doing so would break implementations of the interface. You should create a new interface to avoid versioning problems.

Except for the situations described in these guidelines, you should, in general, choose classes rather than interfaces in designing managed code reusable libraries.

4.7 Struct Design

The general-purpose value type is most often referred to as a struct, its C# keyword. This section provides guidelines for general struct design.

Section 4.8 presents guidelines for the design of a special case of value type, the enum.

X DO NOT provide a default constructor for a struct.

Many CLR languages do not allow developers to define default constructors on value types. Users of these languages are often surprised to learn that `default(SomeStruct)` and `new SomeStruct()` don't necessarily produce the same value. Even if your language does allow defining a default constructor on a value type, it probably isn't worth the confusion it would cause if you did it.

X DO NOT define mutable value types.

Mutable value types have several problems. For example, when a property getter returns a value type, the caller receives a copy. Because the copy is created implicitly, developers might not be aware that they are mutating the copy, not the original value. Also, some languages (dynamic languages, in particular) have problems using mutable value types because even local variables, when dereferenced, cause a copy to be made.

```
// bad design
public struct ZipCode {
    public int FiveDigitCode { get; set; } // get/set properties
    public int PlusFourExtension { get; set; }
}

// good design
public struct ZipCode {
    public ZipCode(int fiveDigitCode, int plusFourExtension){...}
    public ZipCode(int fiveDigitCode):this(fiveDigitCode,0){}

    public int FiveDigitCode { get; } // get-only properties
    public int PlusFourExtension { get; }
}
```

✓ DO declare immutable value types with the `readonly` modifier.

Newer compiler understand the `readonly` modifier on a value type and avoid making extra value copies on operations such as invoking a method on a field declared with the `readonly` modifier.

```

public readonly struct ZipCode {
    public ZipCode(int fiveDigitCode, int plusFourExtension){...}
    public ZipCode(int fiveDigitCode):this(fiveDigitCode,0){}

    public int FiveDigitCode { get; } // get-only properties
    public int PlusFourExtension { get; }

    public override string ToString() {
        ...
    }
}

public partial class Other {
    private readonly ZipCode _zipCode;

    ...

    private void Work() {
        // Because ZipCode is declared as "readonly struct" the
        // ToString() method call does not involve a defensive copy
        string zip = _zipCode.ToString();
        ...
    }
}

```

■ **JAN KOTAS** Be careful when marking an existing struct as `readonly`. Existing structs that appear to be immutable may be mutable in very subtle ways. We learned this lesson hard way when `Nullable<T>` was marked as `readonly` in .NET Core. We quickly found that it broke existing code that depended on the `GetHashCode` and `Equals` methods mutating the `T` for caching, and the change was rolled back.

✓ **DO** declare nonmutating methods on mutable value types with the `readonly` modifier.

For better or worse, there are public mutable value types in .NET. As mentioned in the guidance against having mutable value types, when a mutable value type is stored in a field declared with the `readonly` modifier, any method or property invocation is performed on a copy of the value. Like the `readonly` modifier at the type level, the `readonly` modifier on a method allows the compiler to skip copying the value before invoking the method.

When the type is declared with the `readonly` modifier, there is no additional benefit from specifying the modifier on each method.

The C# compiler automatically applies the `readonly` modifier to the `get` method of a property declared using the auto-implemented property syntax.

```
// Using the "bad design" mutable version of ZipCode
public struct ZipCode {
    private int _plusFour;

    // This get method is implicitly readonly
    public int FiveDigitCode { get; set; }

    // This one needs it explicitly
    public int PlusFourExtension {
        readonly get => _plusFour;

        set {
            if (value > 9999) {
                value = -1;
            }

            _plusFour = value;
        }
    }

    // Any methods also need the readonly modifier
    // (if they don't mutate)
    public override readonly string ToString() { ... }
}
```

- ✓ **DO** ensure that a state where all instance data is set to zero, false, or null (as appropriate) is valid.

This prevents accidental creation of invalid instances when an array of the structs is created. For example, the following struct is incorrectly designed. The parameterized constructor is meant to ensure a valid state, but the constructor is not executed when an array of the struct is created. Thus, the instance field `value` gets initialized to 0, which is not a valid value for this type.

```
// bad design
public struct PositiveInteger {
```

```

private int value;

public PositiveInteger(int value) {
    if (value <= 0) throw new ArgumentException(...);
    _value = value;
}

public override string ToString() {
    return _value.ToString();
}
}

```

The problem can be fixed by ensuring that the default state (in this case, the value field equal to 0) is a valid logical state for the type.

```

// good design
public struct PositiveInteger {
    private int _value; // the logical value is value+1

    public PositiveInteger(int value) {
        if (value <= 0) throw new ArgumentException(...);
        _value = value-1;
    }

    public override string ToString() {
        return (_value+1).ToString();
    }
}

```

X DO NOT define ref-like value types (`ref struct` types), other than for specialized low-level purposes where performance is critical.

A `ref struct` type is an advanced concept that comes with several restrictions. Values from a `ref struct` type are allowed to exist only on the stack, and can never be boxed into the heap. Consequently, a `ref struct` type cannot be used as the type for a field in another type, except for other `ref struct` types, and cannot be used in asynchronous methods generated with the `async` keyword.

These usability limitations are a source of confusion and frustration for less advanced developers, and should generally be avoided.

■ **JEREMY BARTON** I've found myself saying, "Yeah, but `ref structs` break all the rules," when discussing certain inconsistencies in the guidelines in this book compared to some features from .NET Core 2.0 (when `ref structs` were first introduced) and beyond. This doesn't mean that the `ref`-like value types have a free license to ignore the guidance, but that the restrictions placed on them can sometimes force a compromise between strict compliance and the target scenario.

The usability concerns and occasional inconsistency with the rest of the platform are why `ref structs` have a "DO NOT" guidance, and almost every one of them involves long discussion in API review.

✓ **DO** implement `IEquatable<T>` on value types.

The `Object.Equals` method on value types causes boxing, and its default implementation is not very efficient, because it uses reflection. `IEquatable<T>.Equals` can have much better performance and can be implemented so that it will not cause boxing. See section 8.6 for guidelines on implementing `IEquatable<T>`.

✗ **DO NOT** explicitly extend `System.ValueType`. In fact, most languages prevent this.

In general, structs can be very useful, but they should be used only for small, single, immutable values that will not be boxed frequently.

4.8 Enum Design

Enums are a special kind of value type. There are two kinds of enums: simple enums and flag enums.

Simple enums represent small closed sets of choices. A common example of the simple enum is a set of colors such as the following:

```
public enum Color {
    Red,
    Green,
    Blue,
    ...
}
```

Flag enums are designed to support bitwise operations on the enum values. A common example of the flags enum is a list of options like the following:

```
[Flags]
public enum AttributeTargets {
    Assembly = 0x0001,
    Module   = 0x0002,
    Class    = 0x0004,
    Struct   = 0x0008,
    ...
}
```

■ **BRAD ABRAMS** We had some debates about what to call enums that are designed to be bitwise OR-ed together. We considered bitfields, bitflags, and even bitmasks—but ultimately decided to use flag enums because the term is clear, simple, and approachable.

■ **STEVEN CLARKE** I'm sure that less experienced developers will be able to understand bitwise operations on flags. The real question, though, is whether they would expect to have to do this. Most of the APIs that I have run through the labs don't require them to perform such operations, so I have a feeling that they would have the same experience that we observed during a recent study—it's just not something that they are used to doing, so they might not even think about it.

Where it could get worse, I think, is that if less advanced developers don't realize they are working with a set of flags that can be combined with one another, they might just look at the list available and think that is all the functionality they can access. As we've seen in other studies, if an API makes it look to them as though a specific scenario or requirement isn't immediately possible, it's likely that they will change the requirement and do what does appear to be possible, rather than being motivated to spend time investigating what they need to do to achieve the original goal.

Historically, many APIs (e.g., Win32 APIs) represented sets of values using integer constants. Enums make such sets more strongly typed, thereby improving compile-time error checking, usability, and readability.

For example, use of enums allows development tools to know the possible choices for a property or a parameter.

- ✓ **DO** use an enum to strongly type parameters, properties, and return values that represent sets of values.
- ✓ **DO** favor using an enum instead of static constants.

IntelliSense provides support for specifying arguments to members with enum parameters. It does not have similar support for static constants.

```
// Avoid the following
public static class Color {
    public static int Red    = 0;
    public static int Green  = 1;
    public static int Blue   = 2;
    ...
}

// Favor the following
public enum Color {
    Red,
    Green,
    Blue,
    ...
}
```

■ **JEFFREY RICHTER** An enum is a structure with a set of static constants. The reason to follow this guideline is because you will get some additional compiler and reflection support if you define an enum versus manually defining a structure with static constants.

- ✗ **DO NOT** use an enum for open sets (such as the operating system version, names of your friends, etc.).

- ✗ **DO NOT** provide reserved enum values that are intended for future use.

You can always simply add values to the existing enum at a later stage. Section 4.8.2 provides more details on adding values to enums. Reserved values just pollute the set of real values and tend to lead to user errors.

```

public enum DeskType {
    Circular,
    Oblong,
    Rectangular,

    // the following two values should not be here
    ReservedForFutureUse1,
    ReservedForFutureUse2,
}

```

X AVOID publicly exposing enums with only one value.

A common practice for ensuring future extensibility of C APIs is to add reserved parameters to method signatures. Such reserved parameters can be expressed as enums with a single default value. This practice should not be followed in managed APIs. Method overloading allows adding parameters in future releases.

```

// bad design
public enum SomeOption {
    DefaultOption
    // we will add more options in the future
}

...
// The option parameter is not needed.
// It can always be added in the future
// to an overload of SomeMethod().
public void SomeMethod(SomeOption option) {
    ...
}

```

VANCE MORRISON While I agree that this example is not a good practice, I think it is OK to define enums with no values (or just one “sentinel” value). The key question is whether the type safety (not being allowed to “accidentally” use an int) is valuable. If you pass out int values as “handles” to your structure, passing them out as an enum rather than an int is a REALLY good thing because it prevents mistakes and makes the intent of the API clear (What can I do with this “handle” you returned to me?). When you do this, you may be defining enums with no values or maybe just one (a “null” value).

■ **JEREMY BARTON** I agree with Vance that using a strong type instead of `int` or `string` provides value for the case “I gave this to you, and need you to give this back to me,” but I would advise using a custom struct rather than an enum. A custom struct can evolve better (such as gaining extra fields, or being `IDisposable`).

■ **JOE DUFFY** Although having enums with one value is a bad idea, enums with just two values are far more common. If you have a Boolean parameter and suspect that the choice may have a third possible value sometime in the future, it’s usually a good idea to proactively add the parameter to a two-valued enum. This can help to avoid versioning problems down the road.

■ **BRAD ABRAMS** Like Joe, I believe an enum with just two values is a fine and common practice. I much prefer this to a Boolean argument. The main reason is for code readability. For example, consider:

```
FileStream f = File.Open ("foo.txt", true, false);
```

This call gives you no context whatsoever for understanding the meaning behind `true` and `false`. Now consider if the call were:

```
FileStream f = File.Open ("foo.txt", CasingOptions.CaseSensitive,  
                           FileMode.Open);
```

X DO NOT include sentinel values in enums.

Although they are sometimes helpful to framework developers, sentinel values are confusing to users of the framework. They are used to track the state of the enum rather than being one of the values from the set represented by the enum. The following example shows an enum with an additional sentinel value used to identify the last value of the enum, which is intended for use in range checks. This is bad practice in framework design.

```

public enum DeskType {
    Circular      = 1,
    Oblong        = 2,
    Rectangular   = 3,

    LastValue     = 3 // this sentinel value should not be here
}

public void OrderDesk(DeskType desk){
    if((desk > DeskType.LastValue){
        throw new ArgumentOutOfRangeException(...);
    }
    ...
}

```

Rather than relying on sentinel values, framework developers should perform the check using one of the real enum values.

```

public void OrderDesk(DeskType desk){
    if(desk > DeskType.Rectangular || desk < DeskType.Circular){
        throw new ArgumentOutOfRangeException(...);
    }
    ...
}

```

■ **RICO MARIANI** You can get yourself into a lot of trouble by trying to be too clever with enums. Sentinel values are a great example of this: People write code like the example shown, but they use the sentinel value `LastValue` instead of `Rectangular`, as recommended. When a new value comes along and `LastValue` is updated, their program “automatically” does the right thing and accepts the new input value without giving an `ArgumentOutOfRangeException`. That sounds grand except for all that we didn’t show—the part that’s doing the actual work and that might not yet expect or even handle the new value. By avoiding sentinel values you will be forced to revisit all the right places to ensure that the new value really is going to work. The few minutes you spend visiting those call sites will be more than repaid in time you save avoiding bugs.

✓ **DO** provide a value of zero on simple enums.

Consider calling the value something like “None.” If such a value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

```
public enum Compression {
    None = 0,
    GZip,
    Deflate,
}
public enum EventType {
    Error = 0,
    Warning,
    Information,
    ...
}
```

■ **JEREMY BARTON** Remember that class and struct fields are initialized to their zero-value by default. When it would be hard to describe a “default” state, or when it’s important that the caller express an intentional value, consider naming the zero value “Undefined” and treating it as an invalid value.

✓ **CONSIDER** using `Int32` (the default in most programming languages) as the underlying type of an enum unless any of the following is true:

- The enum is a flags enum and you have more than 32 flags, or expect to have more in the future.
- The underlying type needs to be different than `Int32` for easier interoperability with unmanaged code expecting different-size enums.

■ **RICO MARIANI** Did you know that the CLR supports enums with an underlying type of `float` or `double` even though most languages don’t choose to expose it? This is very handy for strongly typed constants that happen to be floating point (e.g., a set of canonical conversion factors for different measuring systems). It’s in the ECMA standard.

- A smaller underlying type would result in substantial savings in space. If you expect the enum to be used mainly as an argument for flow of control, the size makes little difference. The size savings might be significant if:
 - You expect the enum to be used as a field in a very frequently instantiated structure or class.
 - You expect users to create large arrays or collections of the enum instances.
 - You expect a large number of instances of the enum to be serialized.

For in-memory usage, be aware that managed objects are always DWORD-aligned. So, you effectively need multiple enums or other small structures in an instance to pack a smaller enum with in order to make a difference, because the total instance size will always be rounded up to a DWORD.

■ **BRAD ABRAMS** Keep in mind that it is a binary breaking change to change the size of the enum type once you have shipped, so choose wisely—with an eye toward the future. Our experience has been that Int32 is usually the right choice; thus we made it the default.

✓ **DO** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.

See section 3.5.3 for details.

✗ **DO NOT** extend System.Enum directly.

System.Enum is a special type used by the CLR to create user-defined enumerations. Most programming languages provide a programming element that gives you access to this functionality. For example, in C# the enum keyword is used to define an enumeration.

Simple enums works well for a closed set of alternatives. When multiple values need to be expressed concurrently, consider a flag enum (section 4.8.1). When the set of alternatives should be open to allow for derived types to add new capabilities, consider the strongly typed string approach discussed in section 4.11.

4.8.1 Designing Flag Enums

■ JEFFREY RICHTER I use flag enums quite frequently in my own programming. They are stored very efficiently in memory, and manipulation is very fast. In addition, they can be used with interlocked operations, making them ideal for solving thread synchronization problems. I'd love to see the `System.Enum` type offer a bunch of additional methods that could be easily inlined by the JIT compiler and that would make source code easier to read and maintain. Here are some of the methods I'd like to see added to `System.Enum`: `IsExactlyOneBitSet`, `CountOnBits`, `AreAllBitsOn`, `AreAnyBitsOn`, and `TurnBitsOnOff`.

- ✓ **DO** apply the `System.FlagsAttribute` to flag enums. Do not apply this attribute to simple enums.

```
[Flags]
public enum AttributeTargets {
    ...
}
```

- ✓ **DO** use powers of 2 for the flag enum values so they can be freely combined using the bitwise OR operation.

```
[Flags]
public enum WatcherChangeTypes {
    None = 0,
    Created = 0x0002,
    Deleted = 0x0004,
    Changed = 0x0008,
    Renamed = 0x0010,
}
```

■ JEREMY BARTON I often find it clearer, when writing, to use the left-shift operator when defining flag enum values to show I've set only one bit and I'm going in bit order.

```
[Flags]
public enum WatcherChangeTypes {
    None = 0,
    // No value is defined for 1 (0x1, 1 << 0)
    Created = 1 << 1,
    Deleted = 1 << 2,
    Changed = 1 << 3,
    Renamed = 1 << 4,
}
```

Since I find hexadecimal nicer when matching the values against file contents or memory views, I'll usually have my IDE do a value replacement before committing the code.

- ✓ **CONSIDER** providing special enum values for commonly used combinations of flags.

Bitwise operations are an advanced concept and should not be required for simple tasks. `FileAccess.ReadWrite` is an example of such a special value.

```
[Flags]
public enum FileAccess {
    Read = 1,
    Write = 2,
    ReadWrite = Read | Write
}
```

- ✗ **AVOID** creating flag enums where certain combinations of values are invalid.

The `System.Reflection.BindingFlags` enum is an example of an incorrect design of this kind. The enum tries to represent many different concepts, such as visibility, staticness, member kind, and so on.

```
[Flags]
public enum BindingFlags {
    Default = 0,
    Instance = 0x4,
    Static = 0x8,
    Public = 0x10,
    NonPublic = 0x20,
```

```

CreateInstance = 0x200,
GetField = 0x400,
SetField = 0x800,
GetProperty = 0x1000,
 SetProperty = 0x2000,
InvokeMethod = 0x100,
...
}

```

Certain combinations of the values are not valid. For example, the `Type.GetMembers` method accepts this enum as a parameter, but the documentation for the method warns users, “You must specify either `BindingFlags.Instance` or `BindingFlags.Static` in order to get a return.” Similar warnings apply to several other values of the enum.

If you have an enum with this problem, you should separate the values of the enum into two or more enums or other types. For example, the Reflection APIs could have been designed as follows:

```

[Flags]
public enum Visibilities {
    None = 0,
    Public = 0x10,
    NonPublic = 0x20,
}

[Flags]
public enum MemberScopes {
    None = 0,
    Instance = 0x4,
    Static = 0x8,
}

[Flags]
public enum MemberKinds {
    None = 0,
    Constructor = 1 << 0,
    Field = 1 << 1,
    PropertyGetter = 1 << 2,
    PropertySetter = 1 << 3,
    Method = 1 << 4,
}

public class Type {
    public MemberInfo[] GetMembers(MemberKinds members,
                                   Visibilities visibility,
                                   MemberScopes scope);
}

```

X AVOID using flag enum values of zero unless the value represents “all flags are cleared” and is named appropriately, as prescribed by the next guideline.

The following example shows a common implementation of a check that programmers use to determine if a flag is set (see the if-statement in the example). The check works as expected for all flag enum values except the value of zero, where the Boolean expression always evaluates to true.

```
[Flags]
public enum SomeFlag {
    ValueA = 0, // this might be confusing to users
    ValueB = 1,
    ValueC = 2,
    ValueBAndC = ValueB | ValueC,
}

SomeFlag flags = GetValue();
if ((flags & SomeFlag.ValueA) == SomeFlag.ValueA) {
    ...
}
```

■ ANDERS HEJLSBERG Note that in C# the literal constant 0 implicitly converts to any enum type, so you could just write:

```
if (Foo.SomeFlag == 0)...
```

We support this special conversion in order to provide programmers with a consistent way of writing the default value of an enum type, which by CLR decree is “all bits zero” for any value type.

✓ DO name the zero value of flag enums `None`. For a flag enum, the value must always mean “all flags are cleared.”

```
[Flags]
public enum BorderStyle {
    Fixed3D   = 0x1,
    FixedSingle = 0x2,
    None      = 0x0
}
if (foo.BorderStyle == BorderStyle.None)....
```

■ **VANCE MORRISON** Note that the zero value is special in that it is the default value that will be assigned if no other assignment is made. Thus, it has the potential to be the most common value. Your design should accommodate this situation. In particular, the zero value should be either of the following:

- A very good default value (when there is such a default)
- An error value that is checked by your APIs (when there is no good default value and you want to ensure that users set something)

4.8.2 Adding Values to Enums

It is very common to discover that you need to add values to an enum after you have already shipped it. A potential application compatibility problem arises when the newly added value is returned from an existing API, because poorly written applications might not handle the new value correctly. Documentation, samples, and code analysis tools encourage application developers to write robust code that can help applications deal with unexpected values. Therefore, it is generally acceptable to add values to enums, but—as with most guidelines—there might be exceptions to the rule based on the specifics of the framework.

✓ **CONSIDER** adding values to enums, despite a small compatibility risk.

If you have real data about application incompatibilities caused by additions to an enum, consider adding a new API that returns the new and old values, and deprecate the old API, which should continue returning just the old values. This will ensure that your existing applications remain compatible.

■ **CLEMENS SZYPERSKI** Adding a value to an enum presents a very real possibility of breaking a client. Before the addition of the new enum value, a client that was throwing unconditionally in the default case presumably never actually threw the exception, and the corresponding catch path is likely untested. Now that the new enum value can pop up, the client will throw an exception and likely fold.

The biggest concern with adding values to enums is that you don't know whether clients perform an exhaustive switch over an enum or a progressive case analysis across wider-spread code. Even with these FxCop rules in place, and even when it is assumed that client apps pass FxCop without warnings, we still would not know about code that performs things like `if (myEnum == someValue) ...` in various places.

Clients might instead perform pointwise case analyses across their code, resulting in fragility under enum versioning. It is important to provide specific guidelines to developers of enum client code, detailing what they need to do to survive the addition of new elements to enums they use. Developing with the suspected future versioning of an enum in mind is the required attitude.

■ **ANTHONY MOORE** Enum values that have a usage pattern of method or property return values should usually not be changed (even by addition) after being shipped because of usage in `switch` or `enum` statements. `DateTimeKind` is an example of this.

It is generally safe to add members to enums that are primarily used as input arguments, because code that has followed the argument validation guidelines will at worst throw an `ArgumentException` if it encounters the new value. `FileAccess` on `FileStream` is an example of this.

In general, it is safe to add members to flag enums. Compared to regular enums, it is much harder to write code against them that would be broken by the presence of new values.

4.9 Nested Types

A nested type is a type defined within the scope of another type, which is called the enclosing type. A nested type has access to all members of its enclosing type. For example, it has access to private fields defined in the

enclosing type and to protected fields defined in all descendants of the enclosing type.

```
// enclosing type
public class OuterType {
    private string _name;

    // nested type
    public class InnerType {
        public InnerType(OuterType outer){
            // the _name field is private, but it works just fine
            Console.WriteLine(outer._name);
        }
    }
}
```

In general, nested types should be used sparingly, for several reasons. Some developers are not fully familiar with the concept. These developers might, for example, have problems with the syntax of declaring variables of nested types. Nested types are also very tightly coupled with their enclosing types, and as such are not suited to be general-purpose types.

Nested types are best suited for modeling implementation details of their enclosing types. The end user should rarely have to declare variables of a nested type, and should almost never have to explicitly instantiate nested types. For example, the enumerator of a collection can be a nested type of that collection. Enumerators are usually instantiated by their enclosing type, and because many languages support the foreach statement, enumerator variables rarely have to be declared by the end user.

✓ DO use nested types when the relationship between the nested type and its outer type is such that member-accessibility semantics are desirable.

For example, the nested type needs to have access to private members of the outer type.

```
public OrderCollection : IEnumerable<Order> {
    private Order[] _data = ...;

    public IEnumerator<Order> GetEnumerator(){
        return new OrderEnumerator(this);
    }
}
```

```
}

// This nested type will have access to the data array
// of its outer type.
private class OrderEnumerator : IEnumerator<Order> {
}
}
```

X DO NOT use public nested types as a logical grouping construct; use namespaces for this purpose.

X AVOID publicly exposed nested types. The only exception to this guideline is if variables of the nested type need to be declared only in rare scenarios such as subclassing or other advanced customization scenarios.

■ **KRZYSZTOF CWALINA** The main motivation for this guideline is that many beginner developers don't understand why some type names have dots in them and some don't. As long as they don't have to type in the type name, they don't care. But the moment you ask them to declare a variable of a nested type, they get lost. Therefore, in general, we avoid nested types and use them only in places where developers almost never have to declare variables of that type (e.g., collection enumerators).

■ **JEREMY BARTON** The main scenario where we expose nested types in the Base Class Libraries is for struct-based enumerators (which also violate the guideline of not making mutable structs). We feel that these are generally OK because (a) they help with performance by avoiding heap-based allocation and interface dispatch and (b) they're almost always used in a foreach statement and most callers aren't aware that a struct enumerator was involved.

X DO NOT use nested types if the type is likely to be referenced outside of the containing type.

For example, an enum passed to a method defined on a class should not be defined as a nested type in the class.

X DO NOT use nested types if they need to be instantiated by client code. If a type has a public constructor, it probably should not be nested.

If a type can be instantiated, that seems to indicate the type has a place in the framework on its own (you can create it, work with it, and destroy it without ever using the outer type), and thus should not be nested. Inner types should not be widely reused outside of the outer type without any relationship whatsoever to the outer type.

X DO NOT define a nested type as a member of an interface. Many languages do not support such a construct.

In general, use nested types sparingly, and avoid their exposure as public types.

4.10 Types and Assembly Metadata

Types reside in assemblies, which in most cases are packaged in the form of DLLs or executables (EXEs). Several important attributes should be applied to assemblies that contain public types. This section describes guidelines related to these attributes.

✓ DO apply the `CLSCCompliant(true)` attribute to assemblies with public types.

```
[assembly:CLSCCompliant(true)]
```

The attribute is a declaration that the types contained in the assembly are CLS³ compliant and so can be used by all .NET languages.⁴ Some languages, such as C#, verify compliance⁵ with the standard if the attribute is applied.

-
3. The CLS standard is an interoperability agreement between framework developers and language developers as to what subset of the CLR type system can be used in framework APIs so that these APIs can be used by all CLS-compliant languages.
 4. All that support the CLS standard—and almost all of the CLR languages do.
 5. C# verifies compliance with most of the CLS rules. Some rules are not automatically verifiable. For example, the standard does not say that an assembly cannot have non-compliant APIs. Instead, it says only that noncompliant APIs must be marked with `CLSCCompliant(false)` and that a compliant alternative must exist. But, of course, the existence of an alternative to a noncompliant API cannot be verified automatically.

For example, unsigned integers are not in the CLS subset. Therefore, if you add an API that uses UInt32 (for example), C# will generate a compilation warning. To comply with the CLS standard, the assembly must provide a compliant alternative and explicitly declare the noncompliant API as `CLSCompliant(false)`.

```
public static class Console {
    [CLSCompliant(false)]
    public void Write(uint value); // not CLS compliant

    public void Write(long value); // CLS-compliant alternative
}
```

- ✓ **DO** apply `AssemblyVersionAttribute` to assemblies with public types.

```
[assembly: AssemblyVersion(...)]
```

- ✓ **DO** apply the following informational attributes to assemblies. These attributes are used by tools, such as Visual Studio, to inform the user of the assembly about its contents.

```
[assembly: AssemblyTitle("System.Core.dll")]
[assembly: AssemblyCompany("Microsoft Corporation")]
[assembly: AssemblyProduct("Microsoft .NET Framework")]
[assembly: AssemblyDescription(...)]
```

- ✓ **CONSIDER** applying `ComVisible(false)` to your assembly. COM-callable APIs need to be designed explicitly. As a rule of thumb, .NET assemblies should not be visible to COM. If you do design the APIs to be COM-callable, you can apply `ComVisible(true)` either to the individual APIs or to the whole assembly.

- ✓ **CONSIDER** applying `AssemblyFileVersionAttribute` and `AssemblyCopyrightAttribute` to provide additional information about the assembly.

- ✓ **CONSIDER** using the format `<V>.<S>..<R>` for the assembly file version, where V is the major version number, S is the servicing number, B is the build number, and R is the build revision number.

For example, this is how the version attribute is applied in `System.Core.dll`, which shipped as part of .NET Framework 3.5:

```
[assembly:AssemblyFileVersion("3.5.21022.8")]
```

4.11 Strongly Typed Strings

Enums generally provide compile-time assurance that an input option is valid.⁶ The IntelliSense-enhanced feature discovery that enums enable comes at a cost of extensibility: Derived types have no way of supporting additional options using the API provided by the base class.

The most extensible input types are `Object` and `String`, but both types suffer from a difficulty in callers' understanding of legal input values.

```
public partial class RSACryptoServiceProvider {
    // What are the legal input values for "halg"?
    //
    // It turns out, it's some string values, some Type values,
    // and some HashAlgorithm values ... but most string values,
    // most Type values, and about half of the built-in
    // HashAlgorithm types' values are invalid.
    public byte[] SignData(byte[] buffer, object halg) { ... }
}
```

Between the flexible (but hard for callers to reason about) string input type, and the inflexible (but easy for callers to reason about) enum-based input type is the “strongly typed string.” A strongly typed string is a string, wrapped in a value type, with static properties providing the most common values.

```
namespace System.Security.Cryptography {
    public readonly struct HashAlgorithmName :
        IEquatable<HashAlgorithmName> {

        public static HashAlgorithmName MD5 { get; } =
            new HashAlgorithmName("MD5");
```

6. A caller can sometimes specify a disallowed option, or end up in a situation where a library was not aware of a new enum value. But usually if an enum compiles, it's a valid value.

```
public static HashAlgorithmName SHA1 { get; } = ...;
public static HashAlgorithmName SHA256 { get; } = ...;
public static HashAlgorithmName SHA384 { get; } = ...;
public static HashAlgorithmName SHA512 { get; } = ...;

public HashAlgorithmName(string name) {
    // Note: No validation because we have to deal with
    // default(HashAlgorithmName) regardless.
    Name = name;
}

public string Name { get; }

public override string ToString() {
    return Name ?? String.Empty;
}

public override bool Equals(object obj) {
    return obj is HashAlgorithmName &&
        Equals((HashAlgorithmName)obj);
}

public bool Equals(HashAlgorithmName other) {
    // NOTE: Intentionally case-sensitive ordinal, matches OS.
    return Name == other.Name;
}

public override int GetHashCode() {
    return Name == null ? 0 : Name.GetHashCode();
}

public static bool operator ==(HashAlgorithmName left,
    HashAlgorithmName right) {
    return left.Equals(right);
}

public static bool operator !=(HashAlgorithmName left,
    HashAlgorithmName right) {
    return !(left == right);
}
```

- ✓ **CONSIDER** defining a strongly typed string value type when a base class supports a fixed set of inputs but a derived type could support more.

When a strongly typed string is used only by a sealed type hierarchy, there is little to no value in supporting values other than the predefined ones, and an enum is a more consistent type choice.

A strongly typed string provides the most benefit when most code treats it as if it were an enum, such as HTTP header names or cryptographic digest algorithms. When the number of possible values is too large, such as all possible filenames, the value in a strongly typed string is reduced to just parameter consistency via type checking.

A variation of strongly typed strings is used in .NET for the `JsonEncodedText` type, where the type indicates that the value has already been processed by a string escaping/normalization process. This is a valuable approach, but does not represent a strongly typed string.

- ✓ **DO** declare a strongly typed string as an immutable value type (`struct`) with a string constructor.

The strongly typed string type should follow other guidance for immutable value types, such as using the `readonly` modifier on the type and implementing `IEquatable<T>`.

- ✓ **DO** override `ToString()` on a strongly typed string to return the underlying string value.

- ✓ **CONSIDER** exposing the underlying string value from a strongly typed string in a get-only property.

`ToString()` should be overridden because the underlying string value provides an obvious “interesting human-readable string” to return, per the guidelines on overriding `ToString()`.

Since `ToString()` is used for debugging and display purposes, rather than runtime decisions, exposing the underlying string value from the property presents a better general pattern to callers when the value is required—such as interoperating with libraries that use the same string values without using the wrapping type.

If callers will only rarely, or never, need the underlying string value, then not having the property allows the type to look more like an enum to IntelliSense.

There is not a prescriptive guideline for the name of the property. If no obvious name presents itself, remember that “Value” is preferred to “String” (section 3.2.3).

For more information on overloading `ToString()`, see section 8.9.3.

✓ **DO** override equality operators for strongly typed string types.

Overriding the equality operators allows for a strongly typed string type to syntactically look like either a string or an enum in common code patterns.

Strongly typed strings should use ordinal string equality by default. However, when a specific domain represents case-insensitive content, the `IEquatable<T>` behavior can be based on other string comparisons.

✓ **DO** allow null inputs to the constructor of a strongly typed string.

Because value types can always be zero-initialized, the constructor of a strongly typed string should not disallow null inputs. This ensures that code logically equivalent to a copy constructor will function.

```
// This should always succeed.  
HashAlgorithmName newName = new HashAlgorithmName(cur.Name);
```

✓ **DO** declare known values for a strongly typed string via static get-only properties on the type.

The enum-like IntelliSense experience is a very strong motivator for strongly typed string types.

✓ **AVOID** creating overloads across `System.String` and a strongly typed string, unless the `System.String` overload was released in a previous version.

The existence of an overload that accepts a `System.String` instance instead of a strongly typed string can save a few characters at the call site for callers that have only the underlying string value available, but

it forces developers new to your API to understand the difference between the two methods.

If you have recently defined a strongly typed string type to help clarify valid inputs to a method, adding an overload that accepts the strongly typed string provides value. In that case, consider marking the original `System.String`-based overload as `[EditorBrowsable(EditorBrowsableState.Advanced)]`, `[EditorBrowsable(EditorBrowsableState.Never)]`, or `[Obsolete]`.

SUMMARY

This chapter presented guidelines that describe when and how to design classes, structs, and interfaces. The next chapter goes to the next level in type design—the design of members.

This page intentionally left blank

 5

Member Design

METHODS, PROPERTIES, EVENTS, constructors, and fields are collectively referred to as members. Members are ultimately the means by which framework functionality is exposed to the end users of a framework.

Members can be virtual or nonvirtual, concrete or abstract, static or instance, and can have several different scopes of accessibility. All this variety provides incredible expressiveness but at the same time requires care on the part of the framework designer.

This chapter offers basic guidelines that should be followed when designing members of any type. Chapter 6 spells out additional guidelines related to members that need to support extensibility.

5.1 General Member Design Guidelines

Most member design guidelines are specific to the kind of member being designed and are described later in this chapter. Nevertheless, some broad design conventions are applicable to different kinds of members. This section discusses these conventions.

5.1.1 Member Overloading

Member overloading means creating two or more members on the same type that differ only in the number or type of parameters but have the same name. For example, in the following, the `WriteLine` method is overloaded:

```
public static class Console {  
    public void WriteLine();  
    public void WriteLine(string value);  
    public void WriteLine(bool value);  
    ...  
}
```

Because only methods, constructors, and indexed properties can have parameters, only those members can be overloaded.

 **BRENT RECTOR** Technically, the CLR allows additional overloading. For example, multiple fields in the same scope can have the same name as long as the fields have differing types, and multiple methods in the same scope can have the same name, number, and type of parameters as long as the methods have differing return types. However, few languages above IL support such distinctions. If you created an API using these kinds of distinctions, the API wouldn't be usable from most programming languages. Therefore, it's a useful fiction to restrict discussion of overloading to methods, constructors, and indexed properties.

Overloading is one of the most important techniques for improving the usability, productivity, and readability of reusable libraries. Overloading on the number of parameters makes it possible to provide simpler versions of constructors and methods. Overloading on the parameter type makes it possible to use the same member name for members performing identical operations on a selected set of different types.

For example, `System.DateTime` has several constructor overloads. The most powerful—but also the most complex—one takes eight parameters. Thanks to constructor overloading, the type also supports a shortened

constructor that takes only three simple parameters: hours, minutes, and seconds.

```
public struct DateTime {
    public DateTime(int year, int month, int day,
                   int hour, int minute, int second,
                   int millisecond, Calendar calendar) { ... }

    public DateTime(int hour, int minute, int second) { ... }
}
```

RICO MARIANI This is one case where a simpler API for the programmer also results in better code—most calls to the library do not need the complicated arguments, so the bulk of the call sites are abbreviated. Even though there might be internal forwarding to the more complicated API, that forwarding code is shared, so overall there is significantly less code. Less code means more cache hits, which means faster programs.

This section does not cover the similarly named but quite different construct called operator overloading. Operator overloading is described in section 5.7.

✓ DO try to use descriptive parameter names to indicate the default used by shorter overloads.

In a family of members overloaded on the number of parameters, the longer overload should use parameter names that indicate the default value used by the corresponding shorter member. This is mostly applicable to Boolean parameters. For example, in the following code, the first short overload does a case-sensitive look-up. The second, longer overload adds a Boolean parameter that controls whether the look-up is case sensitive. The parameter is named `ignoreCase` rather than `caseSensitive` to indicate that the longer overload should be used to ignore case and that the shorter overload probably defaults to the opposite, a case-sensitive look-up.

```
public class Type {
    public MethodInfo GetMethod(string name); //ignoreCase = false
    public MethodInfo GetMethod(string name, Boolean ignoreCase);
}
```

X DO NOT arbitrarily vary parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name.

```
public class String {  
    // correct  
    public int IndexOf (string value) { ... }  
    public int IndexOf (string value, int startIndex) { ... }  
  
    // incorrect  
    public int IndexOf (string value) { ... }  
    public int IndexOf (string str, int startIndex) { ... }  
}
```

X AVOID being inconsistent in the ordering of parameters in overloaded members. Parameters with the same name should appear in the same position in all overloads.

```
public class EventLog {  
    public EventLog();  
    public EventLog(string logName);  
    public EventLog(string logName, string machineName);  
    public EventLog(string logName, string machineName, string source);  
}
```

In some specific cases, this otherwise very strict guideline can be broken. For example, a `params` array parameter has to be the last parameter in a parameter list. If a `params` array parameter appears in an overloaded member, the API designer might need to make a trade-off and either settle for inconsistent parameter ordering or not use the `params` modifier. For more information on the `params` array parameters, see section 5.8.4.

Another case where the guideline might need to be violated is when the parameter list contains `out` parameters. These parameters should typically appear at the end of the parameter list, which again means that the API designer might need to settle for a slightly inconsistent order of parameters in overloads with `out` parameters. See section 5.8 for more information on `out` parameters.

✓ **DO** make only the longest overload virtual (if extensibility is required).

Shorter overloads should simply call through to a longer overload.

```
public class Encoding {  
    public bool IsAlwaysNormalized(){  
        return IsAlwaysNormalized(NormalizationForm.FormC);  
    }  
    public virtual bool IsAlwaysNormalized(NormalizationForm form){  
        //do real work here  
    }  
}
```

For more information on the design of virtual members, see Chapter 6.

■ **BRIAN PEPIN** Remember that you can apply this pattern to abstract classes, too. In your abstract class, you can perform all necessary argument checking in nonabstract, nonvirtual methods and then provide a single abstract method for the developer to implement.

■ **CHRIS SELLS** I love this guideline, not only because it makes the library code easier to reason about, but also because it makes it easier for me to write. If I have to implement each overload from scratch instead of making the less complicated ones call the more complicated ones, then I have to test and maintain all that copied code, too, which I'm far too lazy to do properly.

Further, if I've got more than two overloads, I always make the less complicated ones call the next more complicated one and not the most complicated one, like the three overloads of `IsAlwaysNormalized` shown previously for `Encoding`. That way, I'm not hard-coding defaults in more than one place—if I had to do that, I'd probably get it wrong.

✗ **DO NOT** use `ref`, `out`, or `in` modifiers to overload members.

For example, you should not do the following:

```
public class SomeType {  
    public void SomeMethod(string name){ ... }  
    public void SomeMethod(out string name){ ... }  
}
```

Some languages cannot resolve calls to overloads like this. In addition, such overloads usually have completely different semantics and probably should not be overloads at all, but rather two separate methods.

- ✗ **DO NOT** have overloads with parameters at the same position and similar types, yet with different semantics.

The following example is a bit extreme, but it does illustrate the point:

```
public class SomeType {  
    public void Print(long value, string terminator){  
        Console.WriteLine("{0}{1}",value,terminator);  
    }  
    public void Print(int repetitions, string str){  
        for(int i=0; i< repetitions; i++){  
            Console.Write(str);  
        }  
    }  
}
```

The methods just presented should not be overloads. They should have two different names.

It's difficult in some languages (in particular, dynamically typed languages) to resolve calls to these kinds of overloads. In most cases, if two overloads called with a number and a string would do exactly the same thing, it would not matter which overload is called. For example, the following is fine, because these methods are semantically equivalent:

```
public static class Console {  
    public void WriteLine(long value){ ... }  
    public void WriteLine(int value){ ... }  
}
```

- ✓ **DO** allow `null` to be passed for optional arguments.

If a method takes optional arguments that are reference types, allow `null` to be passed to indicate that the default value should be used. This avoids the problem of having to check for `null` before calling an API, as shown here:

```
if (geometry==null) DrawGeometry(brush, pen);  
else DrawGeometry(brush, pen, geometry);
```

■ **BRAD ABRAMS** Notice that this guideline is not intended to encourage developers to use `null` as a magic constant, but rather to help them avoid explicit checking, as previously described. In fact, whenever you have to use literal `null` when calling an API, it indicates an error in your code or in the framework not providing the appropriate overload.

✗ **DO NOT** have overloads with a generic type parameter in one method and a specific type in another.

When designing methods on a generic type, it may sometimes seem appropriate to have a generic-based overload and a `System.String` based overload, as in this example:

```
public class PrettyPrinter<T> {  
    public void PrettyPrint(string format){ ... }  
    public void PrettyPrint(T otherPrinter){ ... }  
    public void PrettyPrint(T otherPrinter, string format){ ... }  
}
```

However, when the generic type is itself `System.String`, the overload that uses the generic type parameter becomes uncallable.

```
PrettyPrinter<string> printer = new PrettyPrinter<string>();  
...  
// C# overload resolution says this is PrettyPrint(string),  
// not PrettyPrint(T).  
// There's no way to call PrettyPrint(T) from C#.  
printer.PrettyPrint("hello, world");
```

■ **JEREMY BARTON** This guidance doesn't apply as strongly for generic methods (as opposed to non-generic methods on a generic types), because a generic method can be explicitly invoked by specifying the generic type arguments. But it still can cause unpleasant syntax, may not be possible to solve in all CLR languages, and is far more likely to have different semantics between the two methods with the same name.

✓ **CONSIDER** using default parameters on the longest overload of a method.

Default parameters are available in C#, F#, and VB.NET, but they are not CLS-compliant and may not be well supported by all languages that can use .NET libraries.

For the languages that do support default parameters, the use of default parameters can reduce the number of overloads that becomes possible for expressing every combination of optional parameters while still allowing the caller to use a somewhat minimal method invocation.

```
public static BigInteger Parse(  
    ReadOnlySpan<char> value,  
    NumberStyles style = NumberStyles.Integer,  
    IFormatProvider provider = null) { ... }  
...  
BigInteger val = BigInteger.Parse(input, provider: formatCulture);
```

■ **STEPHEN TOUB** It's worth considering the performance implications of default parameters. A method such as `Parse` (in the preceding example) likely needs to validate all of its arguments and also have all of the branches necessary to support any valid combination of those arguments, even when it's called with the default values for all of the optional arguments. In contrast, if this example used multiple overloads, the overload that just took `value` wouldn't need to validate either `style` or `provider`, and it could delegate directly to the internal implementation that's able to handle the equivalent of `NumberStyles.Integer` when no format provider is specified. For methods called on hot code paths in high-performance scenarios, that difference can be measurable.

When looking at tools for tree shaking/linking, overloads with lots of default parameters can also be problematic. Let's say the `Parse` method had different implementations to handle integer and hex number styles. A linker that sees this overload being used might end up keeping both the integer and hex support, even if it's only ever called with the default integer argument. If this were instead an overload that didn't take a style at all, a linker has a better chance of being able to strip away portions of the implementation that go unused.

- ✓ **DO** provide a simple overload, with no default parameters, for any method with two or more defaulted parameters.

Usability studies have shown that default parameters are a comparatively advanced feature. Many developers are confused with the presentation in IntelliSense, and sometimes think that they have to provide the defaulted arguments as exactly the value shown by IntelliSense. Providing the simple overload makes your method easier to approach for users who need only the default behaviors.

```
public static BigInteger Parse(ReadOnlySpan<char> value)
    // Since a provider is 'specified' this calls the overload
    // which uses default parameters.
    => Parse(value, provider: null);

public static BigInteger Parse(
    ReadOnlySpan<char> value,
    NumberStyles style = NumberStyles.Integer,
    IFormatProvider provider = null) { ... }
```

- ✗ **DO NOT** use default parameters, for any parameter type other than `CancellationToken`, on interface methods or virtual methods on classes.

A disagreement between an interface declaration and the implementation on a default value creates an unnecessary source of confusion for your users. The same confusion also arises when a virtual method and an override of that method disagree on the default values for a parameter. The most straightforward solution to this problem is to avoid the situation, and only use default parameters on nonvirtual methods.

`CancellationToken` is specifically exempted from this guideline, because the guidance for asynchronous methods (section 9.2) is to always have a `CancellationToken` parameter with a default value. Since there's only one legal default value for a `CancellationToken`—that is, `default(CancellationToken)`—no default value disagreement is possible between implementations.

```
public interface IExample {
    void PrintValue(int value = 5);
}
```

```
public class Example : IExample {
    public virtual void PrintValue(int value = 10) {
        Console.WriteLine(value);
    }
}
public class DerivedExample : Example {
    public override void PrintValue(int value = 20) {
        base.PrintValue(value);
    }
}
...
// What gets printed?
// If you have to think about it, the API is not self-documenting.
DerivedExample derived = new DerivedExample();
Example e = derived;
IExample ie = derived;
derived.PrintValue();
e.PrintValue();
ie.PrintValue();
```

For types that use the Template Method Pattern (section 9.9), the longest overload of the public nonvirtual method can use default parameters when appropriate, then defer to the virtual implementation method without using default parameters.

```
public partial class Control {
    // The public method uses a default parameter
    public void SetBounds(
        int x,
        int y,
        int width,
        int height,
        BoundsSpecified specified = BoundsSpecified.All) {
        ...
        SetBoundsCore(x, y, width, height, specified);
    }

    // The protected (virtual) method does not use default parameters
    protected virtual void SetBoundsCore(
        int x,
        int y,
        int width,
```

```
    int height,
    BoundsSpecified specified) {
    // Do the real work here.
}
}
```

Default parameters can be provided for interface methods via extension methods.

```
public interface IExample {
    void PrintValue(int value);
}

public static class ExampleExtensions : IExample {
    public static void PrintValue(
        this IExample example,
        int value = 10) {

        // While this may look like a recursive call,
        // the C# method resolution rules say the instance member
        // from the interface is a better match than this extension
        // method.
        example.PrintValue(value);
    }
}
```

One proposed alternative to not using default parameters in virtual methods was to say that virtual overrides should always match the defaults from the original method declaration. However, that guideline leads to versioning problems when derived types exist in a different assembly:

- Changing the last required parameter of a method to be a default parameter is generally not a breaking change, but would cause the override to provide a different set of defaults.
- The guidance for adding a new default parameter to an existing method (provided next) calls for removing the defaults from the current method overload, which can result in ambiguous method compile-time failures due to the overrides that have yet to be updated.

- ✗ **DO NOT** change the default value for a parameter once it has been publicly released.

Default parameters do not create logical overloads in the assembly that defines them, but are just a convenience mechanism to enable the compiler to fill in any missing values for the calling code at compile-time. When the value changes between two releases of a library, any existing callers will use the old value until they recompile, at which point they will switch to the new value. This can lead to confusion when diagnosing issues raised by users, when their reproduction cases don't exhibit the bad behavior due to the changed default value.

If you want to be able to change the value over time, use an otherwise illegal sentinel value such as zero or -1 to indicate the runtime default value should be used instead.

```
// In this method, the style parameter should continue to use
// NumberStyles.Integer in all future versions of the library.
//
// Since the provider parameter has a default value of null,
// the library can change the default as an implementation detail
// (assuming that's an acceptable breaking change for the method).
public static BigInteger Parse(
    ReadOnlySpan<char> value,
    NumberStyles style = NumberStyles.Integer,
    IFormatProvider provider = null) { ... }
```

- ✗ **DO NOT** have two overloads of a method with “compatible” required parameters that both use default parameters.
- ✗ **AVOID** having two overloads of the same method that both use default parameters.
- ✗ **DO NOT** have different defaults for the same parameter in two overloads of the same method.

The only time that two overloads of the same method should both have default parameters is when their required parameters have incompatible signatures. The parameters shared in common by the two overloads

should have the same default values, or no default value. This makes it abundantly clear which overload is being called.

```
// Given this overload already exists
public static OperationStatus DecodeFromUtf8(
    ReadOnlySpan<byte> utf8,
    Span<byte> bytes,
    out int bytesConsumed,
    out int bytesWritten,
    bool isFinalBlock = true) { ... }

// OK: default parameters have the same value,
// and the signatures are incompatible
public static byte[] DecodeFromUtf8(
    byte[] utf8,
    out int bytesConsumed,
    bool isFinalBlock = true) { ... }

// BAD: default parameters have a different value
public static byte[] DecodeFromUtf8(
    byte[] utf8,
    out int bytesConsumed,
    bool isFinalBlock = false) { ... }

// BAD: This longer overload's required parameters are compatible
// with the original overload
public static OperationStatus DecodeFromUtf8(
    ReadOnlySpan<byte> utf8,
    Span<byte> bytes,
    out int bytesConsumed,
    out int bytesWritten,
    bool isFinalBlock = true,
    int bufferSize = 512) { ... }
```

- ✓ **DO** move all default parameters to the new, longer overload when adding optional parameters to an existing method.

In the previous example, when adding the `bufferSize` parameter, the existing `DecodeFromUtf8` method should change to no longer have a default value for the `isFinalBlock` parameter when the new overload is added. This method cannot be deleted without introducing a runtime breaking change (a `MissingMethodException`) in upgrade scenarios. Since having a simple parameter after an `out` parameter makes this `DecodeFromUtf8` overload no longer conform to design guidelines,

it is advisable to attribute it as [EditorBrowsable(EditorBrowsableState.Never)]. IntelliSense will only show the new overload with the additional optional parameters.

```
// This overload no longer has a default value for isFinalBlock,
// and has been marked as hidden from IntelliSense.
[EditorBrowsable(EditorBrowsableState.Never)]
public static OperationStatus DecodeFromUtf8(
    ReadOnlySpan<byte> utf8,
    Span<byte> bytes,
    out int bytesConsumed,
    out int bytesWritten,
    bool isFinalBlock) { /* call the longer overload */ }

public static OperationStatus DecodeFromUtf8(
    ReadOnlySpan<byte> utf8,
    Span<byte> bytes,
    out int bytesConsumed,
    out int bytesWritten,
    bool isFinalBlock = true,
    int bufferSize = 512) { ... }
```

5.1.2 Implementing Interface Members Explicitly

Explicit interface member implementation allows an interface member to be implemented so that it is only callable when the instance is cast to the interface type. For example, consider the following definition:

```
public struct Int32 : IConvertible {
    int IConvertible.ToInt32 () {...}
    ...
}

// callingToInt32 defined on Int32
int i = 0;
i.ToInt32(); // does not compile
((IConvertible)i.ToInt32()); // works just fine
```

In general, implementing interface members explicitly is straightforward and follows the same general guidelines as those for methods, properties, or events. However, some specific guidelines apply to implementing interface members explicitly, as described next.

■ **ANDERS HEJLSBERG** Programmers working in other environments consistently complain about the need for internal methods to be public just so a class can implement some worker interface. Explicit member implementations are the correct solution to that problem. Yes, it is true that C# doesn't provide any syntax to call the base implementation of such members, but I have seen very few actual requests for that feature.

✗ **AVOID** implementing interface members explicitly without having a strong reason to do so.

Explicitly implemented members can be confusing to developers because such members don't appear in the list of public members and can also cause unnecessary boxing of value types.

■ **KRZYSZTOF CWALINA** You need to be especially careful when considering explicitly implementing members on value types. Casting a value type to an interface causes boxing.

■ **STEPHEN TOUB** It's true that casting a value type to an interface causes boxing. However, in some cases the runtime is able to undo it. In other cases, boxing can be avoided via generics and generic specialization. Consider the method:

```
public int CallToInt32<T>(T convertible) where T : IConvertible =>
    convertible.ToInt32();
```

When calling this method with a value type that implements `IConvertible`, even explicitly, it shouldn't be boxed.

■ **RICO MARIANI** Value types are often very simple types with very simple operations on them. You must be especially careful about incurring extra dispatching costs on any of these operations. If the main work at hand is something as simple as comparing one value to another or reinterpreting an integer in another format, then just the cost of the function calls might be more than the actual work you have to do. Keep this in mind or you might find that you have created a value type that is fundamentally unusable for its primary (cheap) purpose.

■ **JEFFREY RICHTER** In general, the reason to explicitly implement an interface method is if your type also has another method with the same name and parameters but with a different return type. For example:

```
class Collection : IEnumerable {  
    IEnumerator IEnumerable.GetEnumerator() { ... }  
    public MyEnumerator GetEnumerator() { ... }  
}
```

The `Collection` class can't have two methods called `GetEnumerator` that differ only in return value unless the version that returns an `IEnumerable` is an explicitly implemented interface method (as shown earlier). Now, if someone calls `GetEnumerator`, they are calling the strongly typed version of the method that returns `MyEnumerator`. Aside from this example, there are few other reasons to implement an interface method explicitly.

■ **STEVEN CLARKE** One common observation we have made in the API usability studies is that many developers assume that the reason members have been implemented explicitly is that they are not supposed to be used in common scenarios. Thus, they sometimes have a tendency to avoid using these members and spend time looking for some other way to accomplish their task.

✓ **CONSIDER** implementing interface members explicitly if the members are intended to be called only through the interface.

This includes mainly members supporting framework infrastructure, such as data binding or serialization. For example, `ICollection<T>.IsReadOnly` is intended to be accessed mainly by the data-binding infrastructure through the `ICollection<T>` interface. It is almost never accessed directly when using types implementing the interface. Therefore, `List<T>` implements the member explicitly.

✓ **CONSIDER** implementing interface members explicitly to simulate variance (change parameters or return types in “overridden” members).

For example, `IList` implementations often change the type of the parameters and returned values to create strongly typed collections by

explicitly implementing (hiding) the loosely typed member and adding the publicly implemented strongly typed member.

```
public class StringCollection : IList {
    public string this[int index]{ ... }
    object IList.this[int index] { ... }
    ...
}
```

- ✓ **CONSIDER** implementing interface members explicitly to hide a member and add an equivalent member with a better name.

You can say that this amounts to renaming a member. For example, `System.Collections.Concurrent.ConcurrentQueue<T>` implements `IProducerConsumerCollection<T>.TryTake` explicitly and renames it to `TryDequeue`.

```
partial class ConcurrentQueue<T> : IProducerConsumerCollection<T> {
    bool IProducerConsumerCollection<T>.TryTake(out T item) =>
        TryDequeue(out item);

    public bool TryDequeue(out T result) { ... }
}
```

Such member renaming should be done extremely sparingly. In most cases, the added confusion is a bigger problem than the suboptimal name of the interface member.

- ✗ **DO NOT** use explicit members as a security boundary.

Such members can be called by any code by simply casting an instance to the interface.

RICO MARIANI Generally, objects that have security issues should expose the fewest possible interfaces and inherit as little as possible. Get your code reuse by encapsulation rather than by inheritance, and seal as much as you can. My biggest dissent with the guidelines is that they do not recommend sealing as often as I believe they should. Where there are security issues, be more careful and consider sealing more aggressively, inheriting less aggressively, and using explicit (sealed) implementations of those interfaces you need to offer.

- ✓ **DO** provide a protected virtual member that offers the same functionality as the explicitly implemented member if the functionality is meant to be specialized by derived classes.

Explicitly implemented members cannot be overridden. They can be redefined, but then subtypes cannot call the base method's implementation. It is recommended that you name the protected member by either using the same name or affixing `Core` to the interface member name.

```
[Serializable]
public class List<T> : ISerializable {
    ...
    void ISerializable.GetObjectData(
        SerializationInfo info, StreamingContext context) {
        GetObjectData(info, context);
    }

    protected virtual void GetObjectData(
        SerializationInfo info, StreamingContext context) {
        ...
    }
}
```

■ **RICO MARIANI** Classes designed to be subclassed in normal use need extra care. The times when the `Core` functions are called is part of the contract, so document it well and try not to change it. Access to protected members might allow partially trusted code to do bad things to the internal state. You must code with the expectation that the subclass will be hostile.

5.1.3 Choosing Between Properties and Methods

When designing members of a type, one of the most common decisions a library designer must make is to choose whether a member should be a property or a method.

On a high level, there are two general styles of API design in terms of usage of properties and methods. In method-heavy APIs, methods have a large number of parameters, and the types have fewer properties.

```
public class PersonFinder {
    public string FindPersonsName (
        int height,
        int weight,
```

```
    string hairColor,  
    string eyeColor,  
    int shoeSize,  
    Connection database  
);  
}
```

In property-heavy APIs, methods have a small number of parameters and more properties to control the semantics of the methods.

```
public class PersonFinder {  
    public int Height { get; set; }  
    public int Weight { get; set; }  
    public string HairColor { get; set; }  
    public string EyeColor { get; set; }  
    public int ShoeSize { get; set; }  
  
    public string FindPersonsName (Connection database);  
}
```

RICO MARIANI Note the huge difference in semantics. With properties, you must (or can, if you see that as a benefit) set each field independently, and the `PersonFinder` can be used for only one call at one time, because it captures the result of the find as well as the input. With the functional contract, multiple threads can use the very same finder and there is only one function call. This is not a small decision we are making here.

CHRIS ANDERSON I have to firmly agree with the guideline encouraging the use of properties. Generally, methods with lots of arguments lend themselves to lots of overloads: You have 15 overloads of a method so you can get every combination of options. This produces APIs that are super hard to understand, and inconsistent. Look at `DrawRectangle` on `System.Drawing.Graphics` as a great example. There are a lot of overloads, and there is always one missing. When you add a new feature to the API, you have to add more overloads, making it less and less understandable over time. Properties provide a natural self-documentation aspect to the API, easy statement completion, progressive understanding, and simple versioning. You always have to balance performance, but in general properties really do add a huge amount of value.

■ **JEREMY BARTON** I prefer an object with a lot of properties for configuration, which is then passed to a method (or constructor) to control the behavior. Essentially, I rarely combine methods and settable properties on the same type.

Be leery of adding properties to control the behavior of virtual methods, especially when adding them on a derived type.

All else being equal, the property-heavy design is generally preferable because methods with many parameters are less approachable to inexperienced developers. This is described in detail in Chapter 2.

Another reason to use properties when they are appropriate is that property values show up automatically in the debugger. By comparison, inspecting a value of a method is much more cumbersome.

■ **CHRIS SELLS** I have yet to find any developer who prefers things to be harder than they need to be. If you make something more approachable for the “inexperienced developer,” then it’s more approachable for everyone.

However, it is worth noting that the method-heavy design has the advantage in terms of performance, and might result in better APIs for advanced users.

A rule of thumb is that methods should represent actions and properties should represent data. Properties are preferred over methods if everything else is equal.

■ **RICO MARIANI** Properties probably result in more performance crimes than any other language feature. You must remember that the property looks like a simple field access to your customers and comes with an expectation that it is no more costly than a field access. You can expect your callers will write straightforward-looking code to access the properties, and they will be astonished if this is expensive. Similarly, they will be astonished if the behavior changes over time so that it becomes costly, or if it is costly with some subtypes and not with others.

■ **JOE DUFFY** Properties should contain a tiny amount of code. If you avoid if-statements, try-catch blocks, calls to other methods, etc., and strive to make them nothing more than a simple field access, it is highly likely the CLR's JIT compiler will inline all accesses to them. Doing this helps to avoid the performance crimes that Rico mentions in his comment.

■ **JEREMY BARTON** It's not uncommon for a property to be read multiple times in the same calling method, or within a loop. If your property getter is complex enough that calling it in a loop is bad, it probably should have been a method.

✓ **CONSIDER** using a property if the member represents a logical attribute of the type.

For example, `Button.Color` is a property because color is an attribute of a button.

✓ **DO** use a property, rather than a method, if the value of the property is stored in the process memory and the property would just provide access to the value.

For example, a member that retrieves the name of a `Customer` from a field stored in the object should be a property.

```
public Customer {  
    public Customer(string name){  
        this.name = name;  
    }  
    public string Name {  
        get { return this.name; }  
    }  
    private string name;  
}
```

✓ **DO** use a method, rather than a property, in the following situations:

- The operation is orders of magnitude slower than a field access would be. If you are even considering providing an asynchronous version of an operation to avoid blocking the thread, it is very likely

that the operation is too expensive to be a property. In particular, operations that access the network or the file system (other than once for initialization) should likely be methods, not properties.

- The operation is a conversion, such as the `Object.ToString` method.
- The operation returns a different result each time it is called, even if the parameters don't change. For example, the `Guid.NewGuid` method returns a different value each time it is called.

■ **JEFFREY RICHTER** The `DateTime.Now` property is an example of a place in .NET where this property should have been a method because the operation returns a different result each time.

- The operation has a significant and observable side effect. Notice that populating an internal cache is not generally considered an observable side effect.

■ **BRIAN PEPIN** The `Handle` property on a Windows Forms control is an example of where too many side effects might occur. If the control has not had its handle created yet, accessing the `Handle` property will create it. I can't tell you how many times I've completely changed the result of a debugging session by setting up a watch on the `Handle` property. The debugger's watch will actually create the control's handle, often hiding your bug.

- The operation returns a copy of an internal state (this does not include copies of value-type objects returned on the stack).
- The operation returns an array.

Properties that return arrays can be very misleading. Usually it is necessary to return a copy of an internal array so that the user cannot change the internal state. This could lead to inefficient code.

In the following example, the `Employees` property is accessed twice in every iteration of the loop. That would be $2n + 1$ copies for the following short code sample:

```
Company microsoft = GetCompanyData("MSFT");
for (int i = 0; i < microsoft.Employees.Length; i++) {
    if (microsoft.Employees[i].Alias == "kcwalina"){
        ...
    }
}
```

This problem can be addressed in one of two ways:

- Change the property into a method, which communicates to callers that they are not just accessing an internal field and probably are creating an array every time they call the method. Given that, users are more likely to call the method once, cache the result, and work with the cached array.

```
Company microsoft = GetCompanyData("MSFT");
Employees[] employees = microsoft.GetEmployees();
for (int i = 0; i < employees.Length; i++) {
    if (employees[i].Alias == "kcwalina"){
        ...
    }
}
```

- Change the property to return a collection instead of an array. You can use `ReadOnlyCollection<T>` to provide public read-only access to a private array. Alternatively, you can use a subclass of `Collection<T>` to provide controlled read-write access, so that you can be notified when the user code modifies the collection. See section 8.3 for more details on using `ReadOnlyCollection<T>` and `Collection<T>`.

```
public ReadOnlyCollection<Employee> Employees {
    get { return roEmployees; }
}
private Employee[] employees;
private ReadOnlyCollection<Employee> roEmployees;
```

■ **BRAD ABRAMS** Some of the guidelines in this book were debated and agreed on in the abstract; others were learned in the school of hard knocks. The guideline on properties that return arrays belongs to the school of hard knocks camp. When we were investigating some performance issues in version 1.0 of the .NET Framework, we noticed that thousands of arrays were being created and quickly trashed. It turns out that many places in the framework itself ran into this pattern. Needless to say, we fixed those instances and the guidelines.

■ **RICO MARIANI** I hope you've read this far—you really must understand that the preceding guideline is designed to avoid some pretty big problems. I'd encourage you to remember this shorthand: Use properties for simple access to simple data with a simple computation. Don't stray from that pattern.

5.2 Property Design

Although properties are technically very similar to methods, they are quite different in terms of their usage scenarios. They should be seen as smart fields. They have the calling syntax of fields, and the flexibility of methods.

✓ **DO** create get-only properties if the caller should not be able to change the value of the property.

If the type of the property is a mutable reference type, the property value can be changed even if the property is get-only.

✗ **DO NOT** provide set-only properties or properties with the setter having broader accessibility than the getter.

For example, do not use properties with a public setter and a protected getter.

If the property getter cannot be provided, implement the functionality as a method instead. Consider starting the method name with `Set` and follow with what you would have named the property. For example,

AppDomain has a method called SetCachePath instead of having a set-only property called CachePath.

- ✓ **DO** provide sensible default values for all properties, ensuring that the defaults do not result in a security hole or terribly inefficient code.
- ✓ **DO** allow properties to be set in any order, even if this results in a temporary invalid state of the object.

It is common for two or more properties to be interrelated to a point where some values of one property might be invalid given the values of other properties on the same object. In such cases, exceptions resulting from the invalid state should be postponed until the interrelated properties are actually used together by the object.

■ **RICO MARIANI** This happens a lot in business objects used in a three-tier system. You cannot do much more than basic validation of the properties when they are set. You have to provide an explicit “commit” method of some kind so that you know the caller is completely done with the update. Avoid validation that you cannot do with local knowledge (e.g., don’t go to the database). Remember that there is a strong expectation that setting a property will not be much more expensive than setting a field.

■ **BRIAN PEPIN** While working on the code generator for the Windows Forms designer, I had a lot of people ask me for a way to tell the code generator how to order properties. I stubbornly refused every time, because it adds a huge amount of complexity for developers. It’s easy to dictate an order for your properties, but how does your property ordering mix in with classes that derive from you? Also, if it is complicated to describe to a code generator how things need to be ordered, imagine how complicated it will be to explain this to developers.

■ **JEREMY BARTON** One place where order-dependent property setting comes up is when assigning property A changes property B. You generally want to avoid doing that, since the confusion it costs is probably worse than the potential error state it was avoiding.

✓ **DO** preserve the previous value if a property setter throws an exception.

✗ **AVOID** throwing exceptions from property getters.

Property getters should be simple operations and should not have any preconditions. If a getter can throw an exception, it should probably be redesigned to be a method. Notice that this rule does not apply to indexers, where we do expect exceptions as a result of validating the arguments.

■ **PATRICK DUSSUD** Notice that this guideline only applies to property getters. It is OK to throw an exception in a property setter. It is very much like setting an array element, which can throw as well (and not just when checking the index bound, but also when checking for a type mismatch between the value and the array element type).

■ **JASON CLARK** Patrick's guidance speaks to one of the key reasons that exceptions and object-oriented programming environments go hand in hand. Object-oriented environments impose many circumstances in which the developer of a method has limited or no control of the methods signature. Properties, events, constructors, virtual overrides, and operator overloads are examples of this. What exceptions do is take the error response of a method call out of band of the signature or return type. This introduces the necessary flexibility to both reflect and react to failure in any method regardless of the syntax sugar that restricts signature decisions.

■ **JOE DUFFY** Another pattern to avoid is blocking the thread inside of a property, because of either I/O or some synchronization operation. This is the extreme example of "doing too much work" inside of a property. In the worst case (deadlock), the property may never actually return to the caller. If the property accesses shared state, it may be necessary to acquire a lock to access it safely, but anything more complicated (like waiting on a `WaitHandle`) is an indication that a method is a better design choice.

5.2.1 Indexed Property Design

An indexed property is a special property that can have parameters and can be called with special syntax similar to array indexing.

```
public class String {
    public char this[int index] {
        get { ... }
    }
}

string city = "Seattle";
Console.WriteLine(city[0]); // this will print 'S'
```

Indexed properties are commonly referred to as indexers. Indexers should be used only in APIs that provide access to items in a logical collection. For example, a string is a collection of characters, and the indexer on `System.String` exists to access its characters.

 **RICO MARIANI** Be extra careful with these—you can expect them to be called in a loop! Keep them very simple.

 **CONSIDER** using indexers to provide access to data stored in an internal array.

 **CONSIDER** providing indexers on types representing collections of items.

 **AVOID** using indexed properties with more than one parameter.

If the design requires multiple parameters, reconsider whether the property really represents an accessor to a logical collection. If it does not, use methods instead. Consider starting the method name with `Get` or `Set`.

 **AVOID** indexers with parameter types other than `System.Int32`, `System.Int64`, `System.String`, `System.Range`, `System.Index`, or an enum, except on dictionary-like types.

If the design requires other types of parameters, strongly reevaluate whether the API really represents an accessor to a logical collection. If it does not, use a method. Consider starting the method name with `Get` or `Set`.

- ✓ **DO** use the name `Item` for indexed properties unless there is an obviously better name (e.g., see the `Chars` property on `System.String`).

In C#, indexers are by default named `Item`. The `IndexerNameAttribute` can be used to customize this name.

```
public sealed class String {  
    [System.Runtime.CompilerServices.IndexerNameAttribute("Chars")]  
    public char this[int index] {  
        get { ... }  
    }  
    ...  
}
```

- ✗ **DO NOT** provide both an indexer and methods that are semantically equivalent.

In the following example, the indexer should be changed to a method.

```
// Bad design  
public class Type {  
    [System.Runtime.CompilerServices.IndexerNameAttribute("Members")]  
    public MemberInfo this[string memberName]{ ... }  
    public MemberInfo GetMember(string memberName, Boolean ignoreCase){  
        ... }  
}
```

- ✗ **DO NOT** provide more than one family of overloaded indexers in one type.

This is enforced by the C# compiler.

- ✗ **DO NOT** use nondefault indexed properties.

This is enforced by the C# compiler.

- ✓ **DO** return the declaring type from an indexer that accepts `System.Range`.

Range-based indexers on a collection should return a collection of the same type as the type that declares them. This includes implicit indexers created by having a `Slice` method with the appropriate signature. Instead of violating this guideline when your goal is to create a different type, provide a separate conversion method. For example, arrays

produce a copy of the requested range in the Range indexer, but arrays also have the `array.AsSpan(Range)` extension method to produce a `Span<T>` for the same range of indices.

```
// OK: Returns the declaring type from a Range-based indexer
public partial class SomeCollection {
    public SomeCollection this[Range range] { get { ... } }
}

// OK: Returns the declaring type from Slice(int, int), which
// makes an implicit indexer in C#
public partial class SomeCollection {
    public SomeCollection Slice(int offset, int length) { ... }
}

// BAD: Returns the wrong type from a Range-based indexer
public partial class SomeCollection {
    public SomeValue[] this[Range range] { get { ... } }
}

// BAD: Returns the wrong type from Slice(int, int), which
// makes an implicit indexer in C#
public partial class SomeCollection {
    public SomeValue[] Slice(int offset, int length) { ... }
}
```

JAN KOTAS The behavior of the `System.Range` indexer for arrays was a subject of long debate. Making it very easy to produce copies of potentially large arrays seems to be a performance trap. In the end, the copying behavior was chosen for consistency. The future will tell whether it was a wise choice.

5.2.2 Property Change Notification Events

Sometimes it is useful to provide an event notifying the user of changes in a property value. For example, `System.Windows.Forms.Control` raises a `TextChanged` event after the value of its `Text` property has changed.

```
public class Control : Component{
    string text = String.Empty;

    public event EventHandler<EventArgs> TextChanged;
```

```
public string Text{
    get{ return text; }
    set{
        if (text!=value) {
            text = value;
            OnTextChanged();
        }
    }
}

protected virtual void OnTextChanged(){
    EventHandler<EventArgs> handler = TextChanged;
    if(handler!=null){
        handler(this,EventArgs.Empty);
    }
}
```

The guidelines that follow describe when property change events are appropriate and their recommended design for these APIs.

■ **RICO MARIANI** Recall some of the previous rules about properties: They should look and act like fields as much as possible, because library users will think of them and use them as though they were fields. Here, we're practically guaranteeing that the property setter will have a side effect because we're allowing arbitrary user code to run. We're also causing multiple function calls to be made for each set. All of that cross-wiring of objects and handlers often causes "object spaghetti." If the notifications are not at a high enough level, they will occur much too frequently and the connections will be too complex, rendering the system both unusable and indescribable.

■ **CHRIS SELLS** For data binding to work properly in WPF or Windows Forms, objects with properties must raise property change events, preferably via `IPropertyChanged`.

✓ **CONSIDER** raising change notification events when property values in high-level APIs (usually designer components) are modified.

If there is a good scenario for a user to know when a property of an object is changing, the object should raise a change notification event for the property.

However, it is unlikely to be worth the overhead to raise such events for low-level APIs such as base types or collections. For example, `List<T>` would not raise such events when a new item is added to the list and the `Count` property changes.

■ **CHRIS SELLS** Because `List<T>` doesn't implement any of the notification APIs for data binding, and because I like data binding, I find myself using `ObservableCollection<T>` instead, which implements `INotifyCollectionChanged`.

✓ **CONSIDER** raising change notification events when the value of a property changes via external forces.

If a property value changes via some external force (in a way other than by calling methods on the object), raising events indicates to the developer that the value is changing or has changed. A good example is the `Text` property of a text box control. When the user types text in a `TextBox`, the property value automatically changes.

5.3 Constructor Design

There are two kinds of constructors: type constructors and instance constructors.

```
public class Customer {  
    public Customer() { ... } // instance constructor  
    static Customer() { ... } // type constructor  
}
```

Type constructors are static and are run by the CLR before the type is used. Instance constructors run when an instance of a type is created.

Type constructors cannot take any parameters; instance constructors can. Instance constructors that don't take any parameters are often called default constructors.

Constructors are the most natural way to create instances of a type. Most developers will search for, and try to use, a constructor before they consider alternative ways of creating instances (such as factory methods).

✓ **CONSIDER** providing simple, ideally default, constructors.

A simple constructor has a very small number of parameters, and all parameters are primitives or enums. Such simple constructors increase the usability of the framework.

■ **PHIL HAACK** If you have a type that has a required dependency on another type—that is, if an instance of your type won't work if it doesn't have a proper instance of the other type—then you should allow passing that type in via a constructor parameter.

I see the constructor parameter list as defining the set of required dependencies for your type. Following this approach makes your type more amenable to dependency injection.

At the same time, I think it's a good practice to have default constructors that provide reasonable defaults for these required dependencies.

✓ **CONSIDER** using a static factory method instead of a constructor if the semantics of the desired operation do not map directly to the construction of a new instance, or if following the constructor design guidelines feels unnatural.

See section 9.5 for more details on factory method design.

■ **STEPHEN TOUB** Factory methods can also afford significantly more flexibility in the future, as they enable returning a type derived from the return type rather than forcing it to be exactly the return type. An example near and dear to me is `Task`. With 20/20 hindsight, if we could do it over, we would not have added constructors that take a delegate to `System.Threading.Tasks.Task`. In doing so, we've made it extremely difficult for ourselves to optimize `Task` for scenarios that don't require a delegate, such as when `Task` is used as the return type of an `async` method. If we didn't have such constructors and only had factories (e.g. `Task.Run`), we could have put the delegate onto a derived type rather than onto the base type, and reduced the size of the `Task` object for all other cases where a delegate isn't needed.

- ✓ **DO** use constructor parameters as shortcuts for setting main properties.

There should be no difference in semantics between using the empty constructor followed by some property sets and using a constructor with multiple arguments. The following three code examples are equivalent:

```
//1
var applicationLog = new EventLog();
applicationLog.MachineName = "BillingServer";
applicationLog.Log = "Application";

//2
var applicationLog = new EventLog("Application");
applicationLog.MachineName = "BillingServer";

//3
var applicationLog = new EventLog("Application", "BillingServer");
```

- ✓ **DO** use the same name for constructor parameters and a property if the constructor parameters are used just to set the property.

The only difference between such parameters and the properties should be casing.

```
public class EventLog {
    public EventLog(string logName){
        this.LogName = logName;
    }
    public string LogName {
        get { ... }
        set { ... }
    }
}
```

- ✓ **CONSIDER** adding a property for each parameter in a constructor.

Developers generally find it helpful when they can inspect the state of an object for debugging or logging purposes. While a debugger can usually inspect the private fields of an object, that isn't easy to do with logging. Exposing at least a get-only property for each constructor parameter enables users to incorporate the state of your object into their diagnostic flow.

Constructor parameters typed as mutable reference types may not be suitable for exposing as properties, particularly when your type is stateful and modifying that parameter can invalidate your state. You should also be mindful of the guidance for when to use a method instead of a property (see section 5.1.3).

You may have other reasons, such as data hiding, for not exposing constructor parameters via properties. Even so, you should generally start from an expectation of adding the properties and then justify why doing so is not appropriate in context.

✓ **DO** minimal work in the constructor.

Constructors should not do much work other than capture the constructor parameters. The cost of any other processing should be delayed until required.

✓ **DO** throw exceptions from instance constructors, if appropriate.

■ **CHRISTOPHER BRUMME** When an exception propagates out of a constructor, the object is already created despite the fact that the new operator does not return the object reference. If the type defines a `Finalize` method, the method will run when the object becomes eligible for garbage collection. This means that you should make sure the `Finalize` method can run on partially constructed objects.

■ **JEFFREY RICHTER** Alternatively, you could call `GC.SuppressFinalize` from within the constructor itself to avoid having the `Finalize` method called and to improve performance.

```
// finalizable type constructor that can throw
public FinalizableType(){
    try{
        SomeOperationThatCanThrow();
        handle = ... // allocate resource that needs to be finalized
    }
    catch(Exception){
        GC.SuppressFinalize(this);
        throw;
    }
}
```

■ **BRIAN GRUNKEMEYER** Note that if your constructor throws an exception, the finalizer for your type will still run! So the finalizer and the `Dispose(false)` code path in your type must be prepared to handle an uninitialized state. Worse yet, if your app must deal with asynchronous exceptions such as `ThreadAbortException` or `OutOfMemoryException`, your finalizer may have to deal with partially initialized state if your constructor threw an exception halfway through! This surprising fact is usually pretty easy to deal with, but it may take you several years to realize it.

■ **JOE DUFFY** A related constructor anti-pattern can lead to the same problem that Chris describes. If a constructor prematurely shares the `this` reference, the object may be accessible even if it has thrown an exception before it was fully constructed. This could happen by setting a field on an object passed in as an argument, or via a static variable, for example. Avoid doing this at all costs.

✓ **DO** explicitly declare the public default constructor in classes, if such a constructor is required.

If you don't explicitly declare any constructors on a type, many languages (such as C#) will automatically add a public default constructor. (Abstract classes get a protected constructor.) For example, the following two declarations are equivalent in C#:

```
public class Customer {  
}  
  
public class Customer {  
    public Customer(){}  
}
```

Adding a parameterized constructor to a class prevents the compiler from adding the default constructor. This often causes accidental breaking changes. Consider a class defined as shown in the following example:

```
public class Customer {  
}
```

Users of the class can call the default constructor, which the compiler automatically added in this case, to create an instance of the class.

```
var customer = new Customer();
```

It is quite common to add a parameterized constructor to an existing type with a default constructor. If the addition is not done carefully, however, the default constructor might no longer be emitted. For example, the following addition to the type just declared will “remove” the default constructor:

```
public class Customer {  
    public Customer(string name) { ... }  
}
```

This will break code relying on the default constructor, and such a problem is unlikely to be caught in a code review. Therefore, the best practice is to always specify the public default constructor explicitly.

Note that this does not apply to structs. Structs implicitly get default constructors even if they have a parameterized constructor defined.

AVOID explicitly defining default constructors on structs.

Many CLR languages do not allow developers to define default constructors on value types.¹ Users of these languages are often surprised to learn that `default(SomeStruct)` and `new SomeStruct()` don’t necessarily produce the same value. Even if your language does allow defining a default constructor on a value type, it probably isn’t worth the confusion it would cause if you did so.

```
public struct Token {  
    public Token(Guid id) { this.id = id; }  
    internal Guid id;  
}  
...  
var token = new Token(); // this compiles and executes just fine
```

The runtime will initialize all of the fields of the struct to their default values (0/null).

1. A struct without an explicitly defined constructor still gets a default constructor that is provided by the CLR implicitly; it assigns all fields to their “zero” value (0, false, null).

■ **JAN KOTAS** The default constructors on structs were enabled for C# 6 previews. We kept discovering cases where parameterless struct constructors caused inconsistent behaviors and decided to pull them out. It is not clear whether they will ever come back.

X AVOID calling virtual members on an object inside its constructor.

Calling a virtual member will cause the most derived override to be called, even if the constructor of the most derived type has not been fully run yet.

Consider the following example, which prints out “What is wrong?” when a new instance of `Derived` is created. The implementer of the derived class assumes that the value will be set before anyone can call `Method`. However, that is not true: The `Base` constructor is called before the `Derived` constructor finishes, so any calls it makes to `Method` might operate on data that is not yet initialized.

```
public abstract class Base {
    public Base() {
        Method();
    }
    public abstract void Method();
}

public class Derived: Base {
    private int value;
    public Derived() {
        value = 1;
    }

    public override void Method() {
        if (value == 1){
            Console.WriteLine("All is good");
        }
        else {
            Console.WriteLine("What is wrong?");
        }
    }
}
```

Occasionally, the benefits associated with calling virtual members from a constructor might outweigh the risks. An example is a helper constructor that initializes virtual properties using parameters passed to the constructor. It's acceptable to call virtual members from constructors, given that all the risks are carefully analyzed and you document the virtual members that you call for the users overriding the virtual members.

■ **CHRISTOPHER BRUMME** In unmanaged C++, the vtable is updated during the construction so that a call to a virtual function during construction only calls to the level of the object hierarchy that has been constructed.

It's been my experience that many programmers are as confused by the C++ behavior as they are about the managed behavior. The fact is that most programmers don't think about the semantics of virtual calls during construction and destruction until they have just finished debugging a failure related to this.

Either behavior is appropriate for some programs and inappropriate for others. Both behaviors can be logically defended. For the CLR, the decision is ultimately based on our desire to support extremely fast object creation.

5.3.1 Type Constructor Guidelines

A type constructor, also called a static constructor, is used to initialize a type. The runtime calls the static constructor before the first instance of the type is created or any static members of the type are accessed.

✓ **DO** make static constructors private.

A static constructor, also called a class constructor, is used to initialize a type. The CLR calls the static constructor before the first instance of the type is created or any static members of that type are called. The user has no control over when the static constructor is called. If a static constructor is not private, it can be called by code other than the CLR. Depending on the operations performed in the constructor, this can cause unexpected behavior.

The C# compiler forces static constructors to be private.

✗ **DO NOT** throw exceptions from static constructors.

If an exception is thrown from a type constructor, the type is not usable in the current application domain.

■ **CHRISTOPHER BRUMME** The only time it's OK to throw from a static constructor is if the type must never again be used in this application domain. You are basically making the type off-limits in the application domain where you throw, so you'd better have a good reason—such as if some important invariant is broken and your application would not be secure if such usage were permitted.

■ **VANCE MORRISON** To be clear on this rule, it does not apply to just explicitly throwing, but to any exception. This implies that if the body of the class constructor has the potential to make method calls (and this is very likely the case), you need a "try-catch" around the body of your class constructor.

■ **STEPHEN TOUB** In addition to not throwing exceptions from static constructors, it's also a good idea to avoid locking in static constructors, or employing any other form of cross-thread dependency. A static constructor on a type is only invoked once, which means if multiple threads are trying to initialize a type concurrently, the runtime may need to lock to provide this guarantee. If the runtime locks while invoking the static constructor, and then the code in the static constructor also locks, it's possible to end up with lock inversion that leads to deadlock. For example, this seemingly simple app will deadlock and never complete (at least on current implementations of .NET):

```
using System.Threading;

class Test
{
    static void Main() { }

    static Test()
    {
        var t = new Thread(ThreadStart);
        t.Start();
        t.Join();
    }

    static void ThreadStart() { }
}
```

✓ **CONSIDER** initializing static fields inline rather than explicitly using static constructors, because the runtime is able to optimize the performance of types that don't have an explicitly defined static constructor.

```
// unoptimized code
public class Foo {
    public static readonly int Value;
    static Foo() {
        Value = 63;
    }
    public static void PrintValue() {
        Console.WriteLine(Value);
    }
}

// optimized code
public class Foo {
    public static readonly int Value = 63;
    public static void PrintValue() {
        Console.WriteLine(Value);
    }
}
```

■ **CHRISTOPHER BRUMME** Be aware that initializing static fields inline has very loose guarantees about when the fields will be initialized. The guarantee is that the fields will be initialized before the first time they are accessed, but it could potentially occur much earlier. The CLR reserves the right to do the initialization before the program even starts running (e.g., using NGEN techniques). Explicit static constructors make a very precise guarantee. They are run before the first static member (code or data) is accessed, but no earlier.

■ **VANCE MORRISON** Generally, doing nontrivial work in a class constructor is bad because if it fails, you make the class unavailable, and you have made some nontrivial semantic effect happen at a potentially surprising time (do all your users know the rules for precise class construction?). Thus you should seriously reconsider any design that does nontrivial work in class constructors. Initializing statics is really the reason class constructors exist, and for that you *don't* need the static class {} syntax (use static initialization functions instead). It has fewer guarantees, but you should not need stronger guarantees, and the flexibility you give the runtime allows class construction to be more efficient.

■ **JAN KOTAS** The nonpredictable timing of static fields initialized inline was an endless source of subtle compatibility issues. .NET Core runtimes and recent versions of .NET Framework no longer take advantage of the loose guarantees, but instead execute the static constructor precisely on the first static field access. Defining fields inline still offers a performance advantage because the static constructor is triggered by the static field access only—and not by access to other members—but it comes without the warning about nonpredictable timing now.

5.4 Event Design

Events are the most commonly used form of callbacks (constructs that allow the framework to call into user code). Other callback mechanisms include members taking delegates, virtual members, and interface-based plug-ins. Data from usability studies indicates that the majority of developers are more comfortable using events than they are using the other callback mechanisms. Events are nicely integrated with Visual Studio and many languages.

Under the covers, events are not much more than fields that have a type that is a delegate, plus two methods to manipulate the field. Delegates used by events have special signatures (by convention) and are referred to as event handlers.

When users subscribe to an event, they provide an instance of the event handler bound to a method that will be called when the event is raised. The method provided by the user is referred to as an event handling method.

The event handler determines the event handling method's signature. By convention, the return type of the method is void and the method takes two parameters. The first parameter represents the object that raised the event. The second parameter represents event-related data that the object raising the event wants to pass to the event handling method. This data is often referred to as event arguments.

```
var timer = new Timer(1000);
timer.Elapsed += new ElapsedEventHandler(TimerElapsedHandlingMethod);
...
```

```
// event handling method for Timer.Elapsed
void TimerElapsedHandlingMethod(object sender, ElapsedEventArgs e){
    ...
}
```

There are two groups of events: events raised before a state of the system changes, called pre-events, and events raised after a state changes, called post-events. An example of a pre-event would be `Form.Closing`, which is raised before a form is closed. An example of a post-event would be `Form.Closed`, which is raised after a form is closed. The following example shows an `AlarmClock` class defining an `AlarmRaised` post-event.

```
public class AlarmClock {
    public AlarmClock() {
        timer.Elapsed += new ElapsedEventHandler(TimerElapsed);
    }

    public event EventHandler<AlarmRaisedEventArgs> AlarmRaised;

    public DateTimeOffset AlarmTime {
        get { return alarmTime; }
        set {
            if (alarmTime != value) {
                timer.Enabled = false;
                alarmTime = value;
                TimeSpan delay = alarmTime - DateTimeOffset.Now;
                timer.Interval = delay.TotalMilliseconds;
                timer.Enabled = true;
            }
        }
    }

    protected virtual void OnAlarmRaised(AlarmRaisedEventArgs e){
        EventHandler<AlarmRaisedEventArgs> handler = AlarmRaised;
        if (handler != null) {
            handler(this, e);
        }
    }

    private void TimerElapsed(object sender, ElapsedEventArgs e){
        OnAlarmRaised(AlarmRaisedEventArgs.Empty);
    }
}
```

```
private Timer timer = new Timer();
private DateTimeOffset alarmTime;
}
public class AlarmRaisedEventArgs : EventArgs {
    new internal static readonly
        AlarmRaisedEventArgs Empty = new AlarmRaisedEventArgs();
}
```

- ✓ **DO** use the term “raise” for events rather than “fire” or “trigger.”

When referring to events in documentation, use the phrase “an event was raised” instead of “an event was fired” or “an event was triggered.”

■ **BRAD ABRAMS** Why did we decide to use “raise” rather than “fire”? Well, we certainly have some prior art on our side on this one, but we also felt like “fire” was too negative a term. After all, you fire a gun or you fire an employee. “Raise” sounds more peaceful.

- ✓ **DO** use `System.EventHandler<T>` instead of manually creating new delegates to be used as event handlers.

```
public class NotifyingContactCollection : Collection<Contact> {
    public event EventHandler<ContactAddedEventArgs> ContactAdded;
    ...
}
```

If you’re adding new events to an existing feature area that uses traditional event handlers, then continue using the existing event handler types to remain consistent within the feature area, and use new custom event handler types if using the generic handler feels inconsistent with the feature area. For example, when defining new events related to types in the `System.Windows.Forms` namespace, you might want to continue using manually created handlers.

■ **BRIAN PEPIN** You wouldn't believe how long we debated this. On the one hand, `EventHandler<T>` doesn't really buy you that much over the one-line declaration of your own event handler. In fact, the syntax is quite a bit more confusing. On the other hand, reducing the number of classes that need to be loaded by your code improves performance. I've never been a big fan of choosing performance over ease of use (performance gets better over time; ease of use doesn't). The current round of compilers allows you to leave out the new `EventHandler<ContactAddedEventArgs>()` business, however, so this doesn't really impact ease of use that much. We also solved the last hurdle—getting the Visual Studio designers to understand generic events—so I'll be glad to finally put this debate to bed.

Creating new custom event handlers is rare, and the guidance has been archived to Appendix B.

✓ **CONSIDER** using a subclass of `EventArgs` as the event argument, unless you are absolutely sure the event will never need to carry any data to the event handling method, in which case you can use the `EventArgs` type directly.

```
public class AlarmRaisedEventArgs : EventArgs {  
}
```

If you ship an API using `EventArgs` directly, you will never be able to add any data to be carried with the event without breaking compatibility. If you use a subclass, even if it is initially completely empty, you will be able to add properties to the subclass when needed.

```
public class AlarmRaisedEventArgs : EventArgs {  
    public DateTimeOffset AlarmTime { get; }  
}
```

✓ **DO** use a protected virtual method to raise each event. This is applicable only to nonstatic events on unsealed classes—not to structs, sealed classes, or static events.

For each event, include a corresponding protected virtual method that raises the event. The purpose of the method is to provide a way for a derived class to handle the event using an override. Overriding is a

more flexible, faster, and more natural way to handle base class events in derived classes. By convention, the name of the method should start with “On” and be followed with the name of the event.

RICO MARIANI Note that using an event handler mechanism gives you maximum flexibility in terms of code that can receive the event—in principle, any object could be notified. Sometimes all that’s necessary is for the object to notify itself or more generally the subtype. If that’s the case, an event might be overkill and all you need is a virtual protected `OnWhatever()` method. This is both cheaper and simpler than an event.

```
public class AlarmClock {
    public event EventHandler<AlarmRaisedEventArgs> AlarmRaised;

    protected virtual void OnAlarmRaised(AlarmRaisedEventArgs e){
        EventHandler<AlarmRaisedEventArgs> handler = AlarmRaised;
        if (handler != null) {
            handler(this, e);
        }
    }
}
```

The derived class can choose not to call the base implementation of the method in its override. Be prepared for this by not including any processing in the method that is required for the base class to work correctly.

✓ **DO** take one parameter to the protected method that raises an event. The parameter should be named `e` and should be typed as the event argument class.

```
protected virtual void OnAlarmRaised(AlarmRaisedEventArgs e){
    AlarmRaised?.Invoke(this, e);
}
```

✗ **DO NOT** pass `null` as the sender when raising a nonstatic event.

✓ **DO** pass `null` as the sender when raising a static event.

```
EventHandler<EventArgs> handler = ....;
if (handler!=null) handler(null,...);
```

X DO NOT pass null as the event data parameter when raising an event.

You should pass `EventArgs.Empty` if you don't want to pass any data to the event handling method. Developers expect this parameter not to be null.

✓ CONSIDER raising events that the end user can cancel. This guideline applies only to pre-events.

Use `System.ComponentModel.CancelEventArgs` or its subclass as the event argument to allow the end user to cancel events. For example, `System.Windows.Forms.Form` raises a `Closing` event before a form closes. The user can cancel the close operation, as shown in the following example:

```
void ClosingHandler(object sender, CancelEventArgs e) {  
    e.Cancel = true;  
}
```

5.5 Field Design

The principle of encapsulation is one of the most important notions in object-oriented design. This principle states that data stored inside an object should be accessible only to that object.

A useful way to interpret the principle is to say that a type should be designed so that changes to fields of that type (name or type changes) can be made without breaking code other than for members of the type. This interpretation immediately implies that all fields must be private.

We exclude constant and static read-only fields from this strict restriction, because such fields, almost by definition, are never required to change.

X DO NOT provide instance fields that are public or protected.

You should provide properties for accessing fields instead of making them public or protected.

Very trivial property accessors, as shown here, can be inlined by the Just-in-Time (JIT) compiler and provide performance on par with that of accessing a field.

```
public struct Point{  
    private int x;  
    private int y;  
  
    public Point(int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
  
    public int X {  
        get{ return x; }  
    }  
  
    public int Y{  
        get{ return y; }  
    }  
}
```

By not exposing fields directly to the developer, the type can be versioned more easily, and for the following reasons:

- A field cannot be changed to a property while maintaining binary compatibility.
- The presence of executable code in get and set property accessors allows later improvements, such as on-demand creation of an object for usage of the property, or a property change notification.

CHRIS ANDERSON Fields are the bane of my existence. Because reflection treats fields and properties as different constructs, any system that walks an object graph must special-case both of them. Data binding always looks only at properties; runtime serialization looks only at fields. The fact that we didn't unify these two (treating properties as smart fields) is definitely a regret of mine. However, as Rico Mariani would say, properties have additional overhead. And as I would say, fields don't version. You can never promote a field to be a property when you want to add validation, change notification, put it in an interface, and so on. Fields are private data; they are the stores behind your public contract, which should be implemented with properties, methods, and events.

■ **JEFFREY RICHTER** Personally, I always make my fields private. I don't even expose fields as internal, because doing so would give me no protection from code in my own assembly.

✓ **DO** use constant fields for constants that will never change.

The compiler burns the values of `const` fields directly into calling code. Therefore, `const` values can never be changed without the risk of breaking compatibility.

```
public struct Int32 {  
    public const int MaxValue = 0x7fffffff;  
    public const int MinValue = unchecked((int)0x80000000);  
}
```

✓ **CONSIDER** using `public static readonly` fields for predefined object instances.

If there are predefined instances of the type, declare them either as `public static readonly` fields or as get-only properties on the type itself.

```
public struct Color{  
    public static readonly Color Red = new Color(0x0000FF);  
    public static readonly Color Green = new Color(0x00FF00);  
    public static readonly Color Blue = new Color(0xFF0000);  
    ...  
}
```

When a type has a significant number of predefined instances, but few are used in any application, you may prefer to use get-only properties. Because the .NET Runtime will initialize all of the static fields at the same time, a type with many predefined instances can suffer from visible start-up performance. When exposed as properties, the predefined values can be built on demand, like the `System.Text.Encoding.UTF8` value:

```
public static Encoding UTF8 {  
    get {  
        return _utf8Encoding ??= new UTF8Encoding();  
    }  
}
```

JEREMY BARTON While the example here shows `Color.Red` as a static readonly field, the real `System.Drawing.Color.Red` is a get-only property. For value types, like `Color`, using an auto-property with a default value is just as good as a field after the JIT inlines the call:

```
public static Color Red { get; } = new Color(0x0000FF);
```

X DO NOT assign instances of mutable types to public or protected readonly fields.

A mutable type is a type with instances that can be modified after they are instantiated. For example, arrays, most collections, and streams are mutable types, but `System.Int32`, `System.Uri`, and `System.String` are all immutable. The `readonly` modifier on a reference type field prevents the instance stored in the field from being replaced, but it does not prevent the field's instance data from being modified by calling members changing the instance. The following example shows how it is possible to change the value of an object referred to by a `readonly` field.

```
public class SomeType {
    public static readonly int[] Numbers = new int[10];
}
...
SomeType.Numbers[5] = 10; // changes a value in the array
```

JOE DUFFY The real distinction here is between *deep* versus *shallow* immutability. A deeply immutable type is one whose fields are all `readonly`, and each field is of a type that itself is also deeply immutable. This ensures that a whole object graph is transitively immutable. While deep immutability is certainly the most useful kind, shallow immutability can also be useful. What this guideline is trying to protect you from is believing you've exposed a deeply immutable object graph when in fact it is shallow, and then writing code that assumes the whole graph is immutable.

5.6 Extension Methods

Extension methods are a language feature that allows static methods to be called using instance method call syntax. These methods must take at least one parameter, which represents the instance the method is to operate on. For example, in C#, you declare extension methods by placing the `this` modifier on the first parameter.

```
public static class StringExtensions {  
    public static bool IsPalindrome(this string s){  
        ...  
    }  
}
```

This extension method can be called as follows:

```
if("hello world".IsPalindrome()){  
    ...  
}
```

The class that defines such extension methods must be declared as a static class. To use extension methods, you must import the namespace defining the class that contains the extension method.

X AVOID frivolously defining extension methods, especially on types outside your library or framework.

If you do own source code of a type, consider using regular instance methods instead. If you don't own the type, but you want to add a method, be very careful. Liberal use of extension methods has the potential to clutter APIs with types that were not designed to have these methods.

Another thing to consider is that extension methods are a compile-time facility, and not all languages provide support for them. Callers using languages without extension method support will have to use the regular static method call syntax to call your extension methods.

There are, of course, scenarios in which extension methods should be employed. These are outlined in the guidelines that follow.

■ **BRIAN PEPIN** When working on the implementation of the WPF designer for Visual Studio, we made careful use of extension methods. In many places in our code we needed to identify properties based on the values of certain attributes defined for XAML. For example, the content property of a control can be identified by looking for a `ContentPropertyAttribute` on the type, and then using the value of that attribute to look for a property on the type. Having this code everywhere was cumbersome, so we wrote a set of internal extension methods to `Type` to do the work for us:

```
internal static class XamlTypeExtensions {  
    internal static PropertyInfo GetContentProperty(this Type source) {  
        var attrs = source.GetCustomAttributes(typeof  
            (ContentPropertyAttribute), true);  
        if (attrs.Length > 0) {  
            return source.GetProperty(((ContentPropertyAttribute)  
                attrs[0]).Name);  
        }  
        return null;  
    }  
}
```

Centralizing these types of operations in a set of internal extension methods allowed our code to be much cleaner and allowed us to optimize implementations in one place.

✓ **CONSIDER** using extension methods to provide helper functionality relevant to every implementation of an interface.

The primary example of a feature using this guidance is the `System.Linq` namespace. The methods defined on `System.Linq.Enumerable`—such as `Select`, `OrderBy`, `First`, and `Last`—are both *capable of* operating on any `IEnumerable<T>` implementation and *relevant to* any `IEnumerable<T>` instance.

Many of the methods on the `Enumerable` class do runtime type checks to use more efficient implementations. For example, `IList<T>` has a positional indexer and a `Count` property, so the `Last` method can call that property to execute much faster than doing a `foreach` over

everything in the `IEnumerable<T>`. This highlights the main drawback of using extension methods for interface functionality—the lack of polymorphism. If polymorphism is required for the helper functionality to be practical, then providing functionality via an extension method will just set your users up for a bad experience.

■ **RICO MARIANI** The value of this guideline cannot be overstated. Extension methods provide you with a way to give interfaces not just one default implementation but as many as you need, and you can choose between them simply by bringing the right namespace(s) into your lexical scope with `using`. However, this could be easily abused in the same way that complex `#include` combinations in C++ can make the final code hard to read.

Another interesting feature of this manner of adding functionality to a class is that you could potentially partition (extension) methods of the same class into different assemblies for performance reasons. Again, this should not be done lightly, because you could cause great confusion.

This is one of the many times I like to quote from *Spiderman*: “With great power comes great responsibility.”

✓ **CONSIDER** using extension methods when an instance method would introduce an unwanted dependency.

For example, when the `ReadOnlySpan<T>` type was originally introduced, it was in a stand-alone library. Rather than waiting for an opportunity for `String` to depend on `ReadOnlySpan<char>` for the `AsSpan()` method and having a strong dependency between these types, the `AsSpan` method for `String` was written as an extension method in the library that defined `ReadOnlySpan<T>`.

✓ **CONSIDER** using generic extension methods when an instance method on a generic type is not well defined for all possible type parameters.

For example, a collection of integers has a well-defined behavior for a method like `GetMinimumValue`, but the method is less well defined when the collection contains `HttpContext` values. By using a generic extension method, the `GetMinimumValue` method can specify more restrictive type constraints than the collection generic type.

```
// The collection type has no type restrictions
public class SomeCollection<T> { ... }

// A non-generic extensions class
public sealed class SomeCollectionComparisons {
    // This method is only defined when the collection is
    // of comparable values
    public static T GetMinimumValue<T>(
        this SomeCollection<T> source) where T : IComparable<T> {
        ...
    }
}
```

In addition to making use of generic constraints to restrict applicable types, some methods only make sense on specific generic instantiations, such as the `IsWhiteSpace` method for `ReadOnlySpan<char>`.

```
public static bool IsWhiteSpace(this ReadOnlySpan<char> span) { ... }
```

- ✓ **DO** throw an `ArgumentNullException` when the `this` parameter in an extension method is `null`.

Extension methods are just static methods with some extra “syntactic sugar” in C# and some other languages, and they should perform argument validation like any other method. The guidance for `NullReferenceException` from section 7.3.5 says it should never be thrown from a publicly callable method, and that also applies to extension methods even when invoked in “instance” syntax.

Silently operating on `null` inputs, and throwing no exception at all, generally confuses developers debugging “impossible” states in their program—when the actually failing code is “clearly” unreachable, because if this was a `null` value the previous statement would have thrown instead. Even if the method would be well behaved with a `null` input as a simple static method, once the `this` modifier is added the method should throw an exception for a `null` first argument.

```
// This should throw an ArgumentNullException
SomeExtensions.SomeMethod(null);

// This is the same at runtime,
// so it should throw the same kind of exception.
((SomeType)null).SomeMethod();
```

X AVOID defining extension methods on System.Object.

VB users will not be able to call such methods on object references using the extension method syntax, because VB does not support calling such methods. In VB, declaring a reference as Object forces all method invocations on it to be late bound (the actual member called is determined at runtime), while bindings to extension methods are determined at compile-time (early bound). For example:

```
// C# declaration of the extension method
public static class SomeExtensions{
    static void SomeMethod(this object o){...}
}

' VB will fail to compile as VB.NET does not support calling extension
' methods on reference types as Object
Dim o As Object = ...
o.SomeMethod();
```

VB users will have to call the method using the regular static method call syntax.

```
SomeExtensions.SomeMethod(o)
```

Note that this guideline applies to other languages where the same binding behavior is present, or where extension methods are not supported.

✓ CONSIDER naming the type that holds extension methods for its functionality—for example, use “Routing” instead of “[ExtendedType] Extensions.”

In other words, design the static class that holds the extension methods as you would any other static class; then make some of the methods be extension methods, when appropriate.

This provides for a much better experience in languages without extension syntax, and allows the static class to be the correct choice for where a related method should go even when it can’t be used in extension syntax.

```
// Bad: This class has too many purposes
public static class SomeCollectionExtensions {
```

```
public static T GetMinimumValue<T>(
    this SomeCollection<T> source) where T : IComparable<T> { ... }

public static T GetMaximumValue<T>(
    this SomeCollection<T> source) where T : IComparable<T> { ... }

public static SomeCollection<TNested> Flatten(
    this SomeCollection<TOuter> source)
    where TOuter : IEnumerable<TNested> { ... }
}

// Good: These classes each have one purpose
public static class SomeCollectionComparisons {
    public static T GetMinimumValue<T>(
        this SomeCollection<T> source) where T : IComparable<T> { ... }

    public static T GetMaximumValue<T>(
        this SomeCollection<T> source) where T : IComparable<T> { ... }
}

public static class SomeCollectionNestedOperations {
    public static SomeCollection<TNested> Flatten(
        this SomeCollection<TOuter> source)
        where TOuter : IEnumerable<TNested> { ... }
}
```

■ ANTHONY MOORE Many developers love the power and expressivity of extension methods and are tempted to use them in ways that clearly violate these guidelines. In many ways, this seems a lot like operator overloading in the early days of C++, where there was initially vigorous and “creative” use of operators for nonintuitive cases, including the runtime libraries of the language.

Over time the painful lesson was learned, and operator overloading was one of the features that many companies using C++ all but banned. It provides a one-off saving of some typing at the expense of future transparency of the code, and code is read a lot more times than it is written. Using extension methods for reasons other than the scenarios for interfaces or layering is very similar to this situation.

As an example, I’ve had some frustrating experiences with using APIs that enthusiastically use extensions. On a multilanguage project I found that a pleasant and intuitive object model in one language was awkward in a language without support for extension methods. I’ve also had cases where I was misled about the type of an instance because some local code was injecting extensions into it.

Please use extension methods with caution.

X DO NOT put extension methods in the same namespace as the extended type unless the intention is to add methods to interfaces, for generic type restriction, or for dependency management.

Of course, in the case of dependency management, the type with the extension methods would be in a different assembly.

X AVOID defining two or more extension methods with the same signature, even if they reside in different namespaces.

For example, if two different namespaces defined the same extension method on the same type, it would be impossible to import both namespaces in the same file. The compiler would report an ambiguity if one of the methods was called.

```
namespace A {  
    public static class AExtensions {  
        public static void ExtensionMethod(this Foo foo){...}  
    }  
}  
namespace B {  
    public static class BExtensions {  
        public static void ExtensionMethod(this Foo foo){...}  
    }  
}
```

RICO MARIANI There can be reasons to violate this guideline. For example, there might be a framework that allowed developers to choose

```
using SomeType.Routing.SpeedOptimized;
```

or

```
using SomeType.Routing.SpaceOptimized;
```

In this case, both namespaces could offer the same services, and developers simply choose which flavor to use.

The key here is that a natural mutual exclusion exists for a given consumer of the extension so that it avoids the ambiguity. If you tried to use both, you would want to see the error so that you would realize you have a conflict.

■ **MIRCEA TROFIN** Such a situation may occur when using a mix of third-party libraries. To resolve it, import in a file only one namespace defining extension methods on a type, and use fully qualified static method calls for extension methods defined on the same type in a different namespace.

```
using A;
...
T someObj = ...
someObj.ExtentionMethod(); //this calls AExtensions.ExtensionMethod
// To avoid compilation errors,
// we call explicitly the extension method defined in namespace B
B.BExtensions.ExtensionMethod(someObj);
```

✓ **CONSIDER** defining extension methods in the same namespace as the extended type if the type is an interface and if the extension methods are meant to be used in most or all cases.

✓ **CONSIDER** defining extension methods in the same namespace as the extended type if the type is generic and if the extension methods apply stronger type parameter restrictions than the extended type does.

The following example uses an extension method to provide a `Trim` method for `ReadOnlyMemory<char>` without defining it for all `ReadOnlyMemory<T>` types.

```
public partial struct ReadOnlyMemory<T> { ... }

public static partial class MemoryExtensions {
    public static ReadOnlyMemory<char> Trim(
        this ReadOnlyMemory<char> memory) { ... }
}
```

This technique can also be used with a generic method with generic restrictions.

```
public static partial class CollectionDisposer {
    public static void DisposeAll<T>(
        this List<T> list) where T : IDisposable { ... }
}
```

X DO NOT define extension methods implementing a feature in namespaces normally associated with other features. Instead, define them in the namespace associated with the feature they belong to.

Similar to the guidance that the name of the class containing the extension method definitions should reflect its functionality, choose the namespace of the defining class to match the functionality instead of based on the fact that it is providing extension methods.

Remember that not all languages can use extension invocation, and consider the static method invocation experience.

■ PHIL HAACK Section 2.2.4.1 covers the topic of layering in namespaces, which I think applies well to extension methods.

One scenario I've seen is to put functionality that is more advanced or esoteric in a separate namespace. That way, for core scenarios, these extra methods do not "pollute" the API. Developers who know about the namespace, or need these alternative scenarios, can add the namespace and gain access to these extra methods.

The drawback, of course, is that this results in a "hidden" API that is not very discoverable.

5.7 Operator Overloads

Operator overloads allow framework types to appear as if they were built-in language primitives. The following snippet shows some of the most important operator overloads defined by `System.Decimal`.

```
public struct Decimal {
    public static Decimal operator+(Decimal d);
    public static Decimal operator-(Decimal d);
    public static Decimal operator++(Decimal d);
    public static Decimal operator--(Decimal d);
    public static Decimal operator+(Decimal d1, Decimal d2);
    public static Decimal operator-(Decimal d1, Decimal d2);
    public static Decimal operator*(Decimal d1, Decimal d2);
    public static Decimal operator/(Decimal d1, Decimal d2);
    public static Decimal operator%(Decimal d1, Decimal d2);
```

```
public static bool operator==(Decimal d1, Decimal d2);
public static bool operator!=(Decimal d1, Decimal d2);
public static bool operator<(Decimal d1, Decimal d2);
public static bool operator<=(Decimal d1, Decimal d2);
public static bool operator>(Decimal d1, Decimal d2);
public static bool operator>=(Decimal d1, Decimal d2);

public static implicit operator Decimal(int value);
public static implicit operator Decimal(long value);
public static explicit operator Decimal(float value);
public static explicit operator Decimal(double value);
...
public static explicit operator int(Decimal value);
public static explicit operator long(Decimal value);
public static explicit operator float(Decimal value);
public static explicit operator double(Decimal value);
...
}
```

Although allowed and useful in some situations, operator overloads should be used cautiously. Operator overloading is often abused, such as when framework designers use operators for operations that should be simple methods. The following guidelines should help you decide when and how to use operator overloading.

CHRIS SELLS Operator overloads don't show up in IntelliSense, so for most developers, they don't exist. Keep that in mind before you go through the complicated exercise of making sure your operators work and act like the same operators on the built-in types.

KRZYSZTOF CWALINA Normally, overloading is understood as having more than one member with the same name but different parameters defined by one type. In the case of operators, they are said to be overloaded despite the fact that there might be only one such operator member on a type. This terminology can be quite confusing, but there is a reason for the seemingly confusing name.

Overloading happens when an addition of a member means the compiler will have to use the argument list, in addition to the member name, to resolve which member should be called. So, for example, the moment you add an `operator+` to a custom type, the compiler has to know the types of arguments (operands) of the following call to know which operator needs to be called.

```
public struct BigInteger {  
    public static BigInteger operator+(  
        BigInteger left, BigInteger right) { ... }  
}  
...  
// if x and y are both BigIntegers the operator above will be used  
object result = x + y;
```

✗ **AVOID** defining operator overloads, except in types that should feel like primitive (built-in) types.

✓ **CONSIDER** defining operator overloads in a type that should feel like a primitive type.

For example, `System.String` has `operator==` and `operator!=` defined.

✓ **DO** define operator overloads in structs that represent numbers (such as `System.Decimal`).

✗ **DO NOT** be cute when defining operator overloads.

Operator overloading is useful in cases in which it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one `DateTime` from another `DateTime` and get a `TimeSpan`. However, it is not appropriate to use the logical union operator to combine two database queries, or to use the shift operator to write to a stream.

✗ **DO NOT** provide operator overloads unless at least one of the operands is of the type defining the overload.

In other words, operators should operate on types that define them. The C# compiler enforces this guideline.

```
public struct RangedInt32 {
    public static RangedInt32 operator-(RangedInt32 left, RangedInt32 right);
    public static RangedInt32 operator-(RangedInt32 left, int right);
    public static RangedInt32 operator-(int left, RangedInt32 right);

    // The following would violate the guideline and in fact does not
    // compile in C#.
    // public static RangedInt32 operator-(int left, int right);
}
```

✓ **DO** overload operators in a symmetric fashion.

For example, if you overload `operator==`, you should also overload `operator!=`. Similarly, if you overload `operator<`, you should also overload `operator>`, and so on.

■ **RICO MARIANI** More generally, if you haven't defined a whole family of overloads (because maybe they don't all make sense), there's a good chance that operator overloading isn't really the best way to express your class. Remember—there are no bonus points for using fewer characters in your method calls.

✓ **DO** provide methods with friendly names that correspond to each overloaded operator.

Many languages do not support operator overloading. For this reason, it is recommended that types that overload operators include a secondary method with an appropriate domain-specific name that provides equivalent functionality. The operator method is generally implemented as a call to the named method, especially if the named method is virtual. The following example illustrates this point.

```
public struct DateTimeOffset {
    public static TimeSpan operator-
        (DateTimeOffset left, DateTimeOffset right)
        => left.Subtract(right);

    public TimeSpan Subtract(DateTimeOffset value) { ... }
}
```

Table 5-1 lists the .NET operators and the corresponding friendly method names.

TABLE 5-1: Operators and Corresponding Method Names

C# Operator Symbol	Metadata Name	Friendly Name
N/A	op_Implicit	To<TypeName>/From<TypeName>
N/A	op_Explicit	To<TypeName>/From<TypeName>
+ (binary)	op_Addition	Add
- (binary)	op_Subtraction	Subtract
* (binary)	op_Multiply	Multiply
/	op_Division	Divide
%	op_Modulus	Mod or Remainder
^	op_ExclusiveOr	Xor
& (binary)	op_BitwiseAnd	BitwiseAnd
	op_BitwiseOr	BitwiseOr
&&	op_LogicalAnd	And
	op_LogicalOr	Or
=	op_Assign	Assign
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
N/A	op_SignedRightShift	SignedRightShift
N/A	op_UnsignedRightShift	UnsignedRightShift

C# Operator Symbol	Metadata Name	Friendly Name
<code>==</code>	<code>op_Equality</code>	<code>Equals</code>
<code>!=</code>	<code>op_Inequality</code>	<code>Equals</code>
<code>></code>	<code>op_GreaterThan</code>	<code>CompareTo</code>
<code><</code>	<code>op_LessThan</code>	<code>CompareTo</code>
<code>>=</code>	<code>op_GreaterThanOrEqual</code>	<code>CompareTo</code>
<code><=</code>	<code>op_LessThanOrEqual</code>	<code>CompareTo</code>
<code>*=</code>	<code>op_MultiplicationAssignment</code>	<code>Multiply</code>
<code>-=</code>	<code>op_SubtractionAssignment</code>	<code>Subtract</code>
<code>^=</code>	<code>op_ExclusiveOrAssignment</code>	<code>Xor</code>
<code><<=</code>	<code>op_LeftShiftAssignment</code>	<code>LeftShift</code>
<code>%=</code>	<code>op_ModulusAssignment</code>	<code>Mod</code>
<code>+=</code>	<code>op>AdditionAssignment</code>	<code>Add</code>
<code>&=</code>	<code>op_BitwiseAndAssignment</code>	<code>BitwiseAnd</code>
<code> =</code>	<code>op_BitwiseOrAssignment</code>	<code>BitwiseOr</code>
<code>,</code>	<code>op_Comma</code>	<code>Comma</code>
<code>/=</code>	<code>op_DivisionAssignment</code>	<code>Divide</code>
<code>--</code>	<code>op_Decrement</code>	<code>Decrement</code>
<code>++</code>	<code>op_Increment</code>	<code>Increment</code>
<code>- (unary)</code>	<code>op_UnaryNegation</code>	<code>Negate</code>
<code>+ (unary)</code>	<code>op_UnaryPlus</code>	<code>Plus</code>
<code>~</code>	<code>op_OnesComplement</code>	<code>OnesComplement</code>

5.7.1 Overloading Operator ==

Overloading operator `==` is quite complicated. The semantics of the operator need to be compatible with several other members, such as `Object.Equals`. For more information on this subject, see sections 8.9.1 and 8.13.

5.7.2 Conversion Operators

Conversion operators are unary operators that allow conversion from one type to another. The operators must be defined as static members on either the operand or the return type. There are two types of conversion operators: implicit and explicit.

```
public struct RangedInt32 {  
    public static implicit operator int(RangedInt32 value){ ... }  
    public static explicit operator RangedInt32(int value) { ... }  
    ...  
}
```

X DO NOT provide a conversion operator if such a conversion is not clearly expected by the end users.

Ideally, you should have some user research data showing that the conversion is expected, or some prior art examples where a similar type needed such conversion.

X DO NOT define conversion operators outside of a type's domain.

For example, `Int32`, `Int64`, `Double`, and `Decimal` are all numeric types, whereas `DateTime` is not. Therefore, there should be no conversion operator to convert `Int64 (long)` to a `DateTime`. A constructor is preferred in such a case.

```
public struct DateTime {  
    public DateTime(long ticks){ ... }  
}
```

- X DO NOT** provide an implicit conversion operator if the conversion is potentially lossy.

For example, there should not be an implicit conversion from Double to Int32 because Double has a wider range than Int32. An explicit conversion operator can be provided even if the conversion is potentially lossy.

- X DO NOT** provide an implicit conversion operator if the conversion requires nontrivial work.

Since implicit conversions are usually not visible in calling code, the potential performance costs of the method will be hidden from anyone reading the code. If a conversion requires more than a small amount of constant-time work, it should instead be exposed as a named method.

- X DO NOT** throw exceptions from implicit conversion operators.

It is very difficult for end users to understand what is happening, because they might not be aware that a conversion is taking place.

■ STEPHEN TOUB From a performance perspective, implicit conversions should also avoid allocations or otherwise having nontrivial overhead. Just as with boxing, if an implicit conversion is expensive, it's likely developers will end up paying these costs without realizing it.

- ✓ DO** throw System.InvalidCastException if a call to a cast operator results in a lossy conversion and the contract of the operator does not allow lossy conversions.

```
public static explicit operator RangedInt32(long value) {
    if (value < Int32.MinValue || value > Int32.MaxValue) {
        throw new InvalidCastException();
    }
    return new RangedInt32((int)value, Int32.MinValue, Int32.MaxValue);
}
```

5.7.3 Inequality Operators

The inequality operators (`<`, `<=`, `>`, `>=`) allow for concise mathematical expressions to be written in code. The use of these operators on custom types should be consistent with the built-in operators. The strict inequality operator (`!=`) is specifically associated with the equality operator (`==`), discussed in section 8.13.

- ✓ **DO** only implement inequality operators on types that implement `IComparable<T>`.
- ✓ **DO** implement inequality operators consistent with the `IComparable<T>` implementation.

Because `IComparable<T>` and the inequality operators are performing similar roles, anything defining the operators automatically has an easy basis for implementing the interface.

```
public static bool operator <(SomeType left, SomeType right) =>
    left.CompareTo(right) < 0;
```

- ✓ **DO** implement operator `<=` on types that implement operator `<` and `IEquatable<T>`.

When both “less than” and “equal” are defined, then “less than or equal” should be well defined.

- ✓ **DO** return a Boolean value from custom inequality operators.
- ✓ **DO** define inequality operators consistent with their mathematical properties.

Custom inequality operators should all behave as they do on built-in numeric types:

- The *transitive* property: If $a < b$ and $b < c$, then $a < c$.
- The *reversal* property: If $a < b$, then $b > a$.
- The *irreflexive* property: If $a < a$ is false, then $a > a$ is false.
- The *asymmetric* property: If $a < b$ is true, then $b < a$ is false.

Following this guideline allows developers using your operators to write code, and their reviewers to read it, and apply the same logical

analysis they would to your type as they would to the built-in numeric types.

Using the inequality operators for a projective map, or for any purpose other than Boolean mathematical comparisons, makes code harder to reason about.

```
// Almost everyone believes that the type of x here is Boolean.  
// Therefore it should be for any inequality operator you define.  
var x = a < b;
```

X AVOID defining inequality operators against different types.

The *reversal* property of inequality says that for any $a < b$ that returns true, $b > a$ should also return true. Therefore, to be consistent with the expected inequality behaviors, you need to define both `ThisType < ThatType` and `ThatType > ThisType`. The *asymmetric* property, and the guidance for implementing operators symmetrically, means that you also have to define `ThisType > ThatType` and `ThatType < ThisType`. The *transitive* property says you also have to define `ThisType > ThisType` and `ThisType < ThisType` if you haven't already. Finally, the guidance about implementing `IComparable<T>` says that both `ThisType` and `ThatType` should be `IComparable<T>` against each other (and themselves)—a task that can be accomplished only when `ThisType` and `ThatType` are in the same assembly.

Rather than implementing six different operators (and even more if \leq is supported), two interfaces, and two methods, consider making use of a conversion operator (section 5.7.2) or a method or property that converts one type to another. For example, `DateTimeOffset` and `DateTime` can be compared via inequality:

```
DateTimeOffset a = GetDateTimeOffset();  
DateTime b = GetDateTime();  
  
// This compiles and behaves as expected  
if (a < b) {  
    ...  
}
```

`DateTimeOffset` doesn't define an operator `<` to compare against a `DateTime`, but instead has an implicit conversion operator to convert a `DateTime` into a `DateTimeOffset` and an operator `<` to compare two `DateTimeOffset` values:

```
public partial struct DateTimeOffset {  
    public static implicit operator DateTimeOffset(DateTime dateTime) {...}  
  
    public static operator <(DateTimeOffset left, DateTimeOffset right){...}  
}
```

`DateTimeOffset` values can be compared to `DateTime` values (the other direction) by using the `DateTimeOffset.UtcDateTime` property.

```
public partial struct DateTimeOffset {  
    public DateTime UtcDateTime { get { ... } }  
}
```

5.8 Parameter Design

This section provides broad guidelines on parameter design, including sections with guidelines for checking arguments. In addition, you should refer to the parameter naming guidelines in Chapter 3.

- ✓ **DO** use the least derived parameter type that provides the functionality required by the member.

For example, suppose you want to design a method that enumerates a collection and prints each item to the console. Such a method should take `IEnumerable<T>` as the parameter, not `List<T>` or `IList<T>`, for example.

```
public void WriteItemsToConsole(IEnumerable<object> items) {  
    foreach(object item in items) {  
        Console.WriteLine(item.ToString());  
    }  
}
```

None of the specific `IList<T>` members needs to be used inside the method. Typing the parameter as `IEnumerable<T>` allows the end user

to pass collections that implement only `IEnumerable<T>` and not `IList<T>`.

■ **RICO MARIANI** Interface isn't everything. If your algorithm needs a more specialized type to get decent performance, there's no point in pretending that you only need a base type. Best to make your needs clear—ask for the type that is required to get the designed behavior. Likewise, if your method requires certain thread-safety or security features provided by subtypes, insist on those features in the contract. There's no point in allowing users to make calls that won't work.

X DO NOT use reserved parameters.

If more input to a member is needed in some future version, a new overload can be added. For example, it would be bad to reserve a parameter as follows:

```
public void Method(SomeOption option, object reserved);
```

It is better to simply add a parameter to a new overload in a future version, as shown in the following example:

```
public void Method(SomeOption option);

// added in a future version
public void Method(SomeOption option, string path);
```

X DO NOT have publicly exposed methods that take pointers, arrays of pointers, or multidimensional arrays as parameters.

Pointers and multidimensional arrays are relatively difficult to use properly. In almost all cases, APIs can be redesigned to avoid taking these types as parameters.

■ **RICO MARIANI** Sometimes people try these sorts of things to squeeze out more performance. But remember that you aren't helping anyone if it's fast but nearly impossible to use correctly.

- ✓ **DO** place all out parameters following all of the by-value and ref parameters (excluding parameter arrays), even if it results in an inconsistency in parameter ordering between overloads (see section 5.1.1).

The out parameters can be seen as extra return values, and grouping them together makes the method signature easier to understand. For example:

```
public struct DateTimeOffset {
    public static bool TryParse(string input, out DateTimeOffset result);
    public static bool TryParse(string input, IFormatProvider
formatProvider, DateTimeStyles styles, out DateTimeOffset result);
}
```

- ✓ **DO** be consistent in naming parameters when overriding members or implementing interface members.

This better communicates the relationship between the methods.

```
public interface IComparable<T> {
    int CompareTo(T other);
}

public class Nullable<T> : IComparable<Nullable<T>> {
    // correct
    public int CompareTo(Nullable<T> other) { ... }
    // incorrect
    public int CompareTo(Nullable<T> nullable) { ... }
}

public class Object {
    public virtual bool Equals(object obj) { ... }
}

public class String {
    // correct; the parameter to the base method is called 'obj'
    public override bool Equals(object obj) { ... }

    // incorrect; the parameter should be called 'obj'
    public override bool Equals(object value) { ... }
}
```

5.8.1 Choosing Between Enum and Boolean Parameters

A framework designer often must decide when to use enums and when to use Booleans for parameters. In general, you should favor using enums if doing so improves the readability of the client code, especially in commonly used APIs. If using enums would add unneeded complexity and actually hurt readability, or if the API is very rarely used, use Booleans instead.

- ✓ **DO** use enums if a member would otherwise have two or more Boolean parameters.

Enums are much more readable when it comes to books, documentation, source code reviews, and so on. For example, look at the following method call:

```
Stream stream = File.Open("foo.txt", true, false);
```

This call gives the reader no context within which to understand the meaning of `true` and `false`. The call would be much more usable if it were to use enums, as follows:

```
Stream stream = File.Open("foo.txt", CasingOptions.CaseSensitive,  
FileMode.Open);
```

■ **ANTHONY MOORE** Some have asked why we don't have a similar guideline for integers, doubles, and so on. Should we find a way to "name" them as well? There is a big difference between numeric types and Booleans. You almost always use constants and variables to pass numeric values around, because it is good programming practice and you don't want to have "magic numbers." However, if you take a look at real-life source code, this is almost never true of Booleans. Eighty percent of the time a Boolean argument is passed in as a literal constant, and its intention is to turn a piece of behavior on or off. We could alternatively try to establish a coding guideline that you should never pass a literal value to a method or constructor, but I don't think it would be practical. I certainly don't want to define a constant for each Boolean parameter I'm passing in.

■ **JON PINCUS** Methods with two Boolean parameters, like the one in the preceding example, allow developers to inadvertently switch the arguments, and the compiler and static analysis tools can't help you find this error. Even with just one parameter, I tend to believe it's still somewhat easier to make a mistake with Booleans—let's see, does `true` mean case-insensitive or case-sensitive?

■ **STEVEN CLARKE** The worst example of an unreadable Boolean parameter that I had to deal with was the `CWnd::UpdateData` method in MFC. It takes a Boolean that indicates whether a dialog is being initialized or data is being retrieved. I always had to look up whether to pass `true` or `false` to this method each time I called it. Likewise, each time I read code that called the method, I had to look it up to see what it meant.

■ **STEPHEN TOUB** This guidance was written before C# added support for naming arguments at the call site. It's true that the code

```
File.Open("foo.txt", true, false);
```

gives no context with which to understand the meaning of those Boolean arguments. But in the code

```
File.Open("foo.txt", caseSensitive: true, create: false);
```

the name of the arguments makes it clear. I still agree with the general guidance, but the ramifications of not following it aren't as stark as they once were.

X DO NOT use Boolean parameters unless you are absolutely sure there will never be a need for more than two values.

Enums give you some room for future addition of values, but you should be aware of all the implications of adding values to enums, as described in section 4.8.2.

■ **BRAD ABRAMS** We have seen a couple of places in the framework where we added a Boolean in one version, and in the next version we were forced to add another Boolean option to account for what could have been a foreseeable change. Don't let this happen to you: If there is even a slight possibility of needing more options in the future, use an enum now.

✓ **CONSIDER** using Booleans for constructor parameters that are truly two-state values and are simply used to initialize Boolean properties.

■ **ANTHONY MOORE** An interesting clarification of this guideline for constructor parameters that map onto properties is that if the value is typically set in the constructor, an enum value is better. If the value is typically set using the property setter, a Boolean value is better. This thinking helped us clarify a CodeDom work item to add `IsGlobal` on `CodeTypeReference`. In this case, it should be an enum because it is typically set in the constructor, but the `IsPartial` property on `CodeTypeDeclaration` should be a Boolean.

5.8.2 Validating Arguments

Rigorous checks on arguments passed to members are a crucial element of modern reusable libraries. Although argument checks might have a slight impact on performance, end users are in general willing to pay the price for the benefit of better error reporting, which becomes possible if arguments are validated as high on the call stack as possible.

■ **RICO MARIANI** The key words for me here are "high on the call stack." When you get low in the call stack, the amount of work that the functions are doing is so small that the argument validation becomes a significant, even dominant, factor in performance. At that point it's a lousy deal. Where do I tend to just let the runtime throw exceptions rather than prevalidate? Typically, in comparison and hashing functions—you can expect those to be called often and with tight requirements. Regular measurements will help you spot any validations that are too low in the stack.

- ✓ **DO** validate arguments passed to public, protected, or explicitly implemented members. Throw `System.ArgumentException`, or one of its subclasses, if the validation fails.

```
public class StringCollection : IList {  
    int IList.Add(object item){  
        string str = item as string;  
        if(str==null) throw new ArgumentNullException(...);  
        return Add(str);  
    }  
}
```

Note that the actual validation does not necessarily have to happen in the public or protected member itself. It could happen at a lower level in some private or internal routine. The main point is that the entire surface area that is exposed to the end users checks the arguments.

- ✓ **DO** throw `ArgumentNullException` if a null argument is passed and the member does not support null arguments.

- ✓ **DO** validate enum parameters.

Do not assume enum arguments will be in the range defined by the enum. The CLR allows casting any integer value into an enum value even if the value is not defined in the enum.

```
public void PickColor(Color color) {  
    if(color > Color.Black || color < Color.White){  
        throw new ArgumentOutOfRangeException(...);  
    }  
    ...  
}
```

■ **VANCE MORRISON** I am not a believer in always validating [Flags] enums. Typically, all you can usefully do for them is to confirm that “unused” flags are not used, but this also makes your code unnecessarily fragile (now adding a new flag is a breaking change!). Your code “naturally” would simply ignore unused flags, and I think that is a reasonable semantic. However, if certain combinations of flags are illegal, that should certainly be checked and an appropriate error thrown.

X DO NOT use `Enum.IsDefined` for enum range checks.

BRAD ABRAMS There are really two problems with `Enum.IsDefined`. First, it loads reflection and a bunch of cold type metadata, making it a surprisingly expensive call. Second, there is a versioning issue here. Consider an alternative way to validate enum values.

```
public void PickColor(Color color) {
    // the following check is incorrect!
    if (!Enum.IsDefined (typeof(Color), color)) {
        throw new InvalidEnumArgumentException(...);
    }
    // issue: never pass a negative color value
    NativeMethods.SetImageColor (color, byte[] image);
}
// call site
Foo.PickColor ((Color) -1); //throws InvalidEnumArgumentException
```

This looks pretty good, even if you know this (mythical) native API has a buffer overrun if you pass a negative color value. You know this because you know the enum only defined positive values and you are sure that any value passed was one of the ones defined in the enum, right? Well, only half right. You don't know what values are defined in the enum. Checking at the moment you write this code is not good enough because `IsDefined` takes the value of the enum at runtime. So if someone later added a new value (say `Ultraviolet = -1`) to the enum, `IsDefined` will start allowing the value `-1` through. This is true whether the enum is defined in the same assembly as the method or in another assembly.

```
public enum Color {
    Red = 1,
    Green = 2,
    Blue = 3,
    Ultraviolet = -1, //new value added this version
}
```

Now, that same call site no longer throws an exception.

```
//causes a buffer overrun in NativeMethods.SetImageColor() Foo.
PickColor ((Color) -1);
```

The moral of the story is twofold. First, be very careful when you use `Enum.IsDefined` in your code. Second, when you design an API to simplify a situation, be sure the fix isn't worse than the current problem.

■ **BRENT RECTOR** `Enum.IsDefined` is surprisingly expensive. However, I think using `Enum.IsDefined` is appropriate *when you want to determine whether a value equals a definition in an enumerated type*—but that's not what the prior example is trying to do.

In the preceding example, the issue is that a condition must be met before calling the native `SetImageColor` method: The `Color` argument cannot have a negative value. As Brad mentions, the values defined in an enumerated type may change from compilation to compilation. In other words, the constraint applicable to the argument value (which must not be negative) doesn't necessarily hold over time for the values defined in the enumerated type. Therefore, validating that the `Color` argument is not negative by testing whether its value matches a definition in an enumerated type simply isn't proper validation.

The example is great, though, in that it demonstrates how easily you can unintentionally misuse `Enum.IsDefined`, so use it thoughtfully.

✓ **DO** be aware that mutable arguments might have changed after they were validated.

If the member is security-sensitive, you are encouraged to make a copy and then validate and process the argument.

■ **RICO MARIANI** This is one of the many places where you can cash in on the fact that CLR strings are immutable; there is no need to copy them in security-sensitive operations.

5.8.3 Parameter Passing

From the perspective of a framework designer, there are four main groups of parameters: `by-value` parameters, `ref` parameters, `in` (`ref readonly`) parameters, and `out` parameters.

When an argument is passed through a `by-value` parameter, the member receives a copy of the actual argument passed in. If the argument is a value type, a copy of the argument is put on the stack. If the argument is a reference type, a copy of the reference is put on the stack. Most popular

CLR languages, such as C#, VB.NET, and C++, default to passing parameters by value.

```
public void Add (object value) {...}
```

■ **PAUL VICK** When moving to .NET, VB changed the default for parameters from by-reference to by-value. We did this because the vast majority of parameters are by-value, so having a default of by-reference meant that parameters could produce unintentional side effects quite easily. Thus, defaulting to by-value was a safer choice and meant that choosing by-reference semantics was a conscious, rather than accidental, decision.

When an argument is passed through a `ref` parameter, the member receives a reference to the actual argument passed in. If the argument is a value type, a reference to the argument is put on the stack. If the argument is a reference type, a reference to the reference is put on the stack. `Ref` parameters can be used to allow the member to modify arguments passed by the caller.

```
public static void Swap(ref object obj1, ref object obj2){  
    object temp = obj1;  
    obj1 = obj2;  
    obj2 = temp;  
}
```

`In` parameters are also called `ref readonly` parameters, and are similar to “normal” `ref` parameters. The most significant difference from `ref` parameters is that the receiving method cannot assign a replacement value through the parameter. For reference types, there is no functional difference between the default pass-by-reference and using the `in` parameter modifier—but using `in` makes all member accesses slower. For value types, a method cannot assign fields of an `in` parameter, and the compiler will make defensive copies of the parameter before invoking methods or properties—unless the type, method, or property accessor was declared

with the `readonly` modifier (section 4.7). These defensive copies ensure that the called method cannot directly or indirectly modify the caller's value.

`Out` parameters are similar to `ref` parameters, with some small differences. Such a parameter is initially considered unassigned and cannot be read in the member body before it is assigned some value. Also, the parameter has to be assigned some value before the member returns. For example, the following example will not compile and generates the compiler error "Use of unassigned out parameter 'uri.'"

```
public class Uri {
    public bool TryParse(string uriString, out Uri uri){
        Trace.WriteLine(uri);
        ...
    }
}
```

■ **RICO MARIANI** `Out` is the same as the `ref` mechanism, but different (stronger) verification rules apply because the intent has been made clear to the compiler and the runtime.

X AVOID using `out` or `ref` parameters except when implementing patterns that require them, such as the Try pattern (section 7.5.2).

Using `out` or `ref` parameters requires experience with pointers, understanding how value types and reference types differ, and handling methods with multiple return values. Also, the difference between `out` and `ref` parameters is not widely understood. Framework architects designing for a general audience should not expect users to master the challenge of working with `out` or `ref` parameters.

■ **ANDERS HEJLSBERG** As a rule, I am not too crazy about `ref` and `out` parameters in APIs. Such APIs compose very poorly, forcing you to declare temporary variables. I much prefer functional designs that convey the entire result in the return value.

BRIAN PEPIN If you need to return several pieces of data from a call, wrap that data up into a class or struct. For example, .NET has an API to perform hit testing of controls that returns a `HitTestResult` object.

JASON CLARK Generics and `ref/out` parameters interact very nicely. Normally, the variable used to make a `ref` call must be an exact type match, which can be inconvenient. However, if the method is defined with a generic argument that is merely constrained to the required base type, then variables of derived types can be used to make the call. Meanwhile, inference by the compiler makes it unlikely that the caller is impacted by the generic syntax for the method. Very nice!

X DO NOT pass reference types by reference (`ref` or `in`).

There are some limited exceptions to the rule, such as when a method can be used to swap references. There is no reason to pass a reference type with the `in` modifier (`ref readonly` semantics).

```
public static class Reference {
    public void Swap<T>(ref T obj1, ref T obj2){
        T temp = obj1;
        obj1 = obj2;
        obj2 = temp;
    }
}
```

CHRIS SELLS Swap always comes up in these discussions, but I have not written code that actually needed a swap function since college. Unless you've got a very good reason, avoid `out` and `ref` altogether.

X DO NOT pass value types by read-only reference (`in`).

The main benefit of passing a value type by read-only reference is to reduce the cost of copying the value when the type is large. Since the guidance for value types is that they should be small (section 4.2), the performance benefit is slight. Conversely, if your method reads a property or invokes a method—without the appropriate `readonly`

modifier—on an `in` parameter, you end up paying the cost silently. Once the compiler has made two defensive copies (or one copy if the type is small), your API now pays a performance penalty for the modifier, rather than receiving a performance boost.

This performance penalty risk, combined with some CLR languages having unpleasant syntax for passing by reference, means you should rarely, if ever, use the `in` modifier in public or protected methods.

■ **JEREMY BARTON** The `in` modifier, like large structs, can be useful in private or internal code. The edge cases and gotchas to this modifier can be noticed in private code and changed to pass-by-value or “full” `ref`. Since public (and protected) members can’t freely change from `in` to `ref`—or from `in` to by-value—when performance testing says to do so, we advise against using the `in` modifier in those members.

5.8.4 Members with Variable Number of Parameters

Members that can take a variable number of arguments are expressed by providing an array parameter. For example, `String` provides the following method:

```
public class String {  
    public static string Format(string format, object[] parameters);  
}
```

A user can then call the `String.Format` method, as follows:

```
String.Format("File {0} not found in {1}",  
    new object[]{filename,directory});
```

Adding the C# `params` keyword to an array parameter changes the parameter to a so-called `params` array parameter and provides a shortcut to creating a temporary array.

```
public class String {  
    public static string Format(string format, params object[] parameters);  
}
```

Doing this allows the user to call the method by passing the array elements directly in the argument list.

```
String.Format("File {0} not found in {1}",filename,directory);
```

Note that the `params` keyword can be added only to the last parameter in the parameter list.

✓ **CONSIDER** adding the `params` keyword to array parameters if you expect the end users to pass arrays with a small number of elements.

If lots of elements will likely be passed in common scenarios, users will probably not pass these elements inline anyway, so the `params` keyword is not necessary.

■ **BRIAN PEPIN** There are several places in the framework where we didn't do this, and it still grates on me whenever I have to write code that creates a bunch of temporary arrays. In many cases, we were able to add `params` in a later version, but in other cases adding `params` made the method ambiguous and was a source-code breaking change.

✗ **AVOID** using `params` arrays if the caller would almost always have the input already in an array.

For example, members with byte array parameters would almost never be called by passing individual bytes. For this reason, byte array parameters in .NET do not use the `params` keyword.

✗ **DO NOT** use `params` arrays if the array is modified by the member taking the `params` array parameter.

Because many compilers turn the arguments to the member into a temporary array at the call site, the array might be a temporary object, and therefore any modifications to the array will be lost.

✓ **CONSIDER** using the `params` keyword in a simple overload, even if a more complex overload could not use it.

Ask yourself if users would value having the params array in one overload, even if it wasn't in all overloads. Consider the following overloaded method:

```
public class Graphics {  
    FillPolygon(Brush brush, params Point[] points) { ... }  
    FillPolygon(Brush brush, Point[] points, FillMode fillMode) {  
        ...  
    }  
}
```

The array parameter of the second overload is not the last parameter in the parameter list, so it cannot use the params keyword. This does not mean that the keyword should not be used in the first overload, where it is the last parameter. If the first overload is used often, users will appreciate the addition.

- ✓ **DO** try to order parameters to make it possible to use the params keyword.

Consider the following overloads on `PropertyDescriptorCollection`:

```
Sort()  
Sort(IComparer comparer)  
Sort(string[] names, IComparer comparer)  
Sort(params string[] names)
```

Because of the order of parameters on the third overload, the opportunity to use the params keyword has been lost. The parameters could be reordered to allow for this keyword in both overloads.

```
Sort()  
Sort(IComparer comparer)  
Sort(IComparer comparer, params string[] names)  
Sort(params string[] names)
```

- ✓ **CONSIDER** providing special overloads and code paths for calls with a small number of arguments in extremely performance-sensitive APIs.

This makes it possible to avoid creating array objects when the API is called with a small number of arguments. Name the parameters by

taking a singular form of the array parameter and adding a numeric suffix.

```
void Format (string formatString, object arg1)
    void Format (string formatString, object arg1, object arg2)
...
    void Format (string formatString, params object[] args)
```

You should do this only if you are going to treat the entire code path as a special case, not just create an array and call the more general method.

RICO MARIANI You might also do this if you want to specialize the code path at some point, even if you can't do it in your first release. That way, you can change your internal implementation without having your clients recompile, and you can do the most important specializations first.

✓ **DO** be aware that `null` could be passed as a `params` array argument.

You should validate that the array is not `null` before processing it.

```
static void Main() {
    Sum(1, 2, 3, 4, 5); //result == 15
    Sum(null);
}
static int Sum(params int[] values) {
    if(values==null) throw ArgumentNullException(...);
    int sum = 0;
    foreach (int value in values) {
        sum += value;
    }
    return sum;
}
```

RICO MARIANI Very low-level functions (those doing only a tiny amount of work) will find the cost of temporary array creation and array validation a significant burden. All of this argues in favor of using the `params` construct higher up in your stack—in bigger functions that are doing more work.

X DO NOT use the varargs methods, otherwise known as the ellipsis.

Some CLR languages, such as C++, support an alternative convention for passing variable parameter lists called varargs methods. This convention should not be used in frameworks, because it is not CLS-compliant.

■ **RICO MARIANI** Of course, “never” in this context means not in the framework APIs—varargs wasn’t added lightly to C++. If varargs helps you in your internal implementation, by all means use it. Just remember it isn’t CLS-compliant, so it’s bad form to use it in public APIs.

■ **JAN KOTAS** The runtime support for varargs is just enough to make managed C++ work on Windows. The performance is not great, and it has not been implemented for non-Windows platforms. For these reasons, I would not recommend using it even for internal implementations. Several teams have done experiments with it, only to find that it is worse across the board than the alternatives.

5.8.5 Pointer Parameters

In general, pointers should not appear in the public surface area of a well-designed managed code framework. Most of the time, pointers should be encapsulated. However, pointers are sometimes required for interoperability reasons, and using pointers in such cases is appropriate.

The `Span<T>` and `ReadOnlySpan<T>` types unify native memory and managed arrays, and can be pinned to a pointer via the `fixed` keyword in C#. Using spans instead of pointers is preferred, when possible. For more information on spans, see section 9.12.

✓ DO provide an alternative for any member that takes a pointer argument, because pointers are not CLS-compliant.

```
[CLSCompliant(false)]
public unsafe int GetBytes(char* chars, int charCount,
    byte* bytes, int byteCount);
```

```

public int GetBytes(char[] chars, int charIndex, int charCount,
byte[] bytes, int byteIndex, int byteCount)

// Consider Span<T> / ReadOnlySpan<T> instead of
// (or in addition to) methods that take pointers
public int GetBytes(ReadOnlySpan<char> source, Span<byte> destination);

```

X AVOID doing expensive argument checking for pointer arguments.

In general, argument checking is well worth the cost, but for APIs that are performance-critical enough to require using pointers, the overhead is often not worth it.

■ RICO MARIANI This is right in line with my general advice. Put the argument checking at the right level of your abstraction stack. That will get you the best diagnostics at the best price. After you get this close to the metal, those extra tests can be a significant fraction of the job at hand.

✓ DO follow common pointer-related conventions when designing members with pointers.

For example, there is no need to pass the start index, because simple pointer arithmetic can be used to accomplish the same result.

```

//Bad practice
public unsafe int GetBytes(char* chars, int charIndex, int charCount,
                           byte* bytes, int byteIndex, int byteCount)

//Better practice
public unsafe int GetBytes(char* chars, int charCount,
                           byte* bytes, int byteCount)

//Example call site
GetBytes(chars + charIndex, charCount, bytes + byteIndex, byteCount);

```

■ BRAD ABRAMS For developers working with pointer-based APIs, it is more natural to think of the world with a pointer-oriented mindset. Although it is common in “safe” managed code, passing an index is uncommon in pointer-based code; it is more natural to use pointer arithmetic.

5.9 Using Tuples in Member Signatures

Loosely speaking, a tuple is any ordered collection of instances of heterogeneous types. If we consider only strongly typed (section 2.2.3.3) heterogeneous structures, then C# (and other .NET languages) have four different mechanisms for providing a tuple under this definition: the `System.Tuple<T1, T2>` class (including other generic counts), the `System.ValueTuple<T1, T2>` struct (including other generic counts), the language feature “named tuples,” and descriptive types.

To highlight the main characteristics of these four alternatives, we show how each of them can define a simple person as an entity with a first name, a last name, and an age.

1. `System.Tuple<T1, ...>` classes

```
public Tuple<string, string, int> GetPerson() {
    return Tuple.Create("John", "Smith", 32);
}
```

2. `System.ValueTuple<T1, ...>` structs

```
public ValueTuple<string, string, int> GetPerson() {
    return ValueTuple.Create("John", "Smith", 32);
}
```

3. Named tuples

```
public (string FirstName, string LastName, int Age) GetPerson() {
    return ("John", "Smith", 32);
}
```

4. Descriptive types

```
public struct Person {
    public string FirstName { get; }
    public string LastName { get; }
    public int Age { get; }

    public Person(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
}
```

```
public Person GetPerson() {
    return new Person("John", "Smith", 32);
}
```

Declaration is only one part of the story for a value. For each of the previous declarations, we will now look at building a message printing a person's target heart rate using the heuristic of 220 beats per minute minus the person's age.

1. System.Tuple<T1, ...> classes

```
Tuple<string, string, int> person = GetPerson();
string message = $"{person.Item1} {person.Item2}'s target heart
rate is {220 - person.Item3}."
```

2. System.ValueTuple<T1, ...> structs

```
ValueTuple<string, string, int> person = GetPerson();
string message = $"{person.Item1} {person.Item2}'s target heart
rate is {220 - person.Item3}."
```

3. Named tuples

```
(string FirstName, string LastName, int Age) person = GetPerson();
string message = $"{person.FirstName} {person.LastName}'s target
heart rate is {220 - person.Age}."

// This syntax is also available to named tuples
//string message = $"{person.Item1} {person.Item2}'s target heart
rate is {220 - person.Item3}."
```

4. Descriptive types

```
Person person = GetPerson();
string message = $"{person.FirstName} {person.LastName}'s target
heart rate is {220 - person.Age}."
```

When using `System.Tuple<T1, T2, T3>` or `System.ValueTuple<T1, T2, T3>`, the members exposed on the enumerated object are "Item1," "Item2," and "Item3." These names do not help convey the purpose of the values (which is more problematic when two or more of the element types are the same). Conversely, for both the descriptive type and the named tuple, the names shown via IntelliSense are the much more expressive "FirstName," "LastName," and "Value."

The descriptive type has additional benefits over the alternatives, not all of which would apply to this example.

- The constructor could have data validation, or other custom logic.
- Methods can be declared on the type.
- Additional fields and properties can be added in the future.
- Program analyzers can track the type to see what places produce one, and what places accept one as input.
- The descriptive type provides a target for documentation.
- The descriptive names are available in all CLR languages.

The .NET runtime considers two generic types with the same base name but a different number of generic parameters to be different types, so changing from returning `Tuple<T1, T2>` to `Tuple<T1, T2, T3>` is just as much a breaking change as changing a return type from `String` to `Int32`.

✓ **DO** prefer descriptive types over tuples, `System.Tuple<T1, ...>`, and `System.ValueTuple<T1, ...>`; they improve discoverability and can better evolve over time.

■ **STEPHEN TOUB** Neither discoverability nor evolution is particularly important for internals. It's good to remember that the guidelines here are about public APIs. Tuples (and the syntax C# provides for them) are really handy in implementation and are a convenient way of passing around multiple pieces of related data, including when providing multiple return values from methods.

✓ **DO** prefer named tuples over unnamed tuples (including `System.Tuple<T1, ...>` and `System.ValueTuple<T1, ...>`).

The previous section shows why descriptive types are better than named tuples in public API, as well as why named tuples are better than unnamed tuples. However, occasionally a method is so specialized that it would never make sense to amend the returned data, a custom descriptive type would only ever be deconstructed into locals, and

the method name really is sufficient for documentation. When all of these conditions are true for a method's return type, then a named tuple might make more sense. For example, .NET Standard 2.1 added `System.Range`, which has a utility method to compute the total length and initial offset for a range:

```
public partial struct Range {  
    public (int Offset, int Length) GetOffsetAndLength(int length);  
}
```

- ✓ **DO** name elements of a named tuple using PascalCase, as if they were properties.

Named tuples in C# allow access to their elements via the same syntax as accessing a property or field on a descriptive type, so they should be named accordingly.

- ✗ **DO NOT** use tuples with more than three fields.

Tuples are a convenient way to represent a data pair or triplet that will be separated into the individual fields, such as via the C# deconstruction language feature. Once more than three fields are being returned together, it seems more likely that the values will stay grouped together, and be passed to other methods or stored in a look-up table—a role better suited to descriptive types.

■ **JAN KOTAS** Named tuples are internally represented as structs. Having more than three fields would often make the named tuple instance size too big and negatively impact performance.

- ✗ **DO NOT** use tuples as method parameters.

When a tuple is used as a parameter, it introduces extra syntax for both the method declaration and the caller, but does not provide much value in return.

Accepting a collection based on tuples (such as `(string FirstName, string LastName, int Age)[] people`) is more justifiable than accepting a single tuple value parameter. However, operating on a collection suggests a level of complexity that would be better served by descriptive types—validating constructors, instance methods, and the ability to add to the type without a breaking change. Callers of the method may also be forced to use awkward syntax to get their data into the right shape.

This guideline does not mean that library authors should go out of their way to prohibit named tuples or unnamed tuples to be specified as generic type parameters, even though that could cause a violation after generic substitution.

```
// OK
public static T Max(IComparer<T> comparer, T first, T second, T third) {...}

// OK
var first = (Base: 1, Exponent: 2);
var second = (Base: 2, Exponent: 1);
var third = (Base: 3, Exponent: -5);
(int Base, int Exponent) max = Max(comparer, first, second, third);

// Not OK if explicitly defined this way
public static (int Base, int Exponent) Max(
    IComparer<(int Base, int Exponent)> comparer, ...) { ... }
```

X **DO NOT** define extension methods over tuples.

This is a special case of not using tuples as method parameters, but is worth mentioning in its own right.

Suppose we were to define a method to calculate the dot product of two tuples used to represent a vector on the X-Y plane:

```
public static int DotProduct(this (int X, int Y) first,
    (int X, int Y) second) {
    return first.X * second.X + first.Y * second.Y;
}
```

This extension method would also show up in IntelliSense on the `(int Base, int Exponent)` variables from the previous example, and the following code would compile:

```
(int Base, int Exponent) value = (2, 20);
ValueTuple<int, int> otherValue = ValueTuple.Create(5, 7);
int dotProduct = value.DotProduct(otherValue);
```

This is because the C# language (and VB.NET) considers all tuples, as well as `System.ValueTuple<T1, ...>`, to be the same type if their ordered list of element types is the same.

- ✓ **CONSIDER** adding an appropriate `Deconstruct` method to any type that was chosen as an alternative to a tuple.

By adding a `void` method on our `Person` class named `Deconstruct`, we can use the C# 7.0 deconstruction feature:

```
public partial struct Person {
    public void Deconstruct(out string firstName, out string lastName,
out int age) {
        firstName = FirstName;
        lastName = LastName;
        age = Age;
    }
}
```

The `Deconstruct` method enables us to build our target heart rate message with the following code:

```
(string firstName, string lastName, int age) = GetPerson();
string message = $"{firstName} {lastName}'s target heart rate is
{220 - age}."
```

This same code works for all four representations of the person because `System.Tuple<T1, ...>`, `System.ValueTuple<T1, ...>`, and named tuples all have similar `Deconstruct` methods and the deconstruction language feature does not require any similarity in the `out` parameter names and the local variable names.

The deconstruction syntax is especially nice for tuples, as their lack of methods means that any logic operating on a tuple has to first break it down to some (or all) of its element values. If a tuple was a candidate for the return type of a method, then even for the descriptive type, a likely common operation for callers will be to break the type down to its element values.

SUMMARY

This chapter offers comprehensive guidelines for general member design. As the annotations suggest, member design is one of the most complex parts of designing a framework. This is a natural consequence of the richness of concepts related to member design.

The next chapter covers design issues relating to extensibility.



6

Designing for Extensibility

ONE IMPORTANT ASPECT of designing a framework is making sure the extensibility of the framework has been carefully considered. This requires that you understand the costs and benefits associated with various extensibility mechanisms. This chapter helps you decide which of the extensibility mechanisms—subclassing, events, virtual members, callbacks, and so on—can best meet the requirements of your framework. This chapter does not cover the design details of these mechanisms. Such details are discussed in other parts of the book, and this chapter simply provides cross-references to sections that describe those details.

A good understanding of OOP is a necessary prerequisite to designing an effective framework and, in particular, to understanding the concepts discussed in this chapter. However, we do not cover the basics of object-orientation in this book, because there are already excellent books entirely devoted to the topic.

6.1 Extensibility Mechanisms

There are many ways to allow extensibility in frameworks. They range from less powerful but less costly to very powerful but expensive. For any given extensibility requirement, you should choose the least costly extensibility mechanism that meets the requirements. Keep in mind that it's

usually possible to add more extensibility later, but you can never take it away without introducing breaking changes.

This section discusses some of the framework extensibility mechanisms in detail.

6.1.1 Unsealed Classes

Sealed classes cannot be inherited from, and they prevent extensibility. In contrast, classes that can be inherited from are called unsealed classes.

```
// string cannot be inherited from
public sealed class String { ... }

// TraceSource can be inherited from
public class TraceSource { ... }
```

Subclasses can add new members, apply attributes, and implement additional interfaces. Although subclasses can access protected members and override virtual members, these extensibility mechanisms result in significantly different costs and benefits. Subclasses are described in sections 6.1.2 and 6.1.4. Adding protected and virtual members to a class can have expensive ramifications if not done with care, so if you are looking for simple, inexpensive extensibility, an unsealed class that does not declare any virtual or protected members is a good way to do it.

✓ **CONSIDER** using unsealed classes with no added virtual or protected members as a great way to provide inexpensive yet much appreciated extensibility to a framework.

Developers often want to inherit from unsealed classes so as to add convenience members such as custom constructors, new methods, or method overloads.¹ For example, `System.Messaging.MessageQueue` is unsealed and thus allows users to create custom queues that default to a particular queue path or to add custom methods that simplify the API for specific scenarios. In the following example, the scenario is for a method sending `Order` objects to the queue.

1. Some convenience methods can be added to sealed types as extension methods.

```
public class OrdersQueue : MessageQueue {  
    public OrdersQueue() : base(OrdersQueue.Path){  
        this.Formatter = new BinaryMessageFormatter();  
    }  
  
    public void SendOrder(Order order){  
        Send(order,order.Id);  
    }  
}
```

■ **PHIL HAACK** Because test-driven development has caught fire in the .NET developer community, many developers want to inherit from unsealed classes (often dynamically using a mock framework) in order to substitute a test double in the place of the real implementation.

At the very least, if you've gone to the trouble of making your class unsealed, consider making key members virtual, perhaps via the Template Method Pattern, to provide more control.

Classes are unsealed by default in most programming languages, and this is also the recommended default for most classes in frameworks. The extensibility afforded by unsealed types is much appreciated by framework users and quite inexpensive to provide because of the relatively low test costs associated with unsealed types.

■ **VANCE MORRISON** The key word in this advice is "CONSIDER." Keep in mind that you always have the option of unsealing a class in the future (it is not a breaking change); however, once unsealed, a class must remain unsealed. Also, unsealing does inhibit some optimizations [e.g., converting virtual calls to more efficient nonvirtual calls (and then inlining)]. Finally, unsealing helps your users only if they control the creation of the class (sometimes true, sometimes not). In short, designs are only rarely usefully extensible "by accident." Being unsealed is part of the contract of a class and its users, and like everything about the contract, it deserves to be a conscious, deliberate choice on the part of the designer.

6.1.2 Protected Members

Protected members by themselves do not provide any extensibility, but they can make extensibility through subclassing more powerful. They can be used to expose advanced customization options without unnecessarily complicating the main public interface. For example, the `SourceSwitch.Value` property is protected because it is intended for use only in advanced customization scenarios.

```
public class FlowSwitch : SourceSwitch {  
    protected override void OnValueChanged() {  
        switch (this.Value) {  
            case "None" : Level = FlowSwitchSetting.None; break;  
            case "Both" : Level = FlowSwitchSetting.Both; break;  
            case "Entering": Level = FlowSwitchSettingEntering; break;  
            case "Exiting" : Level = FlowSwitchSettingExiting; break;  
        }  
    }  
}
```

Framework designers need to be careful with protected members because the name “protected” can give a false sense of security. Anyone is able to subclass an unsealed class and access protected members, so all the same defensive coding practices used for public members apply to protected members.

✓ **CONSIDER** using protected members for advanced customization.

Protected members are a great way to provide advanced customization without complicating the public interface.

✓ **DO** treat protected members in unsealed classes as public for the purpose of security, documentation, and compatibility analysis.

Anyone can inherit from a class and access the protected members.

■ **BRAD ABRAMS** Protected members are just as much a part of your publicly callable interface as public members. In designing the framework, we considered protected and public to be roughly equivalent. We generally did the same level of review and error checking in protected APIs as we did in public APIs because they can be called from any code that just happens to subclass.

6.1.3 Events and Callbacks

Callbacks are extensibility points that allow a framework to call back into user code through a delegate. These delegates are usually passed to the framework through a parameter of a method.

```
List<string> cityNames = ...  
cityNames.RemoveAll(delegate(string name) {  
    return name.StartsWith("Seattle");  
});
```

Events are a special case of callbacks that supports convenient and consistent syntax for supplying the delegate (an event handler). In addition, Visual Studio's statement completion and designers provide help in using event-based APIs.

```
var timer = new Timer(1000);  
timer.Elapsed += delegate {  
    Console.WriteLine("Time is up!");  
};  
timerStart();
```

General event design is discussed in section 5.4.

Callbacks and events can be used to provide quite powerful extensibility, comparable to virtual members. At the same time, callbacks—and even more so, events—are more approachable to a broader range of developers because they don't require a thorough understanding of object-oriented design. Also, callbacks can provide extensibility at runtime, whereas virtual members can be customized only at compile-time.

The main disadvantage of callbacks is that they are more heavyweight than virtual members. The performance when calling through a delegate is worse than it is when calling a virtual member. In addition, delegates are objects, so their use affects memory consumption.

You should also be aware that by accepting and calling a delegate, you are executing arbitrary code in the context of your framework. Therefore, a careful analysis of all such callback extensibility points from the security, correctness, and compatibility points of view is required.

- ✓ **CONSIDER** using callbacks to allow users to provide custom code to be executed by the framework.
- ✓ **CONSIDER** using events, instead of virtual members, to allow users to customize the behavior of a framework without the need for understanding object-oriented design.
- ✓ **CONSIDER** using events instead of plain callbacks, because events are more familiar to a broader range of developers and are integrated with Visual Studio statement completion.
- ✗ **AVOID** using callbacks in performance-sensitive APIs.

■ **KRZYSZTOF CWALINA** Delegate calls were made much faster in CLR 2.0, but they are still about two times slower than direct calls to virtual members. In addition, delegate-based APIs are generally less efficient in terms of memory usage. Having said that, the differences are relatively small and should only matter if the API is called very frequently.

■ **STEPHEN TOUB** In a performance-critical method, you want to think about all forms of extensibility and what kind of impact they may have on throughput. This goes beyond delegates. In fact, in some situations it may actually be better for your common case to use delegates instead of virtual methods. For example, consider a design where you want a default behavior that can then be potentially replaced if a delegate is provided. If you made the functionality virtual, you'd be paying for the virtual dispatch (unless the JIT could devirtualize the call) regardless of whether a replacement was provided. But with a delegate, you could have a nonvirtual, inlineable implementation that just does a null check on the delegate instance and only pays the delegate invocation costs if there is something else to do instead of the default behavior.

- ✓ **DO** use the `Func<...>`, `Action<...>`, or `Expression<...>` types instead of custom delegates when possible, when defining APIs with callbacks.

`Func<...>` and `Action<...>` represent generic delegates. The following is how .NET defines them:

```
public delegate void Action()
public delegate void Action<T1, T2>(T1 arg1, T2 arg2)
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3)
public delegate void Action<T1, T2, T3, T4>(T1 arg1, T2 arg2,
    T3 arg3, T4 arg4)
public delegate TResult Func<TResult>()
public delegate TResult Func<T, TResult>(T arg)
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2,
    T3 arg3)
public delegate TResult Func<T1, T2, T3, T4, TResult>(T1 arg1, T2 arg2,
    T3 arg3, T4 arg4)
```

They can be used as follows:

```
Func<int,int,double> divide = (x,y)=>(double)x/(double)y;
Action<double> write = (d)=>Console.WriteLine(d);
write(divide(2,3));
```

`Expression<...>` represents function definitions that can be compiled and subsequently invoked at runtime but can also be serialized and passed to remote processes. Continuing with our example:

```
Expression<Func<int,int,double>> expression =
    (x,y)=>(double)x/(double)y;
Func<int,int,double> divide2 = expression.Compile();
write(divide2(2,3));
```

Notice how the syntax for constructing an `Expression<>` object is very similar to that used to construct a `Func<>` object. In fact, the only difference is the static type declaration of the variable (`Expression<>` instead of `Func<...>`).

■ **STEPHEN TOUB** In general, if these generic delegate types *can* be used, they *should* be used. However, there are some relatively rare situations where these generic delegates can't be used. One such category is when the types being passed as arguments or return values can't be used as generic type parameters, such as pointer types or `ref struct` types. Another category is when arguments or return values need to be passed as something other than by value—for example, when you want an argument to be `ref`. In such situations, you will need to find an existing delegate (generic or otherwise) that's been declared with an appropriate signature, or else define a new one.

■ **JAN KOTAS** The `Action` and `Func` delegates do not allow naming arguments. That makes it impractical to use these delegates for callbacks with more complex signatures where the meaning of the arguments is not obvious and it is important to name the arguments for clarity. For example, the `System.Runtime.InteropServices.DllImportResolver` delegate violates this rule for this reason.

■ **RICO MARIANI** Most times you're going to want `Func` or `Action` if all that needs to happen is to run some code. You need `Expression` when the code needs to be analyzed, serialized, or optimized before it is run. `Expression` is for thinking about code; `Func/Action` is for running it.

✓ **DO** measure and understand the performance implications of using `Expression<...>`, instead of using `Func<...>` and `Action<...>` delegates.

`Expression<...>` types are, in most cases, logically equivalent to `Func<...>` and `Action<...>` delegates. The main difference between them is that the delegates are intended to be used in local process scenarios; expressions are intended for cases where it's beneficial and possible to evaluate the expression in a remote process or machine.

■ **RICO MARIANI** The remoteness of the evaluation is sort of incidental. The main thing about Expressions is that you use them when you are going to need to reason over the code to be executed, often over a composition of expressions such as in a LINQ query, and then, having considered the whole and the execution options, you create some kind of optimized plan for doing the work. This is how LINQ to SQL is able to create a single SQL fragment from a composition of loose-looking expressions.

This plan could easily go wrong. You could do too much analysis of expressions or too little. You could use up too much space holding expression trees, or you could avoid all the trees but then find you have bad performance because you have so many small anonymous delegates.

If you look at the patterns used in the LINQ implementations in .NET, you'll see several good ways to make use of these constructs:

- Use expressions only if you need to "think" about the code and not just run it.
- Don't blindly compose and run code that could be meaningfully optimized if you "thought" about it before running it.
- Don't create systems that optimize the code so much before running it that it would have been faster to just run it directly without optimizing.
- Optimization isn't the only use for expression trees, but it is an important one.

✓ **DO** understand that by calling a delegate, you are executing arbitrary code, and that could have security, correctness, and compatibility repercussions.

■ **BRIAN PEPIN** The Windows Forms team bumped up against this issue when writing some of the low-level code in SystemEvents. System Events defines a static API and therefore needs to be threadsafe. Internally, it uses locks to ensure thread safety. Early code in SystemEvents would grab a lock and then raise an event. Here's an example:

```
lock(someInternalLock) {  
    if(eventHandler!=null) eventHandler(sender, EventArgs.Empty);  
}
```

This is bad because you have no idea what the user code in the event handler is going to do. If the user code signals a thread and waits on its own lock, you might have just introduced a deadlock. This would be better code:

```
EventHandler localHandler = eventHandler;  
if(localHandler != null) localHandler(sender, EventArgs.Empty);
```

This way, the user's code will never deadlock due to your own internal implementation. Note that because assignments in managed code are atomic, I didn't need a lock at all in this case. That won't always be true. For example, if your code needed to check more than one variable, you'd still need a lock:

```
EventHandler localHandler = null;  
lock(someInternalLock) {  
    if (eventHandler != null && shouldRaiseEvents) {  
        localHandler = eventHandler;  
    }  
}  
if(localHandler!=null) localHandler(sender,EventArgs.Empty);
```

■ **JEREMY BARTON** The null-conditional operator introduced in C# 6 can simplify the event invocation.

```
eventHandler?.Invoke(sender, EventArgs.Empty);
```

This has the same effect as Brian's second example (invoking outside the lock), including only ever reading from the "eventHandler" value once:

```
EventHandler localHandler = eventHandler;  
if(localHandler != null) localHandler(sender, EventArgs.Empty);
```

■ **JOE DUFFY** In addition to deadlock, invoking a callback under a lock like this can cause reentrancy. Locks on the CLR support recursive acquires, so if the callback somehow manages to call back into the same object that initiated the callback, the results are often not good. Locks are typically used to isolate invariants that are temporarily broken, yet this practice can expose them at the reentrant boundary. Needless to say, this is apt to cause weird exceptions and unexpected behavior.

That said, sometimes this practice is necessary. If the callback is being used to make a decision—as would be the case with a predicate—and that decision needs to be made under a lock, you will have no choice. When invoking a callback under a lock is unavoidable, be sure to carefully document the restrictions (no inter-thread communication, no reentrancy). You must also ensure that, should a developer violate these restrictions, the result will not lead to security vulnerabilities. The risk here is usually greater than the reward.

■ **STEPHEN TOUB** From an API design perspective, this whole discussion is really interesting as it applies to compatibility. You may find yourself in a situation where you've invoked a user-supplied callback while holding a lock, and you decide to "fix" it by employing approaches like that outlined. In doing so, however, you're impacting potentially visible behaviors. The invocation will no longer be synchronized with whatever else might be using the same lock. It's possible the user's callback was actually relying on that synchronization for safety, whether they knew it or not.

Extensibility is hard.

6.1.4 Virtual Members

Virtual members can be overridden, thereby changing the behavior of the subclass. They are quite similar to callbacks in terms of the extensibility they provide, but they are better in terms of execution performance and memory consumption. Also, virtual members feel more natural in scenarios that require creating a special kind of an existing type (specialization).

The main disadvantage of virtual members is that the behavior of a virtual member can be modified only at the time of compilation. The behavior of a callback can be modified at runtime.

Virtual members, like callbacks (and maybe more than callbacks), are costly to design, test, and maintain because any call to a virtual member can be overridden in unpredictable ways and can execute arbitrary code. Also, much more effort is usually required to clearly define the contract of virtual members, so the cost of designing and documenting them is higher.

■ **KRZYSZTOF CWALINA** A common question I get is whether documentation for virtual members should say that the overrides must call the base implementation. The answer is that overrides should preserve the contract of the base class. They can do that by calling the base implementation or by some other means. It is rare that a member can claim that the only way to preserve its contract (in the override) is to call it. In a lot of cases, calling the base might be the easiest way to preserve the contract (and documentation should point that out), but it's rarely absolutely required.

Because of the risks and costs, limiting extensibility of virtual members should be considered. Extensibility through virtual members today should be limited to those areas that have a clear scenario requiring extensibility. This section presents guidelines for when to allow it and when and how to limit it.

X DO NOT make members virtual unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members.

Virtual members are less forgiving in terms of changes that can be made to them without breaking compatibility. Also, they are slower than nonvirtual members, mostly because calls to virtual members are not inlined.

■ **RICO MARIANI** Be sure you understand your extensibility requirements completely before you make decisions in the name of extensibility. A common mistake is sprinkling classes with virtual methods and properties, only to find that the needed extensibility still can't be realized and everything is now (and forever) slower.

■ **JAN GRAY** The peril: If you ship types with virtual members, you are promising to forever abide by subtle and complex observable behaviors and subclass interactions. I think framework designers underestimate their peril. For example, we found that `ArrayList` item enumeration calls several virtual methods for each `MoveNext` and `Current`. Fixing those performance problems could (but probably doesn't) break user-defined implementations of virtual members on the `ArrayList` class that are dependent on virtual method call order and frequency.

- ✓ **CONSIDER** limiting extensibility to only what is absolutely necessary through the use of the Template Method Pattern, described in section 9.9.
- ✓ **DO** prefer protected accessibility over public accessibility for virtual members. Public members should provide extensibility (if required) by calling into a protected virtual member.

The public members of a class should provide the right set of functionality for direct consumers of that class. Virtual members are designed to be overridden in subclasses, and protected accessibility is a great way to scope all virtual extensibility points to where they can be used.

```
public Control{
    public void SetBounds(...){
        ...
        SetBoundsCore(...);
    }

    protected virtual void SetBoundsCore(...){
        // Do the real work here.
    }
}
```

Section 9.9 provides more insight into this subject.

■ **JEFFREY RICHTER** It is common for a type to define multiple overloaded methods for caller convenience. These methods typically allow the caller to pass fewer arguments to the method and then, internally, the method calls a more complex method, passing additional arguments with good default values. If your type offers convenience methods, these methods should not be virtual, but internally they should call the one virtual method that contains the actual implementation of the method (which can be overridden).

6.1.5 Abstractions (Abstract Types and Interfaces)

An abstraction is a type that describes a contract but does not provide a full implementation of that contract. Abstractions are usually implemented as abstract classes or interfaces, and they come with a well-defined set of reference documentation describing the required semantics of the types

implementing the contract. Some of the most important abstractions in .NET include `Stream`, `IEnumerable<T>`, and `Object`. Section 4.3 discusses how to choose between an interface and a class when designing an abstraction.

You can extend frameworks by implementing a concrete type that supports the contract of an abstraction and then using this concrete type with framework APIs consuming (operating on) the abstraction.

A meaningful and useful abstraction that is able to withstand the test of time is very difficult to design. The main difficulty is getting the right set of members—no more and no fewer. If an abstraction has too many members, it becomes difficult or even impossible to implement. If it has too few members for the promised functionality, it becomes useless in many interesting scenarios. Also, abstractions without first-class documentation that clearly spells out all the pre- and post-conditions often end up being failures in the long term. Because of this, abstractions have a very high design cost.

■ **JEFFREY RICHTER** The `ICloneable` interface is an example of very simple abstraction with a contract that was never explicitly documented. Some types that implement this interface's `Clone` method implement it so that it performs a shallow copy of the object, whereas some implementations perform a deep-copy. Because what this interface's `Clone` method should do was never fully documented, when using an object with a type that implements `ICloneable`, you never know what you're going to get. This makes the interface useless.

Too many abstractions in a framework also negatively affect usability of the framework. It is often quite difficult to understand an abstraction without understanding how it fits into the larger picture of the concrete implementations and the APIs operating on the abstraction. Also, names of abstractions and their members are necessarily abstract, which often makes them cryptic and unapproachable without first understanding the broader context of their usage.

However, abstractions provide extremely powerful extensibility that the other extensibility mechanisms cannot often match. They are at the core of many architectural patterns, such as plug-ins, inversion of control (IoC), pipelines, and so on. They are also extremely important for testability of frameworks. Good abstractions make it possible to stub out heavy dependencies for the purpose of unit testing. In summary, abstractions are responsible for the sought-after richness of the modern object-oriented frameworks.

- ✗ **DO NOT** provide abstractions unless they are tested by developing several concrete implementations and APIs consuming the abstractions.
- ✓ **DO** choose carefully between an abstract class and an interface when designing an abstraction. See section 4.3 for more details on this subject.
- ✓ **CONSIDER** providing reference tests for concrete implementations of abstractions. Such tests should allow users to test whether their implementations correctly implement the contract.

■ **JEFFREY RICHTER** I like what the Windows Forms team did: They defined an interface called `System.ComponentModel.IComponent`. Of course, any type can implement this interface. But the Windows Forms team also provided a `System.ComponentModel.Component` class that implements the `IComponent` interface. So a type could choose to derive from `Component` and get the implementation for free, or the type could derive from a different base class and then manually implement the `IComponent` interface. By having available an interface and a base class, developers get to choose whichever works best for them.

■ **STEPHEN TOUB** Before shipping an abstraction, you should plan to validate it by building at least two or three distinct implementations and by using the abstraction in at least two or three distinct consumers. Those tests will provide you with a lot more confidence that you've built something that will actually be usable, and in my experience, will most likely help you to find issues that need to be addressed before you ship.

6.2 Base Classes

Strictly speaking, a class becomes a base class when another class is derived from it. For the purpose of this section, however, a base class is defined as a class designed mainly to provide a common abstraction or for other classes to reuse some default implementation through inheritance. Base classes usually sit in the middle of inheritance hierarchies, between an abstraction at the root of a hierarchy and several custom implementations at the bottom.

Base classes serve as implementation helpers for implementing abstractions. For example, one of the abstractions for ordered collections of items in .NET is the `IList<T>` interface. Implementing `IList<T>` is not trivial, so the framework provides several base classes, such as `Collection<T>` and `KeyedCollection< TKey, TItem >`, that serve as helpers for implementing custom collections.

```
public class OrderCollection : Collection<Order> {
    protected override void SetItem(int index, Order item) {
        if(item==null) throw new ArgumentNullException(...);
        base.SetItem(index,item);
    }
}
```

Base classes are usually not suited to serve as abstractions by themselves because they tend to contain too much implementation. For example, the `Collection<T>` base class contains lots of implementation related to the fact that it implements the non-generic `IList` interface (to integrate better with non-generic collections) and to the fact that it is a collection of items stored in memory in one of its fields.

■ KRZYSZTOF CWALINA `Collection<T>` can also be used directly, without the need to create subclasses, but its main purpose is to provide an easy way to implement custom collections.

As previously discussed, base classes can provide invaluable help for users who need to implement abstractions, but at the same time they can be a significant liability. They add surface area and increase the depth of

inheritance hierarchies, thereby conceptually complicating the framework. For this reason, base classes should be used only if they provide significant value to the users of the framework. They should be avoided if they provide value only to the implementers of the framework, in which case delegation to an internal implementation instead of inheritance from a base class should be strongly considered.

- ✓ **CONSIDER** making base classes abstract even if they don't contain any abstract members. This clearly communicates to the users that the class is designed solely to be inherited from.

▪ **JEREMY BARTON** My interpretation of this guideline is that it's OK to declare the class abstract even if there are no abstract members, but you still need a reason why. If the class works fine on its own, it should probably be instantiable.

- ✓ **CONSIDER** placing base classes in a separate namespace from the main-line scenario types. By definition, base classes are intended for advanced extensibility scenarios and are not interesting to the majority of users. See section 2.2.4 for details.

- ✗ **AVOID** naming base classes with a "Base" suffix if the class is intended for use in public APIs.

For example, despite the fact that `Collection<T>` is designed to be inherited from, many frameworks expose APIs typed as the base class, not as its subclasses, mainly because of the cost associated with a new public type.

```
public Directory {  
    public Collection<string> GetFilenames(){  
        return new FilenameCollection(this);  
    }  
  
    private class FilenameCollection : Collection<string> {  
        ...  
    }  
}
```

The fact that `Collection<T>` is a base class is irrelevant for the user of the `GetFilename` method, so the “Base” suffix would simply create an unnecessary distraction for the user of the method.

6.3 Sealing

One of the features of object-oriented frameworks is that developers can extend and customize them in ways unanticipated by the framework designers. This is both the power and the danger of extensible design. When you design your framework, it is very important to carefully design for extensibility when it is desired, and to limit extensibility when it is dangerous.

■ **KRZYSZTOF CWALINA** Sometimes framework designers want to limit the extensibility of a type hierarchy to a fixed set of classes. For example, let’s say you want to create a hierarchy of living organisms that is split into two and only two subgroups: animals and plants. One way to do so is to make the constructor of `LivingOrganism` internal, and then provide two subclasses (`Plant` and `Animal`) in the same assembly and give them protected constructors. Because the constructor of `LivingOrganism` is internal, third parties can extend `Animal` and `Plant`, but not `LivingOrganism`.

```
public class LivingOrganism {  
    internal LivingOrganism(){}  
    ...  
}  
public class Animal : LivingOrganism {  
    protected Animal() {}  
    ...  
}  
public class Plant : LivingOrganism {  
    protected Plant() {}  
    ...  
}
```

Sealing is a powerful mechanism that prevents extensibility. You can seal either the class or individual members. Sealing a class prevents users

from inheriting from the class. Sealing a member prevents users from overriding a particular member.

```
public class NonNullCollection<T> : Collection<T> {  
    protected sealed override void SetItem(int index, T item) {  
        if(item==null) throw new ArgumentNullException();  
        base.SetItem(index,item);  
    }  
}
```

Because one of the key differentiating points of frameworks is that they offer some degree of extensibility, sealing classes and members will likely feel very abrasive to developers using your framework. Therefore, you should seal only when you have good reasons to do so.

X DO NOT seal classes without having a good reason to do so.

Sealing a class because you cannot think of an extensibility scenario is not a good reason. Framework users like to inherit from classes for various nonobvious reasons, such as adding convenience members. See section 6.1.1 for examples of nonobvious reasons users want to inherit from a type.

Good reasons for sealing a class include the following:

- The class is a static class. For more information on static classes, see section 4.5.
- The class inherits many virtual members, and the cost of sealing them individually would outweigh the benefits of leaving the class unsealed.
- The class is an attribute that requires very fast runtime look-up. Sealed attributes have slightly higher performance levels than unsealed ones. For more information on attribute design, see section 8.2.

■ **BRAD ABRAMS** Having classes that are open to some level of customization is one of the core differences between a framework and a library. With an API library (such as the Win32 API), you basically get what you get. It is very difficult to extend the data structures and APIs. With a framework such as MFC or AWT, clients can extend and customize the classes. The productivity boost from this flexibility is obvious.

■ **KRZYSZTOF CWALINA** People often ask about the cost of sealing individual members. This cost is relatively small, but it is nonzero and should be taken into account. There is development cost (typing in the overrides), testing cost (have you called the base class from the override?), assembly size cost (new overrides), and working set cost (if both the overrides and the base implementation are ever called).

X DO NOT declare protected or virtual members on sealed types.

By definition, sealed types cannot be inherited from. This means that protected members on sealed types cannot be called, and virtual methods on sealed types cannot be overridden.

✓ CONSIDER sealing members that you override.

```
public class FlowSwitch : SourceSwitch {  
    protected sealed override void OnValueChanged() {  
        ...  
    }  
}
```

Problems that can result from introducing virtual members (discussed in section 6.1.4) apply to overrides as well, although to a slightly lesser degree. Sealing an override shields you from these problems starting from that point in the inheritance hierarchy.

In short, part of designing for extensibility is knowing when to limit it, and sealed types are one of the mechanisms by which you do that.

SUMMARY

Designing for extensibility is a critical aspect of designing frameworks. Understanding the costs and benefits provided by various extensibility mechanisms permits the design of frameworks that are flexible while avoiding many of the pitfalls that could lead to trouble later.

This page intentionally left blank

 7

Exceptions

EXCEPTION HANDLING HAS many advantages over return-value-based error reporting. Good framework design helps the application developer realize the benefits of exceptions. This section discusses the benefits of exceptions and presents guidelines for using them effectively.

Exception handling offers the following benefits:

- Exceptions integrate well with object-oriented languages.
Object-oriented languages tend to impose constraints on member signatures that are not imposed by functions in non-object-oriented languages. For example, in the case of constructors, operator overloads, and properties, the developer has no choice in the return value. For this reason, it is not possible to standardize on return-value-based error reporting for object-oriented frameworks. An error reporting method, such as exceptions, which is out of band of the method signature is the only option.

■ **JEFFREY RICHTER** From my perspective, this is the most important reason why exceptions must be used to report problems. In an OO system (like .NET), return codes cannot be used for certain constructs, and an out-of-band mechanism must be used. Now the question becomes whether to use exceptions for everything or whether to use them for the special constructs and use return codes for methods. Obviously, having two different error reporting mechanisms is worse than having one, so it should be obvious that exceptions should be used to report all errors for all code constructs.

■ **CHRIS SELLS** Jeff is right: Always use exceptions for communicating errors and never use them for anything else.

- Exceptions promote API consistency because they are designed to be used for failure reporting and nothing else. In contrast, return values have many uses, of which failure reporting is only a subset. For this reason, it is likely that APIs that report failure through return values will utilize a number of patterns, whereas exceptions can be constrained to specific patterns. The Win32 API is a clear example of this inconsistency: It uses `BOOLs`, `HRESULTS`, and `GetLastError`, among others.
- With return-value-based error reporting, error handling code is always placed very near to the code that could fail. However, with exception handling, the application developer has a choice. Code can be written to catch exceptions near the failure point, or the handling code can be centralized by locating it further up in the call stack.
- Error handling code is more easily localized. Very robust code that reports failure through return values tends to have an if-statement for nearly every functional line of code. The job of these if-statements is to handle the failure case. With exception-based error reporting, robust code can often be written so that a number of methods and operations can be performed with the error handling logic grouped just after the try block or even higher in the call stack.

■ **STEVEN CLARKE** In one API usability study we performed, developers had to call an `Insert` method to insert one or more records into a database. If the method did not throw an exception, the implication was that the records had been inserted successfully. However, this wasn't clear to participants in the study. They expected the method to return the number of records that were successfully inserted. Although return codes should not be used to indicate failure, you can still consider returning status information in the case of a successful operation.

- Error codes can be easily ignored, and often are. Exceptions, by contrast, take an active role in the flow of your code. This makes failures reported as exceptions impossible to ignore and improves the robustness of code.

■ **JEFFREY RICHTER** It should be pointed out that this means that more bugs are caught during the testing of your code and that the code that ships will be more robust. Also, because the shipping code is more robust, there will probably be very few exceptions that get thrown when the shipping code is running out in the wild.

For example, the Win32 API `CloseHandle` fails very rarely, so it is common (and appropriate) for many applications to ignore its return value. However, if any application has a bug that causes `CloseHandle` to be called twice on the same handle, that error would not be obvious unless the code was written to test the return value. A managed equivalent to this API would throw an exception in this case; the unhandled exception handler would log the failure, and the bug would be found.

■ **BRAD ABRAMS** With the return-code error handling model, if the API you are calling fails, the program will muddle on with incorrect results, causing your program to crash or corrupt data at some point in the future. With the exception handling model, when an error occurs the thread is suspended and the calling code is given a chance to handle the exception. When that method doesn't handle the exception, the method that calls it is given a chance to handle it. When no method up the stack handles the exception, your application is terminated. It is better to terminate the application than to let it muddle on—at least the error is eventually fixed.

■ **JEFFREY RICHTER** I completely agree with Brad here. There are also potential security issues that could come up when you let a program continue to run after something fails. I know there are a lot of programmers who do not want their application to crash out in the field, and they are willing to do almost anything to stop that from happening—like swallow exceptions and let the program continue to run. But this is absolutely the wrong thing to do: It is much better for a program to crash than to continue running with unpredictable behavior and potential security vulnerabilities.

Many other examples exist. For example, Windows blue screens occur due to an unhandled exception in kernel-mode code. If a kernel-mode operation fails unexpectedly (an unhandled exception), then Windows doesn't want to just keep running, so it blue screens, all applications are stopped, and all data in memory is lost. In fact, all Microsoft applications, such as Office and Visual Studio, display dialog boxes when they experience an unhandled exception, and they terminate the application. The operating system and these applications are doing the right thing: Do not let your application keep running in the case of an unexpected failure.

■ **BRENT RECTOR** A corollary to the preceding point is that your application should handle only those exceptions that it understands. It is generally impossible to restore the possibly corrupted state of an application to a normal state after “something” has gone wrong. Handle only those exceptions for which your application can respond reasonably. Let all others go unhandled and let the operating system terminate your application.

- Exceptions can carry very rich information describing the cause of the failure.

■ **JEREMY BARTON** I've spent a lot of time in debuggers tracking down the source of an `E_FAIL`, `ERROR_INVALID_PARAM`, or other not-exactly-obvious error code from Win32. Many hours of that time would have been spared if I had the single most valuable part of an exception: the call-stack information.

- Exceptions allow for unhandled exception handlers. Ideally, every application is written to intelligently handle all forms of failure. However, this is unrealistic, because all forms of failure can't be known. With error codes, unexpected failures can be easily ignored by calling code, and the application continues to run with undefined results. With well-written exception-based code, unexpected failures eventually cause an unhandled exception handler to be called. This handler can be designed to log the failure and can also make the choice to shut down the application. This is by far preferable to running with indeterminate results and also provides for logging that makes it possible to add more significant error handling for the previously unexpected case. For example, Microsoft Office uses an unhandled exception handler to gracefully recover and relaunch the application, as well as to send error information to Microsoft to improve the product.

CHRISTOPHER BRUMME You should have a custom unhandled exception handler only if you have application-specific work to do in the handler. If you just want error reporting to occur, the runtime will handle that task automatically for you, and you do not need to (and should not) use an unhandled exception filter (UEF). An application should only register a UEF if it can provide functionality above and beyond the standard error reporting that comes with the system and runtime.

JEFFREY RICHTER In addition to what Chris says, only applications should even think about using a UEF. Components should not be using one of these at all. UEFs are always application-model-specific. In other words, a Windows Form application will probably pop up a window, a Windows NT service will probably log to the event log, and a Web service will probably send a SOAP fault. The component doesn't know in which application model it is being used, so components should leave this decision up to the application developer who is using their component.

- Exceptions promote instrumentation. Exceptions are a well-defined method-failure model. Because of this, it is possible for tools such as debuggers, profilers, performance counters, and others to be intimately aware of exceptions. For example, the Performance Monitor keeps track of exception statistics, and debuggers can be instructed to break when an exception is thrown. Methods that return failure do not share in the benefits of instrumentation.

■ **BRAD ABRAMS** It is worthwhile to note that the exception handling feature can easily be defeated by poorly designed frameworks. If the framework designer chooses to use return codes for error handling, none of the benefits apply.

7.1 Exception Throwing

The exception-throwing guidelines described in this section require a good definition of the meaning of execution failure. Execution failure occurs whenever a member cannot do what it was designed to do (what the member name implies). For example, if the `OpenFile` method cannot return an opened file handle to the caller, it would be considered an execution failure.

■ **KRZYSZTOF CWALINA** One of the biggest misconceptions about exceptions is that they are for “exceptional conditions.” The reality is that they are intended for communicating error conditions. From a framework design perspective, there is no such thing as an “exceptional condition.” Whether a condition is exceptional or not depends on the context of usage, but reusable libraries rarely know how they will be used. For example, `OutOfMemoryException` might be exceptional for a simple data entry application; it’s not so exceptional for applications doing their own memory management (e.g., SQL Server). In other words, one man’s exceptional condition is another man’s chronic condition.

Most developers have become comfortable with using exceptions for usage errors such as division by zero or null references. In .NET, exceptions are used for all error conditions, including execution errors. At first, it can be difficult to embrace exception handling as the means of reporting all failures. However, it is important to design all public methods of a framework to report method failures by throwing an exception.

A variety of excuses might be offered for not using exceptions, but most boil down to one of two perceptions: that exception handling syntax is undesirable, so returning an error code is somehow preferable, or that a thrown exception does not perform as well as returning an error code. The performance concerns are addressed in sections 7.5.1 and 7.5.2. The concern related to syntax is largely a matter of familiarity. We recognize that developers who are new to exceptions find the syntax awkward, but this becomes much less of a problem over time as developers get used to exception handling.

■ JEFFREY RICHTER In addition, different programming languages will offer different syntax for developers to express exception handling. Even for existing languages, like C#, new syntax could be provided in the future. Or code editors could spit out the code, making the coding a little less tedious. Certainly, I agree that syntax should not be a factor in determining exception usage.

X DO NOT return error codes.

Exceptions are the primary means of reporting errors in frameworks. The beginning section of the chapter describes the benefits of exceptions in detail.

■ **KRZYSZTOF CWALINA** It's OK for exceptions to have a property returning some kind of error code, but I would be very careful about this. Each exception can carry two main pieces of information: the exception message explaining to the developer what went wrong and how to fix it, and the exception type that should be used by handlers to decide what programmatic action to take. If you think you need to have a property on your exception that would return additional error code, ask yourself who this code is for. Is it for the developer or for the exception handler? If for the developer, add additional information to the message. If for the handlers, add a new exception type.

■ **CHRISTOPHE NASARRE** The error code property in a `System.Exception` is useful when you need to bind the issue to external documentation that details how to fix the problem. This is particularly important with different layers of a framework where the upper layer does not always (and should not) know how the lower layer is implemented. For example, a Web service call could get an exception because the associated database on the server is not started. In that case, you don't expect the client code to explain how to restart the database when the exception is caught. Instead, you provide a pointer to the documentation and the right section where the whole restarting process is explained.

X DO report execution failures by throwing exceptions.

If a member cannot successfully do what it is designed to do, it should be considered an execution failure, and an exception should be thrown.

■ **JASON CLARK** A good rule of thumb is that if a method does not do what its name suggests, it should be considered a method-level failure, resulting in an exception. For example, a method called `ReadByte` should throw an exception if there are no more bytes left in a stream to be read. Meanwhile, a method named `ReadChar` should not throw an exception when it reaches end of stream, because EOF is a valid char (in most character sets) that can be returned in this case while still achieving what the method's name suggests. So a char can be "read" successfully at the end of a stream, whereas a byte cannot.

X **CONSIDER** terminating the process by calling `System.Environment.FailFast` instead of throwing an exception if your code encounters a situation where immediate process termination is necessary.

■ **CHRISTOPHER BRUMME** An example of the operating system performing a fail fast occurs when the stack is so corrupt that the operating system cannot propagate exceptions through it. In this case, the invariant expectations of the application (i.e., that exceptions propagate) can no longer be satisfied, so the application must terminate.

An example of the CLR performing a fail fast occurs when the Garbage Collector (GC) heap is so corrupt that we can no longer track managed objects. In this case, a process-wide resource is required for further processing, but it is now corrupt and cannot be returned to a functional state.

Similar legitimate reasons for fail fast can occur in managed code. For example, if the application cannot revert a security impersonation on a thread (i.e., exceptions are thrown from `WindowsImpersonationContext.Dispose`), that thread must be doomed. But if there is no good way for the application to ensure that no more code runs on this thread—perhaps because it is the Finalizer thread or a ThreadPool thread—then the process must be destroyed. `Environment.FailFast` can be used for this purpose.

■ **JEREMY BARTON** `Environment.FailFast` is a very big hammer and shouldn't be wielded without thought. `FailFast` will cause all threads and I/O to terminate immediately, which might leave half-written files or other data corruption and server processes will just cease replying to their clients. If you've somehow detected that you're leaking data across tenant partitions, *maybe* it's worth an immediate shutdown.

X **DO NOT** use exceptions for the normal flow of control, if possible. Prefer Tester–Doer or the Try pattern to throwing exceptions in common cases. Except for system failures and operations with potential race conditions, framework designers should design APIs so users can write code that does not throw exceptions. For example, you can provide a way to check preconditions before calling a member so users can write code that does not throw exceptions.

```
ICollection<int> collection = ...  
if(!collection.IsReadOnly){  
    collection.Add(additionalNumber);  
}
```

The member used to check preconditions of another member is often referred to as a tester, and the member that actually does the work is called a doer. See section 7.5.1 for more information on the Tester–Doer Pattern.

Sometimes the Tester–Doer Pattern can have an unacceptable performance overhead. In such cases, consider the Try Pattern instead (see section 7.5.2 for more information).

■ **JEFFREY RICHTER** The Tester–Doer Pattern should be used with caution. The potential problem occurs when you have multiple threads accessing the object at the same time. For example, one thread could execute the tester method, which reports that all is OK, and before the doer method executes, another thread could change the object, causing the doer to fail. Although this pattern might improve performance, it introduces race conditions and must be used with extreme caution.

✓ **CONSIDER** the performance implications of throwing exceptions. Throw rates greater than 100 per second are likely to noticeably impact the performance of most applications. See section 7.5 for details.

✓ **DO** document all exceptions thrown by publicly callable members because of a violation of the member contract (rather than a system failure) and treat them as part of your contract.

Exceptions that are a part of the contract should not change from one version to the next (i.e., the exception type should not change, and new exceptions should not be added).

✗ **DO NOT** have public members that can either throw or not throw based on some option.

```
// bad design  
public Type GetType(string name, bool throwError)
```

■ **BRAD ABRAMS** An API such as this one usually reflects the inability of the framework designer to make a decision. A method either completes successfully or it does not, in which case it should throw an exception. Failure to decide forces the decision on the caller of the API, which likely does not have enough context on implementation details of the API to make an informed decision.

✗ **DO NOT** have public members that return exceptions as the return value or an out parameter.

Returning exceptions from public APIs instead of throwing them defeats many of the benefits of exception-based error reporting.

```
// bad design
public Exception DoSomething() { ... }
```

✓ **CONSIDER** using exception builder methods.

It is common practice to throw the same exception from different places. To avoid code bloat, use helper methods that create exceptions and initialize their properties.

Also, members that throw exceptions are less likely to be inlined. Moving the throw statement inside the builder might allow the member to be inlined. For example:

```
class File{
    string fileName;

    public byte[] Read(int bytes){
        if (!ReadFile(handle, bytes)) ThrowNewFileIOException(...);
    }

    void ThrowNewFileIOException(...){
        string description = // build localized string
        throw new FileIOException(description);
    }
}
```

X DO NOT throw exceptions from exception filter blocks.

When an exception filter raises an exception, the exception is caught by the CLR, and the filter returns `false`. This behavior is indistinguishable from the filter executing and returning `false` explicitly, so it is very difficult to debug.

```
' VB sample
' This is bad design. The exception filter (When clause)
' may throw an exception when the InnerException property
' returns null.

Try
    ...
Catch e As ArgumentException _
When e.InnerException.Message.StartsWith("File")
    ...
End Try
```

X AVOID explicitly throwing exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

This section covered the general issues on throwing exceptions. The next section deals with deciding which type of exception to throw.

7.2 Choosing the Right Type of Exception to Throw

After you have decided when you need to throw exceptions, the next step is to pick the right type of exception to throw. This section provides those guidelines.

You should first decide whether the error condition represents a usage error or an execution error.

A usage error represents an incorrectly written program and is something that can be avoided by changing the code that calls your API. There is no reason to programmatically handle usage errors; instead, the calling code should be changed. For example, if a routine gets into an error state when a null is passed as one of its arguments (an error condition usually represented by an `ArgumentNullException`), the calling code can be modified to ensure that a null is never passed. In other words, usage errors can be fixed at compile-time, and the developer can ensure that they never occur at runtime.

An execution error is something that cannot be completely avoided by writing “better” code. For example, `File.Open` throws `FileNotFoundException` when it tries to open a file that does not exist. Even if the caller of the API checks whether the file exists before calling `File.Open`, the file can get deleted or corrupted between the call to `File.Exists` and the call to `File.Open`. Execution errors need to be further divided into two groups: program errors and system failures.

A program error is an execution error that can be handled programmatically. For example, if `File.Open` throws `FileNotFoundException`, the calling code can catch the exception and handle it by creating a new file. This is in contrast to the usage error example described earlier, where you would never write code that first passes an argument to a method, catches the `ArgumentNullException`, and in the handler calls the method with a non-null argument.

A system failure is an execution error that cannot be handled programmatically. For example, you really cannot handle an out-of-memory exception resulting from the Just-in-Time (JIT) compiler running out of memory.¹

X DO NOT create new exception types to communicate usage errors. Throw one of the existing base library exceptions instead. See section 7.3 for detailed usage guidelines for the most common standard exception types.

Usage errors need to be communicated to the developer calling the API and should not be handled directly in code. The exception type is generally less important to developers trying to fix their code than the message string. Therefore, for exceptions that are thrown as a result of usage errors, you should concentrate on designing a really good (i.e., explanatory) exception message and using one of the existing .NET Framework exception types: `ArgumentNullException`, `ArgumentException`, `InvalidOperationException`, `NotSupportedException`, and so on.

1. So, by default, the CLR shuts down the process in such a case, instead of throwing an exception.

- ✓ **CONSIDER** creating and throwing custom exceptions if you have a program error condition that can be programmatically handled in a different way than any other existing exception. Otherwise, throw one of the existing exceptions. See section 7.4 for details on creating custom exceptions.

Program errors are errors that can be, and often are, handled in code. The way to handle such errors is to catch the exception and execute some “compensating” logic. Whether the catch statement executes is determined by the type of the exception the catch block claims it can handle. Thus, a program error is an error condition (actually the only error condition) where the exception type matters, so you should consider creating a new exception type.

If you think you are dealing with a program error, validate this belief by actually writing code or describing very precisely what the catch handler would do when it catches the exception to allow the program to continue its execution. If you cannot describe it, or if the error can be avoided by changing the calling code, you are probably dealing with a usage error or a system failure.

- ✗ **DO NOT** create a new exception type if the exception would not be handled differently than an existing framework exception. Throw the existing framework exception in such case. See section 7.3 for detailed usage guidelines for the most common standard exception types.

Of course, the existing exception type must make sense for your error condition. For example, you don’t want to throw `FileNotFoundException` from an API unrelated to accessing files.

Also, make sure that reusing the exception will not make an error condition ambiguous. That is, make sure that the code that wants to handle the specific error will always be able to tell whether the exception was thrown because of the error or because of some other error that happens to use the same exception. For example, you don’t want to throw (reuse) `FileNotFoundException` from a routine that calls the built-in file I/O APIs, and so can throw the same exception, unless it does not matter to the calling code whether your code or the underlying framework method threw the exception.

- ✓ **DO** create a new exception type to communicate a unique program error that cannot be communicated using an existing framework exception. See section 7.4 for details on creating custom exceptions.

■ **KRZYSZTOF CWALINA** Keep in mind that it's not a breaking change to change an exception your code throws to a subtype of that exception. For example, if the first version of your framework throws `FooException`, in any future version of your library, you can start throwing a subtype for `FooException` without breaking code that was compiled against the previous version of your library. This means, when in doubt, I would consider not creating new exception types until you are sure you need them. At that point, you can create a new exception type by inheriting from the currently thrown type. Sometimes it might result in slightly strange exception hierarchy (e.g., a custom exception inheriting from `InvalidOperationException`), but it's not a big deal in comparison to the cost of having unnecessary exception types in your library, which makes the library more complex, adds development cost, increases the working set, and so on.

- ✗ **AVOID** creating APIs that when called can result in a system failure. If such a failure can occur, call `Environment.FailFast` instead of throwing an exception when the system failure occurs.

System failures are errors that cannot be handled by the developer or the program. The good thing is that system failures are extremely rare in reusable libraries. They are mostly caused by the execution engine. The best way to shut down a process in such cases is to call `Environment.FailFast`, which logs the state of the system—something that is very useful in diagnosing the problem.

- ✗ **DO NOT** create and throw new exceptions just to have your feature name in the exception type or namespace name.

- ✓ **DO** throw the most specific (the most derived) exception that makes sense.

For example, throw `ArgumentNullException` and not its base type, `ArgumentException`, if a null argument is passed.

■ **JEFFREY RICHTER** Throwing `System.Exception`, the base class of all CLS-compliant exceptions, is always the wrong thing to do.

■ **BRENT RECTOR** As described in more detail later, catching `System.Exception` is nearly always the wrong thing to do as well.

Now that you have chosen the correct exception type, you can focus on ensuring that the error message your exception delivers says what you need it to say.

7.2.1 Error Message Design

The guidelines in this section define best practices for creating the exception message text.

✓ **DO** provide a rich and meaningful message text targeted at the developer when throwing an exception.

The message should explain the cause of the exception and clearly describe what needs to be done to avoid the exception.

■ **BRAD ABRAMS** This guideline applies to frameworks. It is likely quite appropriate in application code for the exception message to be targeted at an admin or even an end user. The high-level advice is that the message text should be targeted at whoever will have to make sense out of it.

✓ **DO** ensure that exception messages are grammatically correct.

Top-level exception handlers can show the exception message to application end users.

✓ **DO** ensure that each sentence of the message text ends with a period.

This way, code that displays exception messages to the user does not have to handle the case in which a developer forgot the final period, which is relatively cumbersome and expensive.

✗ **AVOID** question marks and exclamation points in exception messages.

✗ **DO NOT** disclose security-sensitive information in exception messages.

✓ **CONSIDER** localizing the exception messages thrown by your components if you expect your components to be used by developers speaking different languages.

■ **JEFFREY RICHTER** Exception messages are not relevant when an exception is handled; they only come into play when an exception is unhandled. An unhandled exception indicates a real bug in the application because the application will be terminated. The only way to fix the application now is to have the failure reported, the source code modified and recompiled, and the deployments in the field updated. So the error messages should be geared toward helping developers fix bugs in their code. End users should not see these messages.

When a Microsoft Office application gets an unhandled exception, all the dialog box says is something like this: "Microsoft Word has encountered a problem and needs to close. We are sorry for the inconvenience." There is a bit more to the message, but there is nothing computerese in the message. The user does have the option of clicking a button that will show what's in the error report that gets sent back to Microsoft, and this error report is very geeky.

7.2.2 Exception Handling

Having decided when to throw exceptions, their type, and the message design, the next thing to focus on is how to handle exceptions. First, let's define some terminology. You handle an exception when you have a catch block for a specific exception type and you fully understand the implications of continuing execution of the application after executing the catch block. For example, if you try to open a configuration file, catch `FileNotFoundException` if the file is not present, and fall back to the default configuration. You swallow an exception when you catch a very generic type of exception (usually) and, without fully understanding or responding to the failure, continue with the execution of the application.

✗ **DO NOT** swallow errors by catching nonspecific exceptions, such as `System.Exception`, `System.SystemException`, and so on, in framework code.

```
try{
    File.Open(...);
}
catch(Exception e){ } // swallow "all" exceptions - don't do this!
```

A legitimate reason for catching a nonspecific exception is so that you can transfer that exception to another thread. This can happen, for example, when a GUI application transfers an operation to the UI thread, when doing asynchronous programming, using thread pool operations, and so on. Obviously, if you are transferring an exception to another thread, you aren't actually swallowing it.

■ **JOE DUFFY** When transferring exceptions across threads, it is imperative that you have put safeguards in place that prevent the exception from getting missed. For example, if you catch the exception and stuff it into a list that no other thread in the system ever gets around to checking, you have essentially swallowed it. And the effects can be as disastrous as ignoring an error code in Win32.

✗ **AVOID** swallowing errors by catching nonspecific exceptions, such as `System.Exception`, `System.SystemException`, and so on, in application code.

There are cases when swallowing errors in applications is acceptable, but such cases are rare.

If you decide to swallow exceptions, you must realize that you don't know exactly what went wrong, so you generally cannot predict what state might now be inconsistent. By swallowing exceptions, you are making a trade-off that the value of continuing to execute code in this application domain or process exceeds the risk of executing in the face of inconsistencies. Because a security attack might be able to exploit those inconsistencies, your decision here has deep consequences.

■ **VANCE MORRISON** There is surprisingly little you can do when catching an exception unless you know the following:

- Exactly what component threw it, and what the contract was (what state the object was left in)
- The WHOLE call stack between the thrower and the catcher, and you have determined that every method in that stack properly cleaned up any global state changes before returning

If you don't know these two items, then you don't know that some global state has been left in a "half done" state, which will cause strange errors if the program continues. This is *especially* true for exceptions like `OutOfMemoryException`, `StackOverflowException`, and `ThreadAbortException`, which can happen in a great many places.

In practice, you don't know the whole call stack unless the caller and the thrower are right next to each other. This means that if you are trying to catch "any" exception, you should not continue execution. You should only be doing things like error logging and persisting critical information as a prelude to shutting down (and possibly relaunching the process). These latter operations are best done in a finally block.

✗ **DO NOT** exclude any special exceptions when catching for the purpose of transferring exceptions.

```
catch (Exception e) {  
    // bad code  
    // do not do this!  
    if (e is OutOfMemoryException ||  
        e is AccessViolationException  
    ) throw;  
    ...  
}
```

✓ **CONSIDER** catching a specific exception when you understand why it was thrown in a given context and can respond to the failure programmatically.

■ **JEFFREY RICHTER** You should only catch an exception when you know you can gracefully recover from it. When performing some operation, you might know why an exception was thrown—but if you don't know how you'd recover from it, do not catch it.

X DO NOT overcatch. Exceptions should often be allowed to propagate up the call stack.

■ **JEFFREY RICHTER** This can't be stressed enough. Far too many times I've seen developers catch an exception, which basically hides a bug in their program. Do not catch; flush out the bugs during testing and ship a better, more robust product.

■ **ERIC GUNNERSON** Exception handling is robust by default—every exception that is thrown will propagate up, looking for a handler. Every place you write an exception handler is a chance for you to remove robustness from the system, and if you make a mistake it will often be hard to track down. So be minimal about it.

✓ DO use try-finally and avoid using try-catch for cleanup code. In well-written exception code, try-finally is far more common than try-catch. It might seem counterintuitive at first, but catch blocks are not needed in a surprising number of cases. However, you should always consider whether try-finally could be of use for cleanup to ensure a consistent state of the system when an exception is thrown. Usually, the cleanup logic rolls back resource allocations.

```
FileStream stream = null;  
try{  
    stream = new FileStream(...);  
    ...  
}finally{  
    if(stream != null) stream.Close();  
}
```

C# and VB.NET provide the `using` statement, which can be used instead of a plain `try-finally` statement to clean up objects implementing the `IDisposable` interface.

```
using(var stream = new FileStream(...)){  
    ...  
}
```

■ **CHRISTOPHER BRUMME** If you use catch clauses for cleanup, you should know that any code that comes after the end of the catch might not be executed. In CLR 2.0, the finally and catch blocks get special protection from thread aborts. Code after a catch does not.

■ **BRENT RECTOR** Don't use catch blocks for cleanup code. Use catch blocks for error recovery code and finally blocks for cleanup code. A catch block only runs when an exception of a particular type occurs within the try block. A finally block always runs. If you always need to clean up (the typical case), you need to perform that logic in a finally block.

■ **JEFFREY RICHTER** I completely agree with this guideline. I recommend that almost all cleanup code go inside finally blocks, and it is quite convenient that C# and VB.NET offer many language constructs that emit `try-finally` blocks automatically for you. Examples are C#/VB's `using` statement, C#'s `lock` statement, VB's `SyncLock` statement, C#'s `foreach` statement, and VB's `For Each` statement. In addition, when you define a finalizer in C#, the compiler makes sure that the base class's `Finalize` is called by placing the call within a finally block. In fact, I have designed many of my own types to have methods that return an `IDisposable` object so that they can be easily used with C#/VB's `using` statement.

✓ **DO** prefer using an empty throw when catching and rethrowing an exception. This is the best way to preserve the exception call stack.

```
public void DoSomething(FileStream file){  
    long position = file.Position;  
    try{
```

```
    ... // do some reading with the file
}catch{
    file.Position = position; // unwind on failure
    throw; // rethrow
}
}
```

■ **CHRISTOPHER BRUMME** Every time you catch and throw and rethrow, you impact the debuggability of the system. Debugging of exceptions is based on the notion that we detect exceptions going unhandled during the first pass, when no state changes have occurred. By attaching a debugger at that time, we see the state at the time when the exception was thrown. If there is a series of catch and throw and rethrow segments up the stack, then debugging might be limited to inspecting the very last segment. This is an arbitrary distance from the original fault. The same state changes will reduce the effectiveness of Watson dumps.

■ **JEFFREY RICHTER** In addition, when you throw a new exception (versus rethrowing the original exception), you are reporting a different failure than the failure that actually occurred. This also hurts your ability to debug the application. Therefore, always prefer a rethrow to a throw, and try to avoid catching and (re)throwing altogether.

✓ **DO** use `ExceptionDispatchInfo` when rethrowing an exception from a different context.

If you need to move an exception to a different thread, or anything else that prevents using the empty throw, the `ExceptionDispatchInfo` class persists the call stack across the rethrow.

```
private ExceptionDispatchInfo _savedExceptionInfo;

private void BackgroundWorker() {
    try {
        ...
    } catch (Exception e) {
        _savedExceptionInfo = ExceptionDispatchInfo.Capture(e);
    }
}
```

```
public object GetResult() {
    if (_done) {
        if (_savedExceptionInfo != null) {
            _savedExceptionInfo.Throw();

            // The method throws,
            // but the compiler flow doesn't understand that,
            // so an extra return statement is required.
            return null;
        }
        ...
    }
    ...
}
```

- X DO NOT** handle non-CLS-compliant exceptions (exceptions that don't derive from `System.Exception`) using a parameterless catch block. Languages that support non-Exception-derived exceptions are free to handle these exceptions.

```
try{ ... }
catch{ ... }
```

CLR 2.0 has been modified to deliver noncompliant exceptions to compliant catch blocks wrapped up in `RuntimeWrappedException`.

7.2.3 Wrapping Exceptions

Sometimes exceptions raised in a lower layer would be meaningless if they were allowed to propagate from a higher layer. In such cases, it is sometimes beneficial to wrap the lower-layer exception in an exception that is meaningful to the users of the higher layer. For example, a `FileNotFoundException` would be completely meaningless if allowed to propagate from transaction management APIs. The user of the transaction APIs might not even be aware that transactions can be stored on the file system. In other cases, the actual exception type is less important than the fact that it was raised in some particular code path. For example, even if an otherwise benign exception is thrown from a static constructor, the type will be unusable in the current application domain. In such a case, it is much more important to

communicate to the user that an exception occurred in the constructor than what caused the exception. Therefore, the CLR wraps all exceptions propagating out of the static constructors into `TypeInitializationException`.

■ **STEVEN CLARKE** Make sure that the terminology used in the error message will make sense in the context in which it is being consumed. In an API usability study we ran, the lower-level details of the API had been factored out so that developers were only exposed to the high-level details. Unfortunately, exceptions were thrown from the lower level and caught at the higher level. The error message described concepts that would have made sense only to someone working at the lower level of the API and thus was no good at communicating the reason for the problem.

✓ **CONSIDER** wrapping specific exceptions thrown from a lower layer in a more appropriate exception if the lower layer is just an implementation detail for the operation.

Such wrapping should be quite rare in a typical framework. It will likely have a negative impact on the ability to debug.

```
try {  
    // read the transaction file  
}  
catch(FileNotFoundException e) {  
    throw new TransactionFileMissingException(...,e);  
}
```

■ **RICO MARIANI** Chris Brumme's previous annotation on rethrowing also applies to this context. The example given illustrates the point nicely: For rethrowing to help, the original exception context has to be nearly meaningless and certainly uninteresting to debug. That the original call stack was uninteresting is the controlling factor, in my opinion. In this case, nobody thinks that there is anything wrong with the file opening services—you might as well rethrow something more meaningful to indicate where we had trouble opening files and what file we were trying to open.

■ **ERIC GUNNERSON** Another way of stating this is: “Don’t wrap if your users will ever want to look at the inner exception.” By far the worst position to be in is to have to look at an inner exception and base your behavior on the type of the inner exception.

✗ **AVOID** catching and wrapping nonspecific exceptions.

The catch and wrap practice is often undesired and is just another form of swallowing errors. Exceptions to this rule include cases in which the wrapper exception communicates a severe condition that is much more interesting to the caller than the actual type of the original exception. For example, the `TypeInitializationException` wraps all exceptions propagating from static constructors.

✓ **DO** specify the inner exception when wrapping exceptions.

```
throw new ConfigurationFileMissingException(...,e);
```

■ **JEFFREY RICHTER** It cannot be stressed enough how carefully this needs to be thought through. When in doubt, do not wrap an exception with another. An example in the CLR where wrapping is known to cause all kinds of trouble is with reflection. When you invoke a method using reflection, if that method throws an exception, the CLR catches it and throws a new `TargetInvocationException`. This is incredibly annoying because it hides the actual method and location in the method that had the problem. I have wasted much time trying to debug my code because of this exception wrapping.

7.3 Using Standard Exception Types

This section describes the standard exceptions provided by .NET and the details of their usage. The list provided here is by no means exhaustive. Please refer to the .NET API reference documentation for usage of other framework exception types.

7.3.1 Exception and SystemException

- ✗ **DO NOT** throw System.Exception or System.SystemException.
- ✗ **DO NOT** catch System.Exception or System.SystemException in framework code, unless you intend to rethrow.
- ✗ **AVOID** catching System.Exception or System.SystemException, except in top-level exception handlers.

7.3.2 ApplicationException

- ✗ **DO NOT** throw or derive from System.ApplicationException.

■ **JEFFREY RICHTER** System.ApplicationException is a class that should not be part of .NET. The original idea was that classes derived from SystemException would indicate exceptions thrown from the CLR (or system) itself, whereas non-CLR exceptions would be derived from ApplicationException. However, a lot of exception classes didn't follow this pattern. For example, TargetInvocationException (which is thrown by the CLR) is derived from ApplicationException. So the ApplicationException class lost all meaning. The reason to derive from this base class is to allow some code higher up the call stack to catch the base class. It was no longer possible to catch all application exceptions.

7.3.3 InvalidOperationException

- ✓ **DO** throw an InvalidOperationException if the object is in an inappropriate state.

The System.InvalidOperationException exception should be thrown if a property set or a method call is not appropriate given the object's current state. An example of this is writing to a FileStream that's been opened for reading.

■ **KRZYSZTOF CWALINA** The difference between `InvalidOperationException` and `ArgumentException` is that `ArgumentException` does not rely on the state of any other object besides the argument itself to determine whether it needs to be thrown. For example, if client code tries to access a nonexistent resource, `InvalidOperationException` should be thrown. On the other hand, if client code tries to access a resource using a malformed identifier, `ArgumentException` should be thrown.

7.3.4 ArgumentException, ArgumentNullException, and ArgumentOutOfRangeException

- ✓ **DO** throw `ArgumentException` or one of its subtypes if bad arguments are passed to a member. Prefer the most derived exception type, if applicable.
- ✓ **DO** set the `ParamName` property when throwing one of the subclasses of `ArgumentExceptions`.

This property represents the name of the parameter that caused the exception to be thrown. Note that the property can be set using one of the constructor overloads.

```
public static FileAttributes GetAttributes(string path){
    if(path==null){
        throw new ArgumentNullException(nameof(path),...);
    }
}
```

- ✓ **DO** use `value` for the name of the implicit value parameter of property setters.

```
public FileAttributes Attributes {
    set {
        if(value==null){
            throw new ArgumentNullException(nameof(value),...);
        }
    }
}
```

■ **JEFFREY RICHTER** It is very unusual to find code that catches any of these argument exceptions. When one of these exceptions is thrown, you almost always want the application to die. Then look at the exception stack trace, fix your source code so that you are passing the right argument, recompile the code, and retest.

■ **JASON CLARK** Jeffrey is absolutely correct here. And it is the infrequency with which these exception types are caught that makes them safe to broadly reuse for throwing in your code.

7.3.5 NullReferenceException, IndexOutOfRangeException, and AccessViolationException

✗ **DO NOT** allow publicly callable APIs to explicitly or implicitly throw `NullReferenceException`, `AccessViolationException`, or `IndexOutOfRangeException`. These exceptions are reserved and thrown by the platform and in most cases indicate a bug.

Do argument checking to avoid throwing these exceptions. Throwing these exceptions exposes implementation details of your method that might change over time.

7.3.6 StackOverflowException

✗ **DO NOT** explicitly throw `StackOverflowException`. This exception should be explicitly thrown only by the CLR.

✗ **DO NOT** catch `StackOverflowException`.

It is almost impossible to write managed code that remains consistent in the presence of arbitrary stack overflows. The unmanaged parts of the CLR remain consistent by using probes to move stack overflows to well-defined places rather than by backing out from arbitrary stack overflows.

■ **CHRISTOPHER BRUMME** Generally you should never special-case StackOverflowException. The default policy in CLR 2.0 will result in a fast shutdown of the process when a stack overflows. There are strong security and reliability reasons for this decision. In a normal process, a stack overflow won't even result in a managed exception.

7.3.7 OutOfMemoryException

✗ **DO NOT** explicitly throw OutOfMemoryException. This exception is to be thrown only by the CLR infrastructure.

■ **CHRISTOPHER BRUMME** At one end of the spectrum, an OutOfMemoryException could be the result of a failure to obtain 12 bytes for implicitly autoboxing, or a failure to JIT some code that is required for critical backout. These cases are catastrophic failures and ideally would result in termination of the process. At the other end of the spectrum, an OutOfMemoryException could be the result of a thread asking for a 1 GB byte array. The fact that we failed this allocation attempt has no impact on the consistency and viability of the rest of the process.

The sad fact is that CLR 2.0 cannot distinguish among any points on this spectrum. In most managed processes, all OutOfMemoryExceptions are considered equivalent, and they all result in a managed exception being propagated up the thread. However, you cannot depend on your backout code being executed, because we might fail to JIT some of your backout methods, or we might fail to execute static constructors required for backout.

Also, keep in mind that all other exceptions can get folded into an OutOfMemoryException if there isn't enough memory to instantiate those other exception objects. Also, we will give you a unique OutOfMemoryException with its own stack trace if we can. But if we are tight enough on memory, you will share an uninteresting global instance with everyone else in the process.

My best recommendation is that you treat OutOfMemoryException like any other application exception. You make your best attempts to handle it and remain consistent. In the future, I hope the CLR can do a better job of distinguishing catastrophic OOM from the 1 GB byte array case. If so, we might provoke termination of the process for the catastrophic cases, leaving the application to deal with the less risky ones. By treating all OOM cases as the less risky ones, you are preparing for that day.

■ **JAN KOTAS** It is common practice to explicitly throw `OutOfMemoryException` when interoperating with unmanaged code to convert unmanaged error codes into managed exceptions. It is perhaps the only situation where it is acceptable to throw `OutOfMemoryException` explicitly.

7.3.8 `ComException`, `SEHException`, and `ExecutionEngineException`

✗ **DO NOT** explicitly throw `ComException`, or `SEHException`, or `ExecutionEngineException`. These exceptions are to be thrown only by the CLR infrastructure.

■ **CHRISTOPHER BRUMME** If you see an `ExecutionEngineException` thrown, treat it like any other framework exception. It was thrown from a place where the CLR is not actually in an invalid state. For instance, some invalid operations in the security code throw this exception, and backward-compatibility requirements prevent us from changing the exception type.

7.3.9 `OperationCanceledException` and `TaskCanceledException`

- ✓ **DO** throw `OperationCanceledException` to represent any caller-initiated abort or cancellation.
- ✓ **DO** prefer catching `OperationCanceledException` to `TaskCanceledException`.

`TaskCanceledException` derives from `OperationCanceledException`. The type is used internally by the Task execution engine, but otherwise is barely distinguishable from the parent type. Catch blocks that have special handling for `TaskCanceledException` should generally also have handling for `OperationCanceledException`.

7.3.10 `FormatException`

- ✓ **DO** throw `FormatException` to indicate the input string in a text parsing method does not conform to the required, or specified, format.

- ✓ **DO** throw `FormatException` to indicate the format specifier string to a text parsing or text formatting method is invalid.

`DateTimeOffset.Parse(string input)` throws a `FormatException` as an execution error indicating that the input value, which is generally read as data, cannot be parsed into a `DateTimeOffset` value. Conversely, `DateTimeOffset.ToString(string format)` throws a `FormatException` as a usage error indicating that the format specifier, which is generally a value explicitly specified in the calling code, is not a value that is supported. The dual usage of this exception can be ambiguous for methods like `DateTimeOffset.ParseExact(string input, string format, IFormatProvider formatProvider)`, but only when both the input value and the format value were read as data. When the format value is specified explicitly in calling code, the developer can use the message to debug an invalid format specifier once; then all future calls that result in a `FormatException` are a result of the input value.

7.3.11 PlatformNotSupportedException

- ✓ **DO** throw `PlatformNotSupportedException` to indicate the operation cannot complete in the current runtime environment but could on a different runtime or operating system.

`PlatformNotSupportedException` is the primary exception to throw when a method is declared in a cross-compiled library to avoid a `MissingMethodException` from the CLR, but cannot be implemented in the current build configuration. This exception is also commonly thrown when the implementation of a method requires a newer version of the current operating system or specialty hardware that is not present.

7.4 Designing Custom Exceptions

In some cases, it will not be possible to use existing exceptions (section 7.2 describes in detail how to make such determination). In those cases, you'll

need to define custom exceptions. The guidelines in this section provide help on doing that.

- ✓ **DO** derive exceptions from `System.Exception` or one of the other common base exceptions.

- ✗ **AVOID** deep exception hierarchies.

There are a limited number of cases where it's a good idea to create a more elaborate hierarchy of exceptions, but they're extremely rare. The reason is that code that handles exceptions almost never cares about the hierarchy, because it almost never wants to handle more than one error at a time. If two or more errors can be handled the same way, they should not be expressed using different exception types. Of course, there are exceptions to the rule and in some cases, it may be better to create a deeper hierarchy.

- ✓ **DO** end exception class names with the "Exception" suffix.
- ✓ **DO** make exceptions runtime serializable. An exception must be serializable to work correctly across application domain and remoting boundaries.
- ✓ **DO** provide (at least) these common constructors on all exceptions. Make sure the names and types of the parameters are exactly as shown in the following example.

```
public class SomeException: Exception, ISerializable {  
    public SomeException();  
    public SomeException(string message);  
    public SomeException(string message, Exception inner);  
  
    // this constructor is needed for serialization.  
    protected SomeException(SerializationInfo info, StreamingContext  
    context);  
}
```

- ✓ **CONSIDER** providing exception properties for programmatic access to extra information (besides the message string) relevant to the exception.

7.5 Exceptions and Performance

One common concern related to exceptions is that if exceptions are used for code that routinely fails, the performance of the implementation will be unacceptable. This is a valid concern. When a member throws an exception, its performance can be orders of magnitude slower. However, it is possible to achieve good performance while strictly adhering to the exception guidelines that disallow using error codes. Two patterns described in this section suggest ways to achieve this goal.

X DO NOT use error codes because of concerns that exceptions might affect performance negatively.

To improve performance, it is possible to use either the Tester–Doer Pattern or the Try Pattern, described in the next two sections.

7.5.1 The Tester–Doer Pattern

Sometimes performance of an exception-throwing member can be improved by breaking the member into two. Let's look at the Add method of the `ICollection<T>` interface.

```
ICollection<int> numbers = ...  
numbers.Add(1);
```

The method Add throws an exception if the collection is read-only. This behavior can be a performance problem in scenarios where the method call is expected to fail often. One way to mitigate the problem is to test whether the collection is writable before trying to add a value.

```
ICollection<int> numbers = ...  
...  
if( !numbers.IsReadOnly){  
    numbers.Add(1);  
}
```

The member used to test a condition—in our example, the property `IsReadOnly`—is referred to as the tester. The member used to perform a potentially throwing operation—the Add method in our example—is referred to as the doer.

✓ **CONSIDER** the Tester–Doer Pattern for members that might throw exceptions in common scenarios to avoid performance problems related to exceptions.

■ **RICO MARIANI** Consider this when the “test” is much cheaper than the “do.”

■ **JEFFREY RICHTER** This pattern needs to be used carefully. A potential problem occurs when you have multiple threads accessing the object at the same time. For example, one thread could execute the tester method, which reports that all is OK, and before the doer method executes, another thread could change the object, causing the doer to fail. Although this pattern might improve performance, it might introduce race conditions and must be used with extreme caution.

■ **VANCE MORRISON** Jeff’s comment about races is true, but only rarely should it impact your design. The reason is that unless you specifically designed your class to be threadsafe (operate properly while multiple threads simultaneously use it), it almost certainly is not. Of the thousands of classes in .NET, only a handful are designed to be threadsafe (and this is specifically noted in the documentation). If your class is not threadsafe (the default), any user must use locks to ensure that only one thread uses it at a time. As long as both the “test” and the “do” are done under the same lock (this will typically happen naturally), there is not a problem.

Thus, Jeff’s concern only applies to the design of threadsafe classes.

7.5.2 The Try Pattern

For extremely performance-sensitive APIs, an even faster pattern than the Tester–Doer Pattern described in section 7.5.1 should be used. This pattern calls for adjusting the member name to make a well-defined test case a part of the member semantics. For example, `DateTime` defines a `Parse` method that throws an exception if parsing of a string fails. It also defines a corresponding `TryParse` method that attempts to parse, but returns `false` if parsing is unsuccessful and returns the result of a successful parsing using an `out` parameter.

```

public struct DateTime {
    public static DateTime Parse(string dateTime){
        ...
    }
    public static bool TryParse(string dateTime, out DateTime result){
        ...
    }
}

```

This pattern generalizes well to solve the race condition inherent in the Tester–Doer Pattern for threadsafe types. The `ConcurrentDictionary` class has a `TryAdd` method to solve both the race condition of Tester–Doer and the performance problem of throwing exceptions on a common operation.

```

public partial class ConcurrentDictionary<TKey, TValue> {
    public bool TryAdd(TKey key, TValue value) {
        ...
    }
}

```

✓ **CONSIDER** the Try Pattern for members that might throw exceptions in common scenarios to avoid performance problems related to exceptions.

■ **JEFFREY RICHTER** I like this guideline a lot. It solves the race-condition problem and the performance problem. Besides `DateTime.TryParse`, the `Dictionary` class has a `TryGetValue` method.

✓ **DO** use the prefix “Try” and Boolean return type for methods implementing the Try pattern.

✓ **DO** pick one kind of reason for why your Try method can return `false`, and throw an exception for any other failure reason.

Many methods have only one kind of failure: a string either represents a `DateTime` or it doesn’t; when using a `Dictionary` the key was either present or it wasn’t; and a `ConcurrentQueue` is either empty or not. In the `CountdownEvent` class, however, the `AddCount` method has several possible errors: the object may have already counted down to done, the

requested increment could overflow the counter, or the object could have already been disposed—all in addition to the usage error of providing a negative count. The most likely error is the counter having reached zero, and this is the only error for which TryAddCount will return false. Developers who call the method can write a handler for the failure case and understand what state it represents, while the less likely errors continue to throw exceptions.

- ✓ **DO** provide an exception-throwing member for each member using the Try Pattern.

```
public struct DateTime {  
    public static DateTime Parse(string dateTime){ ... }  
    public static bool TryParse(string dateTime, out DateTime result){  
        ...  
    }  
}
```

■ **RICO MARIANI** But don't provide members that nobody could reasonably use. There's no reason to set your clients up for failure by giving them an API that's almost guaranteed to be too slow to use. The most obvious way to use your system should also be the best way.

7.5.2.1 Value-Producing Try Methods

- ✓ **DO** “return” the value from a Try method via an out parameter.
Unlike CountdownEvent.TryAddCount, most Try methods are expected to produce an additional value from the method—the return value of the exception-throwing alternative member. Since the Try pattern reserves the return value for reporting success or failure, the additional value must be returned via an out parameter (see section 5.8 for more information on out parameters).
- ✓ **DO** write default(T) to the out parameter from a Try method that returns false.

```
public partial class HashSet<T> {  
    public bool TryGetValue(T equalValue, out T actualValue) {  
        Node node = FindNode(equalValue);  
        if (node != null) {
```

```

        actualValue = node.Item;
        return true;
    }
    actualValue = default(T);
    return false;
}
}

```

X AVOID writing to the `out` parameter from a Try method when throwing an exception.

Because the exception indicates that an error occurred, no additional value should be produced. If a Try method “leaks” a partially complete value on failure, then callers could potentially depend on the partially complete value for either error recovery or their main program flow. Once callers are depending on the internal state of your method, it becomes much harder to alter the otherwise private implementation details of the method without causing a runtime breaking change for someone. This situation is compounded when the caller is writing the `out` to a static field, since the value will persist if the exception is handled.

It’s good practice to exclusively assign the `out` parameter a value immediately prior to a return statement in the method. This helps to avoid leaking implementation details in the event of an exception.

```

public static bool TryDecode(byte[] data, out ParsedType value) {
    // Careful: The caller might depend on this value being
    // non-null on an exception from LoadFields.
    value = new ParsedType();
    if (!LoadFields(value, data)) {
        value = null;
        return false;
    }
    return true;
}

public static bool TryDecode(byte[] data, out ParsedType value) {
    // This variant hides the implementation details better
    ParsedType parsedType = new ParsedType();
    if (!LoadFields(parsedType, data)) {
        value = null;
        return false;
    }
    value = parsedType;
    return true;
}

```

7.5.2.1 Static TryParse(string, out T) Methods

- ✓ **DO** return false on null inputs to a method of the form `static bool TryParse(string, out T value)`.

The “kind of reason” for failure that is common to all static TryParse methods that take only a single string input is “the input is not a valid string representation of this type.” All static TryParse methods in the .NET BCL types consider a null input as “not a valid string representation” rather than a usage error. For consistency with the rest of .NET, your static TryParse methods should do the same.

Argument validation should still be performed for any parameters other than the main string input parameter in overloads of a TryParse method.

SUMMARY

In designing frameworks, it is important to use exceptions as your error handling mechanism, for all of the reasons detailed in this chapter. In the end, it will make your and your users’ lives easier.

8

Usage Guidelines

THIS CHAPTER CONTAINS guidelines for using common types in publicly accessible APIs. It deals with direct usage of built-in framework types (e.g., `Collection<T>`, serialization attributes), implementing common interfaces, and inheriting from common base classes. The last section of the chapter discusses overloading common operators.

8.1 Arrays

This section presents guidelines for using arrays in publicly accessible APIs.

✓ DO prefer using collections over arrays in public APIs. Section 8.3.3 provides details about how to choose between collections and arrays.

```
public class Order {  
    public Collection<OrderItem> Items { get { ... } }  
    ...  
}
```

✓ DO remember that arrays are mutable types—even arrays of immutable types.

It's easy to fall into the logical fallacy that “one `Int32` is immutable, so `new Int32[] { 1, 2 }` is also immutable,” especially when it comes to the `readonly` field modifier. But the `readonly` field modifier only

prevents replacing the entirety of the array; changing element values via the indexer is still possible.

This example demonstrates the pitfalls of using read-only array fields:

```
//bad code
public sealed class Path {
    public static readonly char[] InvalidPathChars =
        { '\"', '<', '>', '|' };
}
```

This allows callers to change the values in the array as follows:

```
Path.InvalidPathChars[0] = 'A';
```

Instead, you can either use a read-only collection (only if the items are immutable) or clone the array before returning it.

```
public static ReadOnlyCollection<char> GetInvalidPathChars() {
    return Array.AsReadOnly(badChars);
}

public static char[] GetInvalidPathChars() {
    return (char[])badChars.Clone();
}
```

Another option available to immutable element types is `ReadOnlySpan<T>`. The `Span` types, and the trade-offs that come from them, are discussed in section 9.12.

X AVOID array properties.

Array-based property get methods can be implemented in one of four basic ways: direct state return, direct unauthoritative return, shallow-copy, and deep-copy. The distinction between the four implementations only matters when a caller writes to the returned array, or modifies an object exposed via the array.

- Direct return is used by `ArraySegment<T>.Array`. This is in keeping with the purpose of `ArraySegment`, which effectively is to associate bounds with an array.

```
public T[] Array => _array;
```

- Direct unauthoritative return means the property uses a computed array, and returns the same array for two successive calls—assuming no object modification occurred between them. A caller writing back to the array doesn't alter the behavior of other properties and methods on the object, but does corrupt the property for other callers. This technique is used by `NameValueCollection.AllKeys`.

```
public string[] AllKeys {
    get {
        if (_allKeys == null) {
            _allKeys = new string[_entries.Count];
            for (int i = 0; i < _entries.Count; i++) {
                _allKeys[i] = _entries[i].Key;
            }
        }
        return _allKeys;
    }
}
```

- Shallow-copy properties return a new array on every call, but reference types are not duplicated to make the new array. Object state can still be modified if the element type is a mutable reference type, but replacing values within the array has no effect on the object. This technique is used by `DataTable.PrimaryKey`.

```
private List<Calendar> _calendars;
public Calendar[] ShallowCalendars => _calendars.ToArray();
```

- Deep-copy properties return a new array on every call, and the array contains new instances of any reference type values. The intent of a deep-copy is usually to completely avoid caller modifications affecting object state. Deep-copies are rare in .NET. Indeed, the few properties that technically perform deep-copy do so as a side effect of using `String.Substring` in the accessor method.

```
private List<Calendar> _calendars;
public Calendar[] DeepCalendars {
    get {
        Calendar[] calendars = new Calendar[_calendars.Count];
        for (int i = 0; i < _calendars.Count; i++) {
```

```
    calendars[i] = (Calendar)_calendars[i].Clone();
}
return calendars;
}
```

- When the array element type is fully immutable, a shallow-copy is as effective as a deep-copy for protecting the object state with a lower memory and CPU cost. The “value copy” deep-copy/shallow-copy hybrid technique is used by `NumberFormatInfo.NumberGroupSizes`.

```
private int[] _numberGroupSizes;
public int[] NumberGroupSizes {
    get => (int[])_numberGroupSizes.Clone();
}
```

Set methods can similarly be implemented in any of the four ways.

Per the guidance that properties should be almost as fast as a field access (section 5.1.3), the direct and unauthoritative direct returns are the only techniques that should be used by properties. Nevertheless, the implementation for a little more than half of the array-based properties in .NET Standard 2.0 returns a copy of one form or another. Given the inconsistency with arrays, the recommendation is to use collection-based properties, `ReadOnlyMemory<T>` (see section 9.12), or methods—where a name like “`GetRawBuffer`” can convey that the array is being returned directly.

If you’re defining an array transport type, like `ArraySegment<T>`, then a property directly returning the array is the most natural way of exposing the array back out.

✓ **CONSIDER** using jagged arrays instead of multidimensional arrays.

A jagged array is an array with elements that are also arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data (e.g., sparse matrix) compared to multidimensional arrays. Furthermore, the CLR optimizes index

operations on jagged arrays, so they might exhibit better runtime performance in some scenarios.

```
// jagged arrays
int[][] jaggedArray = {
    new int[] {1,2,3,4},
    new int[] {5,6,7},
    new int[] {8},
    new int[] {9}
};

// multidimensional arrays
int [,] multiDimArray = {
    {1,2,3,4},
    {5,6,7,0},
    {8,0,0,0},
    {9,0,0,0}
};
```

■ BRAD ABRAMS In general, I have found that usage of non-SZ (one dimension, zero lower bound) arrays in mainstream public APIs is very rare. Their usage should be constrained to problem areas that inherently lend themselves to multidimensional cases (such as matrix multiplication). All other usages should prefer defining a custom data structure or passing multiple arrays.

8.2 Attributes

`System.Attribute` is a base class used to define custom attributes. The following is an example of a custom attribute:

```
[AttributeUsage(...)]
public class NameAttribute : Attribute {
    public NameAttribute (string userName) {...} // required argument
    public string UserName { get{...} }
    public int Age { get{...} set{...} } // optional argument
}
```

Attributes are annotations that can be added to programming elements such as assemblies, types, members, and parameters. They are stored in the metadata of the assembly and can be accessed at runtime using the

reflection APIs. For example, .NET defines the `ObsoleteAttribute`, which can be applied to a type or a member to indicate that the type or member has been deprecated.

Attributes can have one or more properties that carry additional data related to the attribute. For example, `ObsoleteAttribute` could carry additional information about the release in which a type or a member became deprecated and the description of the new APIs replacing the obsolete API.

Some properties of an attribute must be specified when the attribute is applied. These are referred to as the required properties or required arguments, because they are represented as positional constructor parameters. For example, the `ConditionString` property of the `ConditionalAttribute` is a required property.

```
public static class Trace {  
    [Conditional("TRACE")]  
    public static void WriteLine(string message) { ... }  
}
```

Properties that do not necessarily have to be specified when the attribute is applied are called optional properties (or optional arguments). They are represented by settable properties. Compilers provide special syntax to set these properties when an attribute is applied. For example, the `AttributeUsageAttribute.Inherited` property represents an optional argument.

```
[AttributeUsage(AttributeTargets.All, Inherited = false)]  
public class SomeAttribute : Attribute {  
}
```

The following guidelines are aimed at designing custom attributes.

- ✓ **DO** name custom attribute classes with the suffix “Attribute.”

```
public class ObsoleteAttribute : Attribute { ... }
```

- ✓ **DO** apply the `AttributeUsageAttribute` to custom attributes.

```
[AttributeUsage(...)]  
public class ObsoleteAttribute{}
```

- ✓ **DO** provide settable properties for optional arguments.

```
public class NameAttribute : Attribute {
    ...
    public int Age { get{..} set{..} } // optional argument
}
```

- ✓ **DO** provide get-only properties for required arguments.

- ✓ **DO** provide constructor parameters to initialize properties corresponding to required arguments. Each parameter should have the same name (although with different casing) as the corresponding property.

```
[AttributeUsage(...)]
public class NameAttribute : Attribute {
    public NameAttribute(string userName){..} // required argument
    public string UserName { get{..} } // required argument
    ...
}
```

■ **KRZYSZTOF CWALINA** This guideline applies equally to case-sensitive and case-insensitive languages. For example, this is how the attribute would look if defined using case-insensitive VB.NET:

```
Public Class FooAttribute
    Dim nameValue As String
    Public Sub New(ByVal name As String)
        nameValue = name
    End Sub

    Public ReadOnly Property Name() As String
        Get
            Return nameValue
        End Get
    End Property
End Class
```

- X DO NOT** provide constructor parameters to initialize properties corresponding to the optional arguments.

In other words, do not have properties that can be set with both a constructor and a setter. This guideline makes very explicit which

arguments are optional and which are required, and it avoids having two ways of doing the same thing.

X AVOID overloading custom attribute constructors.

Having only one constructor clearly communicates to the user which arguments are required and which are optional.

✓ DO seal custom attribute classes, if possible. This makes the look-up for the attribute faster.

```
public sealed class NameAttribute : Attribute { ... }
```

JASON CLARK In the exception section, I cautioned against the reuse of somebody else's exception type. Here I am going to do the same for custom attributes. Reuse of somebody else's custom attribute, unless it means exactly the same thing, puts your mutual clients in the awkward position of having to choose between avoiding your API and applying the co-opted attribute, which involves the risk of unexpected or expected (but undesirable) side effects from the original code that used the attribute.

8.3 Collections

Any type designed specifically to manipulate a group of objects having some common characteristic can be considered a collection. It is almost always appropriate for such types to implement `IEnumerable` or `IEnumerable<T>`. So, in this section, we only consider types implementing one or both of those interfaces to be collections.

X DO NOT use weakly typed collections in public APIs.

The type of all return values and parameters representing collection items should be the exact item type, not any of its base types (this applies only to public members of the collection). For example, a collection storing `Components` should not have a public `Add` method that takes an `object` or a public indexer returning `IComponent`.

```
// bad design
public class ComponentDesigner {
    public IList Components { get { ... } }
    ...
}

// good design
public class ComponentDesigner {
    public Collection<Component> Components { get { ... } }
    ...
}
```

X DO NOT use `ArrayList` or `List<T>` in public APIs.

These types are data structures designed to be used in internal implementation, not in public APIs. `List<T>` is optimized for performance and power at the cost of cleanliness of the APIs and flexibility. For example, if you return `List<T>`, you will never be able to receive notifications when client code modifies the collection. Also, `List<T>` exposes many members, such as `BinarySearch`, that are not useful or applicable in many scenarios. Sections 8.3.1 and 8.3.2 describe types (abstractions) designed specifically for use in public APIs.

Instead of using these concrete types, consider `Collection<T>`, `IEnumerable<T>`, `IList<T>`, `IReadOnlyList<T>`, or other collection abstractions.

```
// bad design
public class Order {
    public List<OrderItem> Items { get { ... } }
    ...
}

// good design
public class Order {
    public Collection<OrderItem> Items { get { ... } }
    ...
}

// also good design
public class Order {
    public IList<OrderItem> Items { get { ... } }
    ...
}
```

X DO NOT use `Hashtable` or `Dictionary<TKey, TValue>` in public APIs.

These types are data structures designed to be used in internal implementation. Public APIs should use `IDictionary`, `IDictionary <TKey, TValue>`, or a custom type implementing one or both of the interfaces.

X DO NOT use `IEnumerator<T>`, `IEnumerator`, or any other type that implements either of these interfaces, except as the return type of a `GetEnumerator` method.

■ ANTHONY MOORE I've seen a lot of violations of this guideline ever since LINQ has been available. The review body made an explicit decision that LINQ should not change this guideline. Your callers can end up with a clumsy object model if they choose not to use LINQ or use a language that does not support it.

Types returning enumerators from methods other than `GetEnumerator` cannot be used with the `foreach` statement.

X DO NOT implement both `IEnumerator<T>` and `IEnumerable<T>` on the same type. The same applies to the non-generic interfaces `IEnumerator` and `IEnumerable`.

In other words, a type should be either a collection or an enumerator, but not both.

8.3.1 Collection Parameters

This section describes guidelines for using collections as parameters.

✓ DO use the least-specialized type possible as a parameter type. Most members taking collections as parameters use the `IEnumerable<T>` interface.

```
public void PrintNames(IEnumerable<string> names){  
    foreach(string name in names){  
        Console.WriteLine(name);  
    }  
}
```

■ ANTHONY MOORE In design meetings, we would often describe this set of guidelines with the statement that you should “require the weakest thing you need, and return the strongest thing you have.”

A nonobvious example that people sometimes miss is “out” parameters, which are really more like return values than input parameters and thus work a little better with the stronger typed objects.

A less clear case is properties on return values on interfaces or abstract base types. In this case, they are both input and output. These usually work better with the weak types because they leave more options to the implementation.

■ STEPHEN TOUB Of course, the strongest thing you have is often a type like `T[]` or `List<T>`, which previous guidelines have suggested not returning. It’s a balance.

■ JAN KOTAS If the actual type returned by the API is a stronger public type than the type declared by the signature, the consumers of the API will tend to take a dependency on the stronger type anyway. We have encountered multiple situations where changing the actual type returned by the API broke real-world code.

X AVOID using `ICollection<T>` or `ICollection` as a parameter just to access the `Count` property.

Instead, consider using `IEnumerable<T>` or `IEnumerable` and dynamically checking whether the object implements `ICollection<T>` or `ICollection`.

```
public List(IEnumerable<T> collection){
    // check if it implements ICollection
    if (collection is ICollection<T> col) {
        this.Capacity = collection.Count;
    }

    foreach(T item in collection){
        Add(item);
    }
}
```

8.3.2 Collection Properties and Return Values

This section offers guidelines for returning collections from methods and from property getters.

X DO NOT provide settable collection properties.

Users can replace the contents of the collection by clearing the collection first and then adding the new contents. If replacing the whole collection is a common scenario, consider providing the AddRange method on the collection.

```
// bad design
public class Order {
    public Collection<OrderItem> Items { get { ... } set { ... } }
    ...
}

// good design
public class Order {
    public Collection<OrderItem> Items { get { ... } }
    ...
}
```

✓ DO use `Collection<T>` or a subclass of `Collection<T>` for properties or return values representing read/write collections.

```
public Collection<Session> Sessions { get; }
```

If `Collection<T>` does not meet some requirement (e.g., the collection must not implement `IList`), use a custom collection by implementing `IEnumerable<T>`, `ICollection<T>`, or `IList<T>`.

✓ DO use `ReadOnlyCollection<T>`, a subclass of `ReadOnlyCollection<T>`, or in rare cases `IEnumerable<T>` for properties or return values representing read-only collections.

```
public ReadOnlyCollection<Session> Sessions { get; }
```

In general, prefer `ReadOnlyCollection<T>`. If it does not meet some requirement (e.g., the collection must not implement `IList`), use a custom collection by implementing `IEnumerable<T>`, `ICollection<T>`,

or `IList<T>`. If you do implement a custom read-only collection, implement `ICollection<T>.ReadOnly` to return false.

```
public class SessionCollection : IList<Session> {  
  
    bool ICollection<Session>.IsReadOnly { get { return false; } }  
  
    ...  
}
```

In cases where you are sure that the only scenario you will ever want to support is forward-only iteration, you can simply use `IEnumerable<T>`.

CHRIS SELLS One of my favorite implementations of `IEnumerable<T>` is sometimes known as a generator. A generator is an iterator class or method that generates collection members on the fly, which is really useful when you'd prefer not to precompute something and then buffer it simply for convenient access. For example, the following is a generator method that computes as many numbers from the Fibonacci sequence as you care to ask for, wrapped in an `IEnumerable` for easy access via `foreach` or for passing to methods that take `IEnumerable` as input:

```
class FibonacciGenerator{  
    public static IEnumerable<long> GetSequence(int count){  
        long fib1 = 0;  
        long fib2 = 1;  
        yield return fib1;  
        yield return fib2;  
        // assume they want at least 2, else what  
        //fun are they?  
        while(--count!= 1) {  
            long fib3 = fib1 + fib2;  
            yield return fib3;  
            fib1 = fib2;  
            fib2 = fib3;  
        }  
    }  
}  
class Program {  
    static void Main() {  
        foreach(long fib in FibonacciGenerator.GetSequence(100)){  
            Console.WriteLine(fib);  
        }  
    }  
}
```

- ✓ **CONSIDER** using subclasses of generic base collections instead of using the collections directly.

This allows for a better name and for adding helper members that are not present on the base collection types. This is especially applicable to high-level APIs.

```
public TraceSourceCollection : Collection<TraceSource> {  
    // optional helper method  
    public void FlushAll {  
        foreach(TraceSource source in this){  
            source.Flush();  
        }  
    }  
    // another common helper  
    public void AddSource(string sourceName){  
        Add(new TraceSource(sourceName));  
    }  
}
```

- ✓ **CONSIDER** returning a subclass of `Collection<T>` or `ReadOnlyCollection<T>` from very commonly used methods and properties.

```
public class ListItemCollection : Collection<ListItem> {}  
public class ListBox {  
    public ListItemCollection Items { get; }  
}  
public class XmlAttributeCollection:ReadOnlyCollection<XmlAttribute>{}  
public class XmlNode {  
    public XmlAttributeCollection Attributes { get; }  
}
```

This will make it possible for you to add helper methods or change the collection implementation in the future.

- ✓ **CONSIDER** using a keyed collection if the items stored in the collection have unique keys (e.g., names, IDs). Keyed collections are collections that can be indexed by both an integer and a key, and they are usually implemented by inheriting from `KeyedCollection< TKey , TItem >`.

For example, a collection of files in a directory could be represented as a subclass of `KeyedCollection<string, FileInfo>`, where `string` is the filename. Users of the collection could then index the collection using filenames.

```
public class FileInfoCollection : KeyedCollection<string, FileInfo> {  
    ...  
  
    public class Directory {  
        public Directory(string root);  
        public FileInfoCollection GetFiles();  
    }  
}
```

Keyed collections usually have larger memory footprints and should not be used if the memory overhead outweighs the benefits of having the keys.

X DO NOT return null values from collection properties or from methods returning collections. Return an empty collection or an empty array instead.

Users of collection properties often assume that the following code will always work:

```
IEnumerable<string> list = GetList();  
foreach(string name in list){  
    ...  
}
```

The general rule is that null and empty (zero-item) collections or arrays should be treated the same.

8.3.2.1 Snapshots Versus Live Collections

Collections representing a state at some point in time are called snapshot collections. For example, a collection containing rows returned from a database query would be a snapshot. Collections that always represent the current state are called live collections. For example, a collection of ComboBox items is a live collection.

- X DO NOT** return snapshot collections from properties. Properties should return live collections.

```
public class Directory {
    public Directory(string root);
    public IEnumerable<FileInfo> Files { get {...} } //live collection
}
```

Property getters should be very lightweight operations. Returning a snapshot generally requires creating a copy of an internal collection in an $O(n)$ operation.

- ✓ DO** use either a snapshot collection or a live `IEnumerable<T>` (or its subtype) to represent collections that are volatile (i.e., that can change without explicitly modifying the collection).

In general, all collections representing a shared resource (e.g., files in a directory) are volatile. Such collections are very difficult or impossible to implement as live collections unless the implementation is simply a forward-only enumerator.

```
public class Directory {
    public Directory(string root);
    public IEnumerable<FileInfo> Files { get; } // live
    // or
    public FileInfoCollection GetFiles(); // snapshot
}
```

8.3.3 Choosing Between Arrays and Collections

Framework designers often need to choose whether to use an array or a collection. These two alternative approaches have very similar usage but somewhat different performance characteristics, usability, and versioning implications.

- ✓ DO** prefer collections over arrays.

Collections provide more control over contents, can evolve over time, and are more usable. In addition, using arrays for read-only scenarios

is discouraged because the cost of cloning the array is prohibitive. Usability studies have shown that some developers feel more comfortable using collection-based APIs.

However, if you are developing low-level APIs, it might be better to use array parameters for read-write scenarios. Arrays have a smaller memory footprint, which helps reduce the working set, and access to elements in an array is faster because it is optimized by the runtime.

- ✓ **CONSIDER** using arrays in low-level APIs to minimize memory consumption and maximize performance.
- ✓ **DO** use byte arrays instead of collections of bytes.

```
// bad design
public Collection<byte> ReadBytes() { ... }

// good design
public byte[] ReadBytes() { ... }

// more performant good designs (see 9.12)
public int ReadBytes(byte[] destination) { ... }
public int ReadBytes(Span<byte> destination) { ... }
```

- ✗ **DO NOT** use arrays for properties if the property would have to return a new array (e.g., a copy of an internal array) every time the property getter is called.

This ensures that users will not write the following inefficient code:

```
//bad design
for(int index=0; index< customer.Orders.Length; index++) {
    Console.WriteLine(customer.Orders[i]);
}
```

8.3.4 Implementing Custom Collections

In implementing custom collections, it is a good idea to follow these guidelines.

- ✓ **CONSIDER** inheriting from `Collection<T>`, `ReadOnlyCollection<T>`, or `KeyedCollection< TKey , TItem >` when designing new collections.

```
public class OrderCollection : Collection<Order> {
    protected override void InsertItem(int index, Order item) {
        ...
    }
}
```

- ✓ **DO** implement `IEnumerable<T>` when designing new collections. Consider implementing `ICollection<T>`, `IReadOnlyList<T>`, or even `IList<T>` where it makes sense.

```
public class TextDecorationCollection : IList<TextDecoration> ... {
    ...
}
```

When implementing such a custom collection, follow the API pattern established by `Collection<T>` and `ReadOnlyCollection<T>` as closely as possible. That is, implement the same members explicitly, name the parameters in the same way that these two collections name them, and so on. In other words, make your custom collection different from these two collections only when you have a very good reason to do so.

- ✓ **CONSIDER** implementing non-generic collection interfaces (`IList` and `ICollection`) if the collection will often be passed to APIs taking these interfaces as input.

```
public class OrderCollection : IList<Order>, IList {
    ...
}
```

- ✗ **AVOID** implementing collection interfaces on types with complex APIs unrelated to the concept of a collection.

In other words, a collection should be a simple type used to store, access, and manipulate items, and not much more.

- ✗ **DO NOT** inherit from non-generic base collections such as `Collection-Base`. Use `Collection<T>`, `ReadOnlyCollection<T>`, and `Keyed-Collection<TKey, TItem>` instead.

8.3.4.1 Naming Custom Collections

Collections (types that implement `IEnumerable`) are created mainly for two reasons: (1) to create a new data structure with structure-specific operations and often different performance characteristics than existing data structures (e.g., `List<T>`, `LinkedList<T>`, `Stack<T>`), and (2) to create a specialized collection for holding a specific set of items (e.g., `StringCollection`). Data structures are most often used in the internal implementation of applications and libraries. Specialized collections are mainly intended to be exposed in APIs (as property and parameter types).

- ✓ **DO** use the “Dictionary” suffix in names of abstractions implementing `IDictionary` or `IDictionary< TKey , TValue >`.
- ✓ **DO** use the “Collection” suffix in names of types implementing `IEnumerable` (or any of its descendants) and representing a list of items.

```
public class OrderCollection : IEnumerable<Order> { ... }
public class CustomerCollection : ICollection<Customer> { ... }
public class AddressCollection : IList<Address> { ... }
```

- ✓ **DO** use the appropriate data structure name for custom data structures.

```
public class LinkedList<T> : IEnumerable<T> , ... { ... }
public class Stack<T> : ICollection<Customer> { ... }
```

- ✗ **AVOID** using any suffixes implying a particular implementation, such as “`LinkedList`” or “`Hashtable`,” in names of collection abstractions.

- ✓ **CONSIDER** prefixing collection names with the name of the item type. For example, a collection storing items of type `Address` (implementing `IEnumerable<Address>`) should be named `AddressCollection`. If the item type is an interface, the “I” prefix of the item type can be omitted. Thus, a collection of `IDisposable` items can be called `DisposableCollection`.
- ✓ **CONSIDER** using the “`ReadOnly`” prefix in names of read-only collections if a corresponding writeable collection might be added or already exists in the framework.

For example, a read-only collection of strings should be called `ReadOnlyStringCollection`.

8.4 DateTime and DateTimeOffset

The original type to represent a point in time in .NET is the `DateTime` structure, which stores a time as either the local system time or UTC. The `DateTimeOffset` structure stores data similar to `DateTime`, but adds information about the offset from UTC; thus it is a more precise representation of a point in time.

■ **ANTHONY MOORE** This was one of the hardest types to name in the history of the Base Class Libraries. The way to think about it is Date + Time + Offset. However, the name confuses people because at first glance it looks like the type of just the offset piece. Many other options were discussed, but they were seen to have problems even worse than this.

The following listing shows the main properties of the `DateTime` and `DateTimeOffset`:

```
public struct DateTime {  
  
    public DateTime Date { get; }  
    public DateTimeKind Kind { get; }  
    public DayOfWeek DayOfWeek { get; }  
    public int Day { get; }  
    public int DayOfYear { get; }  
    public int Hour { get; }  
    public int Millisecond { get; }  
    public int Minute { get; }  
    public int Month { get; }  
    public int Second { get; }  
    public int Year { get; }  
    public long Ticks { get; }  
    public TimeSpan TimeOfDay { get; }  
  
    ...  
}  
  
public struct DateTimeOffset {  
  
    public DateTime Date { get; }  
    public DateTime DateTime { get; }  
    public DateTime LocalDateTime { get; }  
}
```

```
public DateTime UtcDateTime { get; }
public DayOfWeek DayOfWeek { get; }
public int Day { get; }
public int DayOfYear { get; }
public int Hour { get; }
public int Millisecond { get; }
public int Minute { get; }
public int Month { get; }
public int Second { get; }
public int Year { get; }
public long Ticks { get; }
public long UtcTicks { get; }
public TimeSpan Offset { get; }
public TimeSpan TimeOfDay { get; }

...
}
```

- ✓ **DO** use `DateTimeOffset` whenever you are referring to an exact point in time. For example, use it to calculate “now,” transaction times, file change times, logging event times, and so on. If the time zone is not known, use it with UTC. These usages are much more common than the scenarios where `DateTime` is preferred, so this should be considered the default.
- ✓ **DO** use `DateTime` for any cases where the absolute point in time does not apply, such as store opening times that apply across time zones.
- ✓ **DO** use `DateTime` when the time zone either is not known or is sometimes not known. This may happen in cases where it comes from a legacy data source.
- ✗ **DO NOT** use `DateTimeKind` if `DateTimeOffset` can be used instead.
`DateTimeKind` is an enum stored in `DateTime` to indicate whether the instance represents UTC, local time, or unspecified time zone.
- ✓ **DO** use `DateTime` with a 00:00:00 time component rather than `DateTimeOffset` to represent whole dates, such as a date of birth.
- ✓ **DO** use `TimeSpan` to represent times of day without a date.

■ **ANTHONY MOORE** We have considered adding Date and Time types in future releases of .NET. Although the idea is not off the table completely, we are concerned that such additions will complicate the framework without providing enough value to offset the negatives.

8.5 ICloneable

The `ICloneable` interface contains a single `Clone` method, which creates a copy of the current object.

```
public interface ICloneable {  
    object Clone();  
}
```

There are two general ways to implement cloning—deep-copy or shallow-copy. Deep-copy copies the cloned object and all objects referenced by the object, recursively, until all objects in the graph are copied. Shallow-copy copies only a part of the object graph.

Because the contract of `ICloneable` does not specify the type of clone implementation required to satisfy the contract, different classes have different implementations of the interface. Consumers cannot rely on `ICloneable` to let them know whether an object is deep-copied. Therefore, we recommend that `ICloneable` not be implemented.

■ **KRZYSZTOF CWALINA** The moral of the story is that you should never ship an interface if you don't have both implementations and consumers of the interface. In the case of `ICloneable`, we did not have consumers when we shipped it. I searched the framework sources and could not find even one place where we take `ICloneable` as a parameter.

- X **DO NOT** implement `ICloneable`.
- X **DO NOT** use `ICloneable` in public APIs.

- ✓ **CONSIDER** defining the `Clone` method on types that need a cloning mechanism. Ensure that the documentation clearly states whether it is a deep-copy or a shallow-copy.

```
public class Customer {  
    public Customer Clone();  
    ...  
}
```

8.6 IComparable<T> and IEquatable<T>

`IComparable<T>` and `IEquatable<T>` can be implemented by types that support either equality or order comparison. `IComparable<T>` specifies ordering (less than, equals, greater than) and is used mainly for sorting. `IEquatable<T>` specifies equality and is used mainly for look-ups.

```
public interface IComparable<T> {  
    // returns a negative integer if this is less than other  
    // returns 0 if this and other are equal  
    // returns a positive integer if this is greater than other  
    public int CompareTo(T other);  
}  
  
public interface IEquatable<T> {  
    public bool Equals(T other);  
}
```

- ✓ **DO** implement `IEquatable<T>` on value types.

The `Object.Equals` method on value types causes boxing, and its default implementation is not very efficient because it uses reflection. `IEquatable<T>.Equals` can offer much better performance and can be implemented so that it does not cause boxing.

```
public struct Int32 : IEquatable<Int32> {  
    public bool Equals(Int32 other){ ... }  
}
```

- ✓ **DO** follow the same guidelines as for overriding `Object.Equals` when implementing `IEquatable<T>.Equals`.

See section 8.9.1 for detailed guidelines on overriding `Object.Equals`.

- ✓ **DO** override `Object.Equals` whenever implementing `IEquatable<T>`.

Both overloads of the `Equals` method should have exactly the same semantics.

```
public struct PositiveInt32 : IEquatable<PositiveInt32> {
    public bool Equals(PositiveInt32 other) { ... }
    public override bool Equals(object obj){
        if (!obj is PositiveInt32) return false;
        return Equals((PositiveInt32)obj);
    }
}
```

- ✓ **CONSIDER** overloading `operator==` and `operator!=` whenever implementing `IEquatable<T>`.

```
public struct Decimal : IEquatable<Decimal>, ... {
    public bool Equals(Decimal other){ ... }
    public static bool operator==(Decimal x, Decimal y) {
        return x.Equals(y);
    }
    public static bool operator!=(Decimal x, Decimal y) {
        return !x.Equals(y);
    }
}
```

See section 8.10 for more details about implementing the equality operators.

- ✓ **DO** implement `IEquatable<T>` any time you implement `IComparable<T>`.

Note that the reverse is not true, and not all types can support ordering.

```
public struct Decimal : IComparable<Decimal>, IEquatable<Decimal> {
    ...
}
```

- ✓ **CONSIDER** overloading comparison operators (`<`, `>`, `<=`, `>=`) whenever you implement `IComparable<T>`.

```
public struct Decimal : IComparable<Decimal>, ... {
    public int CompareTo(Decimal other){ ... }
```

```

public static bool operator<(Decimal x, Decimal y) {
    return x.CompareTo(y)<0;
}
public static bool operator>(Decimal x, Decimal y) {
    return x.CompareTo(y)>0;
}
...
}

```

See section 5.7 for details on when to overload operators.

8.7 IDisposable

`IDisposable` is known as the Dispose Pattern and is discussed in section 9.3.

8.8 Nullable<T>

`Nullable<T>` is a simple type added to the .NET Framework 2.0. The type is designed to be able to represent value types with “null” values.

```

Nullable<int> x = null;
Nullable<int> y = 5;
Console.WriteLine(x == null); // prints true
Console.WriteLine(y == null); // prints false

```

Note that C# provides special support for `Nullable<T>` in the form of language aliases for nullable types, lifted operators, and the null coalescing operator.

```

int? x = null; // alias for Nullable<int>
long? d = x; // calls cast operator from Int32 to Int64
Console.WriteLine(d??10); // coalescing; prints 10 because d == null

```

✓ **CONSIDER** using `Nullable<T>` to represent values that might not be present (i.e., optional values). For example, use it when returning a strongly typed record from a database with a property representing an optional table column.

X DO NOT use `Nullable<T>` unless you would use a reference type in a similar manner, taking advantage of the fact that reference type values can be null.

For example, you would not use null to represent optional parameters.

```
// bad design
public class Foo {
    public Foo(string name, int? id);
}

// good design
public class Foo {
    public Foo(string name, int id);
    public Foo(string name);
}
```

X AVOID using `Nullable<bool>` to represent a general three-state value.

`Nullable<bool>` should only be used to represent truly optional Boolean values: `true`, `false`, and `not available`. If you simply want to represent three states (e.g., yes, no, cancel), consider using an enum.

X AVOID using `System.DBNull`. Prefer `Nullable<T>` instead.

■ PABLO CASTRO `Nullable<T>` is, in general, a better representation of optional database values. One thing to consider, though, is that while `Nullable<T>` gives you the ability to represent null values, you don't get database null operational semantics. Specifically, you don't get null propagation through operators and functions. If you deeply care about the propagation semantics, consider sticking with `DBNull`.

8.9 Object

`System.Object` has several members that are very commonly overridden. The following sections describe when and how to override these members.

8.9.1 Object.Equals

The default implementation of `Object.Equals` on value types returns true if all fields of the values being compared compare themselves as equal. We call such equality “value equality.” The implementation uses

reflection to access the fields; in consequence, it is often unacceptably inefficient and should be overridden.

The default implementation of `Object.Equals` on reference types returns true if the two references being compared point to the same object. We call such equality “reference equality.” Some reference types override the default implementation to provide value equality semantics. For example, the value of a string is based on the characters of the string, so the `Equals` method of the `String` class returns true for any two string instances that contain exactly the same characters in the same order.

The following guidelines describe when and how to override the default behavior of the `Object.Equals` method.

✓ **DO** comply with the contract defined for `Object.Equals` when overriding the method.

For convenience, the contract taken directly from the `System.Object` documentation is provided here.

- `x.Equals(x)` returns true.
- `x.Equals(y)` returns the same value as `y.Equals(x)`.
- If `(x.Equals(y) && y.Equals(z))` returns true, then `x.Equals(z)` returns true.
- Successive invocations of `x.Equals(y)` return the same value as long as the objects `x` and `y` are not modified.
- `x.Equals(null)` returns false.

✓ **DO** override `GetHashCode` whenever you override `Equals`.

The contracts of `Equals` and `GetHashCode` are interdependent. For more information, see section 8.9.2 on implementing `GetHashCode`.

✓ **CONSIDER** implementing `IEquatable<T>` whenever overriding `Object.Equals`.

X DO NOT throw exceptions from Equals.

Two objects should either be equal or not equal. So, for example, even if the argument passed to Equals is null, returning false is better than throwing an exception.

8.9.1.1 Equals on Value Types

✓ DO override Equals on value types.

The default implementation uses reflection to access and compare all the fields. In consequence, it is often unacceptably inefficient.

✓ DO provide an overload of Equals taking the value type parameter by implementing IEquatable<T>.

This provides a way to compare two value types without boxing the parameter passed to Equals.

```
public struct MyStruct {
    public bool Equals (MyStruct value) { ... }
    ...
}
```

8.9.1.2 Equals on Reference Types

✓ CONSIDER overriding Equals to provide value equality if a reference type represents a value. For example, you might want to override Equals in reference types representing numbers or other mathematical entities.

X DO NOT implement value equality on mutable reference types.

Reference types that implement value equality (e.g., System.String) should be immutable. Mutable reference types with value equality can, for example, be “lost” in hashtables when their value (and so the hash code) changes.

8.9.2 `Object.GetHashCode`

A hash function is used to generate a number (hash code) that corresponds to the identity of an object as determined by an associated implementation of equality. Hash codes are used by hashtables; thus, it is important to understand how hashtables work to be able to implement hash functions properly.

- ✓ **DO** override `GetHashCode` if you override `Object.Equals`.

This guarantees that two objects considered equal have the same hash code. The following guidelines provide more information.

- ✓ **DO** ensure that if the `Object.Equals` method returns true for any two objects, `GetHashCode` returns the same value for these objects.

Types that do not follow this guideline will not work correctly when used as hashtable keys.

■ **CHRISTOPHER BRUMME** This means that if `obj1.Equals(obj2)` returns true, both of the objects should have the same hash code. If the objects aren't equal, they might or might not have the same hash code. Strictly speaking, all objects could have a hash code of 1. This would really be terrible from a performance point of view when looking up such items in a hashtable, of course.

- ✓ **DO** make every effort to ensure that `GetHashCode` generates a uniform distribution of numbers for all objects of a type.

This will minimize hashtable collisions, which degrade performance.

For example, two strings will return the same hash code if they represent the same string value, as defined by the `String.Equals` implementation. Also, the method uses all the characters in the string to generate reasonably uniformly distributed output, even when the input is clustered in certain ranges (e.g., many users could have strings that contain only the lower 128 ASCII characters, even though a string can contain any of the several thousand Unicode characters).

The built-in `HashCode` class allows for easy implementations of `GetHashCode()` that have good distribution characteristics.

```
public partial struct Size {  
    public override readonly int GetHashCode() =>  
        HashCode.Combine(Width, Height);  
}
```

- ✓ **DO** ensure that `GetHashCode` returns exactly the same value regardless of any changes that are made to the object.

Note that there are several related guidelines throughout the book. In particular, some guidelines advise against mutable value types, and others advise against mutable reference types implementing value equality. See sections 8.9.1.2 and 4.7.

■ **BRIAN PEPIN** This has tripped me up more than once: Make sure `GetHashCode` always returns the same value across the lifetime of an instance. Remember that hash codes are used to identify “buckets” in most hashtable implementations. If an object’s “bucket” changes, a hashtable may not be able to find your object. These can be very hard bugs to find, so get it right the first time.

- ✗ **AVOID** throwing exceptions from `GetHashCode`.

8.9.3 `Object.ToString`

The `Object.ToString` method is intended to be used for general display and debugging purposes. The default implementation simply provides the object type name. The default implementation is not very useful, and it is recommended that the method be overridden.

- ✓ **DO** override `ToString` whenever an interesting human-readable string can be returned.

The default implementation is not very useful, and a custom implementation can almost always provide more value.

■ **CHRIS SELLS** I consider `ToString` an especially dangerous method to provide for UI-generic types, because it's likely to be implemented with some specific UI in mind, making it useless for other UI needs. To avoid tempting myself in this way, I prefer to make my `ToString` output as geeky as possible to emphasize that the only "humans" who should ever see the output are "developer humans" (a subspecies all their own).

■ **BRIAN PEPIN** I tend to be very careful with `ToString`. I treat it as a diagnostic API and seldom use it for presentation to users unless I know exactly how it works. Back when I was working with the source code to Expression Blend and had an instance of an object that represented a value converted to a string, there was no obvious API to return the string—but what do you know, `ToString` had just what I needed! It turned out I was wrong: `ToString` was returning debugging information that looked just like the value I wanted most of the time. Other times, however, it returned different diagnostic information and broke my code. The moral is use `ToString` only for diagnostics and define a separate method for end-user presentation.

■ **VANCE MORRISON** The most important value of `ToString` is that the debugger uses it as the default way of displaying the object. This is really valuable and is well worth doing. Sadly, all too often we don't do this and the debuggability of our code suffers. In my own code, for nontrivial types, I found that writing something that looked like an XML fragment was quite useful (it is unambiguous, and programmers understand its syntax).

✓ **DO** try to keep the string returned from `ToString` short.

The debugger uses `ToString` to get a textual representation of an object to be shown to the developer. If the string is longer than the debugger can display (typically less than one screen length), the debugging experience is hindered.

■ **CHRISTOPHE NASARRE** In terms of debugging experience, you should decorate your type with `DebuggerDisplayAttribute` in addition to overriding `ToString` for that particular purpose.

- ✓ **CONSIDER** returning a unique string associated with the instance.
- ✓ **DO** prefer a friendly name over a unique but not readable ID.
- ✓ **DO** format the output of `ToString()` based on the current thread culture when returning culture-dependent information.

■ **CHRISTOPHE NASARRE** To be more explicit, use the `CultureInfo` instance returned by a thread's `CurrentCulture` property to format any numeric value or date, and the one returned by `CurrentUICulture` to look up any resource. People often confuse these two properties.

- ✓ **DO** overload `ToString(string format)`, or implement `IFormattable`, if the string returned from `ToString()` is culture-sensitive or there are various ways to format the string. For example, `DateTime` provides the overload and implements `IFormattable`.

✗ **DO NOT** return an empty string or null from `ToString`.

✗ **AVOID** throwing exceptions from `ToString`.

- ✓ **DO** ensure that `ToString` has no observable side effects.

One reason for this is that `ToString` is called by debuggers during debugging, and such side effects can make debugging difficult.

- ✓ **CONSIDER** having the output of `ToString` be a valid input for any parsing methods on this type.

For example, the string returned from `DateTime.ToString` can be successfully parsed using `DateTime.Parse`.

```
DateTime now = DateTime.Now;
DateTime parsed = DateTime.Parse(now.ToString());
```

8.10 Serialization

Serialization is the process of converting an object into a format that can be readily persisted or transported. For example, you can serialize an object, transport it over the Internet using HTTP, and deserialize it at the destination machine.

.NET has many built-in serialization technologies, each optimized for different serialization scenarios. The second edition of this book offered guidance regarding the serialization technologies built into .NET at the time. However, because serialization formats change fairly rapidly—and few serializers handle multiple formats gracefully—most of the serialization guidance has been moved to Appendix B.

X AVOID using serialization attributes or interfaces on public types in a general-purpose library, except for types that need to support serialization across AppDomains—such as Exception types.

In general, if your type needs attributes on the type, or any of its members, to work well with a specific serialization technology, then your type will need attributes for most of the other serialization technologies. This can provide significant frustration for your users when you support some serialization technology that they don't (or can't) use, and don't support the technology that they are using.

Supporting particular serializers will also frequently require adding assembly references you otherwise wouldn't need. That can lead to unnecessary application bloat for applications that don't use your chosen serializer.

General-purpose types within general-purpose libraries should concentrate on functionality and usability within a programming context, and leave the decisions on serialization technologies to the application developer.

Exception types and other types that explicitly support serialization across AppDomain boundaries—in the .NET runtimes that support multiple AppDomains per process—should continue to use the [Serializable] attribute.

- ✓ **DO** consider backward and forward compatibility when creating or changing serializable types.

Keep in mind that serialized streams of future versions of your type can be deserialized into the current version of the type, and vice versa.

Make sure you understand the versioning semantics of your serializer. For example, some serializers use reflection to serialize fields. If your type supports one of those serializers, then renaming (and possibly reordering) even private fields can cause errors when deserializing data from a previous version of your library.

Forward compatibility is also important, but is an often overlooked area. When an unknown property is encountered, should you ignore it, or should the application fail? Unfortunately, the answer is usually context-specific and depends on what the property represented. The most conservative answer is to fail on unknown data and to take the stance that once you've released a serializable type, you can't ever add to it.

Test compatibility of serialization when making changes to serializable types. Try deserializing the new version into an old version, and vice versa, across as broad a range as you expect your users to need.

- ✓ **DO** make the `(SerializationInfo info, StreamingContext context)` serialization constructor `protected` (`private` for sealed types) when implementing `ISerializable`.

```
[Serializable]
public class Person : ISerializable {
    protected Person(SerializationInfo info, StreamingContext context) {
        ...
    }
}
```

- ✓ **DO** implement the `ISerializable` members explicitly.

```
[Serializable]
public class Person : ISerializable {
    void ISerializable.GetObjectData(...) {
        ...
    }
}
```

- ✓ **DO** apply a link demand to `ISerializable.GetObjectData` implementation. This ensures that only fully trusted assemblies and the Runtime Serializer have access to the member.

```
[Serializable]
public class Person : ISerializable {
    [SecurityPermission(
        SecurityAction.LinkDemand,
        Flags = SecurityPermissionFlag.SerializationFormatter)
    ]
    void ISerializable.GetObjectData(...) {
        ...
    }
}
```

■ **JEREMY BARTON** Link demands are part of the now-deprecated Code Access Security (CAS) system. The guidance may feel antiquated, but since the platforms that support CAS and the platforms that support AppDomains are the same, it's still something you should do when using the `ISerializable` interface.

8.11 Uri

`System.Uri` is a type that can be used to represent uniform resource identifiers (URIs). These concepts can also be represented using strings. Some of the most important guidelines in this section are intended to help you choose between `System.Uri` and `System.String` for representing URIs.

- ✓ **DO** use `System.Uri` to represent URI and URL data.

This applies to parameter types, property types, and return value types.

```
public class Navigator {
    public Navigator(Uri initialLocation);
    public Uri CurrentLocation { get; }
    public Uri NavigateTo(Uri location);
}
```

■ **MARK ALCAZAR** `System.Uri` is a much safer and richer way of representing URIs. Extensive manipulation of URI-related data using plain strings has been shown to cause many security and correctness problems.

- ✓ **CONSIDER** providing string-based overloads for the most commonly used members with `System.Uri` parameters.

In cases where the usage pattern of taking a string from a user will be common enough, you should consider adding a convenience overload accepting a string. The string-based overload should be implemented in terms of the Uri-based overload.

```
public class Navigator {  
    public void NavigateTo(Uri location);  
    public void NavigateTo(string location) {  
        NavigateTo (new Uri(location));  
    }  
}
```

- ✗ **DO NOT** automatically overload all Uri-based members with a version that accepts a string.

Generally, Uri-based APIs are preferred. String-based overloads are meant to be helpers for the most common scenarios. Therefore, you should not automatically provide string-based overloads for all variants of the Uri-based members. Instead, be selective and provide such helpers just for the most commonly used variants.

```
public class Navigator {  
    public void NavigateTo(Uri location);  
    public void NavigateTo(Uri location, NavigationMode mode);  
    public void NavigateTo(string location);  
}
```

8.11.1 System.Uri Implementation Guidelines

The guidelines in this section help with the implementation of code using `System.Uri`.

✓ **DO** call the Uri-based overloads, if available.

✗ **DO NOT** store URI/URL data in a string.

When you accept a URI/URL input as a string, you should convert the string to `System.Uri` and store the instance of `System.Uri`.

```
public class SomeResource {
    Uri location;
    public SomeResource(string location) {
        this.location = new Uri(location);
    }
    public SomeResource(Uri location){
        this.location = location;
    }
}
```

8.12 System.Xml Usage

This section discusses usage of several types residing in `System.Xml` namespaces that can be used to represent XML data.

✗ **DO NOT** use `XmlNode` or `XmlDocument` to represent XML data. Instead, favor using instances of `IXPathNavigable`, `XmlReader`, `XmlWriter`, or subtypes of `XNode`. `XmlNode` and `XmlDocument` are not designed for exposing in public APIs.

```
// bad design
public class ServerConfiguration {
    ...
    public XmlDocument ConfigurationData { get { ... } }
}

// good design
public class ServerConfiguration {
    ...
    public IXPathNavigable ConfigurationData { get { ... } }
}
```

- ✓ **DO** use `XmlReader`, `IXPathNavigable`, or subtypes of `XNode` as input or output of members that accept or return XML.

Use these abstractions instead of `XmlDocument`, `XmlNode`, or `XPathDocument`, because this decouples the methods from specific implementations of an in-memory XML document and allows them to work with virtual XML data sources that expose `XNode`, `XmlReader`, or `XPathNavigator`.

- ✗ **DO NOT** subclass `XmlDocument` if you want to create a type representing an XML view of an underlying object model or data source.

This guideline means that `XmlDataDocument` is an example of what not to do.

■ **DARE OBASANJO** There are several problems with implementations like `XmlDataDocument`. One is inefficiency. Because `XmlNode`s need to be distinct objects, such implementations result in large memory consumption. A second problem is that the data model of the `DataSet` does not map 1:1 with that of XML. There are all sorts of edge cases when one does things like insert comments, PIs, or CDATA sections into an `XmlDataDocument`.

Implementing a custom `XPathNavigator` gets around the inefficiency problems because the navigator is a cursor; there is no need to create objects for each node in the tree. It also reduces the data/XML impedance mismatch. Because the main goal of `XmlDataDocument` was to enable users to either write out the `DataSet` as XML or query it with `XPath`, there is no need to support editability via the DOM. In addition, the simpler data model of the `XPathNavigator` leads to fewer edge cases in mapping your data model to XML.

8.13 Equality Operators

This section discusses overloading equality operators. It refers to `operator==` and `operator!=` as equality operators.

- ✗ **DO NOT** overload one of the equality operators and not the other.

It is very surprising to developers when they discover that a type overloads just one of the operators.

- ✓ **DO** ensure that `Object.Equals` and the equality operators have exactly the same semantics and similar performance characteristics.

This often means that `Object.Equals` needs to be overridden when the equality operators are overloaded.

```
public struct PositiveInt32 : IEquatable<PositiveInt32> {
    public bool Equals(PositiveInt32 other) { ... }
    public override bool Equals(object obj) { ... }
    public static bool operator==(PositiveInt32 x, PositiveInt32 y){
        return x.Equals(y);
    }
    public static bool operator!=(PositiveInt32 x, PositiveInt32 y){
        return !x.Equals(y);
    }
    ...
}
```

- ✗ **DO NOT** throw exceptions from equality operators.

For example, return false if one of the arguments is null instead of throwing `NullReferenceException`.

- ✓ **DO** implement `IEquatable<T>`, with the same behavior as the operator, on any type that defines `operator==`.

For more information on `IEquatable<T>`, see section 8.6.

- ✓ **DO** implement `operator==` with the mathematical reflexive and transitive properties.

The mathematical reflexive property states that when `a == b`, then `b == a`. To maintain the reflexive property when you define the equality operator against another type, you need to define it twice—once with this type as the first parameter, and once with this type as the second parameter. You also need to ensure that the values are the same, of course.

```
// Bad: (int == SomeType) is not defined when (SomeType == int) is
public partial struct SomeType {
    public static bool operator==(SomeType left, int right){ ... }
}

// OK: Both (int == SomeType) and (SomeType == int) are defined
```

```
// OK: Both operators return the same answer for the same values
// Bad: Not transitive (the next explanation)
public partial struct SomeType {
    public static bool operator==(SomeType left, int right){ ... }

    public static bool operator==(int left, SomeType right)
        => right == left;
}
```

The mathematical transitive property states that when $a == b$ and $b == c$, then $a == c$. To maintain the transitive property when you define the equality operator against another type, you need to also define an overload that compares two instances of the defining type.

In the previous example, if we had two `SomeType` instances, we could compare them against each other by indirectly comparing them against some `Int32` value that returns true for at least one instance, but we should be able to directly compare the two `SomeType` instances.

```
// OK: Now == is both transitive and reflexive.
// Defining operator!= is an exercise left to the reader.
public struct SomeType {
    public static bool operator==(SomeType left, int right){ ... }

    public static bool operator==(int left, SomeType right)
        => right == left;

    public static bool operator==(SomeType left, SomeType right) { ... }
}
```

X **AVOID** defining `operator==` with arguments of different types.

The addition of one different-type equality operator should be accompanied by the reflexive variant, the instance `Equals` method in the defining type, the instance (or extension) method for the comparison type, and both non-equality operators.

Instead of defining six methods, consider defining a conversion operator (section 5.7.2) or exposing the compared value via a property. For example, equality between a `DateTime` and a `DateTimeOffset` value is

possible because of a conversion operator from `DateTime` to `DateTimeOffset`, or more explicitly because `DateTimeOffset` has a property to return an equivalent `DateTime` value (`UtcDateTime`).

```
// Why do this much work?
public partial struct DateTimeOffset {
    public static bool operator==(DateTimeOffset left, DateTime right) => left.Equals(right);

    public static bool operator==(DateTime left, DateTimeOffset right) => right.Equals(left);

    public static bool operator!=(DateTimeOffset left, DateTime right) => !left.Equals(right);

    public static bool operator!=(DateTime left, DateTimeOffset right) => !right.Equals(left);

    public bool Equals(DateTime other) { ... }
}

public static partial class DateTimeOffsetComparisons {
    public static bool Equals(this DateTime left, DateTimeOffset right) => right.Equals(left);
}

// When you instead can do this much work?
public partial struct DateTimeOffset {
    public DateTime UtcDateTime { get { ... } }
}

// Or this much work?
public partial struct DateTimeOffset {
    public static implicit operator DateTimeOffset(DateTime value) { ... }
}
```

8.13.1 Equality Operators on Value Types

✓ **DO** overload the equality operators on value types, if equality is meaningful.

In most programming languages, there is no default implementation of `operator==` for value types.

8.13.2 Equality Operators on Reference Types

X AVOID overloading equality operators on mutable reference types.

Many languages have built-in equality operators for reference types. The built-in operators usually implement the reference equality, and many developers are surprised when the default behavior is changed to the value equality.

This problem is mitigated for immutable reference types because immutability makes it much harder to notice the difference between reference equality and value equality.

X AVOID overloading equality operators on reference types if the implementation would be significantly slower than that of reference equality.

The next chapter discusses a set of design patterns used in the design of types in .NET that we feel will be of help to other framework designers.



9

Common Design Patterns

HERE ARE NUMEROUS books on software patterns, pattern languages, and anti-patterns that address the very broad subject of patterns. Thus, this chapter provides guidelines and discussion related to a very limited set of patterns that are used frequently in the design of .NET APIs.

9.1 Aggregate Components

Many feature areas might benefit from one or more façade types that act as simplified views over more complex but also more powerful APIs. A façade that supports component-oriented design (see section 9.1.1) is called an aggregate component.

An aggregate component ties multiple lower-level factored types into a higher-level component to support common scenarios, and is the entry point for developers exploring its namespace. An example might be an e-mail component that ties together Simple Mail Transfer Protocol (SMTP), sockets, encodings, and so on. It is important for an aggregate component to provide a higher abstraction level rather than just a different way of doing things.

Providing simplified high-level operations is crucial for those developers who do not want to learn the whole extent of the functionality

provided by the feature and just need to get their (often very simple) tasks done.

■ **KRZYSZTOF CWALINA** `System.Net.Http.HttpClient` is an example of an aggregate component. It provides an API for simple scenarios in the `System.Net` namespace. Other examples of such components include `System.Messaging.MessageQueue`, `System.IO.SerialPort`, and `System.Diagnostics.EventLog`.

Aggregate components, as high-level APIs, should be implemented so they magically work without the user being aware of the sometimes complicated things happening underneath. We often refer to this concept as It-Just-Works. For example, the `EventLog` component hides the fact that a log has a read handle and a write handle that need to be opened. As far as the user is concerned, the component can be instantiated, properties can be set, and log events can be written.

Sometimes a bit more transparency is required. We recommend greater transparency for operations if the user would be required to take an explicit action as a result of an operation. For example, implicitly opening a file and then requiring the user to explicitly close it is probably taking the It-Just-Works principle a bit too far.

■ **KRZYSZTOF CWALINA** An important API design principle is that complexity should be either completely (or very close to completely) hidden or not hidden at all. The worst thing you can do is design an API that looks simple but as developers start to use it, they discover (usually the hard way) that it is not.

It is often possible to design clever solutions that hide even those complexities. For example, reading a file can be implemented as a single operation that opens a file, reads its content, and closes it, thus shielding the user from all the complexities related to opening and closing the file handles.

```
string[] lines = File.ReadAllLines("foo.txt");
```

Users of aggregate components should not be required to implement any interfaces, modify any configuration files, and so on. Framework designers should ship default implementations for all interfaces they declare. All configuration settings should be optional and backed by sensible defaults. Tools and IDE features should be considered for all common development tasks that are required beyond writing simple lines of code. In other words, framework designers should provide full end-to-end solutions, not just the APIs.

■ **KRZYSZTOF CWALINA** The `System.ServiceProcess` namespace greatly simplifies development of Windows Service applications. The API would be more successful with a wider range of developers if writing a service relied on hooking up event handlers instead of requiring the developer to override methods.

Aggregate components are frequently integrated with Visual Studio designers by implementing `IComponent` or deriving from one of the UI Element classes.

The next section describes component-oriented design, an important concept in the design of high-level APIs, particularly in the design of aggregate components.

9.1.1 Component-Oriented Design

Component-oriented design is a design in which APIs are exposed as types, with constructors, properties, methods, and events. It actually has more to do with the way the API is used than with the mere inclusions of the constructors, methods, properties, and events. The usage model for component-oriented design follows a pattern of instantiating a type with a default or relatively simple constructor, setting some instance properties, and then calling simple instance methods. We call this pattern the Create–Set–Call Pattern.

```
' VB.NET sample code
' Instantiate
Dim t As New T()
```

```

' Set properties/options
t.P1 = v1
t.P2 = v2
t.P3 = v3

' Call methods and optionally change options between calls
t.M1()
' t.P3 = v4
t.M2()

```

A concrete example showing the Create–Set–Call usage pattern would look like the following:

```

' Instantiate
Dim queue As New MessageQueue()

' Set properties
queue.Path = queuePath
queue.EncryptionRequired = EncryptionRequired.Body
queue.Formatter = New BinaryMessageFormatter()

' Call methods
queue.Send("Hello World")
queue.Close()

```

It is very important that all aggregate components support this pattern. The Create–Set–Call pattern is something that users of aggregate components expect and for which tools such as IntelliSense and designers are optimized.

■ **STEVEN CLARKE** We have investigated this design pattern extensively in our usability labs. Our observations highlighted just how critical this pattern is for some developers. Without it, reading, writing, and debugging code can be much more difficult, because it can be more difficult to learn about the purpose of each parameter that a method takes.

One problem with component-oriented design is that it sometimes results in types that can have modes and invalid states. For example, a default constructor allows users to instantiate a `MessageQueue` component without providing a valid path. Also, properties, which can be set

optionally and independently, sometimes cannot enforce consistent and atomic changes to the state of the object. The benefits of component-oriented design often outweigh these drawbacks for mainline scenario APIs, such as aggregate components, where usability is the top priority.

Also, some problems can and should be mitigated with proper error reporting. When users call methods that are not valid in the current state of the object, an `InvalidOperationException` should be thrown. The exception's message should clearly explain what properties need to be changed to get the object into a valid state.

■ **STEVEN CLARKE** Following this pattern means that it is very easy to learn how to use the API through actual usage, rather than having to resort to documentation.

Often, API designers try to design types so objects cannot exist in an invalid state. This is accomplished by having all required settings as parameters to the constructor, having get-only properties for settings that cannot be changed after instantiation, and breaking functionality into separate types so that properties and methods do not overlap. This approach is strongly recommended for factored types (see section 9.1.2) but does not work for aggregate components. For aggregate components, we recommend relying on clear exceptions for communicating invalid states to the user. Exceptions should be thrown when an operation is being performed, not when the component is initialized (i.e., don't throw from the constructor or the when a property is being set). This is important for avoiding situations in which the invalid state is temporary and gets "fixed" in a subsequent line of code.

```
var workingSet = new PerformanceCounter();
workingSet.Instance = process.ProcessName;
// Exception is not thrown here even though the counter is in an
// invalid state (counter is not specified).

workingSet.Counter = "Working Set"; // State is "fixed" here!
workingSet.Category = "Process";

Debug.WriteLine(workingSet.NextValue());
```

■ **CHRISTOPHE NASARRE** It is not recommended to define a class with the opposite pattern: Create–Call–Get. For example, don't define a Session class that provides a Login method that pops up a dialog in which the end user enters his credential. The Get properties for the user name, for example, are valid for use only after the call to Login. Instead, you should implement the Login method to return another type that contains the credential details. It is even worse if the class provides other methods that depend on methods to be called in specific order, such as GetSessionInfo, only if the Login method has already been called.

An aggregate component is a façade based on component-oriented design with the following additional requirements:

- **Constructors:** An aggregate component should provide a simple constructor.
- **Constructors:** All constructor parameters correspond to and initialize properties.
- **Properties:** Most properties have getters and setters.
- **Properties:** All properties have sensible defaults.
- **Methods:** Methods do not take parameters if the parameters specify options that stay constant across method calls (in main scenarios). Such options should be specified using properties.
- **Events:** Methods do not take delegates as parameters. All callbacks are exposed as events.

9.1.2 Factored Types

As described in the preceding section, an aggregate component provides shortcuts for most common high-level operations and is usually implemented as a façade over a set of more complex but also richer types. We call these types factored types.

Factored types should not have modes and should have very clear lifetimes. An aggregate component might provide access to its internal factored types through some properties or methods. Users would access the

internal factored types in advanced scenarios or in scenarios where integration with different parts of the system is required. The following example shows an aggregate component (`SerialPort`) exposing its internal factored type (a serial port `Stream`) through the `BaseStream` property.

```
var port = new SerialPort("COM1");
port.Open();
GZipStream compressed;
compressed = new GZipStream(port.BaseStream,
    CompressionMode.Compress);
compressed.Write(data, 0, data.Length);
port.Close();
```

■ **PHIL HAACK** Since factored types have an explicit lifetime, it probably makes good sense to implement the `IDisposable` interface so that developers can make use of the `using` statement. The code sample here could then be refactored to:

```
using(SerialPort port = new SerialPort("COM1")) {
    port.Open();
    GZipStream compressed;
    compressed = new GZipStream(port.BaseStream,
        CompressionMode.Compress);
    compressed.Write(data, 0, data.Length);
}
```

9.1.3 Aggregate Component Guidelines

The following guidelines apply to designing aggregate components.

✓ **CONSIDER** providing aggregate components for commonly used feature areas.

Aggregate components provide high-level functionality and are starting points for exploring given technology. They should provide shortcuts for common operations and add significant value over what is already provided by factored types. They should not simply duplicate the functionality. Many main scenario code samples should start with an instantiation of an aggregate component.

■ **KRZYSZTOF CWALINA** An easy trick to increase the visibility of an aggregate component is to choose the most “attractive” name for the component and less attractive names for the corresponding factored types. For example, a name representing a well-known system entity like `File` will attract more attention than `StreamReader`.

✓ **DO** model high-level concepts (physical objects) rather than system-level tasks with aggregate components.

For example, the components should model files, directories, and drives, rather than streams, formatters, and comparers.

✓ **DO** increase the visibility of aggregate components by giving them names that correspond to well-known entities of the system, such as `MessageQueue`, `Process`, or `EventLog`.

✓ **DO** design aggregate components so they can be used after very simple initialization. If some initialization is necessary, the exception resulting from not having the component initialized should clearly explain what needs to be done.

✗ **DO NOT** require the users of aggregate components to explicitly instantiate multiple objects in a single scenario.

Simple tasks should be done with just one object. The next best thing is to start with one object that in turn creates other supporting objects. Your top five scenario samples showing aggregate component usage should not have more than one new statement.

```
var queue = new MessageQueue();
queue.Path = ...;
queue.Send("Hello World");
```

■ **KRZYSZTOF CWALINA** Book publishers say that the number of copies a book will sell is inversely proportional to the number of equations in the book. The API designer version of this law is that the number of developers who will use your API is inversely proportional to the number of new statements in your simple scenarios.

- ✓ **DO** make sure aggregate components support the Create–Set–Call usage pattern, where developers expect to be able to implement most scenarios by instantiating the component, setting its properties, and calling simple methods.
- ✓ **DO** provide a default or a very simple constructor for all aggregate components.

```
public class MessageQueue {
    public MessageQueue() { ... }
    public MessageQueue(string path) { ... }
}
```

- ✓ **DO** provide properties with getters and setters corresponding to all parameters of aggregate component constructors.

It should always be possible to use the default constructor and then set some properties instead of calling a parameterized constructor.

```
public class MessageQueue {
    public MessageQueue() { ... }
    public MessageQueue(string path) { ... }

    public string Path { get { ... } set { ... } }
}
```

- ✓ **DO** use events instead of delegate-based APIs in aggregate components.

Aggregate components are optimized for ease of use, and events are much easier to use than APIs using delegates. See section 5.4 for more details.

- ✓ **DO** use events in aggregate components instead of virtual members that need to be overridden.

- ✗ **DO NOT** require users of aggregate components to inherit, override methods, or implement any interfaces in common scenarios.

Components should mostly rely on properties and composition as the means of modifying their behavior.

✗ **DO NOT** require users of aggregate components to do anything besides writing code in common scenarios. For example, users should not have to configure components in the configuration file, generate any resource files, and so on.

✓ **CONSIDER** making changes to aggregate components' modes automatic.

For example, a single instance of `MessageQueue` can be used to send and receive messages, but the user should not be aware that mode switching is occurring.

✗ **DO NOT** design factored types that have modes.

Factored types should have a well-defined life span scoped to a single mode. For example, instances of `Stream` can either read or write, and an instantiated stream is already opened.

✓ **CONSIDER** integrating your aggregate components with Visual Studio designers.

Integration allows placing the component on the Visual Studio Toolbox and adds support for drag and drop, property grid, event hookup, and so on. The integration is simple and can be done by implementing `IComponent` or inheriting from a type implementing this interface, such as `Component` or `Control`.

✓ **CONSIDER** separating aggregate components and factored types into different assemblies.

This allows the component to aggregate arbitrary functionality provided by factored types without circular dependencies.

✓ **CONSIDER** exposing access to internal factored types of an aggregate component.

Factored types are ideal for integrating different feature areas. For example, the `SerialPort` component exposes access to its stream, thus allowing integration with reusable APIs, such as compression APIs that operate on streams.

9.2 The Async Patterns

Operations that feature I/O (file, network, interprocess communication, external hardware, or any other transfer-control-and-wait model) can often benefit from asynchronous APIs. Asynchronous APIs use threads more efficiently and so are better choices for highly concurrent applications.

.NET has three distinct API patterns to model asynchronous APIs: the Classic Async Pattern (a.k.a. Async Pattern, the Begin/End pattern, IAsyncResult, or Asynchronous Programming Model [APM]), the Event-Based Async Pattern (a.k.a. EAP, or Async Pattern for Components), and the Task-Based Async Pattern.

9.2.1 Choosing Between the Async Patterns

Historically, when only the Classic Async Pattern and the Event-Based Async Pattern were available, the main difference between the patterns was that the Event-Based Async Pattern was optimized for usability and integration with visual designers, whereas the Classic Async Pattern was optimized for power and small surface area.

Especially when combined with the language support from the C#, Visual Basic, and F# languages, the Task-Based Async Pattern offers the best usability and smallest surface area of the three asynchronous method patterns. Therefore, we now consider the Event-Based Async Pattern and the Classic Async pattern to be legacy patterns, not suitable for use on new types.

- ✓ **DO** implement new asynchronous API using the Task-Based Async Pattern.
- ✓ **CONSIDER** upgrading a Classic Async Pattern or Event-Based Async Pattern API to the Task-Based Async Pattern.

The TaskFactory and TaskFactory<TResult> types can be used to provide Task-Based Async Pattern methods by wrapping existing Classic Async Pattern and Event-Based Async Pattern APIs. However, the prevalence of Task-Based Async Pattern APIs in .NET generally makes it easier to write the new Task-Based Async Pattern methods

directly and continue to support existing async methods from the older patterns in terms of the new methods.

```
// Synchronous method
public ParsedData Parse(string filename) { ... }

// Classic Async Pattern
public IAsyncResult BeginParse(
    string filename,
    AsyncCallback callback,
    object state) { ... }

public ParsedData EndParse(IAsyncResult asyncResult) { ... }

// Task-Based Async Pattern
//
// Written to wrap an existing Classic Async Pattern method,
// which does not support cancellation.
public Task<ParsedData> ParseAsync(string filename) {
    if (filename == null)
        throw new ArgumentNullException(nameof(filename));

    return TaskFactory<ParsedData>.FromAsync(
        BeginParse, EndParse, filename, null);
}

// Event-Based Async Pattern
public event EventHandler<ParseEventArgs> ParseCompleted;

// Written to call the new Task-Based Async Pattern method instead
public void ParseAsync(string filename, object userState) {
    ParseAsync(filename).
        ContinueWith(
            FireEapEvent,
            AsyncOperationManager.CreateOperation(userState));
}

private void FireEapEvent(Task<ParsedData> task, object userState){
    EventHandler<ParseEventArgs> completed = ParseCompleted;

    if (completed != null) {
        AsyncOperation op = (AsyncOperation)userState;
        ParseEventArgs args;
```

```

        if (task.IsFaulted) {
            args = ParseEventArgs.FromException(
                task.Exception.InnerException, op.UserSuppliedState);
        } else {
            args = ParseEventArgs.FromResult(
                task.Result, op.UserSuppliedState);
        }

        op.PostOperationCompleted(
            a => completed(this, (ParseEventArgs)a), args);
    }
}

```

9.2.2 Task-Based Async Pattern

The Task-Based Async Pattern is a naming, method signature, and behavioral convention for providing APIs that can be used to execute asynchronous operations. The pattern is based primarily around two types in .NET: `System.Threading.Tasks.Task`, for methods that do not return a result, and `System.Threading.Tasks.Task<TResult>`, for methods that have a result of type `TResult`. The Task-Based Async Pattern can be implemented manually, using these types directly, or in conjunction with language support (such as `async/await` in C#, `Async/Await` in Visual Basic .NET, and `async/let!/use!/do!` in F#). The pattern consists of the following elements:

- The “-`Async`” method, which initiates an asynchronous operation
- The `Task` (or `Task<TResult>`) object, which is returned from the “-`Async`” method and provides a way of getting the status of the operation as well as its result or a resultant exception
- An optional `CancellationToken` input parameter, which allows a caller to request early termination of the operation

```

public partial class File {
    public Task<byte[]> ReadAllBytesAsync(
        string path,
        CancellationToken cancellationToken = default) { ... }
}

```

■ **JEREMY BARTON** Creating a method with the C# `async` keyword is not enough to appropriately conform to the Task-Based Async Pattern, but it is a good start. The keyword restricts the method return type to an appropriate value (in addition to the less-appropriate `void`), prohibits `ref` `struct` types as parameters, and prohibits the use of the `ref` and `out` parameter modifiers. But the keyword does not enforce naming, proper handling of usage exceptions, parameter alignment to synchronous method variants, cancellation, or other nuances.

✓ **DO** use the suffix “`Async`” when naming an asynchronous method.

The “`Async`” suffix provides a clear indicator to both callers and code reviewers that the method is asynchronous, and thus requires special handling. The suffix also disambiguates synchronous and asynchronous variants of the same functionality when both methods have the same parameter list (e.g., in cases where the synchronous method accepts a `CancellationToken`).

The “`Async`” suffix should not be used when naming types, because even a type that is primarily designed for asynchronous operations may have synchronous methods for configuring the instance.

■ **JEREMY BARTON** One reason that I’ve heard for why the “`Async`” suffix isn’t needed is that it’s obvious from usage, and that you’ll fail to compile if you forget to `await` the result. But reading a property named `IsCompleted` or `Exception`, or calling a method named `Dispose` or `Wait`, is easy to mistakenly do on the returned Task-like value instead of the value that should have been produced from `await`.

✓ **CONSIDER** adding synchronous variants of `Async` methods.

```
// asynchronous method
public Task<ParsedData> ParseAsync(
    string filename,
    CancellationToken cancellationToken = default) { ... }
```

```
// synchronous variant  
public ParsedData Parse(string filename) { ... }
```

Existing codebases may already have a large number of synchronous methods, and converting them to support asynchronous calls may not be a high priority for those developers compared to the functional value that your component offers. If you only provide asynchronous operations, then a synchronous caller is forced to call the blocking `GetResult()` method, creating a condition called “sync-over-async.” See section 9.2.5.3 for more information.

- ✓ **DO** accept a `CancellationToken` parameter, named “cancellation-`Token`,” in asynchronous methods.

Long-running operations generally warrant an “abort” option, and asynchronous operations tend to be long-running. The Task-Based Async Pattern uses the `System.Threading.CancellationToken` type to control cancelations.

```
// Incorrect: The caller has no way of cancelling this operation  
public Task WriteAsync(string text) { ... }
```

```
// Correct: Cancellation is supported, as an optional parameter  
public Task WriteAsync(  
    string text,  
    CancellationToken cancellationToken = default) { ... }
```

- ✓ **DO** provide a default value to the `CancellationToken` value.

- ✓ **CONSIDER** keeping the `CancellationToken` as the last parameter, for better alignment with a synchronous version of the method.

The last parameter rule is an exception to the normal rule of method overloads, which states that the additional parameters of the overloads go last. The C# language requires that all optional parameters (any parameter with a default value) come after all required parameters, and the combination of the `CancellationToken` rules plays well with that requirement.

For example, the following example is incorrect for overloads involving `CancellationToken`—even if you make a conscious decision to not provide a default value:

```
public Task WriteAsync(
    string text,
    CancellationToken cancellationToken) { ... }

public Task WriteAsync(
    string text,
    CancellationToken cancellationToken,
    Encoding encoding) { ... }
```

These overloads should instead have been

```
public Task WriteAsync(
    string text,
    CancellationToken cancellationToken) { ... }

public Task WriteAsync(
    string text,
    Encoding encoding,
    CancellationToken cancellationToken) { ... }
```

✓ **CONSIDER** accepting a `CancellationToken` in long-running or blocking synchronous methods.

The `CancellationToken` type isn't tied to asynchronous execution or `Tasks`, so it is equally suited to synchronous operations that would benefit from timeout-based or user-initiated abort signals. Synchronous methods that pause until some other action is completed—known as blocking methods—are often good candidates for a `CancellationToken`. The two most common blocking operations are waiting on thread synchronization and an I/O operation such as a file download.

As an example, the `BlockingCollection<T>` class supports a cancelable version of `Add`:

```
public partial class BlockingCollection<T> {
    public void Add(T item) {}
    public void Add(T item, CancellationToken cancellationToken) {}
}
```

The cancelable Add overload is not an asynchronous operation: It uses a synchronous `Wait()` instead of an asynchronous yield to pause until there's space in the collection. This same type of cancellation support would apply to a long-running, nonblocking operation such as prime factorization.

X DO NOT return Tasks for long-running synchronous methods.

If a method is not using asynchronous yield (either it's a long-running computation or it's using a synchronous block), it should not return Tasks or use the "Async" suffix. Since synchronous methods are not part of the Task-Based Async Pattern, neither the guidance to keep the `CancellationToken` parameter as the final parameter nor the guidance to provide a default value for the parameter applies.

■ STEPHEN TOUB Related to this, you are strongly discouraged from exposing a Task-returning method that simply wraps a synchronous method by queueing the call to the synchronous method—doing so makes it harder for a consumer of your API to choose the best method for their needs. The main exception to this guidance is when you are exposing the async version as virtual and expect derived overrides to be able to provide truly asynchronous implementations.

Because the previous example of `BlockingCollection<T>.Add` uses synchronous `Wait()`, it is correct that it doesn't follow the Task-Based Asynchronous Pattern.

X DO NOT use `ref` or `out` parameter modifiers in an Async method.

The .NET runtime only permits value references on the stack (method parameters or locals), which cannot be maintained if an asynchronous method needs to yield execution. The C# compiler understands this limitation, so declaring a `ref` or `out` parameter on a method that uses the `async` keyword is a compile error, but it will not issue an error on arbitrary methods that return Tasks.

```
public Task<bool> TryParseAsync(
    TextReader reader,
    out ParsedData value) {
```

```
Task<ParsedData> parseTask = ParseAsync(reader);

// This method isn't actually asynchronous;
// it does a synchronous wait!
parseTask.Wait();

if (parseTask.IsFaulted) {
    value = default;
    return Task.FromResult(false);
}

value = parseTask.Result;
return Task.FromResult(true);
}
```

To maintain the ability to write to the `out` parameter, the `TryParseAsync` method instead became a blocking synchronous method masquerading as an asynchronous method. This is not only misleading to the callers of this method, but also can cause deadlock due to the use of `Task.Wait()` (see section 9.2.5.3 for more information).

X DO NOT use the `in` (`ref readonly`) parameter modifier in a virtual or abstract `Async` method.

X AVOID the `in` (`ref readonly`) parameter modifier in nonvirtual `Async` methods.

The C# compiler will issue an error if the `in` parameter modifier is used in a signature of a method with the `async` keyword, which will prevent the developers extending the type from using the language support for implementing an override for the asynchronous method.

When implementing a virtual asynchronous method with the Template Method Pattern (section 9.9), it is not a compile-time error to have a parameter with the `in` modifier. Be aware that if the parameter type is not a `readonly struct`, a caller might expect that the asynchronous method can react to the caller modifying the value; however, since a value copy is made between the public method and the protected method, the asynchronous operation will not see mutations done by the caller. Since a `readonly struct` cannot be modified, there is no possibility of confusion, so the `in` modifier can be used if appropriate.

For more information on when it is appropriate to use the `in` modifier, see section 5.8.3.

```
public Task WriteParametersAsync(
    string filename,
    in RSAParameters rsaParameters) {

    if (rsaParameters.Modulus == null) {
        throw new ArgumentException(
            "Modulus is required.",
            nameof(rsaParameters));
    }

    // A value copy of rsaParameters is made here.
    return WriteParametersAsyncCore(filename, rsaParameters);
}

protected abstract Task WriteParametersAsyncCore(
    string filename,
    RSAParameters rsaParameters);

...

RSAParameters rsaParameters = GetRSAParameters();
// The optional 'in' keyword is shown here for clarity.
Task saveToFile = WriteParametersAsync(filename, in rsaParameters);

// Because the struct was passed as 'in' the code behind the Task
// will see this change, right? (Wrong.)
rsaParameters.Exponent = s_someOtherExponent;
await saveToFile;
```

JEREMY BARTON While the example here just has a linear flow, things get trickier if the mutable `struct` is a field and the application is multi-threaded. A concurrent write could happen between the argument validation phase and the value copy to the Core method, violating the contract that the base class was trying to uphold. Since this awkwardness only comes up when ignoring the guideline against public mutable value types (section 4.7) and the guidelines against using `in` (`ref readonly`) parameters in public APIs (section 5.8.3), hopefully no one will run into it when using your library.

9.2.3 Async Method Return Types

The C# `async` keyword requires that a method return a “Task-like” type.¹ .NET has four predefined options:

- `System.Threading.Tasks.Task`: A reference type representing an operation with no return value.
- `System.Threading.Tasks.Task<TResult>`: A reference type representing an operation returning a value of type `TResult`.
- `System.Threading.Tasks.Task.ValueTask`: A value type representing an operation with no return value.
- `System.Threading.Tasks.Task.ValueTask<TResult>`: A value type representing an operation returning a value of type `TResult`.

Before the introduction of `ValueTask<TResult>`,² it was easy to choose the appropriate return type: The asynchronous variant should use `Task` when a synchronous equivalent method would be `void`-returning, and `Task<TResult>` otherwise. While it is still easy to choose between the generic types and the non-generic ones, choosing between `Task<TResult>` and `ValueTask<TResult>` is a little less straightforward.

The primary motivation behind the `ValueTask<TResult>` type is to reduce memory allocations when calling asynchronous methods that happen to complete synchronously. Unfortunately, the performance benefits of reduced memory usage come with a reduction in usability and utility. The most obvious usability problem with `ValueTask` is that many methods in base class libraries, such as `Task.WaitAny(Task[])`, cannot directly accept a `ValueTask` or `ValueTask<TResult>`. Also `ValueTask` and `ValueTask<TResult>` are easier to misuse, as discussed in section 9.2.5.4.

1. Methods built with the `async` keyword are also permitted to have a `void` return type. Such `async void` methods are closer to the Event-Based Async Pattern than the Task-Based Async Pattern, and are not recommended in public code.
2. `Task<TResult>` was introduced in .NET Framework 4.0 (2010), and `ValueTask<TResult>` was introduced in .NET Core 1.0 (2016).

The simple flow for determining the best return type is as follows:

- If your method produces no result: Task.
- If your method commonly completes synchronously:
`ValueTask<TResult>`.
- Otherwise: `Task<TResult>`.

This flow is justified with the subsequent guidance and explanations.

✓ **DO** use (non-generic) Task as the preferred return type for an asynchronous method that does not produce a value.

The Task class has better usability than the ValueTask struct, and both types have similar performance characteristics for methods generated with the `async` keyword.

✓ **DO** use `Task<TResult>` as the preferred return type for an asynchronous method that produces a value.

Since the `Task<TResult>` class is a reference type, and the value it carries can be different every time an asynchronous method is called, it comes at a slight memory allocation cost compared to `ValueTask<TResult>`. Often, the extra memory allocation doesn't matter, so the better usability of the `Task<TResult>` class makes it a better default.

■ **JEREMY BARTON** There is a limited amount of synchronous-success caching for `Task<TResult>` for methods built with the `async` keyword. For .NET Core 3.1, this is the Boolean values `true` and `false`, the `Int32` values -1 through 9, and the 0 value for `Byte`, `SByte`, `Int16`, `UInt16`, `UInt32`, `Int64`, `UInt64`, `IntPtr`, `UIntPtr`, and `Char`.

Some asynchronous methods that are built manually also cache their own common return values to provide both good critical-path performance and better usability of `Task<TResult>`.

✓ **CONSIDER** returning `ValueTask<TResult>` from an asynchronous method when the method commonly completes synchronously.

Analysis of the behavior of `System.IO.Stream` shows that in many cases there is already data available when the `ReadAsync` method is called, due to buffering; thus, the method often returns synchronously. Since the synchronous `Read` method returns an `Int32`, it can usually avoid having any memory impact when data is already available, but the `ReadAsync` method needs to produce a `Task<int>` to report the number of bytes read. The resulting small memory impact can become noticeable when done in a loop.

The newer `ReadAsync` overload on `Stream` that writes to `System.Memory<byte>` returns a `ValueTask<int>` so that the asynchronous method can similarly have no memory impact when data is already available.

```
public partial class Stream {  
    // The older, .NET Standard 1.0 versions  
    public virtual Task<int> ReadAsync(  
        byte[] buffer, int offset, int count) { ... }  
  
    public virtual Task<int> ReadAsync(  
        byte[] buffer, int offset, int count,  
        CancellationToken cancellationToken) { ... }  
  
    // The newer, .NET Standard 2.1 version  
    public virtual ValueTask<int> ReadAsync(  
        Memory<byte> buffer,  
        CancellationToken cancellationToken = default);  
}
```

`ValueTask<TResult>` doesn't always provide better performance than `Task<TResult>`. When a `ValueTask<TResult>` is returned by a method that is built with the `async` keyword, and the method did not complete synchronously, a new `Task<TResult>` instance is created to track the remainder of the operation. For operations that regularly complete asynchronously, returning a `ValueTask<TResult>` leads to a usability loss and a minor performance *penalty* for the caller, with no significant gain. Therefore the recommendation for methods that don't have an expectation of synchronous completion is to return the reference type, `Task<TResult>`.

■ **STEPHEN TOUB** This guidance may make you question why the non-generic ValueTask exists at all. It exists as a performance optimization for critical-path methods that frequently complete asynchronously and where the implementation is able to employ some kind of object reuse to avoid allocation. Many more details are available in <https://devblogs.microsoft.com/dotnet/understanding-the-whys-whats-and-whens-of-valuetask/>.

9.2.4 Making an Async Variant of an Existing Synchronous Method

Strictly speaking, method overloading only applies to methods with the same name on the same type. However, .NET API design reviews frequently refer to the asynchronous version of a synchronous method as the “async overload.” While this does stretch the definition of method overloading, it is useful to think of the methods as overloads when modeling their behavior—but not their names.

✓ **DO** keep parameters in the same order in the synchronous and asynchronous variants of a method, when possible.

When the synchronous method has no parameters passed as `out` or `ref`, and neither the return value or any of the parameters are `ref`-like value types (`ref struct` types), then you can declare the asynchronous variant by just adding the “`Async`” suffix, changing the return type to the appropriate Task-like type (section 9.2.3), and accepting a `CancellationToken`. For example:

```
public partial class XDocument {
    public static XDocument Load(
        TextReader textReader, LoadOptions options) { ... }

    // The real XDocument.LoadAsync does not have a default value for
    // cancellationToken; this version is better aligned with the
    // guidance in this section.
    public static Task<XDocument> LoadAsync(
        TextReader textReader, LoadOptions options,
        CancellationToken cancellationToken = default) { ... }
}
```

✓ **DO** convert the signature as needed to remove any `ref` or `out` parameters from an asynchronous method.

If the method had a single `out` parameter and was `void`-returning, then you can define the `Async` variant to simply return the value via a `Task`-like type.

```
public void CalculateValue(int input, out int result) { ... }
public Task<int> CalculateValueAsync(
    int input,
    CancellationToken cancellationToken = default) { ... }
```

If there is a preexisting return value or more than one `out` parameter, then you can create a new type to describe the more complex return value.

```
public partial class SomeClass {
    public int Divide(int divisor, int dividend, out int remainder);

    public Task<DivisionResult> DivideAsync(
        int divisor, int dividend,
        CancellationToken cancellationToken = default);
}

public readonly struct DivisionResult {
    public int Quotient { get; }
    public int Remainder { get; }

    public DivisionResult(int quotient, int remainder) { ... }
}
```

■ **JEREMY BARTON** This replacement strategy for `out` does not work if the method is designed to both write to the `out` and throw an exception. While that's generally considered bad design, it can be handled by the reference wrapper for `ref` replacement.

A `ref` parameter is more difficult to replace, since it describes zero or more value reads and zero or more value writes. If the intention of the method is single-read single-write, then one possibility is to logically separate the parameter into two parameters—a simple `input` parameter and an `out`—and then apply the suggested `out` replacement strategy.

```
public partial class SomeClass {
    public int SomeMethod(ref string value) { ... }
```

```

// Logical intermediate replacement; this method does not
// need to ever actually be defined
// public int SomeMethod(string value, out string updatedValue);

// Async variant based on the parameter separation
public Task<SomeResult> SomeMethodAsync(
    string value,
    CancellationToken cancellationToken = default);
}

public readonly struct SomeResult {
    public int CalculatedResult { get; }
    public string UpdatedValue { get; }

    public DivisionResult(int result, string updatedValue) { ... }
}

```

If the method expects to have `ref` reads and writes interact with another thread, have intermediate writes be visible to other threads, or just wants to avoid creating a new return type, you can wrap the value in a reference type.

```

public bool ContrivedMethod(
    byte[] data, ref string value, out int valueUtf8Length) { ... }

public Task<bool> ContrivedMethodAsync(
    byte[] data, RefWrapper<string> value,
    RefWrapper<int> valueUtf8Length,
    CancellationToken cancellationToken = default) { ... }

public class RefWrapper<T> {
    public T Value { get; set; }
    ...
}

```

It may be difficult to assess which read and write models the implementations and callers are expecting for virtual methods.

The final replacement strategy is to not convert a particularly troublesome method to `Async`. Consider that parameter guidance in section 5.8.3 recommends avoiding `ref` and `out`, design a new synchronous method avoiding those parameter passing conventions, and make an asynchronous version of that method instead.

■ **JEREMY BARTON** All of the examples in this section are contrived. I looked at all of the types and members in .NET Standard 2.1 and couldn't find any representative examples. The best example that I can provide is that during the transformation from the Classic Async Pattern to the Task-Based Async Pattern, `System.Net.Socket` simplified the `async` completion value from a return value and two `out` parameters to just the one return value.

That, combined with the fact that there is no public `RefWrapper<T>` type in .NET, should suggest that the answer most commonly chosen by the BCL team is to just redesign the method.

✓ **CONSIDER** replacing synchronous callbacks with asynchronous callbacks in the `Async` method, or accepting both via overloading.

Given a method that accepts a synchronous callback, such as `public void DoWork(Action<State> callback)`, you need to understand if the caller is expected to do synchronous or asynchronous work in the callback. If the callback is intended to save data, and the `DoWorkAsync` will call the callback asynchronously, then `DoWorkAsync` probably only needs an asynchronous callback. Conversely, a progress notification callback from an asynchronous version of loading saved state might only warrant a synchronous callback. If it's unclear, consider offering both.

```
// The original synchronous method
public void DoWork(Action<State> callback) { ... }

// An asynchronous variant with a synchronous callback
public void DoWorkAsync(
    Action<State> callback,
    CancellationToken cancellationToken = default) { ... }

// An asynchronous variant with an asynchronous callback
public void DoWorkAsync(
    Func<State, CancellationToken, Task> asyncCallback,
    CancellationToken cancellationToken = default) { ... }
```

9.2.5 Implementation Guidelines for Async Pattern Consistency

Using two-tab indentation, GNU-style bracing, or only single-letter variable names are not things that a caller can observe—but returning a null Task is. The guidance in this section focuses on those portions of method implementation that are observable to calling code, which should be consistent with other Task-Based Async Pattern methods in these observable effects.

X DO NOT return a null Task or `Task<TResult>`.

Very few callers would ever check an Async method for a null Task being returned. Instead of returning null, consider what this value means and then choose a better alternative. For example, if an Async method cannot be started, it should throw an exception (see section 9.2.6.5). Alternatively, if the work is already done, return `Task.CompletedTask` or `Task.FromResult(result)`.

9.2.5.1 `Task.Status` Consistency

Most methods interacting with `System.Threading.Tasks.Task` instances don't directly interact with the `Status` property. Nevertheless, to the machinery that drives Task, it is important that the `Status` value actually matches the state the operation is in.

✓ DO NOT return tasks in the `Created` state.

Task values created directly with a Task constructor are in the `Created` state, which allows customization before the Task begins work. If a Task is returned in the `Created` state (and the object isn't later started by another thread), then any caller that awaits the Task will wait indefinitely.

Tasks created by the compiler via the `async` keyword are always started before they are returned, as are Tasks from `Task.Run(...)` and `Task.Factory.StartNew(...)`.

■ **STEPHEN TOUB** This `Created` state is a good example of a public API that probably shouldn't have ever existed. The ability to create a `Task` separately from `Starting` it helps as an implementation detail in a very, very small number of cases. In catering to that need by exposing the `Created` state, the relevant constructors, and the `Start` method, we ended up creating a pit of failure that developers can easily fall into, while also making it a lot more difficult for us to optimize certain common use cases. This is an example where actually cutting back on the functionality exposed would have led to a greater good.

- ✓ **DO** throw an `OperationCanceledException` when aborting work due to a `CancellationToken`.

The `Task` execution engine handles `OperationCanceledException` and updates the `Task` to be in the `Canceled` state. Awaiting a `Task` in the `Canceled` state results in another `OperationCanceledException` being thrown, which normally results in a cascade that ends with program termination or a caller explicitly handling the `Canceled` state. If you use the `CancellationToken.IsCancellationRequested` value to do an early return from a method created with the `async` keyword, then the `Task` for your method moves to the `RanToCompletion` state—not the `Canceled` state—and any caller awaiting the `Task` continues as if the method had completed successfully. This behavior is often a source of bugs.

The easiest way to ensure that the correct behaviors happen is to invoke the `ThrowIfCancellationRequested()` method on the `CancellationToken`, but manually throwing an instance of `OperationCanceledException` works as well.

```
try {
    await SaveData(data, cancellationToken);
    QueueNotification("Data saved successfully!");
} catch (OperationCanceledException) {
    QueueNotification("Save was canceled.");
}
...
```

```
private async Task SaveData(  
    byte[] data, CancellationToken cancellationToken) {  
  
    // Bad: this causes the success message  
    if (cancellationToken.IsCancellationRequested) {  
        return;  
    }  
  
    // Good: this causes the canceled message for our caller  
    if (cancellationToken.IsCancellationRequested) {  
        throw new OperationCanceledException(cancellationToken);  
    }  
  
    // Good: this causes the canceled message for our caller  
    cancellationToken.ThrowIfCancellationRequested();  
  
    ...  
}
```

9.2.5.2 Awaiting on the Correct Context

- ✓ **DO** use `await task.ConfigureAwait(false)` when awaiting an `Async` operation, except in application models that depend on the synchronization context.

By default, .NET uses `SynchronizationContext.Current` to determine how to continue processing the Task. In a console application, the default `SynchronizationContext` sends Tasks to be executed on background threads with no special handling.

In graphical user interface (GUI) application models—such as WinForms or WPF—the default `SynchronizationContext` dispatches all Task completion callbacks to the UI thread to make it easier to interact with UI elements. Since the `SynchronizationContext` in a GUI application is using a single thread for execution, this behavior can lead to a reduction in throughput, and makes deadlock situations possible. Unless your method is interacting with a UI element, or another component that you know needs the synchronization context, then you should use `await task.ConfigureAwait(false)` to avoid unnecessarily tying up the UI thread.

■ **STEPHEN TOUB** There are many intricacies related to ConfigureAwait not captured in this short summary. For more details, <https://devblogs.microsoft.com/dotnet/configureawait-faq/> is a good source of information.

■ **JEREMY BARTON** Be aware that ConfigureAwait returns a wrapper around the Task—it doesn't modify the Task. Unless you await that expression, or save it to a local to await later, it has no effect.

```
Task t = SomeAsync(cancellationToken);
// This has no effect, the result was not awaited.
t.ConfigureAwait(false);
// This still comes back on the captured context.
await t;
```

9.2.5.3 Avoiding Deadlock

X DO NOT invoke Task.Wait() or read the Task.Result property in an Async method; instead, use await.

The await keyword causes the Task to yield, which allows other Tasks to continue. The Task.Wait() method—and the Task.Result property—instead do a blocking synchronous wait, which is inefficient and can lead to deadlock in some situations.

✓ DO call asynchronous method variants, instead of synchronous method variants, in the implementation of Async methods.

In general, if there are both a synchronous variant and an asynchronous variant for a method, the synchronous method will potentially block and the asynchronous method will potentially yield. Calling a blocking method from an asynchronous method is inefficient, and can lead to resource starvation and deadlock.

9.2.5.4 Properly Handling ValueTask and ValueTask<TResult>

X DO NOT operate on an instance of ValueTask or ValueTask<TResult> more than once or save it; only ever await or return it.

As mentioned in section 9.2.3, `ValueTask` and `ValueTask<TResult>` have the potential to provide a performance benefit in some cases, but at the cost of some robustness and usability.

Unlike with `Task` and `Task<TResult>`, awaiting the same `ValueTask` or `ValueTask<TResult>` twice is not safe, and has undefined behavior. The “undefined behavior” extends to anything that can observe the result of a `ValueTask<TResult>`, such as reading the `Result` property or calling `ToString()`. If you need to do anything more complicated than a simple `await` or `return`, get a `Task` (or `Task<TResult>`) from the `AsTask()` method and ignore the `ValueTask`.

■ **JEREMY BARTON** An easy oversimplification is “never have a `ValueTask` or `ValueTask<T>` local variable.” Once you need to talk about the value, write something like

```
Task<int> task = SomeMethodAsync(...).AsTask();
// complicated stuff goes here
```

9.2.5.5 Exceptions from Async Methods

✓ **DO** throw usage error exceptions directly from the `Async` method, to aid in debuggability.

Usage error exceptions (see section 7.2) are indicative of an error in the calling code. If you throw a usage error exception directly from the called method that conveys a few benefits over the exception being wrapped inside `Tasks`:

1. The callstack very clearly indicates where the usage violation came from, making it easier for the caller to diagnose and fix.
2. There’s no chance that the exception will be ignored when the `Task` is never awaited.
3. The invalid inputs don’t end up spinning up the `Task` machinery, only to cause it to spin down again.

One effect of this guidance is that usage error exceptions should not be thrown from `public async` or `protected async` methods. Instead, the public (or protected) method should do usage validation and then defer to a non-public `async` method (or an `async` local function, or another means of separating “inside” and “outside” the Task).

```
// Wrong: The exception is thrown inside the Task;
// the call stack will be less clear
public async Task SaveAsync(string filename) {
    if (filename == null)
        throw new ArgumentNullException(nameof(filename));

    ...
}

// Right: The exception is thrown instead of a Task being returned.
public Task SaveAsync(string filename) {
    if (filename == null)
        throw new ArgumentNullException(nameof(filename));

    return SaveAsyncCore(uri);
}

private async Task SaveAsyncCore(string filename){ ... }
```

- ✓ **DO** throw execution error exceptions via the Task-like value returned from an Async method.

Execution error exceptions can be the result of invalid data, which might only be available after a nested Async operation has finished. Since these “late” execution error exceptions would be captured within the Task, the most consistent model is to ensure that all execution error exceptions are held in the Task.

This example can incorrectly throw a `FileNotFoundException`, an execution error exception, in such a way that it is not captured within the `Task<byte[]>`:

```
public static Task<byte[]> ReadAllBytesAsync(string path) {
    // Maybe we can just let OpenRead throw the
    // ArgumentNullException for us?
    Stream stream = File.OpenRead(path);
```

```
// Unfortunately, it also could have thrown  
// AccessDeniedException, FileNotFoundException,  
// and a number of other execution error-based exceptions.  
return ReadAllBytesAsyncCore(stream);  
}
```

9.2.7 Classic Async Pattern

The Classic Async Pattern is not recommended for new APIs. When adding new asynchronous functionality to an existing type that already implements the Classic Async Pattern, the recommendation is to upgrade the existing API to the Task-Based Async Pattern and to implement the new functionality only with the Task-Based Async Pattern.

For historical reasons, the information from this section in previous editions has been moved to Appendix B, section B.9.2.2.

```
// Classic Async Pattern  
// This pattern is deprecated.  
// Use the Task-Based Async Pattern instead.  
public IAsyncResult BeginParse(  
    string filename,  
    AsyncCallback callback,  
    object state) { ... }  
  
public ParsedData EndParse(IAsyncResult asyncResult) { ... }
```

9.2.8 Event-Based Async Pattern

The Event-Based Async Pattern is not recommended for new APIs. When adding new asynchronous functionality to an existing type that already implements the Event-Based Async Pattern, the recommendation is to upgrade the existing API to the Task-Based Async Pattern and to implement the new functionality only with the Task-Based Async Pattern. If the mix of the Event-Based Async Pattern and the Task-Based Async Pattern seems messy, or if you think callers might be confused by the existing event model with respect to the new APIs, then it may make more sense to introduce a new type to carry the Task-Based Async Pattern APIs. Then use your judgment to decide if the new functionality should be available only for the new Task-Based Async Pattern-based type, or if you also need

to provide the new functionality on the existing type via the Event-Based Async Pattern.

For historical reasons, the information from this section in previous editions has been moved to Appendix B, section B.9.2.4.

```
// Event-Based Async Pattern
// This pattern is deprecated.
// Use the Task-Based Async Pattern instead.
public event EventHandler<ParseEventArgs> ParseCompleted;
public void ParseAsync(string filename, object userState) { ... }
```

9.2.9 IAsyncDisposable

`IAsyncDisposable` is discussed along with `IDisposable` and the `Dispose` Pattern (section 9.4). The `IAsyncDisposable`-specific content appears in section 9.4.4.

9.2.10 IAsyncEnumerable<T>

The `IAsyncEnumerable<T>` interface is primarily used for generating asynchronous iterators, also known as “async streams,” for use with the C# 8.0 `await foreach` statement.

- ✓ **DO** use the suffix “`Async`” for methods returning `IAsyncEnumerable<T>`.

Methods that return `IAsyncEnumerable<T>` follow the same naming conventions as methods returning Tasks with the Task-Based Async Pattern, including using the combined suffix “`AsyncCore`” when making use of the Template Method Pattern (section 9.9).

- ✓ **DO** add the `[EnumeratorCancellation]` attribute to the `CancellationToken` parameter of an `IAsyncEnumerable<T>` method using `yield return`.

The `[EnumeratorCancellation]` attribute indicates that the compiler should bind the `CancellationToken` value from the call to `GetAsyncEnumerator` to the parameter with the attribute, or combine the `CancellationToken` values from `GetAsyncEnumerator` and the token passed as a parameter to the `IAsyncEnumerable<T>` method. When the

attribute is not specified, the `CancellationToken` value from `GetAsyncEnumerator` is ignored for compiler-generated implementations.

```
// incorrect
public static async IAsyncEnumerable<int> ValueGenerator(
    int start,
    int count,
    CancellationToken cancellationToken = default) {

    int end = start + count;
    for (int i = start; i < end; i++) {
        await Task.Delay(i, cancellationToken).ConfigureAwait(false);
        yield return i;
    }
}

// correct
public static async IAsyncEnumerable<int> ValueGenerator(
    int start,
    int count,
    [EnumeratorCancellation]
    CancellationToken cancellationToken = default) {

    int end = start + count;
    for (int i = start; i < end; i++) {
        await Task.Delay(i, cancellationToken).ConfigureAwait(false);
        yield return i;
    }
}
```

X DO NOT use `IAsyncEnumerable<T>`, or any other type that implements or extends that interface, except as the return type of a `GetAsyncEnumerator` method.

X DO NOT implement both `IAsyncEnumerable<T>` and `IEnumerable<T>` on the same public type.

These guidelines are the asynchronous versions of the guidelines for `IEnumerable<T>` and `IEnumerable<T>` from section 8.3. See that section for more information.

9.2.6.5 Usage Guidelines for `await foreach`

✓ DO use the `WithCancellation` modifier when using `await foreach` on an `IAsyncEnumerable<T>` parameter for the purpose of using your `CancellationToken` with the enumerator.

The `IAsyncEnumerable<T>` interface accepts a `CancellationToken` in the call to `GetAsyncEnumerator`, but the `async foreach` statement doesn't have anywhere to directly accept a `CancellationToken` value to pass on your behalf. The `WithCancellation` extension method produces a value that wraps the `IAsyncEnumerable<T>` value and passes on the appropriate `CancellationToken` value.

In the first example that follows, the `MaxAsync` method correctly accepts a `CancellationToken` value, but it does not use the `CancellationToken` during the `async foreach`. The second example corrects this error.

```
// incorrect
public Task<int> MaxAsync(
    IAsyncEnumerable<int> source,
    CancellationToken cancellationToken = default) {

    int max = int.MinValue;
    bool hasValue = false;

    // cancellationToken isn't used
    async foreach (int value in source.ConfigureAwait(false)) {
        hasValue = true;
        if (value > max) {
            max = value;
        }
    }

    return hasValue ? max : throw new InvalidOperationException();
}

// correct
public Task<int> MaxAsync(
    IAsyncEnumerable<int> source,
    CancellationToken cancellationToken = default) {

    int max = int.MinValue;
    bool hasValue = false;

    // cancellationToken is passed into the enumerator
    async foreach (int value in source.WithCancellation(cancellationToken).ConfigureAwait(false)) {
        hasValue = true;
    }
}
```

```

    if (value > max) {
        max = value;
    }
}

return hasValue ? max : throw new InvalidOperationException();
}

```

When using `await foreach` directly on the result of a method that accepted a `CancellationToken`, there is no need, nor any value, in also calling `WithCancellation`. But there's also no significant penalty, other than line length.

```

// WithCancellation(cancellationToken) is not needed,
// because it was passed into ValueGenerator already.
async foreach (int value in
    ValueGenerator(10, 5, cancellationToken).ConfigureAwait(false)) {
    ...
}

```

- ✓ **DO** use the `ConfigureAwait` modifier when using `await foreach` in the same way that you would with `await`.

The `ConfigureAwait` extension method for `IAsyncEnumerable<T>` produces a value that effectively proxies the `ConfigureAwait` to each call to `MoveNextAsync()`. You should use it the same way, and in the same situations, as the `ConfigureAwait` modifier for Tasks (see section 9.2.5.2).

9.3 Dependency Properties

A dependency property (DP) is a regular .NET property that stores its value in a property store, instead of storing it in a field.

An attached dependency property is a kind of dependency property modeled as static Get and Set methods representing “properties” that describe relationships between objects and their containers (e.g., the position of a `Button` object on a `Panel` container).

This section describes when this might be useful and guidelines related to the design of such properties.

- ✓ **DO** provide properties as dependency properties if you need the properties to support WPF features such as styling, triggers, data binding, animations, dynamic resources, and inheritance.

In the following example, the `TextButton.Text` property is a dependency property supporting WPF style triggers.

```
<Style TargetType="TextButton">
    <Setter Property="Text" Value="Move here and click" />
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Text" Value="Now click" />
        </Trigger>
    </Style.Triggers>
</Style>
```

Here's an example using the property with data binding:

```
<TextButton Text="{Binding FirstName}" />
```

9.3.1 Dependency Property Design

The following guidelines describe details of dependency property design.

- ✓ **DO** inherit from `DependencyObject`, or one of its subtypes, when implementing dependency properties. The type provides a very efficient implementation of a property store and automatically supports WPF data binding.
- ✓ **DO** provide a regular CLR property and public static read-only field storing an instance of `System.Windows.DependencyProperty` for each dependency property.

```
public class TextButton : DependencyObject {

    public string Text {
        get { return (string)GetValue(TextProperty); }
        set { SetValue(TextProperty, value); }
    }

    public static readonly DependencyProperty TextProperty =
        DependencyProperty.Register(
```

```

    "Text",
    typeof(string),
    typeof(TextButton));
}

```

- ✓ **DO** implement dependency properties by calling instance methods `DependencyObject.GetValue` and `DependencyObject.SetValue`.

```

public class TextButton : DependencyObject {

    public string Text {
        get { return (string)GetValue(TextProperty); }
        set { SetValue(TextProperty, value); }
    }

    public static readonly DependencyProperty TextProperty = ...
}

```

- ✓ **DO** name the dependency property static field by suffixing the name of the property with “Property.”

The first parameter to the `DependencyProperty.Register` method should be the name of the wrapper property.

```

public class TextButton : DependencyObject {
    public static readonly DependencyProperty TextProperty =
        DependencyProperty.Register(
            "Text",
            typeof(string),
            typeof(TextButton));
}

```

- ✗ **DO NOT** set default values of dependency properties explicitly in code; instead, set them in metadata.

If you set a property default explicitly, you might prevent that property from being set by some implicit means, such as a styling.

```

public class TextButton : DependencyObject {

    public TextButton(){
        // do not set DP default values explicitly
        Text = String.Empty; // this is bad!
    }
}

```

```
public static readonly DependencyProperty TextProperty =
    DependencyProperty.Register(
        "Text",
        typeof(string),
        typeof(TextButton),
        new PropertyMetadata(String.Empty)); // this is good!
}
```

X DO NOT put code in the property accessors other than the standard code to access the static field.

That code won't execute if the property is set by implicit means, such as a styling, because styling uses the static field directly.

```
public string Text {
    get { return (string)GetValue(TextProperty); }
    set {
        SetValue(TextProperty, value);
        DoWorkOnTextChanged(); // this is bad!
    }
}
```

X DO NOT use dependency properties to store secure data. Even private dependency properties can be accessed publicly.

```
public class BadType : DependencyObject {
    // Do not do this! It's not secure!
    private static readonly DependencyProperty SecretProperty =
        DependencyProperty.Register(
            "Secret",
            typeof(string),
            typeof(BadType));

    public BadType() {
        SetValue(SecretProperty, "password");
    }

    private static void Main() {
        var b = new BadType();
        var enumerator = b.GetLocalValueEnumerator();
        while (enumerator.MoveNext()) {
            Console.WriteLine("{0}={1}",
                enumerator.Current.Property,
                enumerator.Current.Value);
        }
    }
}
```

9.3.2 Attached Dependency Property Design

The dependency properties described in the preceding section represent intrinsic properties of the declaring type; for example, the `Text` property is a property of `TextButton`, which declares it. One special kind of dependency property is the attached dependency property. The dependency property guidelines apply to attached dependency properties as well, but there are some additional guidelines to consider when implementing attached dependency properties. This section describes these special considerations.

Attached dependency properties represent properties that are defined on one type but that can be set on an object of another type.

 **CHRISTOPHE NASARRE** This is close to what you would achieve with a `PropertyExtender` in Windows Forms.

A classic example of an attached property is the `Grid.Column` property. The property represents `Button`'s (not `Grid`'s) column position, but it is only relevant if the `Button` is contained in a `Grid`, so it's "attached" to `Buttons` by `Grids`.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Button Grid.Column="0">Click</Button>
  <Button Grid.Column="1">Clack</Button>
</Grid>
```

The definition of an attached property looks mostly like that of a regular dependency property, except that the accessors are represented by static `Get` and `Set` methods:

```
public class Grid {

  public static int GetColumn(DependencyObject obj) {
    return (int)obj.GetValue(ColumnProperty);
  }
```

```

public static void SetColumn(DependencyObject obj, int value) {
    obj.SetValue(ColumnProperty, value);
}

public static readonly DependencyProperty ColumnProperty =
    DependencyProperty.RegisterAttached(
        "Column",
        typeof(int),
        typeof(Grid));
}

```

9.3.3 Dependency Property Validation

Properties often implement validation. Validation logic executes when an attempt is made to change the value of a property. The following example shows the typical validation code ensuring that a property is not ever set to null.

```

public string Text {
    get { ... }
    set {
        if (value == null) {
            throw new ArgumentNullException(nameof(value));
        }
        ...
    }
}

```

Unfortunately, dependency property accessors cannot contain arbitrary validation code. Instead, dependency property validation logic needs to be specified during property registration.

X DO NOT put dependency property validation logic in the property's accessors. Instead, pass a validation callback to `DependencyProperty.Register` method.

```

public static readonly DependencyProperty TextProperty =
    DependencyProperty.Register(
        "Text",
        typeof(string),
        typeof(TextButton),
        new PropertyMetadata(string.Empty),
        value => value != null); // validation

```

9.3.4 Dependency Property Change Notifications

Similar to the validation logic described in the previous section, dependency property accessors should not have custom code that would be normally used to trigger property change notification events, because these accessors are not called from XAML-generated code.

```
public string Text {
    set {
        if(_text != value) {
            _text = value;
        }
        OnTextChanged(); // bad
    }
}
```

X DO NOT implement change notification logic in dependency property accessors. Dependency properties have a built-in change notifications feature that is used by supplying a change notification callback to the `PropertyMetadata`.

```
public static readonly DependencyProperty TextProperty =
DependencyProperty.Register(
    "Text",
    typeof(string),
    typeof(TextButton),
    new PropertyMetadata(
        string.Empty,
        (obj, args) => {
            // property changed
            ...
        }));
}
```

9.3.5 Dependency Property Value Coercion

Property coercion takes place when the value given to a property setter is modified by the setter before the property store is actually modified. The following example shows simple coercion logic.

```
public string Text {
    set {
        if (value == null) {
```

```

        value = string.Empty;
    }
    _text = value;
}
}

```

Similar to the validation and change notification logic described in the previous sections, dependency properties must specify coercion logic during property registration, not directly in the property accessor.

X DO NOT implement coercion logic in dependency property accessors. Dependency properties have a built-in coercion feature, and it can be used by supplying a coercion callback to the `PropertyMetadata`.

```

public static readonly DependencyProperty TextProperty =
DependencyProperty.Register(
    "Text",
    typeof(string),
    typeof(TextButton),
    new PropertyMetadata(
        string.Empty,
        (obj, args) => {
            // change notification callback
        },
        (obj, value) => value ?? string.Empty)); // coercion

```

9.4 Dispose Pattern

All programs acquire one or more system resources, such as memory, system handles, or database connections, during the course of their execution. Developers must be careful when using such system resources, because they must be released after they have been acquired and used.

The CLR provides support for automatic memory management. Managed memory (memory allocated using the C# operator `new`) does not need to be explicitly released; instead, it is released automatically by the Garbage Collector (GC). This frees developers from the tedious and difficult task of releasing memory and is one of the main reasons for the unprecedented productivity afforded by .NET.

Unfortunately, managed memory is just one of many types of system resources. Resources other than managed memory still need to be released explicitly; thus, they are referred to as unmanaged resources. The GC was specifically not designed to manage such unmanaged resources, which means that the responsibility for managing unmanaged resources lies in the hands of the developers.

The CLR provides some help for releasing unmanaged resources. `System.Object` declares a virtual method `Finalize` (also called the finalizer) that is called by the GC before the object's memory is reclaimed by the GC; this method can be overridden to release unmanaged resources. Types that override the finalizer are referred to as finalizable types.

Although finalizers are effective in some cleanup scenarios, they have two significant drawbacks:

- The finalizer is called when the GC detects that an object is eligible for collection. This happens at some undetermined period of time after the resource is no longer needed. The delay between when the developer could or would like to release the resource and the time when the resource is actually released by the finalizer might be unacceptable in programs that acquire many scarce resources (resources that can be easily exhausted) or in cases in which resources are costly to keep in use (e.g., large unmanaged memory buffers).
- When the CLR needs to call a finalizer, it must postpone collection of the object's memory until the next round of garbage collection (the finalizers run between collections). As a result, the object's memory (and all objects it refers to) will not be released for a longer period of time.

For these reasons, relying exclusively on finalizers might not be appropriate in many scenarios when it is important to reclaim unmanaged resources as quickly as possible, when dealing with scarce resources, or in highly performant scenarios in which the added GC overhead of finalization is unacceptable.

.NET provides the `System.IDisposable` interface that should be implemented to provide the developer with a manual way to release unmanaged resources as soon as they are no longer needed. It also provides the `GC.SuppressFinalize` method, which can tell the GC that an object was manually disposed of and no longer needs to be finalized, in which case the object's memory can be reclaimed earlier. Types that implement the `IDisposable` interface are referred to as disposable types.

■ **BRIAN PEPIN** The idea behind `Dispose` is that you call it to release scarce or unmanaged resources. We've designed it so if you don't call `Dispose`, the object will finalize and release the resource anyway. It's great that we'll clean up eventually for you, but in reality you probably have a bug if you're not disposing of an object when you're done with it. The exception here is when you're sharing an object with other code. For example, Windows Forms controls that have an `ImageList` property never dispose of the `ImageList`, because they don't know if it is being used by other controls.

■ **JOE DUFFY** The advice I would like to give people is roughly as follows. When a type implements `IDisposable`, and ownership is obvious, you should do your best to call `Dispose` when you are done with the object. But if ownership becomes tricky (because the object is referenced from multiple places or shared across threads, for example), neglecting to call `Dispose` will do no harm. This is along the lines of what Brian says above. Unfortunately, there are a few instances in .NET where failure to call `Dispose` can lead to surprising behavior. `FileStream`, for example, will keep the file handle open until finalization time. If that handle was opened in write-exclusive mode, nobody else on the machine will be able to open the file until the finalizer runs. Moreover, the contents written won't be flushed to disk until then. This is a nondeterministic event, and it is terrible to rely on it for correctness.

The `Dispose` Pattern is intended to standardize the usage and implementation of finalizers and the `IDisposable` interface. The main motivation for this pattern is to reduce the complexity of the implementation of

the `Finalize` and `Dispose` methods. The complexity stems from the fact that the methods share some, but not all, code paths (the differences are described later in the chapter). In addition, there are historical reasons for some elements of the pattern related to the evolution of language support for deterministic resource management.

- ✓ **DO** implement the Basic Dispose Pattern on types containing instances of disposable types. See section 9.4.1 for details on the basic pattern.

If a type is responsible for the lifetime of other disposable objects, developers need a way to dispose of them, too. Using the container's `Dispose` method is a convenient way to make this possible.

- ✓ **CONSIDER** implementing the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects, but are likely to have subtypes that do.

A great example of this is the `System.IO.Stream` class. Although it is an abstract base class that doesn't hold any resources, most of its subclasses do—and because of this, it implements this pattern.

9.4.1 Basic Dispose Pattern

The basic implementation of the pattern involves implementing the `System.IDisposable` interface and declaring a `Dispose(bool)` method that implements all resource cleanup logic to be shared between the `Dispose` method and the optional finalizer. Note that this section does not discuss providing a finalizer: Finalizers are extensions to this basic pattern and are discussed in section 9.4.2.

The following example shows a simple implementation of the basic pattern:

```
public class DisposableResourceHolder : IDisposable {
    private SafeHandle _resource; // handle to a resource

    public DisposableResourceHolder() {
        _resource = ... // allocates the resource
    }

    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing) {
        if (disposing) {
            if (_resource != null) {
                _resource.Close();
                _resource = null;
            }
        }
    }
}
```

```
public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing) {
    if (disposing) {
        _resource?.Dispose();
    }
}
```

■ **HERB SUTTER** If using C++, simply write the usual destructor (`~T()`), and the compiler will automatically generate all of the machinery described later in this section. In the rare cases in which you do want to write a finalizer (`!T()`) as well, the recommended way to share code is to put as much of the work into the finalizer as the finalizer is able to handle (e.g., the finalizer cannot reliably touch other objects, so don't put code in there that needs to use other objects), put the rest in the destructor, and have your destructor call your finalizer explicitly.

The Boolean parameter `disposing` indicates whether the method was invoked from the `IDisposable.Dispose` implementation or from the finalizer. The `Dispose(bool)` implementation should check the parameter before accessing other reference objects (e.g., the `resource` field in the preceding sample). Such objects should only be accessed when the method is called from the `IDisposable.Dispose` implementation (when the `disposing` parameter is equal to true). If the method is invoked from the finalizer (`disposing` is false), other objects should not be accessed. The reason is that objects are finalized in an unpredictable order, so they—or any of their dependencies—might already have been finalized.

Also, this section applies to classes with a base that does not already implement the Dispose Pattern. If you are inheriting from a class that already implements the pattern, simply override the `Dispose(bool)` method to provide additional resource cleanup logic.

- ✓ **DO** declare a protected virtual void `Dispose(bool disposing)` method to centralize all logic related to releasing managed and unmanaged resources.

All resource cleanup should occur in this method. The method is called from both the finalizer and the `IDisposable.Dispose` method. The parameter will be false if being invoked from inside a finalizer or `DisposeAsync()`. It should be used to ensure that any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in section 9.4.2.

```
protected virtual void Dispose(bool disposing) {
    if (disposing) {
        _resource?.Dispose();
    }
}
```

■ **JEFFREY RICHTER** The idea here is that `Dispose(bool)` knows whether it is being called to do explicit cleanup (the Boolean is true) versus being called due to a garbage collection (the Boolean is false). This distinction is useful because, when being disposed explicitly, the `Dispose(bool)` method can safely execute code using reference type fields that refer to other objects, knowing for sure that these other objects have not been finalized. When the Boolean is false, the `Dispose(Boolean)` method should not execute code that refers to reference type fields, because those objects might have already been finalized.

■ **JOE DUFFY** Jeff's comment might seem incorrect at first glance—that is, can't you safely access reference type objects that aren't finalizable? The answer is yes, if and only if you are certain that the object doesn't rely on the finalizable state itself! This is a nontrivial thing to figure out and is subject to change from release to release. Unless you're 100 percent certain, perhaps because you own the type in question, just avoid doing it.

- ✓ **DO** implement the `IDisposable` interface by simply calling `Dispose(true)` followed by `GC.SuppressFinalize(this)`.

The call to `SuppressFinalize` should only occur if `Dispose(true)` executes successfully.

```
public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
}
```

■ **BRAD ABRAMS** We had a fair amount of debate about the relative ordering of calls in the `Dispose()` method. We opted for the ordering where `SuppressFinalize` is called after `Dispose(bool)`. It ensures that `GC.SuppressFinalize()` only gets called if the `Dispose` operation completes successfully.

■ **JEFFREY RICHTER** I, too, wrestled back and forth with the order of these calls. Originally, I felt that `SuppressFinalize` should be called prior to `Dispose`. My thinking was this: If `Dispose` throws an exception then, it will throw the same exception when `Finalize` is called and there is no benefit to this, and the second exception should be prevented. However, I have since changed my mind, and I now agree with this guideline that `SuppressFinalize` should be called after `Finalize`. The reason is that `Dispose()` calls `Dispose(true)`, which might throw, but when `Finalize` is called later, `Dispose(false)` is called. This might be a different code path than before, and it would be good if this different code path executed. In addition, the different code path might not throw the exception.

X DO NOT make the parameterless `Dispose` method virtual.

The `Dispose(bool)` method is the one that should be overridden by subclasses.

```
// bad design
public class DisposableResourceHolder : IDisposable {
    public virtual void Dispose() { ... }
    protected virtual void Dispose(bool disposing) { ... }
}
```

```
// good design
public class DisposableResourceHolder : IDisposable {
    public void Dispose() { ... }
    protected virtual void Dispose(bool disposing) { ... }
}
```

 **BRIAN PEPIN** If you look hard enough, there are still places in the framework where we don't follow this pattern. By the time we finalized the `Dispose` pattern, quite a bit of the framework had already been written. Although we scrubbed everything to the best of our ability, a few things still slipped through the cracks.

 **DO NOT** declare any overloads of the `Dispose` method other than `Dispose()` and `Dispose(bool)`.

`Dispose` should be considered a reserved word to help codify this pattern and prevent confusion among implementers, users, and compilers. Some languages might choose to automatically implement this pattern on certain types.

 **DO** allow the `Dispose(bool)` method to be called more than once. The method might choose to do nothing after the first call.

```
public class DisposableResourceHolder : IDisposable {

    bool _disposed = false;

    protected virtual void Dispose(bool disposing) {
        if (_disposed) {
            return;
        }
        // cleanup
        ...
        disposed = true;
    }
}
```

 **AVOID** throwing an exception from within `Dispose(bool)` except under critical situations where the containing process has been corrupted (e.g., leaks, inconsistent shared state).

Users expect that a call to `Dispose` will not raise an exception. For example, consider the manual try-finally statement in this snippet:

```
TextReader tr = new StreamReader(File.OpenRead("foo.txt"));
try {
    // do some stuff
}
finally {
    tr.Dispose();
    // more stuff
}
```

If `Dispose` throws an exception, further finally-block cleanup logic will not execute. To work around this, the user would need to wrap every call to `Dispose` (within the finally block!) in another try block, which leads to very complex cleanup handlers. If executing a `Dispose(bool disposing)` method, never throw an exception if `disposing` is false: Doing so will terminate the process if executing inside a finalizer context!

- ✓ **DO** throw an `ObjectDisposedException` from any member that cannot be used after the object has been disposed of.

```
public class DisposableResourceHolder : IDisposable {
    bool _disposed = false;
    SafeHandle _resource; // handle to a resource

    public void DoSomething() {
        if (_disposed) {
            throw new ObjectDisposedException(...);
        }
        // now call some native methods using the resource
        ...
    }

    protected virtual void Dispose(bool disposing) {
        if (_disposed) {
            return;
        }
        // cleanup
        ...
        disposed = true;
    }
}
```

✗ **AVOID** allowing an object to regain meaningful state after a call to `Dispose()`.

Object “rehydration” often creates hard-to-diagnose performance problems, and is an easy bug to create when the important-to-dispose portion of an object is created lazily. When an object is disposed, it should generally stay disposed. However, if the type has a method that clearly conveys new resources are being acquired, such as “Open,” it may make sense to allow the object to stop throwing `ObjectDisposedException` and resume functioning.

```
public partial class ImplicitRehydration : IDisposable {
    private SafeHandle _resource;
    private string _resourceId;

    public ImplicitRehydration(string resourceId) {
        _resourceId = resourceId;
    }

    public int Size {
        get {
            // Because Dispose sets _resource back to null,
            // and no other disposed state is tracked,
            // reading this property accidentally reopens the resource
            return GetSize(EnsureResource());
        }
    }

    private SafeHandle EnsureResource() {
        if (_resource == null) {
            _resource = ...;
        }
        return _resource;
    }

    protected virtual void Dispose(bool disposing) {
        if (disposing) {
            _resource?.Dispose();
            _resource = null;
        }
    }
}
```

Rehydration is particularly problematic on unsealed types, because a derived type may have had a finalizer that is now suppressed. When rehydrating a finalizable object, be sure to call `GC.ReRegisterForFinalize(this)`, so as to undo the `GC.SuppressFinalize(this)` from `Dispose()`.

```
public partial class ExplicitRehydration : IDisposable {
    private IntPtr _resource;

    ~ExplicitRehydration() {
        Dispose(false);
    }

    public ExplicitRehydration(...) {
        _resource = ...;
    }

    public int Size {
        get {
            if (_resource == IntPtr.Zero) {
                throw new ObjectDisposedException(...);
            }
            return GetSize(_resource);
        }
    }

    public void Open(...) {
        ...
        GC.ReRegisterForFinalize(this);
    }
}
```

■ **JEREMY BARTON** Object rehydration on an unsealed type is really something that you have to design for, following the extensibility principles from Chapter 6. Your `Dispose` method cleared the finalization-required marker of the derived type needed. If you rehydrate, how does the derived type know? Without some sort of event, the derived type can easily be in a corrupt state. The easiest fix? Don't have your public types be rehydratable. Internal types have free rein, though.

✓ **CONSIDER** providing a `Close()` method, in addition to the `Dispose()` method, if “close” is standard terminology in the area.

When doing so, it is important that you make the `Close` implementation identical to `Dispose` and consider implementing the `IDisposable`. `Dispose` method explicitly. See section 5.1.2, on implementing interfaces explicitly.

```
public class Stream : IDisposable {
    IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

■ JEFF PROSISE Implementing `Close` and `Dispose` so that they're semantically equivalent helps avoid a lot of confusion. More than once, I've had developers tell me that you shouldn't instantiate a `SqlConnection` in a `using` statement because the resultant code will call `Dispose` but not `Close`. In reality, it's fine to combine `using` and `SqlConnection` because `SqlConnection.Dispose` calls `SqlConnection.Close`.

9.4.2 Finalizable Types

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the `Dispose(bool)` method.

Finalizers are notoriously difficult to implement correctly, primarily because you cannot make certain (normally valid) assumptions about the state of the system during their execution. The following guidelines should be taken into careful consideration.

Note that some of the guidelines apply not just to the `Finalize` method, but to any code called from a finalizer. In the case of the Basic Dispose Pattern, this means logic that executes inside `Dispose(bool disposing)` when the `disposing` parameter is `false`.

If the base class already is finalizable and implements the Basic Dispose Pattern, you should not override `Finalize` again. You should instead just

override the `Dispose(bool)` method to provide additional resource cleanup logic.

■ **HERB SUTTER** You really don't want to write a finalizer if you can help it. Besides the problems noted earlier in this chapter, writing a finalizer on a type makes that type more expensive to use even if the finalizer is never called. For example, allocating a finalizable object is more expensive because it must also be put on a list of finalizable objects. This cost can't be avoided, even if the object immediately suppresses finalization during its construction (as when creating a managed object semantically on the stack in C++).

The following code shows an example of a finalizable type:

```
internal class ComplexResourceHolder : IDisposable {  
  
    private IntPtr _buffer; // unmanaged memory buffer  
    private SafeHandle _resource; // disposable handle to a resource  
  
    public ComplexResourceHolder() {  
        _buffer = ... // allocates memory  
        _resource = ... // allocates the resource  
    }  
  
    protected virtual void Dispose(bool disposing) {  
        if (_buffer != IntPtr.Zero) {  
            ReleaseBuffer(_buffer); // release unmanaged memory  
            _buffer = IntPtr.Zero;  
        }  
        if (disposing) { // release other disposable objects  
            _resource?.Dispose();  
        }  
    }  
  
    ~ComplexResourceHolder() {  
        Dispose(false);  
    }  
  
    public void Dispose() {  
        Dispose(true);  
        GC.SuppressFinalize(this);  
    }  
}
```

X DO NOT make public types finalizable.

Public types that hold finalizable resources should instead make an internal, or private nested, type to be a finalizable resource holder. Prefer using existing resource wrappers, such as SafeHandle, to encapsulate unmanaged resources where possible.

```
// wrong
public partial class NetworkCard : IDisposable {
    private IntPtr _deviceHandle;

    ~NetworkCard() {
        Dispose(false);
    }

    public bool Dispose() => Dispose(true);

    protected bool Dispose(bool disposing) {
        if (_deviceHandle != IntPtr.Zero) {
            Interop.CloseHandle(_deviceHandle);
            _deviceHandle = IntPtr.Zero;
        }
    }
}

// right
public partial class NetworkCard : IDisposable {
    private SafeDeviceHandle _deviceHandle;

    public bool Dispose() => Dispose(true);

    protected bool Dispose(bool disposing) {
        if (disposing) {
            _deviceHandle?.Dispose();
        }
    }
}
internal sealed class SafeDeviceHandle :
    SaveHandleZeroOrMinusOneIsInvalid {

    internal SafeDeviceHandle()
        : base(ownsHandle: true)
    {
    }
}
```

```
protected override bool ReleaseHandle() {
    return Interop.CloseHandle(_handle);
}
```

- ✓ **DO** implement the Basic Dispose Pattern on every finalizable type. See section 9.4.1 for details on the basic pattern.

This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

Consistent use of the Basic Dispose Pattern for internal finalizable types allows new developers of your library to transfer their skills between projects, and to better interact with the few public finalizable types in .NET.

- ✗ **DO NOT** access any finalizable objects in the finalizer code path, because there is significant risk that they will have already been finalized.

For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A's finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if `Environment.HasShutdownStarted` returns true.

■ **JAN KOTAS** Finalization of all objects during shutdown was a source of intractable reliability issues in .NET Framework. The forceful finalization of all objects during process exit was removed in .NET Core. `AppDomain.ProcessExit` event can still be used to perform explicit cleanup during shutdown.

■ **JEFFREY RICHTER** Note that it is OK to touch unboxed value type fields.

X DO NOT let exceptions escape from the finalizer logic, except for system-critical failures.

If an exception is thrown from a finalizer, the CLR will shut down the entire process, preventing other finalizers from executing and resources from being released in a controlled manner.

X DO NOT remove a finalizer from an unsealed public type once it has one.

Guidance from previous editions of this book suggested not redefining the finalizer when a base type was known to already implement the finalizer via the Basic Dispose Pattern. Removing the finalizer once a type has taken that dependency causes the finalizer to not run on that type, which can lead to difficult-to-diagnose resource leaks.

Since sealed types cannot have inheritors, removing a finalizer can't have such drastic consequences. For internal types, you are empowered to check on any derived types and move the finalizer as appropriate.

9.4.3 Scoped Operations

The majority of operations defined in .NET are one-off operations such as computing a single value, batch-processing the contents of a list, and setting a single property. A few operations, however, are inherently scoped: obtain a lock and release a lock, begin logging this request and this request has completed, indent a writer scope and resume previous indentation. The `IDisposable` interface and C# `using` statement permit for alignment of logical scopes and method scopes as well as simplifying the caller experience by eliminating the need to determine the appropriate "end" method to pair with the "begin" call.

✓ CONSIDER returning a disposable value instead of making callers manually pair "begin" and "end" methods.

The `IndentedTextWriter` class permits increasing and decreasing the indentation level by setting a property. A very common pattern is to increase indentation, do some work, and then restore the indentation.

```
public void WriteMethod(
    MethodData method,
    IndentedTextWriter writer) {

    WriteMethodDeclaration(method, writer);
    if (!method.IsAbstract) {
        writer.Indent += 4;
        WriteStatements(method.Statements, writer);
        writer.Indent -= 4;
        writer.WriteLine('}');
    }
}
```

If `IndentedTextWriter` had a method returning a disposable instance, such as `OpenScope`, then the caller code could more closely map lexical code scopes and logical operation scopes.

```
public void WriteMethod(
    MethodData method,
    IndentedTextWriter writer) {

    WriteMethodDeclaration(method, writer);
    if (!method.IsAbstract) {
        using (writer.OpenScope(4)) {
            WriteStatements(method.Statements, writer);
        }
        writer.WriteLine('}');
    }
}
```

When making a disposable value for the scoped operation, it is important to still follow the guidance for `Dispose` methods for resource cleanup—for example, being safe to invoke these methods multiple times. One potential implementation for `OpenScope` is provided here as an example:

```
public partial class IndentedTextWriter {
    public IndentationScope OpenScope(int indentAmount) {
        if (indentAmount < 0) {
            throw new ArgumentOutOfRangeException(nameof(indentAmount));
        }
        Indent += indentAmount;
        return new IndentationScope(this, Indent, indentAmount);
    }
}
```

```

public readonly struct IndentationScope : IDisposable {
    private IndentedTextWriter _writer;
    private int _pushedIndent;
    private int _indentAmount;

    internal IndentationScope(...) { ... }

    public void Dispose() => Dispose(true);

    protected void Dispose(bool disposing) {
        // default(IndentationScope)
        if (_writer == null) {
            return;
        }
        if (_writer.Indent == _pushedIndent) {
            _writer.Indent -= _indentAmount;
        }
        if (_writer.Indent > _pushedAmount) {
            throw new InvalidOperationException(...);
        }
    }
}
}

```

This implementation of `IndentationScope` uses a value type with no public members, other than `Dispose`, to track the disposable scope. It can use a nested type because the vast majority of callers are expected to put the `OpenScope` call in a `using` statement and not save an explicit local. This particular implementation throws an exception if the indentation is deeper than it should have been, but ignores the state when indentation is less than expected—which avoids an exception when the second of two `Dispose()` calls occurs. Other implementations, with different consequences, are also possible.

✓ CONSIDER returning a disposable value instead of making callers manually pair “begin” and “end” methods when the “end” method must be in a `finally` block for program correctness.

In the previous `IndentedTextWriter` example, neglecting to call the “end” operation (reducing the `Indent` property back to the previous value) simply causes a functional bug. For some operations, like locking, failing to call the “end” method can result in deadlock or other difficult program states.

In the following example, the developer did not consider the possibility that if the code between lock acquisition and lock release throws an exception, the lock is never released.

```
public void DoStuff () {
    _lock.EnterReadLock();
    DoStuffCore();
    // If DoStuffCore throws, we never get here...
    _lock.ExitReadLock();
}
```

The C# using statement, and equivalent statements in other .NET languages, calls the `Dispose()` method in a finally block. If we assume that the average caller uses a using statement instead of—or in addition to—manually calling `Dispose()`, then using a disposable return value for the operation starts the callers off on the more reliable path of releasing the lock in a finally block.

```
public void DoStuff () {
    // The expanded version of this, in IL, is to
    // call the Dispose method in a finally block.
    using (_lock.GetReadLock()) {
        DoStuffCore();
    }
}
```

The following example implementation of a disposable-valued `GetReadLock` uses a reference type to implement the `IDisposable` operation type, and simplifies the caller understanding by declaring the return type as `IDisposable`.

```
public partial class ReaderWriterLockSlim {
    public IDisposable GetReadLock() {
        HeldLock lock = new HeldLock();
        lock.EnterReadLock(this);
        return lock;
    }

    private sealed class HeldLock : IDisposable {
        private ReaderWriterLockSlim _lock;

        internal void EnterReadLock(ReaderWriterLockSlim lock) {
            _lock = lock;
        }
    }
}
```

```
    lock.EnterReadLock();
}

public void Dispose() {
    ReaderWriterLockSlim lock =
        Interlocked.Exchange(ref _lock, null);

    lock?.ExitReadLock();
}
}
```

■ **JEREMY BARTON** Since locking is a common, low-level operation, the disposable value would be better if it was a public value type, for the same reasons as described in the `IndentedTextWriter` example. Making a functionally correct implementation that doesn't involve a mutable value type or per-operation allocation—but guarantees execution will occur exactly once—is difficult, but possible. It wouldn't do any better as an example to illustrate the point, and bug-fixing print books is hard, so the example here was kept simple, even though we'd never actually write it this way in the BCL.

9.4.4 IAsyncDisposable

The `IAsyncDisposable` interface permits the same resource cleanup and operation termination as `IDisposable`, but allows for the implementation to use asynchronous methods when the work from `Dispose()` involves I/O or other blocking operations.

- ✓ **DO** follow the guidance for both asynchronous design (section 9.2) and synchronous Dispose (section 9.4), except as otherwise noted, when implementing `IAsyncDisposable`.

`IAsyncDisposable` inherits the guidance from both the “`Async`” and the “`Disposable`” portions of its name. For example, the method should be safe to call multiple times (section 9.4), should use `ConfigureAwait(false)` unless designed for an app model that requires using the ambient task scheduler (section 9.2.5.2), and should result in an `ObjectDisposedException` for many members on the type (section 9.4.1).

X DO NOT declare any overloads of `DisposeAsync()`.

Synchronous `Dispose` recommends the Basic `Dispose` Pattern (section 9.4.1), deferring `Dispose()` to `Dispose(disposing: true)`, so as to allow the unification of cleanup code between `Dispose()` and any potential finalizer on the type. Since a finalizer is always executed in a synchronous context, it should only ever call the synchronous `Dispose` variant.

✓ DO implement `DisposeAsync` as only calling and awaiting `DisposeAsyncCore()`, calling `Dispose(false)`, and calling `GC.SuppressFinalize(this)`, in that order—except on sealed types.

In the Basic `Dispose` Pattern (section 9.4.1), the only code unique to the zero-argument `Dispose()` method is the call to `GC.SuppressFinalize(this)`. Providing virtual `DisposeAsync` via the Template Method Pattern (section 9.9) enables you to ensure that `DisposeAsync` performs as an asynchronous equivalent to `Dispose` when it comes to finalizable types, keeping `Dispose` and `DisposeAsync` as synchronous and asynchronous versions of the same functionality, respectively. The call to `Dispose(false)` ensures that any finalizer-specific code that was intermixed in the `Dispose` implementation of a base class still gets invoked. In most type hierarchies, it will have no effect, but it's necessary to complete the functional equivalence from synchronous `Dispose`.

```
public partial class SomeType : IDisposable, IAsyncDisposable {
    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing) { ... }

    public async ValueTask DisposeAsync() {
        await DisposeAsyncCore();
        Dispose(false);
        GC.SuppressFinalize(this);
    }

    protected virtual ValueTask DisposeAsyncCore() { ... }
}
```

9.4.4.1 Usage Guidelines for await using

- ✓ **DO** use the ConfigureAwait modifier when using await using in the same way that you would with await.

The ConfigureAwait extension method for IAsyncDisposable produces a value that proxies the ConfigureAwait modifier to DisposeAsync(). You should use it the same way, and in the same situations, as the ConfigureAwait modifier for Tasks (see section 9.2.5.2).

Invoking ConfigureAwait means that you can't use the variable assignment syntax of await using. Instead, the value has to be saved to a local variable first.

```
// (possibly) incorrect
await using (var value = new SomeType()) {
    ...
}

// (probably) correct
// Extra braces were added to preserve the restricted scope of
// the "value" variable
{
    var value = new SomeType();
    using (value.ConfigureAwait(false)) {
        ..
    }
}
```

Be aware that if an exception is thrown between the local variable assignment and the await using statement, the DisposeAsync method will not be invoked. As of C# 8.0, there is no easy way to get the statement-by-statement safety of “stacked usings” while also invoking the ConfigureAwait modifier. The blockless “simplified using” syntax comes closest to this goal.

```
// (possibly) incorrect: no use of ConfigureAwait
await using (var value1 = new SomeType())
// If this statement throws, value1 still has DisposeAsync called
await using (var value2 = ValueFactory.BuildSomeOtherType()) {
    ...
}
```

```
// Uses ConfigureAwait, but does not call value1.DisposeAsync
// when ValueFactory.BuildSomeOtherType() throws
{
    var value1 = new SomeType();
    var value2 = ValueFactory.BuildSomeOtherType();
    using (value.ConfigureAwait(false))
    using (value.ConfigureAwait(false)) {
        ..
    }
}

// Fixes the problem with not calling value1.DisposeAsync, but may
// violate your project's style guidelines
{
    var value1 = new SomeType();
    await using var ignored = value1.ConfigureAwait(false);
    var value2 = ValueFactory.BuildSomeOtherType();
    await using (value2.ConfigureAwait(false)) {
        ..
    }
}
```

9.5 Factories

The most common and consistent way to create an instance of a type is via its constructor. However, sometimes a preferable alternative is to use a factory.

A factory is an operation or collection of operations that abstract the object creation process for the users, allowing for specialized semantics and finer granularity of control over an object's instantiation. Simply put, a factory's primary purpose is to generate and provide instances of objects to callers.

There are two main groups of factories: factory methods and factory types (also called abstract factories).

`File.Open` and `Activator.CreateInstance` are examples of factory methods.

```
public class File {
    public static FileStream Open(String path, FileMode mode) { ... }
```

```
public static class Activator {  
    public static object CreateInstance(Type type){ ... }  
}
```

Factory methods often appear on the types for which instances are to be created and are typically static. Such static factory methods are often limited to creating instances of a specific type determined at the time of compilation (this is true of constructors as well). This behavior is sufficient in most scenarios, but sometimes it is necessary to return a dynamically selected subclass.

Factory types can address these scenarios. These special-purpose types have factory methods implemented as virtual (usually abstract) instance functions.

For example, consider the following scenario, in which factory types inherited from `StreamFactory` can be used to dynamically select the actual type of the `Stream`:

```
public abstract class StreamFactory {  
    public abstract Stream CreateStream();  
}  
  
public class FileStreamFactory: StreamFactory {  
    ...  
}  
  
public class IsolatedStorageStreamFactory: StreamFactory {  
    ...  
}
```

✓ **DO** prefer constructors to factories, because they are generally more usable, consistent, and convenient than specialized construction mechanisms.

Factories sacrifice discoverability, usability, and consistency for implementation flexibility. For example, IntelliSense will guide a user through the instantiation of a new object using its constructors, but it won't point users in the direction of factory methods.

■ **KRZYSZTOF CWALINA** I often hear criticism that we take tool support into account when making API design decisions. To answer this, I have to say that I strongly believe that a modern framework is more than just a piece of stand-alone reusable code. It is a part of a large ecosystem of run-times, languages, documentation packages, support networks, and, finally, tools. All parts of the ecosystem must influence each other to provide an optimal solution. A modern framework designed outside of its ecosystem loses its competitive potential.

✓ **CONSIDER** using a factory if you need more control than can be provided by constructors over the creation of the instances.

For example, consider the Singleton, Builder, or other similar patterns that constrain the ways in which objects are created. A constructor is very limited in its ability to enforce rich patterns such as these, whereas a factory method can easily perform caching, throttling, and sharing of objects, for example.

■ **ANDERS HEJLSBERG** The advantage of the factory pattern is that `SomeClass.GetReader` can return an object whose runtime type is derived from `SomeReader`.

```
SomeClass c = new SomeClass(...)  
SomeReader r = c.GetReader(...);
```

The choice of an actual runtime type can be based on state in the `SomeClass` instance as well as on arguments passed to `GetReader`. Say, for example, that a `SomeClass` can represent a local file or a URL. `GetReader` could then return instances of either `FileReader` or `UrlReader`, both of which are derived from `SomeReader`. The folks building the managed `XmlReader` and `XmlWriter` switched from constructors in .NET Framework 1.1 to a factory pattern in .NET Framework 2.0 for exactly this reason. Another advantage of the factory pattern is that `GetReader` isn't required to return a fresh instance—it could cache objects and return previously allocated instances.

The advantages of the constructor pattern are simplicity and the extensibility argument given previously. Most users find the constructor pattern simpler and more discoverable because they are used to objects being created by constructors. The disadvantage is that you can't dynamically decide the runtime type of what you return, nor can you return a previously allocated instance. If you are confident you will never need these capabilities, then constructors are probably the better choice. Simplicity is always a good thing.

In short: Factory patterns give you more degrees of freedom, but constructors are simpler for the user.

Now, pick your poison ;-)

■ STEPHEN TOUB Factories have both advantages and disadvantages when it comes to performance. In terms of advantages, factories enable developers to create custom implementations tailored to specific scenarios; for example, the `System.Threading.Channels` library uses a factory method to create a channel instance that is optimized based on various constraints the caller specifies, and a future implementation of the library could specialize even more combinations with additional concrete implementations. In terms of disadvantages, it can be very difficult for a tool like a tree shaker/linker to determine exactly which code paths inside of a factory are reachable, and thus which concrete types are actually used by an app; this can lead to larger app sizes when such a tool is being used to minimize code size by removing all unused code paths.

✓ **DO** use a factory in cases where a developer might not know which type to construct, such as when coding against a base type or interface.

A factory can often use parameters and other context-based information to make this decision for the user.

```
public class Type {
    // This factory returns instances of various types including
    // PropertyInfo, ConstructorInfo, MethodInfo, etc.
    public MemberInfo[] GetMember(string name);
}
```

- ✓ **CONSIDER** using a factory if having a named method is the only way to make the operation self-explanatory.

Constructors cannot have names, and sometimes using a constructor lacks sufficient context to inform a developer of an operation's semantics. For example, consider this statement:

```
public String(char c, int count);
```

This operation generates a string of repeated characters. Its semantics would have been clearer if a static factory was provided instead, because the method name makes the operation self-explanatory, as in this example:

```
public static String Repeat(char c, int count);
```

■ **BRAD ABRAMS** This is, in fact, the pattern we use for the same concept in `ArrayList`.

- ✓ **DO** use a factory for conversion-style operations, such as `Parse` or `Decode`.

For example, consider the standard `Parse` method available on the primitive value types.

```
int i = int.Parse("35");
DateTime d = DateTime.Parse("10/10/1999");
```

The semantics of the `Parse` operation is such that information is converted from one representation of the value into another. In fact, it doesn't feel like we are constructing a new instance at all, but rather rehydrating one from an existing state (the string). The `System.Convert` class exposes many such static factory methods that take a value type in one representation and convert it to an instance of a different value type, retaining the same logical state in the process. Constructors have a very rigid contract with callers: A unique instance of a specific type will be created, initialized, and returned.

- ✓ **DO** prefer implementing factory operations as methods rather than properties.
- ✓ **DO** return created instances as method return values, not as out parameters.

■ **KRZYSZTOF CWALINA** Methods implementing the Try-Parse Pattern (see section 7.5.2) are factory methods that return created instances through out parameters. This is unfortunate, but using the return value for the Boolean is the best way to implement the pattern. This is an example showing that sometimes even **DO** and **DO NOT** guidelines need to be broken.

■ **JEREMY BARTON** The guidelines generally avoid conflict, and when they do there's usually an ordering that makes sense. For this guideline and the Try Pattern (section 7.5.2) it looks like this: (1) Design your factory method that returns the created instance (this guideline); (2) CONSIDER the Try Pattern ...; (3) observe DO create an exception-throwing member ... (which means leave this method as is); and (4) DO return the value from a Try method via an out parameter (applies specifically to the Try method and overrides this general factory guideline). In the end, you're left with our general Try-Parse:

```
public SomeType Parse(string s) { ... }
public bool TryParse(string s, out SomeType value) { ... }
```

- ✓ **CONSIDER** naming factory methods by concatenating Create and the name of the type being created, when the factory method is declared on a distinct factory type.

For example, consider naming a factory method that creates buttons `CreateButton`. In some cases, a domain-specific name can be used, as in `File.Open`.

- ✓ **CONSIDER** naming factory types by concatenating the name of the type being created and Factory. For example, consider naming a factory type that creates `Control` objects `ControlFactory`.

9.6 LINQ Support

Writing applications that interact with data sources, such as databases, XML documents, or Web Services, was made easier in .NET Framework 3.5 with the addition of a set of features collectively referred to as LINQ (Language-Integrated Query). The following sections provide a very brief overview of LINQ and list guidelines for designing APIs related to LINQ support, including the Query Pattern.

9.6.1 Overview of LINQ

Quite often, programming requires processing over sets of values. Examples include extracting a list of the most recently added books from a database of products, finding the e-mail address of a person in a directory service such as Active Directory, transforming parts of an XML document to HTML to allow for Web publishing, or an operation as frequently performed as looking up a value in a hashtable. LINQ allows for a uniform language-integrated programming model for querying data sets, independent of the technology used to store that data.

■ **RICO MARIANI** Like everything else, there are good and bad ways to use these patterns. The Entity Framework and LINQ to SQL offer good examples of how you can provide rich query semantics and still get very good performance using strong typing and by offering query compilation.

The Pit of Success notion is very important in LINQ implementations. I've seen some cases where the code that runs as a result of using a LINQ pattern is simply terrible in comparison to what you would write the conventional way. That's really not good enough—EF and LINQ to SQL let you write it nicely, and you get high-quality database interactions. That's what to aim for.

In terms of concrete language features and libraries, LINQ is embodied as:

- A specification of the notion of extension methods. (Extension methods are described in detail in section 5.6.)

- Lambda expressions, a language feature for defining anonymous delegates.
- The `Func<...>` and `Action<...>` types, representing generic delegates to functions and procedures.
- Representation of a delay-compiled delegate, the `Expression<...>` family of types.
- The `System.Linq.IQueryable<T>` interface.
- The Query Pattern, a specification of a set of methods that a type must provide to be considered a LINQ provider. A reference implementation of the pattern can be found in the `System.LinqEnumerable` class. Details of the pattern are discussed later in this chapter.
- Query Expressions, an extension to language syntax allowing for queries to be expressed in an alternative, SQL-like format.

```
//using extension methods:  
var names = set.Where(x => x.Age>20).Select(x=>x.Name);  
  
//using SQL-like syntax:  
var names = from x in set where x.Age>20 select x.Name;
```

MIRCEA TROFIN The interplay between these features is the following: Any `IEnumerable` can be queried upon using the LINQ extension methods, most of which require one or more lambda expressions as parameters; this leads to an in-memory generic evaluation of the queries. For cases where the set of data is not in memory (e.g., in a database) and/or queries may be optimized, the set of data is presented as an `IQueryable`. If lambda expressions are given as parameters, they are transformed by the compiler to `Expression<...>` objects. The implementation of `IQueryable` is responsible for processing said expressions. For example, the implementation of an `IQueryable` representing a database table would translate `Expression` objects to SQL queries.

9.6.2 Ways of Implementing LINQ Support

There are three ways by which a type can support LINQ queries:

- The type can implement `IEnumerable<T>` (or an interface derived from it).
- The type can implement `IQuerybable<T>`.
- The type can implement the Query Pattern.

The following sections will help you choose the right method of supporting LINQ.

9.6.3 Supporting LINQ through `IEnumerable<T>`

✓ **DO** implement `IEnumerable<T>` to enable basic LINQ support.

Such basic support should be sufficient for most in-memory data sets. The basic LINQ support will use the extension methods on `IEnumerable<T>` provided in .NET. For example, simply define the methods as follows:

```
public class RangeOfInt32s : IEnumerable<int> {
    public IEnumerator<int> GetEnumerator() {...}
    IEnumerator IEnumerable.GetEnumerator() {...}
}
```

Doing so allows for the following code, despite the fact that `RangeOfInt32s` did not implement a `Where` method:

```
var a = new RangeOfInt32s();
var b = a.Where(x => x > 10);
```

■ **RICO MARIANI** Keep in mind that you'll get your same enumeration semantics, and putting a LINQ façade on them does not make them execute any faster or use less memory.

✓ **CONSIDER** implementing `ICollection<T>` to improve the performance of query operators.

For example, the `System.Linq.Enumerable.Count` method's default implementation simply iterates over the collection. However, when the `IEnumerable<T>` implementation also implements `ICollection<T>`, the `Enumerable.Count` method returns the `ICollection<T>.Count` property value. Since most collection types that implement `ICollection<T>` offer an $O(1)$ implementation of the `Count` property, this noticeably speeds up the `Enumerable.Count` method.

- ✓ **CONSIDER** supporting selected methods of `System.Linq.Enumerable` or the Query Pattern (see section 9.6.5) directly on new types implementing `IEnumerable<T>` if it is desirable to override the default `System.Linq.Enumerable` implementation (e.g., for performance optimization reasons).

The specialized support methods can also be provided as extension methods. For example, the `ImmutableArrayExtensions.Select` extension method provides a more optimized implementation of the LINQ `from` expression than the default target of `Enumerable.Select` when the source collection is strongly typed as an `ImmutableArray<T>`.

```
public partial static class ImmutableArrayExtensions {
    public static IEnumerable<TResult> Select<T, TResult>(
        this ImmutableArray<T> immutableArray,
        Func<T, TResult> selector) { ... }
}
```

9.6.4 Supporting LINQ through `IQueryable<T>`

- ✓ **CONSIDER** implementing `IQueryable<T>` when access to the query expression, passed to members of `IQueryable`, is necessary.

When querying potentially large data sets generated by remote processes or machines, it might be beneficial to execute the query remotely. An example of such a data set is a database, a directory service, or a Web service.

- ✗ **DO NOT** implement `IQueryable<T>` without understanding the performance implications of doing so.

Building and interpreting expression trees is expensive, and many queries can actually get slower when `IQueryable<T>` is implemented.

This trade-off is acceptable in the LINQ to SQL case, since the alternative overhead of performing queries in memory would be far greater than the transformation of the expression to an SQL statement and the delegation of the query processing to the database server.

- ✓ **DO** throw `NotSupportedException` from `IQueryable<T>` methods that cannot be logically supported by your data source.

For example, imagine representing a media stream (e.g., an Internet radio stream) as an `IQueryable<byte>`. The `Count` method is not logically supported—the stream can be considered as infinite, so the `Count` method should throw a `NotSupportedException`.

9.6.5 Supporting LINQ through the Query Pattern

The Query Pattern refers to defining the methods in Figure 9-1 without implementing the `IQueryable<T>` (or any other LINQ interface).

```
S<T> Where(this S<T>, Func<T,bool>)

S<T2> Select(this S<T1>, Func<T1,T2>)
S<T3> SelectMany(this S<T1>, Func<T1,S<T2>>, Func<T1,T2,T3>)
S<T2> SelectMany(this S<T1>, Func<T1,S<T2>>)

O<T> OrderBy(this S<T>, Func<T,K>), where K is IComparable
O<T> ThenBy(this O<T>, Func<T,K>), where K is IComparable

S<T> Union(this S<T>, S<T>)
S<T> Take(this S<T>, int)
S<T> Skip(this S<T>, int)
S<T> SkipWhile(this S<T>, Func<T,bool>)

S<T3> Join(this S<T1>, S<T2>, Func<T1,K1>, Func<T2,K2>,
Func<T1,T2,T3>)

T ElementAt(this S<T>,int)
```

FIGURE 9-1: Query pattern method signatures

The notation in Figure 9-1 is not meant to be valid code in any particular language, but rather is simply used to present the type signature pattern. In this notation, S indicates a collection type (e.g., `IEnumerable<T>`, `ICollection<T>`), and T indicates the type of elements in that collection. Additionally, we use `O<T>` to represent subtypes of `S<T>` that are ordered. For example, `S<T>` is a notation that could be replaced with `IEnumerable<int>`, `ICollection<Foo>`, or even `MyCollection` (as long as the type is an enumerable type).

The first parameter of all the methods in the pattern (marked with `this`) is the type of the object the method is applied to. The notation uses extension-method-like syntax, but the methods can be implemented as extension methods or as member methods. In the latter case, the first parameter should be omitted, of course, and the `this` pointer should be used.

Also, anywhere `Func<...>` is being used, pattern implementations may substitute `Expression<Func<...>>` for it. You can find guidelines later in this section that describe when this substitution is preferable.

✓ DO implement the Query Pattern as instance members on the new type, if the members make sense on the type even outside the context of LINQ. Otherwise, implement them as extension methods.

For example, instead of the following:

```
public partial class MyDataSet<T> : IEnumerable<T>{...}

public static partial class MyDataSetQueries {
    public static MyDataSet<T> Where(
        this MyDataSet<T> data, Func<T, bool> query) { ... }
}
```

Prefer the following, because it's completely natural for data sets to support `Where` methods:

```
public partial class MyDataSet<T>:IEnumerable<T> {
    public MyDataSet<T> Where(Func<T, bool> query) { ... }
}
```

- ✓ **DO** implement `IEnumerable<T>` on types implementing the Query Pattern.
- ✓ **CONSIDER** designing the LINQ operators to return domain-specific enumerable types. Essentially, you can return anything from a `Select` query method; however, the expectation is that the query result type should be at least enumerable.

This allows the implementation to control which query methods get executed when they are chained. Otherwise, consider a user-defined type `MyType`, which implements `IEnumerable<T>`. The `MyType` type has an optimized `Count` method defined, but the return type of the `Where` method is `IEnumerable<T>`. In the example here, the optimization is lost after the `Where` method is called; the method returns `IEnumerable<T>`, so the built-in `Enumerable.Count` extension method is called instead of the optimized version defined on `MyType`.

```
var result = myInstance.Where(query).Count();
```

- ✗ **AVOID** implementing just a part of the Query Pattern if fallback to the basic `IEnumerable<T>` implementations is undesirable.

For example, consider a user-defined type `MyType`, which implements `IEnumerable<T>`. The `MyType` type has an optimized `Count` method defined but does not have `Where`. In the example here, the optimization is lost after the `Where` method is called; the method returns `IEnumerable<T>`, so the built-in `Enumerable.Count` method is called instead of the optimized one defined on `MyType`.

```
var result = myInstance.Where(query).Count();
```

- ✓ **DO** represent ordered sequences as a separate type from their unordered counterpart.
- ✓ **DO** define the `ThenBy` method, or implement `IOrderedEnumerable<T>`, on ordered sequence types.

This follows the current pattern in the LINQ to Objects implementation and allows for early (compile-time) detection of errors such as applying ThenBy to an unordered sequence.

For example, .NET provides the `IOrderedEnumerable<T>` type, which is returned by `OrderBy`. The `ThenBy` extension method is defined for this type, but not for `IEnumerable<T>`.

- ✓ **DO** defer execution of query operator implementations. The expected behavior of most of the Query Pattern members is that they simply construct a new object which, upon enumeration, produces the elements of the set that match the query.

The following methods are exceptions to this rule: `All`, `Any`, `Average`, `Contains`, `Count`, `ElementAt`, `Empty`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Max`, `Min`, `Single`, and `Sum`.

In the example here, the expectation is that the time necessary for evaluating the second line will be independent from the size or nature (e.g., in-memory or remote server) of `set1`. The general expectation is that this line simply prepares `set2`, delaying the determination of its composition to the time of its enumeration.

```
var set1 = ...  
var set2 = set1.Select(x => x.SomeInt32Property);  
foreach(int number in set2){...} // this is when actual work happens
```

- ✓ **DO** place query extensions methods in a “Linq” subnamespace of the main namespace. For example, extension methods for `System.Data` features reside in `System.Data.Linq` namespace.
- ✓ **DO** use `Expression<Func<...>>` as a parameter instead of `Func<...>` when it is necessary to inspect the query. See section 9.6.5 for more details.

As discussed earlier, interacting with an SQL database is already done through `IQueryable<T>` (and therefore expressions) rather than `IEnumerable<T>`, since this gives an opportunity to translate lambda expressions to SQL expressions.

An alternative reason for using expressions is performing optimizations. For example, a sorted list can implement look-up (`Where` clauses) with binary search, which can be much more efficient than the standard `IEnumerable<T>` or `IQueryable<T>` implementations.

9.7 Optional Feature Pattern

When designing an abstraction, you might want to allow cases in which some implementations of the abstraction support a feature or a behavior, whereas other implementations do not. For example, stream implementations can support reading, writing, seeking, or any combination thereof.

One way to model these requirements is to provide a base class with APIs for all nonoptional features and a set of interfaces for the optional features. The interfaces are implemented only if the feature is actually supported by a concrete implementation. The following example shows one of many ways to model the stream abstraction using such an approach.

```
// framework APIs
public abstract class Stream {
    public abstract void Close();
    public abstract int Position { get; }
}
public interface IInputStream {
    byte[] Read(int numberOfBytes);
}
public interface IOutputStream {
    void Write(byte[] bytes);
}
public interface ISekableStream {
    void Seek(int position);
}
public interface IFiniteStream {
    int Length { get; }
    bool EndOfStream { get; }
}

// concrete stream
public class FileStream : Stream, IOutputStream, IInputStream,
ISekableStream, IFiniteStream {
    ...
}
```

```
// usage
void OverwriteAt(IOutputStream stream, int position, byte[] bytes){
    // do dynamic cast to see if the stream is seekable
    ISeekableStream seekable = stream as ISeekableStream;
    if (seekable==null){
        throw new NotSupportedException(...);
    }
    seekable.Seek(position);
    stream.Write(bytes);
}
```

You will notice that the `System.IO` namespace does not follow this model, and with good reason. Such factored design requires adding many types to the framework, which increases general complexity. Also, using optional features exposed through interfaces often requires dynamic casts, which in turn results in usability problems.

KRZYSZTOF CWALINA Sometimes framework designers provide interfaces for common combinations of optional interfaces. For example, the `OverwriteAt` method would not have to use the dynamic cast if the framework design provided `ISeekableOutputStream`. The problem with this approach is that it results in an explosion of the number of different interfaces for all combinations.

Sometimes the benefits of factored design are worth the drawbacks, but often they are not. It is easy to overestimate the benefits and underestimate the drawbacks. For example, the factorization did not help the developer who wrote the `OverwriteAt` method avoid runtime exceptions (the main reason for factorization). It is our experience that many designs incorrectly err on the side of too much factorization.

The Optional Feature Pattern provides an alternative to excessive factorization. It has drawbacks of its own but should be considered as an alternative to the factored design described previously. This pattern provides a mechanism for discovering whether the particular instance supports a feature through a query API and uses the features by accessing optionally supported members directly through the base abstraction.

```

// framework APIs
public abstract class Stream {
    public abstract void Close();
    public abstract int Position { get; }

    public virtual bool CanWrite { get { return false; } }
    public virtual void Write(byte[] bytes){
        throw new NotSupportedException(...);
    }

    public virtual bool CanSeek { get { return false; } }
    public virtual void Seek(int position){
        throw new NotSupportedException(...);
    }
    ...
}

// concrete stream
public class FileStream : Stream {
    public override bool CanSeek { get { return true; } }
    public override void Seek(int position) { ... }
    ...
}

// usage
void OverwriteAt(Stream stream, int position, byte[] bytes){
    if (!stream.CanSeek || !stream.CanWrite){
        throw new NotSupportedException(...);
    }
    stream.Seek(position);
    stream.Write(bytes);
}

```

In fact, the `System.IO.Stream` class uses this design approach. Some abstractions might choose to use a combination of factoring and the Optional Feature Pattern. For example, the .NET collection interfaces are factored into indexable and nonindexable collections (`IList<T>` and `ICollection<T>`), but they use the Optional Feature Pattern to differentiate between read-only and read-write collections (`ICollection<T>.IsReadOnly` property).

✓ CONSIDER using the Optional Feature Pattern for optional features in abstractions.

The pattern minimizes the complexity of the framework and improves usability by making dynamic casts unnecessary.

STEVE STARCK If your expectation is that only a very small percentage of classes deriving from the base class or interface would actually implement the optional feature or behavior, then using interface-based design might be better. There is no real need to add additional members to all derived classes when only one of them provides the feature or behavior. Also, factored design is preferred in cases when the number of combinations of the optional features is small and the compile-time safety afforded by factorization is important.

- ✓ **DO** provide a simple Boolean property that clients can use to determine whether an optional feature is supported.

```
public abstract class Stream {
    public virtual bool CanSeek { get { return false; } }
    public virtual void Seek(int position){ ... }
}
```

Code that consumes the abstract base class can query this property at runtime to determine whether it can use the optional feature.

```
if (stream.CanSeek){
    stream.Seek(position);
}
```

- ✓ **DO** use virtual methods on the base class that throw NotSupportedException to define optional features.

```
public abstract class Stream {
    public virtual bool CanSeek { get { return false; } }
    public virtual void Seek(int position){
        throw new NotSupportedException(...);
    }
}
```

The method can be overridden by subclasses to provide support for the optional feature. The exception should clearly communicate to the user that the feature is optional and which property the user should query to determine if the feature is supported.

9.8 Covariance and Contravariance

Generics provide a very powerful type system feature that allows creation of so-called parameterized types. For example, `List<T>` is such a type: It represents a list of objects of type `T`. The type `T` is specified when the instance of the list is created.

```
var names = new List<string>();
names.Add("John Smith");
names.Add("Mary Johnson");
```

Such generic data structures have many benefits over their non-generic counterparts. But they also have some—sometimes surprising—limitations. For example, some users expect that a `List<string>` can be cast to `List<object>`, just as a `String` can be cast to `Object`. But unfortunately, the following code won't even compile:

```
List<string> names = new List<string>();
List<object> objects = names; // this won't compile
```

There is a very good reason for this limitation: It allows for full strong typing. For example, if you could cast `List<string>` to a `List<object>`, the following incorrect code would compile, but the program would fail at runtime:

```
static void Main(){
    var names = new List<string>();

    // This of course does not compile, but if it did
    // the whole program would compile, but would be incorrect as it
    // attempts to add arbitrary objects to a list of strings.
    AddObjects((List<object>)names);

    string name = names[0]; // how could this work?
}

// This would (and does) compile just fine.
static void AddObjects(List<object> list) {
    list.Add(new object()); // It's a list of strings, really. Should we
    throw?
    list.Add(new Button());
}
```

Unfortunately, this limitation can also be undesired in some scenarios. For example, let's consider the following type:

```
public class CountedReference<T> {  
    public CountedReference(T value);  
    public T Value { get; }  
    public int Count { get; }  
    public void AddReference();  
    public void ReleaseReference();  
}
```

There is nothing conceptually wrong with casting a `CountedReference<string>` to `CountedReference<object>`, as in the following example:

```
var reference = new CountedReference<string>(...);  
CountedReference<object> obj = reference; // this won't compile
```

In general, having a way to represent any instance of this generic type is very useful, but there is no automatic common root for constructed types (types constructed by specifying type arguments to a generic type).

```
// What type should ??? be?  
// CountedReference<object> would be nice but it won't work  
static void PrintValue(??? anyCountedReference) {  
    Console.WriteLine(anyCountedReference.Value);  
}
```

■ **KRZYSZTOF CWALINA** Of course, `PrintValue` could be a generic method taking `CountedReference<T>` as the parameter.

```
static void PrintValue<T>(CountedReference<T> any) {  
    Console.WriteLine(any.Value);  
}
```

This would be a fine solution in many cases. But it does not work as a general solution and might have negative performance implications. For example, this trick does not work for properties. If a property needed to be typed as “any reference,” you could not use `CountedReference<T>` as the type of the property. In addition, generic methods might have undesirable performance implications. If such generic methods are called with many differently sized type arguments, the runtime will generate a new method for every argument size. This might introduce unacceptable memory consumption overhead.

Unfortunately, unless `CountedReference<T>` implemented the Simulated Covariance Pattern described in section 9.8.3, the only common representation of all `CountedReference<T>` instances would be `System.Object`. But `System.Object` is too limiting and would not allow the `PrintValue` method to access the `Value` property.

The reason that casting to `CountedReference<object>` is just fine, but casting to `List<object>` can cause all sorts of problems, is that in the former case the object appears only in the output position (the return type of `Value` property). In the case of `List<object>`, the object represents both output and input types. For example, `object` is the type of the input to the `Add` method.

```
// T does not appear as input to any members except the constructor
public class CountedReference<T> {
    public CountedReference(T value);
    public T Value { get; }
    public int Count { get; }
    public void AddReference();
    public void ReleaseReference();
}

// T does appear as input to members of List<T>
public class List<T> {
    public void Add(T item); // T is an input here
    public T this[int index]{
        get;
        set; // T is actually an input here
    }
}
```

In other words, in `CountedReference<T>`, the `T` is only in covariant positions (outputs). In `List<T>`, the `T` is in both covariant (output) and contravariant (input) positions.

There's a quote, commonly attributed to Albert Einstein, that says: "You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: You send signals here, they receive them there. The only difference is that there is no cat." Similarly, covariance and contravariance can be thought of as making an adapter type to translate from one form to another, so long as no cast operators are required... and there is no adapter type.

```
public class EnumeratorAdapter<T1,T2> : Ienumerator<T2> {
    private Ienumerator<T1> _wrapped;

    public EnumeratorAdapter(Ienumerator<T1> wrapped) {
        _wrapped = wrapped;
    }

    public bool MoveNext() => _wrapped.MoveNext();
    public void Reset() => _wrapped.Reset();
    public T2 Current => _wrapped.Current;
    object Ienumerator.Current => Current;
}

public class ComparerAdapter<T1, T2> : IComparer<T2> {
    private IComparer<T1> _wrapped;

    public ComparerAdapter(IComparer<T1> wrapped) {
        _wrapped = wrapped;
    }

    public int Compare(T2 x, T2 y) => _wrapped.Compare(x, y);
}
```

The `EnumeratorAdapter` example is demonstrating covariance: because the generic parameter is only used in an output position, the adapter would work for a `T1` and `T2` where `T2` is between `T1` and `System.Object`. The `ComparerAdapter` example is demonstrating contravariance: because the generic parameter is only used in an input position, the adapter would work for a `T1` and `T2` where `T1` is between `T2` and `System.Object` (the opposite relationship as covariant).

9.8.1 Contravariance

Contravariance for generic type parameters is indicated in C# with the `in` keyword, and in VB.NET with the `In` modifier.

```
// C# invariant delegate
public delegate void Invariant<T>(T arg);
// C# contravariant delegate
public delegate void Contravariant<in T>(T arg);

// VB invariant delegate
Public Delegate Sub Invariant(Of T)(arg as T);
// VB contravariant delegate
Public Delegate Sub Contravariant(Of In T)(arg as T);
```

- ✓ **DO** mark generic type parameters on generic delegates as contravariant when the type parameter is used as input, but not output, and your language supports creating contravariant types.
- ✓ **DO** mark generic type parameters on generic interfaces as contravariant when the type parameter is only ever used as an input parameter to methods on the interface.

These two guidelines really boil down to “declare contravariance when the compiler lets you do so.” In addition to the simple cases where the generic type parameter is used as a parameter type, the type parameter can be used as a generic type argument to covariant interfaces or delegates as methods inputs, or to contravariant interfaces or delegates as method return types.

```
public interface ICalculator<in T> {
    int Sum(IEnumerable<T> inputs);
}
```

Since changing the signature of a delegate and altering an interface are both breaking changes, declaring contravariance whenever it’s possible to do so doesn’t artificially limit future changes to the type.

- ✗ **DO NOT** change generic type parameters from contravariant to invariant in newer versions of your library.

Changing a generic type parameter from contravariant to invariant would be a breaking change for any callers that exercised the variance. Given that adding new members to an interface is a breaking change, the most likely situation in which you might want to remove the contravariant indicator is when rewriting your library from a language that can express variance (such as C#) to one that cannot (such as F#, as of language version 4.7).

- ✓ **CONSIDER** making a sub-interface of a new generic interface if removing members would allow a generic type parameter to be contravariant and the remaining members provide a coherent interface.

When an interface uses a generic type parameter for both input and output purposes, that parameter must be declared as invariant. Any

methods that only use the generic type parameter in an input role could be moved to a sub-interface to empower your callers with the flexibility that contravariance provides. If the applicable methods wouldn't make for a coherent interface—or set of interfaces—then don't artificially split your new interface.

Because interface members cannot be moved without a breaking change—only copied—and adding a sub-interface to an interface is also a breaking change, this should not be done for any public interface that has already been released.

```
// Instead of
public interface IEqualityComparer<T> : ICollection<T> {
    bool SequenceEqual(IEnumerable<T> other);
}

// Consider
public interface ISequenceEqual<in T> {
    bool SequenceEqual(IEnumerable<T> other);
}
public interface IEqualityComparer<T> :
    ICollection<T>, ISequenceEqual<T> {
}
```

9.8.2 Covariance

Covariance for generic type parameters is indicated in C# with the `out` keyword, and in VB.NET with the `Out` modifier.

```
// C# invariant delegate
public delegate T Invariant<T>();
// C# covariant delegate
public delegate T Covariant<out T>();

// VB invariant delegate
Public Delegate Sub Invariant(Of T)() as T;
// VB covariant delegate
Public Delegate Sub Covariant(Of In T)() as T;
```

- ✓ **DO** mark generic type parameters on generic delegates as covariant when the type parameter is used as output, but not input, and your language supports creating covariant types.

- ✓ **DO** mark generic type parameters on generic interfaces as covariant when the type parameter is only ever used in output roles for members on the interface.

As with contravariance, these two guidelines essentially boil down to “declare covariance when the compiler lets you do so.” In addition to the simple cases where the generic type parameter is used as a parameter type, the type parameter can be used as a generic type argument to contravariant interfaces or delegates as method inputs, or to covariant interfaces or delegates as method return types.

```
public interface IFilter<out T> {
    IEnumerable<T> Filter(Func<T, bool> predicate);
}
```

Since changing the signature of a delegate and altering an interface are both breaking changes, declaring covariance whenever it’s possible to do so doesn’t artificially limit future changes to the type.

- ✗ **DO NOT** declare a generic type parameter covariant on an interface that returns an array based on the type parameter.

The C# variance rules permit an interface to be covariant when returning generic arrays, but writing to an array produced by covariance can result in an `ArrayTypeMismatchException` at runtime from safe-looking code. Change the return value to be a covariant collection interface, remove the array-returning member altogether, or declare the generic type parameter as invariant on the interface.

```
public interface IArrayProducer<out T> {
    T[] ProduceArray(int length);
}
public class ArrayProducer<T> : IArrayProducer<T> {
    public T[] ProduceArray(int length) {
        return new T[length];
    }
}
...
IArrayProducer<object> producer = new ArrayProducer<string>();
object[] array = producer.ProduceArray(1);
// The array is actually a string[], so this write fails.
array[0] = 3;
```

X DO NOT change generic type parameters from covariant to invariant in newer versions of your library.

Changing a generic type parameter from contravariant to invariant would be a breaking change for any callers that exercised the variance. Given that adding new members to an interface is a breaking change, the most likely situation in which you might want to remove the contravariant indicator is when rewriting your library from a language that can express variance (such as C#) to one that cannot (such as F#, as of language version 4.7).

✓ CONSIDER making a sub-interface of a new generic interface if removing members would allow a generic type parameter to be covariant and the remaining members provide a coherent interface.

When an interface uses a generic type parameter for both input and output purposes, that parameter must be declared invariant. Any methods or properties that only use the generic type parameter in an output role could be moved to a sub-interface to empower your callers with the flexibility covariance provides. If the applicable methods wouldn't make for a coherent interface—or set of interfaces—then don't artificially split your new interface. Any get/set properties on the interface can be declared as get-only on the new sub-interface.

Because interface members cannot be moved without a breaking change—only copied—and adding a sub-interface to an interface is also a breaking change, this should not be done for any public interface that has already been released.

Had .NET supported covariance on generic interfaces when generics were originally introduced, `IList<T>` would almost certainly have been separated into `IReadOnlyList<out T>` and `IList<T>`.

In this example, `IEasyCollection<T>` is split into `IEasyCollection<T>` and `IReadOnlyEasyCollection<out T>`. The `Clear()` method could have been declared on `IReadOnlyEasyCollection<T>`, but since `Clear()` doesn't belong on a "read-only" collection, it was not moved. The `Clear()` method could be split off to a third interface, `IClearable`, but that method doesn't seem to stand alone as an interface.

```

// Instead of
public interface IEasyCollection<T> : IEnumerable<T> {
    int Count { get; }
    bool IsReadOnly { get; }
    void Add(T item);
    void Clear();
}

// Consider
public interface IReadOnlyEasyCollection<out T> : IEnumerable<T>{
    int Count { get; }
    bool IsReadOnly { get; }
}
public interface IEasyCollection<T> : IReadOnlyEasyCollection<T> {
    void Add(T item);
    void Clear();
}

```

9.8.3 Simulated Covariance Pattern

To solve the problem of not having a common type representing the root of all constructions of a generic type, you can implement the Simulated Covariance Pattern.

Consider a generic type (class or interface) and its dependencies described in the following code fragment:

```

public class Foo<T> {
    public T Property1 { get; }
    public T Property2 { set; }
    public T Property3 { get; set; }
    public void Method1(T arg1);
    public T Method2();
    public T Method3(T arg);
    public Type1<T> GetMethod1();
    public Type2<T> GetMethod2();
}
public class Type1<T> {
    public T Property { get; }
}
public class Type2<T> {
    public T Property { get; set; }
}

```

Create a new interface (root type) in which all members containing a T at contravariant positions are removed. In addition, feel free to remove all members that might not make sense in the context of the trimmed-down type.

```
public interface IFoo<out T> {
    T Property1 { get; }
    T Property3 { get; } // setter removed
    T Method2();
    Type1<T> GetMethod1();
    IType2<T> GetMethod2(); // note that the return type changed
}
public interface IType2<T> {
    T Property { get; } // setter removed
}
```

The generic type should then implement the interface explicitly and “add back” the strongly typed members (using T instead of object) to its public API surface.

```
public class Foo<T> : IFoo<object> {
    public T Property1 { get; }
    public T Property2 { set; }
    public T Property3 { get; set; }
    public void Method1(T arg1);
    public T Method2();
    public T Method3(T arg);
    public Type1<T> GetMethod1();
    public Type2<T> GetMethod2();

    object IFoo<object>.Property1 { get; }
    object IFoo<object>.Property3 { get; }
    object IFoo<object>.Method2() { return null; }
    Type1<object> IFoo<object>.GetMethod1();
    IType2<object> IFoo<object>.GetMethod2();
}

public class Type2<T> : IType2<object> {
    public T Property { get; set; }
    object IType2<object>.Property { get; }
}
```

Now, all constructed instantiations of `Foo<T>` have a common root type `IFoo<object>`.

```
var foos = new List<IFoo<object>>();
```

```

foos.Add(new Foo<int>());
foos.Add(new Foo<string>());
...
foreach(IFoo<object> foo in foos){
    Console.WriteLine(foo.Property1);
    Console.WriteLine(foo.GetMethod2().Property);
}

```

In the case of the simple `CountedReference<T>`, the code would look like the following:

```

public interface ICountedReference<out T> {
    T Value { get; }
    int Count { get; }
    void AddReference();
    void ReleaseReference();
}

public class CountedReference<T> : ICountedReference<object> {
    public CountedReference(T value) {...}
    public T Value { get { ... } }
    public int Count { get { ... } }
    public void AddReference(){...}
    public void ReleaseReference(){...}

    object ICountedReference<object>.Value { get { return Value; } }
}

```

✓ **CONSIDER** using the Simulated Covariance Pattern if there is a need to have a representation for all instantiations of a generic type.

This pattern should not be used frivolously, because it results in additional types in the framework and can make the existing types more complex.

✓ **DO** ensure that the implementation of the root's members is equivalent to the implementation of the corresponding generic type members.

There should not be an observable difference between calling a member on the root type and calling the corresponding member on the generic type. In many cases, the members of the root are implemented by calling members on the generic type.

```

public class Foo<T> : IFoo<object> {

    public T Property3 { get { ... } set { ... } }
}

```

```

object IFoo<object>.Property3 { get { return Property3; } }

}
...

```

- ✓ **CONSIDER** using an abstract class instead of an interface to represent the root.

This might sometimes be a better option, because interfaces are more difficult to evolve (see section 4.3). On the other hand, some problems can arise with using abstract classes for the root. Abstract class members cannot be implemented explicitly, and the subtypes need to use the new modifier. This makes it tricky to implement the root's members by delegating to the generic type members.

- ✓ **CONSIDER** using a non-generic root type if such type is already available.

For example, `List<T>` implements `IEnumerable` for the purpose of simulating covariance.

9.9 Template Method

The Template Method Pattern is a very well-known pattern described in much greater detail in many sources, such as the classic book *Design Patterns* by Gamma et al. Its intent is to outline an algorithm in an operation. The Template Method Pattern allows subclasses to retain the algorithm's structure while permitting redefinition of certain steps of the algorithm. We include a simple description of this pattern here, because it is one of the most commonly used patterns in API frameworks.

The most common variation of the pattern consists of one or more nonvirtual (usually public) members that are implemented by calling one or more protected virtual members.

```

public Control{
    public void SetBounds(int x, int y, int width, int height){
        ...
        SetBoundsCore (...);
    }

    public void SetBounds(int x, int y, int width, int

```

```
height, BoundsSpecified specified){  
    ...  
    SetBoundsCore (...);  
}  
  
protected virtual void SetBoundsCore(int x, int y, int width,  
int height, BoundsSpecified specified){  
    // Do the real work here.  
}  
}
```

The goal of the pattern is to control extensibility. In the preceding example, the extensibility is centralized to a single method (a common mistake is to make more than one overload virtual). This helps to ensure that the semantics of the overloads stay consistent, because the overloads cannot be overridden independently.

Also, public virtual members basically give up all control over what happens when the member is called. This pattern is a way for the base class designer to enforce some structure of the calls that happen in the member. The nonvirtual public methods can ensure that certain code executes before or after the calls to virtual members and that the virtual members execute in a fixed order.

As a framework convention, the protected virtual methods participating in the Template Method Pattern should use the suffix “Core.”

X AVOID making public members virtual.

If a design requires virtual members, follow the template pattern and create a protected virtual member that the public member calls. This practice provides more controlled extensibility.

✓ CONSIDER using the Template Method Pattern to provide more controlled extensibility.

In this pattern, all extensibility points are provided through protected virtual members that are called from nonvirtual members.

✓ CONSIDER naming protected virtual members that provide extensibility points for nonvirtual members by suffixing the nonvirtual member name with “Core.”

```
public void SetBounds(...){
    ...
    SetBoundsCore (...);
}
protected virtual void SetBoundsCore(...){ ... }
```

- ✓ **DO** perform argument and state validation in the nonvirtual member of the Template Method Pattern before calling the virtual member.
- ✓ **DO** only perform argument and state validation unique to the derived type in the virtual member of the Template Method Pattern. That is, do not repeat validation from the nonvirtual member.

Any argument states that would surprise callers if they varied across implementations should be performed in the public member. This includes throwing exceptions for arguments that cannot be null, index and count values that don't make sense for the target collection, and state exceptions like `ObjectDisposedException`. If the derived type has additional invalid states, or needs to restrict arguments further than the base class did, then it should add those checks as with any other method.

By moving the common checks into the public member, a developer for the derived type only needs to focus on the logic for their specific implementation. This approach eliminates the entire class of bugs associated with argument validation being slightly different, or outright missing, in a specific derived type.

```
public void SetBounds(int left, int right) {
    if (left < MinLeft || left >= right) {
        throw new ArgumentOutOfRangeException(nameof(left));
    }
    if (right > MaxRight) {
        throw new ArgumentOutOfRangeException(nameof(right));
    }
    if (_disposed) {
        throw new ObjectDisposedException(...);
    }
    SetBoundCore(left, right);
}

protected virtual void SetBoundsCore(int left, int right){ ... }
```

9.10 Timeouts

Timeouts occur when an operation returns before its completion because the maximum time allocated for the operation (timeout time) has elapsed. The user often specifies the timeout time. For example, it might take a form of a parameter to a method call.

```
server.PerformOperation(timeout);
```

An alternative approach is to use a property.

```
server.Timeout = timeout;  
server.PerformOperation();
```

The following short list of guidelines describes best practices for the design of APIs that need to support timeouts.

- ✓ **DO** prefer method parameters as the mechanism for users to provide timeout time.

Method parameters are favored over properties because they make the association between the operation and the timeout much more apparent. The property-based approach might be better if the type is designed to be a component used with visual designers.

- ✓ **DO** prefer using `TimeSpan` to represent timeout time.

Historically, timeouts have been represented by integers. Integer timeouts can be hard to use for the following reasons:

- It is not obvious what the unit of the timeout is.
- It is difficult to translate units of time into the commonly used millisecond. (How many milliseconds are in 15 minutes?)

Often, a better approach is to use `TimeSpan` as the timeout type. `TimeSpan` solves the preceding problems.

```
class Server {  
    void PerformOperation(TimeSpan timeout){  
        ...
```

```
    }  
}  
  
var server = new Server();  
server.PerformOperation(TimeSpan.FromMinutes(15));
```

Integer timeouts are acceptable if:

- The parameter or property name can describe the unit of time used by the operation—for example, if a parameter can be called `milliseconds` without making an otherwise self-describing API cryptic.
- The most commonly used value is small enough that users won't have to use calculators to determine the value—for example, if the unit is milliseconds and the commonly used timeout is less than 1 second.

✓ **DO** throw `System.TimeoutException` when a timeout elapses.

A timeout equal to `TimeSpan(0)` means that the operation should throw if it cannot complete immediately. If the timeout equals `TimeSpan.MaxValue`, the operation should wait forever without timing out. Operations are not required to support either of these values, but they should throw an `ArgumentOutOfRangeException` if an unsupported timeout value is specified.

If a timeout expires and the `System.TimeoutException` is thrown, the server class should cancel the underlying operation.

✗ **DO NOT** return error codes to indicate timeout expiration.

Expiration of a timeout means the operation could not complete successfully and thus should be treated and handled as any other runtime error (see Chapter 7).

9.11 XAML Readable Types

XAML is an XML format used by WPF (and other technologies) to represent object graphs. The following guidelines describe design considerations for ensuring that your types can be created using XAML readers.

- ✓ **CONSIDER** providing the default constructor if you want a type to work with XAML.

For example, consider the following XAML markup:

```
<Person Name="John" Age="22" />
```

It is equivalent to the following C# code:

```
new Person() { Name = "John", Age = 22 };
```

Consequently, for this code to work, the `Person` class needs to have a default constructor. Markup extensions, discussed in the next guideline in this section, are an alternative way of enabling XAML.

■ **CHRIS SELLS** In my opinion, this one should really be a DO, not a CONSIDER. If you're designing a new type to support XAML, it's far more preferable to do it with a default constructor than with markup extensions or type converters.

- ✓ **DO** provide markup extension if you want an immutable type to work with XAML readers.

Consider the following immutable type:

```
public class Person {
    public Person(string name, int age){
        Name = name;
        Age = age;
    }
    public string Name { get; }
    public int Age { get; }
}
```

Properties of such a type cannot be set using XAML markup, because the reader does not know how to initialize the properties using the parameterized constructor. Markup extensions address this problem.

```
[MarkupExtensionReturnType(typeof(Person))]
public class PersonExtension : MarkupExtension {
    public string Name { get; set; }
    public int Age { get; set; }
```

```

    public override object ProvideValue(IServiceProvider
serviceProvider){
    return new Person(Name, Age);
}
}

```

Keep in mind that immutable types cannot be written using XAML writers.

- X AVOID** defining new type converters unless the conversion is natural and intuitive. In general, limit type converter usage to the ones already provided by .NET.

Type converters are used to convert a value from a string to the appropriate type. They're used by XAML infrastructure and in other places, such as graphical designers. For example, the string "#FFFF0000" in the following markup gets converted to an instance of a red Brush thanks to the type converter associated with the Rectangle.Fill property.

```
<Rectangle Fill="#FFFF0000"/>
```

But type converters can be defined too liberally. For example, the Brush type converter should not support specifying gradient brushes, as shown in the following hypothetical example:

```
<Rectangle Fill="HorizontalGradient White Red" />
```

Such converters define new “minilanguages,” which add complexity to the system.

- ✓ CONSIDER** applying the ContentPropertyAttribute to enable convenient XAML syntax for the most commonly used property.

```

[ContentProperty("Image")]
public class Button {
    public object Image { get; set; }
}

```

The following XAML syntax would work without the attribute:

```
<Button>
  <Button.Image>
    <Image Source="foo.jpg" />
  </Button.Image>
</Button>
```

The attribute makes the following much more readable syntax possible.

```
<Button>
  <Image Source="foo.jpg" />
</Button>
```

9.12 Operating on Buffers

A number of operations—such as character encoding, cryptography, data serialization, and text escaping—operate on multiple input values and output multiple values, typically bytes or characters. Within .NET, these operations usually take an array as input, and return a new array as output:

```
public partial class Encoding {
  public virtual byte[] GetBytes(char[] chars) { ... }
  public virtual char[] GetChars(byte[] bytes) { ... }
}
...
public partial class HashAlgorithm {
  public byte[] ComputeHash(byte[] buffer) { ... }
}
...
public partial class Convert {
  public static string ToBase64String(byte[] inArray) { ... }
  public static byte[] FromBase64String(string s) { ... }
}
```

Arrays are the traditional data types used for arbitrary-length input and arbitrary-length output, with a few very common operations also accepting pointers. Accepting an array for input data is generally good, but in some cases this approach introduces inefficiencies. For example,

operating on only part of an array sometimes requires the caller to make a smaller copy of the array to get the size correct. Similarly, returning an array, precisely sized to match how much data was produced, has performance overhead. Text-based operations frequently operate on `String` input, which functions as a read-only variant of a `char[]`, and comes with similar performance problems when only a portion of the string is intended to be passed into a method.

The `System.Span<T>` type, introduced in .NET Core 2.0, addresses these inefficiencies without returning entirely to C-style pointer-and-length inputs and the risks that entails. The `Span<T>` type represents contiguous memory from a managed array, from a `stackalloc` array, a pointer and length, or a `string`. The `Span<T>` type also supports $O(1)$ slicing operations, through the `Slice` methods. Slicing a `Span` does not make a copy of the data, so it eliminates the memory and time costs associated with copying to a smaller array.

`Span<T>` is a ref-like value type (`ref struct`), which means it cannot be a field in a class or (non-ref) `struct`. Operations that need to store a buffer can use the `System.Memory<T>` type, which essentially functions as a `Span<T>` holder. The `Memory<T>` type can represent a range within an array or be associated with a `MemoryManager<T>` object for more complicated uses (including pointer-based memory), making it more versatile than the `ArraySegment<T>` type.

The guidance in this section can result in a large number of variations of what would otherwise be a single method. The various patterns have differences in exception handling, ease of use, and support across all of the CLR languages. It is really up to you to decide how many are appropriate to your feature and your user base. A very flexible set of methods to parse hexadecimal input to bytes might look like this:

```
public partial class ByteOperations {
    public static byte[] FromHexadecimal(
        ReadOnlySpan<char> source) { ... }
    public static byte[] FromHexadecimal(
        char[] source) { ... }
    public static byte[] FromHexadecimal(
        char[] source, int sourceIndex, int count) { ... }
```

```
public static byte[] FromHexadecimal(
    ArraySegment<char> source) { ... }
public static byte[] FromHexadecimal(
    string source) { ... }

public static int FromHexadecimal(
    ReadOnlySpan<char> source,
    Span<byte> destination) { ... }
public static int FromHexadecimal(
    char[] source,
    byte[] destination) { ... }
public static int FromHexadecimal(
    byte[] source, int sourceIndex,
    char[] destination, int destinationIndex, int count) { ... }
public static int FromHexadecimal(
    ArraySegment<char> source,
    ArraySegment<byte> destination) { ... }

public static bool TryFromHexadecimal(
    ReadOnlySpan<char> source,
    Span<byte> destination,
    out int bytesWritten) { ... }
public static bool TryFromHexadecimal(
    char[] source,
    byte destination,
    out int bytesWritten) { ... }
public static bool TryFromHexadecimal(
    char[] source, int sourceIndex,
    byte[] destination, int destinationIndex,
    int count,
    out int bytesWritten) { ... }
public static bool TryFromHexadecimal(
    ArraySegment<char> source,
    ArraySegment<byte> destination,
    out int bytesWritten) { ... }

public static OperationStatus FromHexadecimal(
    ReadOnlySpan<char> source,
    Span<byte> destination,
    out int charsConsumed, out int bytesWritten) { ... }
public static OperationStatus FromHexadecimal(
    char[] source,
    byte[] destination,
    out int charsConsumed, out int bytesWritten) { ... }
public static OperationStatus FromHexadecimal(
    char[] source, int sourceIndex,
    byte[] destination, int destinationIndex, int count,
    out int charsConsumed, out int bytesWritten) { ... }
```

```
public static OperationStatus FromHexadecimal(
    ArraySegment<char> source,
    ArraySegment<byte> destination,
    out int charsConsumed, out int bytesWritten) { ... }
}
```

JEREMY BARTON The list of possible overloads here is far from complete. Maybe you want to support offset and count for both the source data and the destination data (these examples showed accepting a source count, but destination was always from `destinationIndex` through the end of the array). Also, the string-accepting methods could have an offset and a length. My recommendation is to do the Span (or Memory)-based overloads and then either an entire-array or `ArraySegment` overload, depending on your target scenarios. You can always add more later, but using `ArraySegment` over `(T[], int, int)` saves on repeatedly writing offset and count argument validation.

✓ **CONSIDER** using Spans as a representation of buffers.

Methods that accept a `Span<T>`, or `ReadOnlySpan<T>`, can operate on a larger range of inputs than methods that accept an array: arrays, `stackalloc` data, pointer-based data, and (for `ReadOnlySpan<char>`) strings.

Because the callers can slice the buffer themselves, Span-based methods do not need overloads that accept an offset and a count. As a result, they are simpler, and are spared from the rote argument validation associated with those parameters.

```
public partial class BulkQueue<T> {
    // A simple, array-returning method
    public T[] DequeueMultiple(int limit) { ... }

    // An overload that lets the caller reuse an array
    public int DequeueMultiple(T[] destination) {
        if (destination == null) {
            throw new ArgumentNullException(nameof(destination));
        }
        return DequeueMultiple(destination.AsSpan());
    }
}
```

```
// An overload that lets the caller use part of an array
public int DequeueMultiple(
    T[] destination, int offset, int count) {

    if (destination == null) {
        throw new ArgumentNullException(nameof(destination));
    }
    // Defer the offset/count validation to AsSpan because the
    // parameter names match.
    return DequeueMultiple(destination.AsSpan(offset, count));
}

// The Span-based overload.
// The signature is simpler than (T[],int,int),
// but with the same functionality.
// This method also allows the destination to be stackalloc or
// pointer-based data (if those options are open to the type T).
public int DequeueMultiple(Span<T> destination) {
    int count = Count;
    int length = destination.Length;
    if (length < count) {
        _pending.AsSpan(_offset, count).CopyTo(destination);
        Count = 0;
        return count;
    }
    _pending.AsSpan(_offset, length).CopyTo(destination);
    Count -= destination.Length;
    _offset += destination.Length;
    return destination.Length;
}
}
```

If you have an existing type that already has array- or string-accepting methods, it generally makes sense to provide the Span- or Memory-based alternative as an overload on the same type, particularly for instance methods. For static methods, it may be more appropriate to have the array- or string-accepting methods on a high-level type, and to create a new low-level type for working with Span or Memory (section 2.2.4).

✓ **DO** use `ReadOnlySpan<T>` instead of `Span<T>` where possible.

The `System.ReadOnlySpan<T>` type does not support value mutation. When a method accepts a `ReadOnlySpan<T>` instead of a `Span<T>`, it provides a signal to the caller that the called method will not be

replacing the contents of the span. Accepting a `ReadOnlySpan<T>` when possible also allows more callers to the method, because all `Span<T>` values can be freely converted to `ReadOnlySpan<T>`, but `ReadOnlySpan<T>` values cannot be converted back to `Span<T>`.

`String` values logically represent an immutable `char[]`, and can be converted to `ReadOnlySpan<char>`, but not `Span<char>`.

```
public static long Sum(ReadOnlySpan<int> values) {
    long sum = 0;
    for (int i = 0; i < values.Length; i++) {
        sum += values[i];
    }
    return sum;
}

// Good: An array overload, either an existing method
// or one added concurrently
public static long Sum(int[] values) {
    if (values == null) {
        throw new ArgumentNullException(nameof(values));
    }
    return Sum(values.AsSpan());
}

// Bad: This method suggests it writes to the parameter, but it only
// reads. Writable spans are freely convertible to ReadOnlySpan.
public static long Sum(Span<int> values) {
    return Sum((ReadOnlySpan<int>)values);
}
```

X AVOID returning a `Span<T>` or `ReadOnlySpan<T>` unless the lifetime of the returned Span is very clear.

Because Span types can represent unmanaged memory, it is hard for a caller to understand the lifetime associated with the returned Span. Even with managed memory, it is entirely possible that the Span represents part of an array that can get repurposed for some other operation.

When the Span was given to you by your caller, it is acceptable to return a slice of it. Usually that means the Span is a parameter on the method

that returns it, but for a `ref`-like value type (`ref struct`) the Span could also come from constructor parameter.

```
public static ReadOnlySpan<char> TrimEnd(ReadOnlySpan<char> span) {
    int index = span.Length - 1;
    while (index >= 0 && char.IsWhiteSpace(span[index])) {
        index--;
    }
    return span.Slice(0, index + 1);
}
```

If the Span is representing stack memory, it should never be returned, as that memory will likely get overwritten during the next method invocation. Also, if the Span is describing unmanaged memory, then it now has no clear owner and would seem to be a memory leak.

■ **JEREMY BARTON** The C# compiler has flow analysis to try to prevent `stackalloc` values from being returned. The analysis isn't perfect, though, so you should still be careful when returning a Span from a method that uses `stackalloc`.

You should avoid returning a Span that represents a slice of a buffer that you have as a field in your type, except when that buffer was the argument to a constructor parameter. This is especially true if your type grows the buffer as needed, because any Span you returned will continue pointing to the previous buffer after you have moved to a new, larger one. As a result, your callers may potentially depend on observing changes in a Span you returned to them, only to have it fail sporadically based on your growth strategy, which can lead to hard-to-diagnose bugs for those callers. Stated more generally, returning a Span makes it easy for callers to depend on your internal implementation details, which can rapidly limit your ability to evolve your type.

■ **JEREMY BARTON** I'd encourage you to think of returning a Span pointing to one of your fields as the same as returning the field. They both over-expose your implementation. Returning a writable Span, like exposing a writable field, can let your caller corrupt your data or state.

The most common alternative to returning a Span is to accept a destination Span<T> as a parameter, and copy from your buffer to your caller's.

```
partial class SpanifiedQueue<T> {
    // Bad: Returning a ReadOnlySpan over-exposes details
    public ReadOnlySpan<T> DequeueMany(int count) {
        ...
        int start = _offset;
        _offset += count;
        return _buffer.AsSpan(start, count);
    }

    // Good: The data moves without exposing the implementation
    public int DequeueMany(Span<T> destination) {
        int count = Math.Min(destination.Length, Count);
        _buffer.AsSpan(_offset, count).CopyTo(destination);
        _offset += count;
        return count;
    }
}
```

- ✓ **DO** provide very clear documentation for the ownership rules on a Span that you return that did not come from the caller.

When you return a Span representing a buffer that your caller isn't already responsible for, you need to provide *very* clear documentation for what you expect the caller to do with it. This documentation needs to state when the value must no longer be used and what, if any, follow-up action is required on the caller's part.

Since reading and understanding the documentation is required to properly use such a method, it is no longer part of a self-documenting framework.

- ✓ **CONSIDER** returning a ReadOnlySpan<T> from a get-only property or parameter-less method to represent fixed data, when T is an immutable type.

File format headers, command text, and other fixed-value sequences that you would consider exposing as a const or a static read-only field if they were a single value can safely be exposed as ReadOnlySpan<T> members.

```
public static partial class ZipHeaders
{
    public static ReadOnlySpan<byte> LocalFile =>
        new byte[] { 0x50, 0x4b, 0x03, 0x04 };
    public static ReadOnlySpan<byte> DataDescriptor =>
        new byte[] { 0x50, 0x4b, 0x07, 0x08 };
    ...
}
```

■ **JEREMY BARTON** While `ZipHeaders.LocalFile` may look like it creates a new array each time it's invoked, the C# compiler has special support for methods—including property getters—that return a `ReadOnlySpan<byte>` and the method's entire body consists only of returning a new array initialized with literals (this optimization also applies for the other single-byte literal types: `sbyte` and `bool`). For Spans of any other type, you'll have to manually make a `private static readonly T[]` field and have the property return the array as a `ReadOnlySpan<T>`.

Usually you would expose these values as static properties. You may find virtual instance properties appropriate when each type in your hierarchy uses different values, but two instances of the same type don't vary. This pattern is used for the `Preamble` property on `System.Text.Encoding`.

```
public partial class Encoding {
    public virtual byte[] GetPreamble() { ... }

    // On any Encoding instance the sequence of bytes in the Preamble
    // value is always the same.
    public virtual ReadOnlySpan<byte> Preamble => GetPreamble();
}

public sealed partial class SomeEncoding {
    private static readonly byte[] s_preamble = { 0x05, 0xA0 };

    // This encoding type optimizes the Preamble property to always
    // use the same array.
    public override ReadOnlySpan<byte> Preamble =>
        _usePreamble ? s_preamble : default;

    // GetPreamble() still needs to return a defensive copy.
    public override byte[] GetPreamble() => Preamble.ToArray();
}
```

```
public static partial class ZipHeaders
{
    public static ReadOnlySpan<byte> LocalFile { get; } =
        new byte[] { 0x50, 0x4b, 0x03, 0x04 };
    public static ReadOnlySpan<byte> DataDescriptor { get; } =
        new byte[] { 0x50, 0x4b, 0x07, 0x08 };
    ...
}
```

✓ **CONSIDER** returning `System.Range` representing the bounds of a `Span` parameter, rather than a slice of the parameter.

You can convert a `Span<T>` value to a `ReadOnlySpan<T>` when you need to, but you can't convert a `ReadOnlySpan<T>` back to a `Span<T>`. So, if you call a method that accepts a `ReadOnlySpan<T>` and returns a slice of the parameter, you can't easily determine the equivalent slice of your writable `Span`.

```
// Unless you make overloads, callers trimming a writable Span
// now have a read-only slice when they may want a writable slice.
public static ReadOnlySpan<char> Trim(ReadOnlySpan<char> span) {
    int end = span.Length - 1;
    while (end >= 0 && char.IsWhiteSpace(span[end])) {
        index--;
    }
    int start = 0;
    while (start < end && char.IsWhiteSpace(span[start])) {
        start++;
    }
    return span.Slice(start, end - start + 1);
}

Span<char> span = ...;
ReadOnlySpan<char> readOnlyTrimmed = Trim(span);
Span<char> writableTrimmed = ???;
```

If you return a `Range` instead of a slice, your callers can do the slicing themselves and easily maintain the appropriate slice of their writable `Span`.

```
public static Range TrimEnd(ReadOnlySpan<char> span) {
    int index = span.Length - 1;
    while (index >= 0 && char.IsWhiteSpace(span[index])) {
        index--;
    }
```

```

int start = 0;
while (start < end && char.IsWhiteSpace(span[start])) {
    start++;
}
return new Range(start, end - start + 1);
}

Span<char> span = ...;
Range trimRange = Trim(span);
Span<char> writableTrimmed = span[trimRange];

```

You could solve the problem by providing an overload like `public static Span<char> Trim(Span<char> span)`, but then you should also create overloads for `ReadOnlyMemory<char>` and `Memory<char>`. By returning the Range you need to have only one method that works for all of these types, as well as any other types that have a Range-based indexer and can be converted to a `ReadOnlySpan<char>`.

- ✓ **DO** use `ReadOnlyMemory<T>` in place of `ReadOnlySpan<T>` in asynchronous methods.
- ✓ **DO** use `Memory<T>` in place of `Span<T>` in asynchronous methods.

Asynchronous methods largely represent the idea of “doing work later.” Since the `Span` types can’t generally be stored as part of the state of pending work, the appropriate `Memory` type should be used instead.

```

public partial class Stream {
    public virtual int Read(Span<byte> buffer) { ... }

    public virtual ValueTask<int> ReadAsync(
        Memory<byte> buffer,
        CancellationToken cancellationToken = default) { ... }

    public virtual int Write(ReadOnlySpan<byte> buffer) { ... }

    public virtual ValueTask<int> WriteAsync(
        ReadOnlyMemory<byte> buffer,
        CancellationToken cancellationToken = default) { ... }
}

```

- ✓ **DO** use `ReadOnlyMemory<T>` in place of `ReadOnlySpan<T>` for parameters when the purpose of the constructor or method is to store a reference to the buffer.

- ✓ **DO** use `Memory<T>` in place of `Span<T>` for parameters when the purpose of the constructor or method is to save a reference to the buffer.

In general, synchronous methods written by the .NET BCL team either process the data in an array before returning or store a copy of the array to perform later work on the array contents. Holding a reference to the buffer avoids the memory and time performance penalties of “defensive copies,” but creates a cost in that the caller has to keep the contents of the buffer stable until the operation completes. In addition, the concept of “complete” can vary for each type or method.

The `JsonDocument` class has a `Parse` method that maintains a reference to the input, without making a copy of the data. This improves performance, but places a responsibility on the caller to prevent data mutation. Since the parameter is a `Memory` type, the caller can expect that the `JsonDocument` instance will continue using the same buffer that was provided to the `Parse` method.

```
public partial class JsonDocument {
    // The returned JsonDocument holds a reference to the input
    // buffer and reads the JSON without making unnecessary copies
    public static JsonDocument Parse(
        ReadOnlyMemory<byte> utf8Json,
        JsonDocumentOptions options = default) { ... }
}
```

- ✗ **AVOID** overloading a method across `Span<T>` and `ReadOnlySpan<T>`, or `Memory<T>` and `ReadOnlyMemory<T>`.

If one method operates on a buffer in a read-only manner and another operates on it in a read-write manner, those methods should have different names to facilitate caller understanding on why the two methods behave differently.

```
public static int Sum(ReadOnlySpan<int> values) { ... }

// Why does this one need a writable Span?
// It must do something different, so it needs a different name.
public static void Sum(Span<int> values) { ... }
```

One exception to this guidance is when returning a slice of the input data, where the method returns the input data type, such as a `TrimEnd` method. As noted in earlier guidance, this overload set can be reduced to a single method if the method returns the slice indices instead of the sliced buffer.

```
public static ReadOnlySpan<char> TrimEnd(ReadOnlySpan<char> span) { ... }
public static Span<char> TrimEnd(Span<char> span) { ... }
```

 **AVOID** overloading a method across `Span<T>` and `Memory<T>`, or `ReadOnlySpan<T>` and `ReadOnlyMemory<T>`.

If one overload stores a reference to a buffer and the other operates on it without storing a reference, the methods should have different names.

```
public static void ProcessText(Span<char> span) { ... }
// What's the difference?
public static void ProcessText(Memory<char> memory) { ... }
```

One exception to this guidance is when returning a slice of the input data, where the method returns the input data type, such as a `TrimEnd` method. As noted in earlier guidance, this overload set can be reduced to a single method if the method returns the slice indices instead of the sliced buffer.

```
public static Span<char> TrimEnd(Span<char> span) { ... }
public static Memory<char> TrimEnd(Memory<char> memory) { ... }
```

 **CONSIDER** adding `Span-` or `Memory-`based alternatives to methods that accept arrays or strings.

The `Span` and `Memory` types support arrays, but also support additional kinds of buffers—such as `ReadOnlyMemory<char>` and `ReadOnlySpan<char>`, which can represent a portion of a `System.String` value. Providing `Span-` or `Memory-`based alternatives allows for a greater number of callers to the functionality you’re offering.

Generally, your users will expect you to provide the alternatives as method overloads on the same type as the array- or string-accepting method.

```

// Previously:
public void AppendData(byte[] data, int offset, int count) {
    if (data == null)
        throw new ArgumentNullException(nameof(data));
    // bounds argument validation here

    // implementation
}

// With a ReadOnlySpan<byte> overload:
public void AppendData(byte[] data, int offset, int count) {
    if (data == null)
        throw new ArgumentNullException(nameof(data));
    // bounds argument validation here so parameter names are correct

    AppendData(data.AsSpan(offset, count));
}

public void AppendData(ReadOnlySpan<byte> data) {
    // implementation, from previous version
}

```

For static methods, it can make sense to have the array- or string-accepting methods on a high-level type, and to create a new low-level type for working with Span or Memory (section 2.2.4).

✓ **CONSIDER** providing array-accepting alternatives for methods that accept Span or Memory values.

✓ **CONSIDER** providing System.String-accepting alternatives for methods that accept `ReadOnlySpan<char>` or `ReadOnlyMemory<char>`.

Not all CLR languages can use Spans as easily as C# does, and not all C# users are familiar with Spans. If you have an array- (or string-) based approach, you make it easier for these callers to call your method.

If you're designing a low-level component that is only intended to be used by other Span- or Memory-based operations, then it's OK to not provide the array or string alternative methods.

✓ **CONSIDER** providing an array-returning alternative to buffer-writing methods.

Novice developers, short-lived applications, one-time operations, and prototyping all benefit from having an array-returning alternative to

buffer-writing methods by getting the data they want without the developer overhead of managing buffers.

For methods that write to `Span<char>`, `Memory<char>`, or `char[]`, the “array-returning” method might be better designed as returning a `System.String`. For example, a `ToBase64` operation might have a String-returning form and a `Span<char>`-writing form.

```
public static string ToBase64(byte[] source) { ... }

public static int ToBase64(
    ReadOnlySpan<byte> source, Span<char> destination);
```

If you’re designing a low-level component that is only intended to be used by other buffer-writing operations, then it’s OK to not provide the array or string alternative methods.

- ✓ **DO** perform normal parameter validation in array or string alternatives to `Span` or `Memory` methods.

The `Span` types and `Memory` types are all value types, where the default value represents an empty buffer. Arrays—and strings—are reference types, where the default value (`null`) represents a lack of a buffer.

Consider your array, or string, alternative method on its own when deciding whether you want to treat a `null` input and an empty input as the same thing. If you decide that `null` values are not appropriate, throw the normal `ArgumentNullException`.

If your array, or string, alternative member accepts offset and count parameters, you should validate them manually so that the `ArgumentException` uses the parameter name that matches your method.

- ✓ **DO** prefer the name “source” for the input buffer parameter of methods that accept an output buffer and a single input buffer.
- ✓ **DO** prefer the name “destination” for the output buffer parameter of methods that write to a single buffer.

A buffer-writing variant for an existing array-returning method should maintain the existing name for the source parameter. As with all other

naming suggestions, if the suggested names are confusing in context, or if a different name would clarify the parameter's role, use your judgment to pick the best name.

In the RSA class, the name of the input buffer to the `SignData` method is "data," and the name of the input buffer to `SignHash` is "hash," to reinforce the different treatment that the buffers get between those two methods. The `TrySignData` and `TrySignHash` methods continue to use the names "data" and "hash" instead of "source" to be consistent with the data-returning method variants, but both still use "destination" as the name for the output buffer because no domain-specific clarity is required.

```
public partial class RSA {
    public virtual byte[] SignData(
        byte[] data,
        HashAlgorithmName hashAlgorithm,
        RSASignaturePadding padding) { ... }

    public virtual bool TrySignData(
        ReadOnlySpan<byte> data,
        Span<byte> destination,
        HashAlgorithmName hashAlgorithm,
        RSASignaturePadding padding,
        out int bytesWritten) { ... }
}
```

9.12.1 Data Transformation Operations

One of the most powerful uses of the `Span` and `Memory` types is in data transformation operations, such as transforming a string to UTF-8 bytes, bytes into Base64 text, or numeric values into a sequence of big-endian—or little-endian—bytes, or changing the casing of text. These operations always involve some input values, either as a formal input parameter or as part of an instance's state, as well as some output values. Before the introduction of `Span<T>` and `Memory<T>`, the most common way to return the output values was to return them in an array. A few operations also accepted a destination array as a parameter, allowing the same array to be used for multiple calls to the method.

```
public partial class Encoding {
    public virtual byte[] GetBytes(char[] chars, int index, int count){...}

    public abstract int GetBytes(
        char[] chars, int charIndex, int charCount,
        byte[] bytes, int byteIndex);
}
```

In general, transformation APIs are used to build bigger messages, and in some applications a significant amount of time and memory was wasted copying the returned array into some other buffer. The transformation APIs that accepted a destination array allowed a caller to avoid both the redundant memory allocation and the redundant copy, but at the expense of a more complex signature. Using a Span to represent a slice of the buffer reduces the complexity of the signature while still maintaining the performance benefits of writing to a destination instead of returning the output as a new array.

```
public partial class Encoding {
    public virtual int GetBytes(
        ReadOnlySpan<char> source,
        Span<byte> destination) { ... }
}
```

Span- and Memory-based transformation APIs come in three major patterns, discussed in the next sections. The correct pattern or patterns to use for your API depend on whether any of the input values can be invalid for your transformation, how predictable the output size of your transformation is, and whether your transformation can make incremental progress.

The simplest transformation API pattern is for transformations that don't have invalid inputs and produce an output size that is easily approximated from the input size, like the `Encoding.GetBytes` method. When you call a method with this pattern, you're expected to ensure that enough space is available in the buffer you are providing as the destination before calling the method, as in the following example, which writes a length-prepended string:

```
Encoding encoding = Encoding.UTF8;
int totalLength = sizeof(int) + encoding.GetMaxBytes(text.Length);
```

```
byte[] buf = new byte[totalLength];
BinaryPrimitives.WriteInt32BigEndian(buf, text.Length);
int written = encoding.GetBytes(text, buf.AsSpan(sizeof(int)));
WriteOutput(buf.AsSpan(0, written + sizeof(int)));
```

JAN KOTAS The simple output size approximations tend to be very conservative. For example, the approximation returned by `UTF8Encoding.GetMaxBytes` can be three times more than the required buffer size. It makes the simple transformation API pattern applicable only when the data has known maximum size, typically less than 10 kB.

The next pattern is best for transformations that don't have invalid input, but estimating the output size is difficult. Data compression is one such transformation, because the output size will vary widely depending on the input values. This pattern uses a specialized version of the Try Pattern (section 7.5.2) called the Try-Write pattern.

```
partial class DataCompressor {
    public static bool TryCompressFile(
        ReadOnlySpan<byte> source,
        Span<byte> destination,
        out int bytesWritten) { ... }
}
```

When you call a Try-Write method, you generally first call it with whatever buffer you have available for the destination. If the method returns false, you need to try again with a bigger buffer. Unless you have insight into a better technique for knowing how much more space is required, the most common technique is to double the size of the buffer and try again (“double and retry”).

```
byte[] output = ...;
int offset = ...;

while (!DataCompressor.TryCompressFile(
    uncompressed, output.AsSpan(offset), out int bytesWritten)) {

    Array.Resize(ref output, output.Length * 2);
}

offset += bytesWritten;
```

The last major pattern is best for transformations that can process partial inputs and can return partial results. Parsing hexadecimal text into a sequence of bytes is one such transformation, because it can write output values for every successfully read pair of characters and report where it stopped. This final pattern uses the `OperationStatus` enumeration as the method return type, to report back to callers why processing ended.

```
partial class TextProcessing {
    public static OperationStatus ParseHexadecimal(
        ReadOnlySpan<char> source,
        Span<byte> destination,
        out int charsConsumed,
        out int bytesWritten) { ... }
}
```

When you call an `OperationStatus`-based method, you use the return value to determine what further action you need to take. For example, the `DestinationTooSmall` response means that the transformation stopped because the next result was larger than the amount of space remaining in the destination buffer. For network protocols, that might be a time to send the current data over the network and start back at the beginning of the buffer, or you might want to grow your buffer and continue.

```
byte[] output = ...;
int outputOffset = ...;
int inputOffset = ...;

while (true) {
    OperationStatus status = TextProcessing.ParseHexadecimal(
        text.Span(inputOffset), output.Span(outputOffset),
        out int charsConsumed, out int bytesWritten);

    outputOffset += bytesWritten;
    inputOffset += charsConsumed;

    if (status == OperationStatus.Done) {
        break;
    }
    else if (status == OperationStatus.DestinationTooSmall) {
        Send(output.Span(0, outputOffset));
        outputOffset = 0;
    }
}
```

```

    else if (...) {
    }
    else {
        throw new InvalidOperationException();
    }
}

```

The remainder of this section discusses guidelines that apply to all transformation-based APIs. Simple transformation APIs are discussed in section 9.12.2, the Try-Write pattern is discussed in section 9.12.3, and OperationStatus-based transformations are discussed in section 9.12.4.

✓ **DO** position the input buffer parameter (“source”) as the first method parameter, when there is an explicit input buffer parameter.

✓ **DO** position the output buffer parameter (“destination”) as the first method parameter after the source parameter(s).

Some methods—instance methods, in particular—may provide the source from ambient state; therefore, no formal source parameter is required. When a formal source parameter is required, it should be the first parameter, followed by the destination parameter, and then any other parameters.

```

public partial class IncrementalHash {
    // No source parameter is required,
    // because the values were built by other instance methods.
    public bool TryGetHashAndReset(
        Span<byte> destination,
        out int bytesWritten) { ... }
}

public partial class ByteOperations {
    // One source, one destination
    // (destination is read-write in this example).
    public static int XorInto(
        ReadOnlySpan<byte> source,
        Span<byte> destination) { ... }

    // Two source parameters, one destination.
    // The source parameters come first.
    public static int Xor(
        ReadOnlySpan<byte> left,
        ReadOnlySpan<byte> right,
        Span<byte> destination) { ... }
}

```

```

// One source, one destination, one other parameter.
public static int CombineInto(
    ReadOnlySpan<byte> source,
    Span<byte> destination,
    BitOperation operation) { ... }

// Two sources, one destination, one other parameter.
public static int Combine(
    ReadOnlySpan<byte> left,
    ReadOnlySpan<byte> right,
    Span<byte> destination,
    BitOperation operation) { ... }

}

public partial class AesGcm {
    // Two (or three) inputs, two outputs.
    public void Encrypt(
        ReadOnlySpan<byte> nonce,
        ReadOnlySpan<byte> plaintext,
        Span<byte> ciphertext,
        Span<byte> tag,
        ReadOnlySpan<byte> associatedData = default) { ... }
}

```

When a buffer-writing method accepts an array and an offset, the offset parameter should immediately follow the array it is associated with. This is one of the only reasons for inserting a parameter between the source and destination parameters.

```

public static int XorInto(
    ReadOnlySpan<byte> source,
    Span<byte> destination) { ... }

public static int XorInto(
    byte[] source,
    byte[] destination) { ... }

public static int XorInto(
    byte[] source, int sourceIndex,
    byte[] destination, int destinationIndex) { ... }

```

- ✓ **DO** let the caller know the number of elements written into each destination buffer.

Whenever you write into a caller-provided buffer, you need to inform the caller how much of the buffer contains the result. For simple

transformations, the best way to do that is by returning the number of elements written. If the return value is already needed for something else—like the Boolean return for Try-Write (section 9.12.3) or an OperationStatus result (section 9.12.4)—then use an out parameter instead.

When your method is always going to fill the entire buffer, as occurs with `RandomNumberGenerator.Fill`, the caller already knows the number of elements written, so you don't need to explicitly return it.

```
partial class RandomNumberGenerator {
    // No return value required; it always writes
    // destination.Length bytes
    public static void Fill(Span<byte> destination) { ... }
}

public partial class Encoding {
    // Returns the number of bytes written to destination
    public virtual int GetBytes(
        ReadOnlySpan<char> source,
        Span<byte> destination) { ... }
}
```

✓ **CONSIDER** naming an out parameter that reports the number of values written by a buffer-writing method “bytesWritten,” “charsWritten,” or “valuesWritten” (as appropriate to the data type).

One simple example of this pattern is the `TryFormat` method on `System.Version`:

```
public bool TryFormat(
    Span<char> destination, out int charsWritten) { ... }
```

9.12.2 Writing Fixed or Predetermined Sizes to a Buffer

You can use the simplest form of a buffer-writing method when the number of elements for your operation is permanently fixed, as with a data hashing or checksum algorithm, or can easily be estimated by the length of the input, as when converting bytes to Base64 text.

```
partial class Crc64 {
    public const int ChecksumSize = 8;
```

```

public static int ComputeChecksum(
    ReadOnlySpan<byte> source,
    Span<byte> destination) { ... }

partial class TextTransforms {
    public int GetBase64TextSize(int sourceLength) =>
        (sourceLength + 2) / 3 * 4;

    public static int ToBase64(
        ReadOnlySpan<byte> source,
        Span<char> destination) { ... }
}

```

- ✓ **DO** provide a simple buffer-writing method for operations that write a small, effectively fixed, number of elements to the destination buffer.

If it's easy for your callers to ensure their buffer is large enough before using your operation, then you should provide a simple signature so they can avoid all of the ceremony associated with the more complex buffer-writing patterns.

For example, a CRC-64 operation will always write an 8-byte output (64 bits), so there should be a simple method for computing a CRC-64 value into a buffer.

```

partial class Crc64 {
    public const int ChecksumSize = 8;

    public static int ComputeChecksum(
        ReadOnlySpan<byte> source,
        Span<byte> destination) {

        if (destination.Length < ChecksumSize) {
            throw new ArgumentException(..., nameof(destination));
        }
        ...
        return ChecksumSize;
    }
}

```

9.12.3 Writing Values to Buffers with the Try-Write Pattern

While some buffer-writing methods operate on an unchanging, predictable number of output values, many others must deal with variable output

values. Since the caller provides the destination buffer, there is the possibility that the destination will not be big enough to hold the response. To accommodate this possibility, we developed a specialized version of the Try Pattern (section 7.5.2) for methods that write variable data to a single buffer. The Try-Write Pattern is useful for communicating that a destination is too small without throwing an exception, allowing the caller to obtain a larger buffer and try again.

The Try-Write Pattern consists of using the prefix “Try,” a Boolean return value that is true on success and false when the destination is too small, an optional input buffer, an output buffer, and an out parameter that reports the number of elements written to the output buffer.

```
partial class IPAddress {
    public bool TryWriteBytes(
        Span<byte> destination,
        out int bytesWritten) { ... }

}

partial class BrotliEncoder {
    public static bool TryCompress(
        ReadOnlySpan<byte> source,
        Span<byte> destination,
        out int bytesWritten) { ... }

}
```

Callers of Try-Write methods are expected to try again with a bigger buffer in a grow-and-retry loop whenever a Try-Write method returns false. Unless a better heuristic specific to the operation is available, most callers will do something like double the size of their buffer on each failed call. Usually a caller will call a Try-Write method without checking the amount of space remaining in the destination buffer.

```
byte[] buf = new byte[256];

while (!obj.TryWrite(buf, out int bytesWritten)) {
    Array.Resize(ref buf, buf.Length * 2);
}

int writeOffset = bytesWritten;
while (!obj2.TryWrite(buf.AsSpan(writeOffset), out bytesWritten)) {
    Array.Resize(ref buf, buf.Length * 2);
}
writeOffset += bytesWritten;
...
```

- ✓ **DO** use the Try-Write Pattern for buffer-writing methods that write variable amounts of data when the amount of data is small, or the operation cannot meaningfully produce partial results.

The Try-Write Pattern is better than a simple buffer-writing method when the number of output elements varies greatly for two inputs of the same size. It is partially built on a caller “hoping” that the destination buffer is big enough, and when it isn’t, getting a bigger buffer and trying again.

However, every time your caller grows the buffer, the caller pays the cost of copying all of the existing contents into the new buffer. The performance penalty is worse if method has to do a nontrivial amount of work before it can determine the destination is too small. If you expect a caller to have to get a bigger buffer several times before it can use your operation successfully, the Try-Write Pattern may be a performance trap. Later guidance in this section addresses avoiding, or reducing, the performance concerns in these situations.

- ✓ **DO** use the prefix “Try” and a Boolean return type for methods implementing the Try-Write Pattern.
- ✓ **DO** report the number of values written by a Try-Write method via an `out` parameter.
- ✓ **DO** return `false` from a Try-Write method if, and only if, the destination is too small.
- ✓ **DO** throw an exception, instead of returning `false`, for any error other than the destination being too small in a Try-Write method.

In the Try-Write Pattern, `true` means success and `false` means the destination is not big enough. This consistency allows callers to reliably interpret `false` as “try again with a bigger buffer.” For any other error condition (e.g., invalid inputs, invalid object state), the correct response is to throw an appropriate exception (see section 7.2).

- ✓ **DO** provide a way of determining a sufficient output length if your Try-Write method can produce large results.

If a caller has to call your Try-Write method more than twice with successively larger buffers, the attempts between the first call and the successful call probably involved needlessly copying data into bigger and bigger buffers. If you have a method that predicts the output size of your operation, your caller can use that information to grow the buffer once, and succeed on the second call.

```
public partial struct BigInteger {
    public int GetByteCount(bool isUnsigned=false) {
        ...
    }
    ...
    BigInteger value = ...;
    byte[] buf = ...;
    int offset = ...;

    while (!value.TryWriteBytes(buf.AsSpan(offset), out bytesWritten)){
        int growBy = Math.Max(buf.Length, value.GetByteCount());
        Array.Resize(ref buf, buf.Length + growBy);
    }
}
```

While `BigInteger.GetByteCount` and `Encoding.GetByteCount` both produce exact answers, you can also—or instead—provide a faster method that returns the maximum possible value for the result size, like `Encoding.GetMaxByteCount`.

```
public partial class Encoding {
    public abstract int GetMaxByteCount(int charCount);
    public abstract int GetMaxCharCount(int byteCount);

    public virtual int GetByteCount(ReadOnlySpan<char> chars) {
        ...
    }

    public virtual int GetCharCount(ReadOnlySpan<byte> bytes) {
        ...
    }
}

int longestString = strings.Max(s => s.Length);
byte[] buf = new byte[encoding.GetMaxByteCount(longestString)];
foreach (string s in strings) {
    int written = encoding.GetBytes(s, buf);
    SendMessage(buf.Slice(0, written));
}
```

- ✓ **CONSIDER** providing an API to compute a sufficient output buffer size for Try-Write methods, even when the output size is small.

If you know all of the sizes, or an upper bound for all of the sizes, of the output from a series of transformations, you can pre-allocate a big enough buffer to avoid needing to grow.

For example, writing a row of Boolean values into a CSV file can pre-allocate a buffer based on knowing that the longest output from `bool.TryFormat` is from the string "false". Pre-allocating buffers is more difficult for TryFormat methods on more complex types, unless those types expose their maximum format length via API.

```
private void WriteRow(ReadOnlySpan<bool> row, ref char[] buf) {
    int length = (bool.FalseString.Length + 1) * row.Length;
    Array.Resize(ref buf, Math.Max(length, buf.Length));
    int offset = 0;
    foreach (bool b in row) {
        while (!b.TryFormat(buf.AsSpan(offset), out int written)) {
            Array.Resize(ref buf, buf.Length * 2);
        }
        offset += written;
        buf[offset] = ',';
    }
    WriteLine(buf.AsSpan(0, offset));
}
```

- ✓ **CONSIDER** providing an exception-throwing alternative for each member using the Try-Write Pattern.

Callers that make use of pre-sized or over-sized buffers can benefit from simplifying their code and eliminating the false branch altogether, without resorting to getting into the habit of calling a Try-Write method and not checking the return value.

By having an alternative that returns the count of values written instead of a Boolean value, a caller can write linear code and still be protected from logic errors in the pre-sizing (or a time-of-check to time-of-use disagreement from data mutation).

For example, suppose a type has both a `TryWriteBytes` method and a `WriteBytes` method:

```
public bool TryWriteBytes(
    Span<byte> destination, out int bytesWritten) { ... }

public int WriteBytes(Span<byte> destination) {
    if (!TryWriteBytes(destination, out int bytesWritten)) {
        throw new ArgumentException(SR.TooSmall, nameof(destination));
    }
    return bytesWritten;
}
```

A caller can simplify the invocation from an if-statement with a (theoretically) unreachable branch to a simple call:

```
// If only TryWriteBytes is available
if (!target.TryWriteBytes(destination, out int bytesWritten)) {
    Debug.Fail("TryWriteBytes failed with a pre-allocated buffer");
    throw new InvalidOperationException();
}

// Using WriteBytes
int bytesWritten = target.WriteBytes(destination);
```

The exception-throwing buffer-writing method should be written according to the guidance for predetermined-length writes in section 9.12.2.

 **CONSIDER** also using the Try-Write Pattern for methods that write fixed or predetermined amounts of data to a buffer.

The Try-Write Pattern is useful both when the amount of data to be written isn't known ahead of time and when called by an operation that has a variable amount of buffer remaining after successfully calling other Try-Write methods. Offering a Try-Write alternative for a fixed-write method may be valuable to callers, as it composes more fluidly with other Try-Write methods.

```
public partial class EventInstance {
    public Guid OperationId { get; private set; }
    public Guid EventTypeId { get; private set; }
    public DateTimeOffset Timestamp { get; private set; }
```

```

public bool TrySerialize(
    Span<byte> destination,
    out int bytesWritten) {

    bytesWritten = 0;
    if (!TryWriteGuid(OperationId, destination, out int written)) {
        return false;
    }
    int totalWritten = written;
    destination = destination.Slice(written);
    if (!TryWriteGuid(EventTypeId, destination, out written)) {
        return false;
    }
    totalWritten += written;
    destination = destination.Slice(written);
    if (!BinaryPrimitives.TryWriteUInt64LittleEndian(destination,
        TimestampToFileTime())) {
        return false;
    }
    bytesWritten = totalWritten + sizeof(ulong);
    return true;
}
}

```

9.12.4 Partial Writes to Buffers and OperationStatus

Methods that return an `OperationStatus` value are similar to Try-Write methods, in that they accept a destination buffer, indicate the amount of data written via an `out` parameter, and return a success or failure value. The difference is that `OperationStatus` methods can also support partial input, partial output, and a non-throwing way to report invalid input.

```

namespace System.Buffers {
    public enum OperationStatus {
        Done,
        DestinationTooSmall,
        NeedMoreData,
        InvalidData,
    }
}

```

The `OperationStatus` return value is well suited to transformation-based operations—such as text encoding—that take in input from an

indexable or enumerable source and write to an indexable destination. For example, a bytes-to-hexadecimal conversion could be written as follows:

```
private static readonly char[] s_hexChars =
    new[] {
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };

public static OperationStatus ToHexadecimal(
    ReadOnlySpan<byte> source,
    Span<char> destination,
    out int bytesConsumed,
    out int charsWritten) {

    int i = 0;
    int j = 0;
    for (; i < source.Length && j + 2 < destination.Length; i++, j += 2) {
        destination[j] = s_hexChars[(source[i] & 0xF0) >> 4];
        destination[j + 1] = s_hexChars[source[i] & 0x0F];
    }
    bytesConsumed = i;
    charsWritten = j;
    return i == source.Length ?
        OperationStatus.Done :
        OperationStatus.DestinationTooSmall;
}
```

The `FromHexadecimal` operation is more complicated than the `ToHexadecimal` operation, and can also return the two other states described by `OperationStatus`:

- **NeedsMoreData:** An odd number of hexadecimal characters were present in the input. All data prior to `charsConsumed` has already been processed into `destination`.
- **InvalidData:** A non-hexadecimal (and optionally non-whitespace) character was read. All data prior to `charsConsumed` has already been processed into `destination`.

Note that the `InvalidData` response does not necessarily exit with `source[charsConsumed]` as the invalid input. When processing an input of "1Q", for example, the method should exit with `charsConsumed` at 0,

because no portion of the data successfully was moved from input to output.

- ✓ **CONSIDER** using the `OperationStatus` method pattern for buffer-writing methods that can produce partial results.

Any algorithm where the input to can be broken down into smaller pieces—such as reading from or writing to Base64, or expanding compressed data—is a good candidate for an `OperationStatus` method. Such a method allows the caller to process whatever amount of data fits in the current buffer and continue without needing to first obtain a bigger buffer.

```
private int Expand(ReadOnlySpan<byte> source, Stream destination){  
    OperationStatus status = _expander.Expand(  
        source, _buf,  
        out int bytesConsumed, out int bytesWritten);  
  
    switch (status) {  
        case OperationStatus.Done:  
        case OperationStatus.DestinationTooSmall:  
        case OperationStatus.NeedMoreData:  
            destination.Write(_buf.AsSpan(0, bytesWritten));  
            return bytesConsumed;  
    }  
    throw new InvalidOperationException();  
}
```

- ✓ **DO** report the number of input values successfully processed by an `OperationStatus` method via an `out` parameter.
- ✓ **CONSIDER** naming the `out` parameter that reports the number of values successfully processed by an `OperationStatus` method “`bytesConsumed`,” “`charsConsumed`,” or “`valuesConsumed`” (as appropriate to the data type).
- ✓ **DO** report the number of output values successfully written by an `OperationStatus` method via an `out` parameter, declared after the `out` parameter that reports the number of input values that were successfully processed.

Following the earlier guidance that the input buffer and output buffer should be the first and second parameters, respectively, an Operation Status method should generally have a method signature like the following example to read a list of numbers:

```
public static OperationStatus ReadInt32List(  
    ReadOnlySpan<char> source,  
    Span<int> destination,  
    out int charsConsumed,  
    out int valuesWritten) { ... }
```

- ✓ **DO** include a Boolean parameter named “isFinalBlock,” if an Operation Status method needs to treat the last input block in a special way. The parameter should be optional and have a default value of true.

For example, ToBase64 needs to know whether the source buffer represents the final segment of the data so it can determine if it needs to treat an incomplete segment as NeedsMoreData or if it should write the padding characters. This would result in something like the following:

```
public static OperationStatus ToBase64(  
    ReadOnlySpan<byte> source,  
    Span<byte> destination,  
    out int bytesConsumed,  
    out int charsWritten,  
    bool isFinalBlock = true) {  
    ...  
    if (i < source.Length) {  
        if (j + 4 <= destination.Length) {  
            if (isFinalBlock) {  
                // write the padded final block  
                ...  
                return OperationStatus.Done;  
            }  
            return OperationStatus.NeedsMoreData;  
        }  
        return OperationStatus.DestinationTooSmall;  
    }  
    return OperationStatus.Done;  
}
```

✓ **CONSIDER** providing a Try-Write alternative to an `OperationStatus` method.

A Try-Write method is easily implemented in terms of an `OperationStatus` method, and has an easier method signature to work with when called from higher level Try-Write methods. Your callers may appreciate the simplified signature when they are in a method that cannot itself obtain a larger destination buffer, and the `InvalidData` state is unexpected or they can't meaningfully handle it.

```
public bool TryEncode(
    Span<char> destination, out int charsWritten) {

    if (s_header.TryCopyTo(destination)) {
        OperationStatus result = WriteNewlineHexadecimal(
            _data,
            destination.Slice(s_header.Length),
            newlineAfter: 32,
            out _,
            out int written);

        if (result == OperationStatus.Done) {
            charsWritten = written + s_header.Length;
            return true;
        }

        Debug.Assert(result == OperationStatus.DestinationTooSmall);
    }
    charsWritten = 0;
    return false;
}
```

If `WriteNewlineHexadecimal` had a Try-Write alternative method, the caller code could be simpler:

```
public bool TryEncode(
    Span<char> destination, out int charsWritten) {

    if (s_header.TryCopyTo(destination) &&
        TryWriteNewlineHexadecimal(
            _data,
            destination.Slice(s_header.Length),
            out int written)) {
```

```
        charsWritten = written + s_header.Length;
        return true;
    }
    charsWritten = 0;
    return false;
}
```

The TryWriteNewlineHexadecimal method itself is easily defined in terms of the OperationStatus-returning WriteNewlineHexadecimal method.

```
public static TryWriteNewlineHexadecimal(
    ReadOnlySpan<byte> source,
    Span<char> data,
    int newlineAfter,
    out int charsWritten) {

    OperationStatus status = WriteNewlineHexadecimal(
        source, data, newlineAfter, out _, out int written));

    switch (status) {
        case OperationStatus.Done:
            charsWritten = written;
            return true;
        case OperationStatus.DestinationTooSmall:
            charsWritten = 0;
            return false;
        default:
            // Depending on the operation, there may not be an
            // InvalidData or NeedMoreData response.
            throw new AppropriateException();
    }
}
```

You shouldn't have a Try-Write alternative method if your Operation Status method is capable of producing large results, but you can't provide an API to estimate the output length.

9.13 And in the End...

The process of creating a great framework is demanding. It requires dedication, knowledge, practice, and a lot of hard work. But in the end, it can be one of the most fulfilling jobs software engineers ever get to do. Large system frameworks can enable millions of developers to build software that was not possible before. Application extensibility frameworks can turn simple applications into powerful platforms and make them shine. Finally, reusable component frameworks can inspire and enable developers to take their applications beyond the ordinary. When you create a framework like that, please let us know. We would like to congratulate you.



C# Coding Style Conventions

UNLIKE THE FRAMEWORK Design Guidelines, these coding style conventions are not required and should be treated as a set of suggestions. The reason we don't insist on following these coding conventions is that they have no direct effect on most users of a framework.

There are many coding style conventions, each with its own history and philosophy. The conventions described here are, more or less, the conventions used by the .NET BCL team since moving .NET to open source. Since many other projects have set their coding style based on the .NET BCL team coding style guidelines, the third edition of this book replaced the previous contents of this appendix with this new de facto standard.

Since most developers have an easier time understanding, and contributing to, codebases that use a style with which they are already familiar, we encourage you to use this appendix as a starting point in making coding style conventions for any new open-source C# project that you create.

The style conventions here start from a few principles:

- If there's a choice between improving the clarity for the reader and the brevity of the writer, clarity wins (for example, `var` is only used in limited situations). This recognizes that code is read more than it is written and that a code reviewer typically has less context than a code author.

- If there are two ways to express something, the one that reduces noise in a future change is preferred (for example, add the trailing comma after the last declared value in an enum).
- The conventions are supposed to be simple to follow. This means both that there are few of them, and that style rules that need fewer exceptions are preferred. For example, Allman-style bracing rules have fewer exceptions than K&R-style bracing rules.

■ **JEREMY BARTON** The BCL team style guidelines, as of when the third edition of this book was written, are expressed in 17 rules and fit on one screen. The guidelines in this appendix are broken down into smaller pieces and include samples.

Please note that the numbered chapters in this book **do not** follow these guidelines. Instead, they use a more compact, “paper-friendly” style.

A.1 General Style Conventions

A.1.1 Brace Usage

The .NET BCL team uses Allman-style bracing. Braces are generally required, even when the language considers them optional, with a few exceptions described in the guidance.

✓ **DO** place the opening brace on the next line, indented to the same level as the block statement.

```
if (someExpression)
{
    DoSomething();
}
```

✓ **DO** align the closing brace with the opening brace.

```
if (someExpression)
{
    DoSomething();
}
```

- ✓ **DO** place the closing brace on its own line, except the end of a do..while statement.

```
if (someExpression)
{
    do
    {
        DoSomething();
    } while(someOtherCondition);
}
else if (someOtherExpression)
{
    ...
}
```

- ✗ **AVOID** omitting braces, even if the language allows it.

Braces should not be considered optional. Even for single-statement blocks, you should use braces. This increases code readability and maintainability.

```
for (int i = 0; i < 100; i++)
{
    DoSomething(i);
}
```

One exception to the rule is braces in case statements. These braces can be omitted, because the case and break statements indicate the beginning and the end of the block.

```
case 0:
    DoSomething();
    break;
```

- ✓ **CONSIDER** omitting braces in an argument validation preamble.

For all parts of the method preamble that are of the form `if (single LineExpression) { throw ... }`, the braces and vertical whitespace can be eliminated. Once you add a blank line, or have code other than `if...throw`, you've left the preamble and braces become required again. Multi-line conditions should ideally be factored out into methods to maintain the visual flow of the preamble.

```
public void DoSomething(SomeCollection coll, int start, int count)
{
    if (coll is null)
        throw new ArgumentNullException(nameof(coll));
    if ((uint)start < (uint)coll.Count)
        throw new ArgumentOutOfRangeException(nameof(start));
    if (coll.Count - count > start)
        throw new ArgumentOutOfRangeException(nameof(count));

    if (count == 0)
    {
        return;
    }

    ...
}
```

X DO NOT use the braceless variant of the using (dispose) statement.

The braced using statement provides a strong visual indicator of when the value is being released. Just as with locking, a disposable value should be disposed as soon as it is practical to do so. The braceless variant makes it too easy to add code that unnecessarily extends the scope of the disposable value, and changing the braceless version to the braced version adds unnecessary noise to a diff representation of a change.

Right:

```
using (Element element = GetElement())
{
    ...
}
```

Wrong:

```
using Element element = GetElement();
...
```

X AVOID using the braceless variant of the await using statement, except to simulate stacked await using statements with ConfigureAwait.

Until an addition to the C# language allows for simple stacked await using statements while also appropriately using ConfigureAwait (discussed in Chapter 9, sections 9.2.6.2 and 9.4.4.1), the braceless await using can be used to achieve the same level of conciseness,

provided it is used only in a series of variable declarations and await using statements in a fresh scope.

The braceless await using statement, as of C# 8.0, requires a variable declaration, but since the result of ConfigureAwait—a Configured AsyncDisposable value—is never going to be directly used in your method, you can use “var ignored” as an exception to the normal restrictions on the use of the var keyword.

Acceptable:

```
{
    Element element1 = GetElement(1);
    await using var ignored1 = element1.ConfigureAwait(false);
    Element element2 = GetElement(2);
    await using var ignored2 = element2.ConfigureAwait(false);

    ...
}
```

Wrong:

```
await using Element element1 = GetElement(1);
await using Element element2 = GetElement(2);
...
```

A.1.2 Space Usage

- ✓ **DO** use one space before and after the opening and closing braces when they share a line with other code. Do not add a trailing space before a line break.

```
public int Foo { get { return foo; } }
```

- ✓ **DO** use a single space after a comma between parameters.

```
Right: public void Foo(char bar, int x, int y)
Wrong: public void Foo(char bar,int x,int y)
```

- ✓ **DO** use a single space between arguments.

```
Right: Foo(myChar, 0, 1)
Wrong: Foo(myChar,0,1)
```

X DO NOT use spaces after the opening parenthesis or before the closing parenthesis.

```
Right: Foo(myChar, 0, 1)
Wrong: Foo( myChar, 0, 1 )
```

X DO NOT use spaces between a member name and an opening parenthesis.

```
Right: Foo()
Wrong: Foo ()
```

X DO NOT use spaces after or before square braces.

```
Right: x = dataArray[index];
Wrong: x = dataArray[ index ];
```

✓ DO use spaces between flow control keywords and the open parenthesis.

```
Right: while (x == y)
Wrong: while(x == y)
```

✓ DO use spaces before and after binary operators.

```
Right: if (x == y) { ... }
Wrong: if (x==y) { ... }
```

X DO NOT use spaces before or after unary operators.

```
Right: if (!y){ ... }
Wrong: if (! y){ ... }
```

A.1.3 Indent Usage

✓ DO use four consecutive space characters for indents.

X DO NOT use the tab character for indents.

- ✓ **DO** indent contents of code blocks.

```
if (someExpression)
{
    DoSomething();
}
```

- ✓ **DO** indent case blocks even if not using braces.

```
switch (someExpression)
{
    case 0:
        DoSomething();
        break;
    ...
}
```

- ✓ **DO** “outdent” goto labels one indentation level.

```
private void SomeMethod()
{
    int iteration = 0;

    start:
    ...

    if (...)

    {
        nested_label:
        iteration++;
        goto start;
    }

    ...
}
```

- ✓ **DO** indent all continued lines of a single statement one indentation level.

```
bool firstResult = list.Count > 0 ?
    list[list.Length - 1] > list[0] :
    list.Capacity > 1000;

string complex =
    _table[index].Property.Method(methodParameter).
        Method2(method2Parameter, method2Parameter2).
        Property2;
```

- ✓ **DO** chop before the first argument or parameter, indent one indentation level, and include one argument or parameter per line when a method declaration or method invocation is exceedingly long.

```
private void SomeMethod(
    int firstParameter,
    string secondParameter,
    bool thirdParameter,
    string fourthParameter)
{
    ...
    SomeOtherMethod(
        firstParameter * 100,
        fourthParameter,
        thirdParameter &&
            secondParameter.Length > fourthParameter.Length,
        22);
    ...
}
```

■ **JEREMY BARTON** We don't have a set line length as part of our style guidelines. I set my IDE to give a guide line at 120 characters, which is generally narrow enough to avoid horizontal scrolling for most IDE configurations and web-based code viewers. However, I treat this as a "soft" limit and violate it when adding the line break makes the line too hard to read.

A.1.4 Vertical Whitespace

- ✓ **DO** add a blank line before control flow statements.
- ✓ **DO** add a blank line after a closing brace, unless the next line is also a closing brace.

```
while (!queue.IsEmpty)
{
    Element toProcess = queue.Pop();
```

```

if (toProcess == null)
{
    continue;
}

if (!toProcess.IsDone)
{
    ...
}
}

```

- ✓ **DO** add a blank line after “paragraphs” of code where it enhances readability.

```

key = keyInfo.Key;

isShift = (keyInfo.Modifiers & ConsoleModifiers.Shift) != 0;

isAlt = (keyInfo.Modifiers & ConsoleModifiers.Alt) != 0;

isCtrl = (keyInfo.Modifiers & ConsoleModifiers.Control) != 0;

ch = ((keyLength == 1) ? _buffer[_startIndex] : '\0');

_startIndex += keyLength;
}

```

■ **JEREMY BARTON** I add a blank line when the next statement is subjectively “less” related to the previous one than the one before.

```

int size = value.BitCount;
size = (size + 7) / 8;

byte[] tmp = new byte[size];

...

```

A.1.5 Member Modifiers

The .NET BCL team considers this the canonical order of modifiers: `public`, `private`, `protected`, `internal`, `static`, `extern`, `new`, `virtual`, `abstract`, `sealed`, `override`, `readonly`, `unsafe`, `volatile`, `async`. The guidelines and examples are mostly just expressing this in more detail.

- ✓ **DO** always explicitly specify visibility modifiers.

Right:

```
internal class SomeClass
{
    private object _lockObject;
    ...
}
```

Wrong:

```
class SomeClass
{
    object _lockObject;
    ...
}
```

- ✓ **DO** specify the visibility modifier as the first modifier.

Right: protected abstract void Reset();
 Wrong: abstract protected void Reset();

- ✓ **DO** specify the static modifier immediately after visibility, for static members or static classes.

Right: private static readonly ConcurrentQueue<T> ...
 Wrong: private readonly static ConcurrentQueue<T> ...

- ✓ **DO** specify the extern method modifier immediately after the static modifier for an extern method.

Right: private static extern IntPtr CreateFile(...);
 Wrong: private extern static IntPtr CreateFile(...);

- ✓ **DO** specify the member slot modifier (new, virtual, abstract, sealed, or override) immediately after the static modifier (if specified).

Right: public static new SomeType Create(...)
 Wrong: public new static SomeType Create(...)

- ✓ **DO** specify the readonly modifier for fields or methods immediately after the member slot modifier (if specified).

Right: private new readonly object _lockObject = new object();
 Wrong: private readonly new object _lockObject = new object();

- ✓ **DO** specify the `unsafe` modifier for methods immediately after the `readonly` modifier (if specified).

Right: `public readonly unsafe Matrix Multiply(...)`
 Wrong: `public unsafe readonly Matrix Multiply(...)`

- ✓ **DO** specify the `volatile` modifier for fields immediately after the `static` modifier (if specified).

This guideline skips the `readonly` modifier for fields since `volatile` and `readonly` are mutually exclusive modifiers.

Right: `private static volatile int s_instanceCount;`
 Wrong: `private volatile static int s_instanceCount;`

- ✓ **DO** specify the `async` modifier as the last method modifier, when used.

Right: `public readonly async Task CloseAsync(...)`
 Wrong: `public async readonly Task CloseAsync(...)`

- ✓ **DO** use “`protected internal`” instead of “`internal protected`” for a member that can be called by derived types or types in the same assembly.

Right: `protected internal void Reset()`
 Wrong: `internal protected void Reset()`

- ✓ **DO** use “`private protected`” instead of “`protected private`” for a member that can be called by derived types within the same assembly.

Right: `private protected void Reset()`
 Wrong: `protected private void Reset()`

A.1.6 Other

- ✓ **DO** add the optional trailing comma after the last enum member.

- ✓ **DO** add the optional trailing comma after a property assignment from an object initializer, or addition via a collection initializer, when the initializer spans multiple lines.

Providing the trailing comma avoids unnecessary lines in a diff view of a change, where the formerly last line gains a comma.

Right:

```
public enum SomeEnum
{
    First,
    Second,
    Third,
}
```

Wrong:

```
public enum SomeEnum
{
    First,
    Second,
    Third
}
```

X DO NOT use this. unless absolutely necessary.

If the private members are named correctly, a “this.” should not be required except to invoke an extension method. The use of the underscore prefix on fields already provides a distinction between field access and variable access.

- ✓ DO** use language keywords (string, int, double, ...) instead of BCL type names (String, Int32, Double, ...), for both variable declarations and method invocation.

Right:

```
string[] split = string.Split(message, Delimiters);
int count = split.Length;
```

Wrong:

```
String split = String.Split(message, Delimiters);
Int32 count = split.Length;
```

X DO NOT use the var keyword except to save the result of a new, as-cast or “hard” cast.

In the core .NET libraries, we only use var when the type name is already specified. We don’t use it when calling factory methods, calling methods for which “everyone knows” the return type (e.g., String.IndexOf), or receiving the result of a TryParse method. We also do not

compel the use of `var` when using a new, as-cast, or “hard” cast; instead, we leave it as a judgment call on the part of the code author.

```
Acceptable (no var):
    StringBuilder builder = new StringBuilder();
    List<int> list = collection as List<int>;
    string built = ProcessList(list, builder);
    SomeType fromCache = cache.Get<SomeType>();

    if (int.TryParse(input, out int parsedValue)
    {
        ...
    }

Acceptable (maximal use of var):
    var builder = new StringBuilder();
    var list = collection as List<int>;
    string built = ProcessList(list, builder);
    SomeType fromCache = cache.Get<SomeType>();

    if (int.TryParse(input, out int parsedValue)
    {
        ...
    }

Wrong (uses var in places that are not permitted):
    var builder = new StringBuilder();
    var list = collection as List<int>;
    var built = ProcessList(list, builder);
    var fromCache = cache.Get<SomeType>();

    if (int.TryParse(input, out var parsedValue)
    {
        ...
    }
```

 **DO** use object initializers where possible.

This rule can be ignored when working with a type that has a required order of property assignment. As with most exceptions to a rule, however, a comment should explain why the code should not be changed to use object initializers. The object initializer does respect the order the properties are assigned, but it doesn’t “feel” as ordered as multiple assignment statements.

Right:

```
SomeType someType = new SomeType
{
    PropA = ...,
    PropB = ...,
    PropC = ...
}
```

Acceptable:

```
// PropB has to be set first
SomeType someType = new SomeType
{
    PropB = ...,
};
someType.PropA = ...;
someType.PropC = ...;
```

Not preferred:

```
SomeType someType = new SomeType();
someType.PropB = ...;
someType.PropA = ...;
someType.PropC = ...;
```

✓ **DO** use collection initializers where possible.

Right:

```
SomeCollection someColl = new SomeCollection
{
    first,
    second,
    third,
};
```

Not preferred:

```
SomeCollection someColl = new SomeCollection();
someColl.Add(first);
someColl.Add(second);
someColl.Add(third);
```

✓ **CONSIDER** using expression-bodied members when the implementation of a property or method is unlikely to change.

```
public partial class UnseekableStream : Stream
{
    public bool CanSeek => false;
```

```
public override long Seek(long offset, SeekOrigin origin) =>
    throw new NotSupportedException();
}
```

- ✓ **DO** use auto-implemented properties when the implementation is unlikely to change.

```
internal partial class SomeCache
{
    internal long HitCount { get; private set; }
    internal long MissCount { get; private set; }
}
```

- ✓ **DO** restrict code to ASCII characters, and use Unicode escape sequences (\uXXXX) when needed for non-ASCII values.

Following the naming rules from Chapter 3 means that a class representing a résumé (a one-sheet description of someone’s work and education history) would be named “Resume” (without the diacritical marks). If you still want to name it “Résumé,” the e-acute can be written as “\u00E9”.

```
public partial class R\u00E9sum\u00E9
{
    ...
}

...
string nihongo = "\u65E5\u672C\u8A9E";
```

- ✓ **DO** use the nameof(. . .) syntax, instead of a literal string, when referring to the name of a type, member, or parameter.

```
Right: throw new ArgumentNullException(nameof(target));
Wrong: throw new ArgumentNullException("target");
```

- ✓ **DO** apply the readonly modifier to fields when possible.

Be careful when applying the readonly modifier to fields where the field type is a struct. If the struct isn’t declared as readonly, then property and method invocations might make local copies of the value and create a performance problem.

- ✓ **DO** use `if...throw` instead of `assignment-expression-throw` for argument validation.

Using `expression-throw` in an assignment statement can lead to subtle bugs in finalizers where some fields are set but others aren't. While finalizers aren't common, always using `if...throw` generalizes without exception to types with and without finalizers, as well as to both methods and constructors.

Right:

```
public SomeType(SomeOtherType target, ...)  
{  
    if (target is null)  
        throw new ArgumentNullException(nameof(target));  
    ...  
  
    _target = target;  
    ...  
}
```

Wrong:

```
public SomeType(SomeOtherType target, ...)  
{  
    _target = target ??  
        throw new ArgumentNullException(nameof(target));  
  
    ...  
}
```

A.2 Naming Conventions

In general, we recommend following the Framework Design Guidelines for naming identifiers. However, some additional conventions and exceptions to using the Framework Design Guidelines arise when naming internal and private identifiers.

- ✓ **DO** follow the Framework Design Guidelines for naming identifiers, except for naming private and internal fields.
- ✓ **DO** use PascalCasing for namespace, type, and member names, except for internal and private fields.

```
namespace Your.MultiWord.Namespace
{
    public abstract class SomeClass
    {
        internal int SomeProperty { get; private set; }
        ...
    }
}
```

- ✓ **DO** use PascalCasing for const locals and const fields except for interop code, where it should match the name from the called code exactly.

```
internal class SomeClass
{
    private const DefaultListSize = 32;

    private bool Encode()
    {
        const int RetryCount = 3;
        const int ERROR_MORE_DATA = 0xEA;

        ...
    }
}
```

- ✓ **DO** use camelCasing for private and internal fields.

- ✓ **DO** use a prefix “`_`” (underscore) for private and internal instance fields, “`s_`” for private and internal static fields, and “`t_`” for private and internal thread-static fields.

These prefixes are very valuable to a code reviewer. Reading from, or writing to, a field indicates why a method isn’t declared as `static`. Writes to a field indicate a persisted state. Reading from, or writing to, a static field indicates a shared persisted state and a potential thread-safety problem. Thread-static fields require even more attention, because they may lose their value after an `await`—or have never been initialized on this thread.

```
internal partial class SomeClass
{
    [ThreadStatic]
    private static byte[] t_buffer;
```

```
    private static int s_instanceCount;  
  
    private int _instanceId;  
}
```

- ✓ **DO** use camelCasing for local variables.
- ✓ **DO** use camelCasing for parameters.
- ✗ **DO NOT** use Hungarian notation (i.e., do not encode the type of a variable in its name).

A.3 Comments

Comments should be used to describe the intent, algorithmic overview, and logical flow. It would be ideal if, from reading the comments alone, someone other than the author could understand the function's behavior and purpose. Although there are no minimum comment requirements and certainly some very small routines need no commenting at all, it is desirable for most nontrivial routines to have comments reflecting the programmer's intent and approach.

- ✗ **DO NOT** use comments unless they describe something not obvious to someone other than the developer who wrote the code.
- ✗ **AVOID** multiline syntax (`/* ... */`) for comments. The single-line syntax (`// ...`) is preferred even when a comment spans multiple lines.

```
// Implements a variable-size list that uses an array of objects  
// to store the elements. A List has a capacity, which is the  
// allocated length of the internal array. As elements are added  
// to a List, the capacity of the List is automatically increased  
// as required by reallocating the internal array.  
//  
public class List<T> : IList<T>, IList  
{  
    ...  
}
```

- ✗ **DO NOT** place comments at the end of a line unless the comment is very short.

✗ **AVOID** placing comments at the end of a line even when the comment is very short.

```
//Avoid  
public class ArrayList  
{  
    private int count; // -1 indicates uninitialized array  
}
```

✗ **AVOID** writing “I” in comments.

Code ownership changes over time, particularly in long-lived software projects, which leaves “I” ambiguous when not consulting code history.

- Rather than “I found that ...,” say something like “Based on the 2019 aggregated TPS report”
- Use “we” to refer to the team or project, collectively. “We already checked for null, so”
- Consider personifying the method: “This method wants a source and a destination; the destination needs to be at least twice as large as the source.”
- Consider using passive voice where necessary: “The inputs were all validated to be monotonic and even, so now”

A.4 File Organization

✗ **DO NOT** have more than one public type in a source file, unless they differ only in the number of generic parameters or one is nested in the other.

Multiple internal types in one file are allowed, though there is a preference for “one (top-level) type per file.”

✓ **DO** name the source file with the name of the public type it contains.

For example, the `String` class should be in a file named “`String.cs`” and the `List<T>` class should be in a file named “`List.cs`”.

✓ **DO** name the source file for a partial type with the name of the primary file and a logical description of the contents of the file separated by a “.” (period), such as “`JsonDocument.Parse.cs`”.

✓ **DO** organize the directory hierarchy just like the namespace hierarchy.

For example, the source file for `System.Collections.Generic.List<T>` should be in the `System\Collections\Generic` directory.

For assemblies that have a common namespace prefix for all types, removing empty top-level directories is a usual practice. Thus, a type named “`SomeProject.BitManipulation.Simd`” from an assembly with a common prefix of “`SomeProject`” would usually be found at `<project root>\BitManipulation\Simd.cs`.

✓ **DO** group members into the following sections in the specified order:

- All const fields
- All static fields
- All instance fields
- All auto-implemented static properties
- All auto-implemented instance properties
- All constructors
- Remaining members
- All nested types

✓ **CONSIDER** grouping the remaining members into the following sections in the specified order:

- Public and protected properties
- Methods
- Events
- All explicit interface implementations
- Internal members
- Private members

- ✓ **DO** place the using directives outside the namespace declaration.

```
using System;

namespace System.Collections
{
    ...
}
```

- ✓ **DO** sort the using directives alphabetically, but place all System namespaces first.

```
using System;
using System.Collections.Generic;
using Microsoft.Win32.SafeHandles;

namespace System.Diagnostics
{
    ...
}
```

This page intentionally left blank



Obsolete Guidance

EVERY NEW FEATURE has the potential to obsolete something that was once considered fundamental to a framework, such as Task replacing IAsyncResult. Even without a real replacement, some features simply don't age well and get deprecated through lack of consistent use because the scenarios are less relevant, such as Code Access Security (CAS). But obsolescence is relative, so this appendix captures the guidance that is no longer generally applicable to new frameworks and new framework features that might be relevant to a project you work on.

The section numbers in this appendix reflect the section number that the content had from the last edition where it was still active guidance. For example, if you had a note that referenced section 3.8 for the rules for naming resources, it will now be B.3.8 (Appendix B, formerly Chapter 3, section 8).

Even if you don't need the guidance from this appendix, you may be interested to see what has changed. At the beginning of every section that was archived to this appendix, a marker, labeled “☞ WHY THIS IS HERE,” describes why the guidance is no longer applicable to new frameworks or new framework features.

Guidance that was merely modified from previous editions is not contained within this appendix.

B.3 Obsolete Guidance from Naming Guidelines

B.3.8 Naming Resources

■  **WHY THIS IS HERE** During the updates to this book for the third edition, we observed that resource strings are rarely, if ever, exposed as public APIs. We codified that in guidance to not directly expose localizable resources—anywhere that a localizable resource is exposed should be done on a case-by-case basis with the same level of thought as all public (or protected) members.

Because localizable resources can be referenced via certain objects as if they were properties, the naming guidelines for resources are similar to property guidelines.

✓ **DO** use PascalCasing in resource keys.

✓ **DO** provide descriptive rather than short identifiers.

Keep them concise where possible, but do not sacrifice readability for space.

✗ **DO NOT** use language-specific keywords of the main CLR languages.

✓ **DO** use only alphanumeric characters and underscores in naming resources.

✓ **DO** use the following naming convention for exception message resources.

The resource identifier should be the exception type name plus a short identifier of the exception:

```
ArgumentExceptionIllegalCharacters  
ArgumentExceptionInvalidName  
ArgumentExceptionFileNameIsMalformed
```

B.4 Obsolete Guidance from Type Design Guidelines

B.4.1 Types and Namespaces

B.4.1.1 Standard Subnamespace Names

 **WHY THIS IS HERE** Between the second and third editions of this book, Code Access Security (CAS) became deprecated, removing the need for the guidance in 4.1.1.2. Primary Interop Assemblies are somewhat of a rarity, making 4.1.1.3 more confusing than insightful. 4.1.1.1 instead was replaced with a single sentence.

The guidance in B.4.1.1 is not contradictory to the guidance in section 4.1 of the third edition; it is simply no longer generally applicable.

Types that are rarely used should be placed in subnamespaces to avoid cluttering the main namespaces. We have identified several groups of types that should be separated from their main namespaces.

B.4.1.1.1 The .Design Subnamespace

Design-time-only types should reside in a subnamespace named `.Design`. For example, `System.Windows.Forms.Design` contains Designers and related classes used to do design of applications based on `System.Windows.Forms`.

```
System.Windows.Forms.Design  
System.Messaging.Design  
System.Diagnostics.Design
```

 **DO** use a namespace with the “`.Design`” suffix to contain types that provide design-time functionality for a base namespace.

B.4.1.1.2 The .Permissions Subnamespace

Permission types should reside in a subnamespace named `.Permissions`.

 **DO** use a namespace with the “`.Permissions`” suffix to contain types that provide custom permissions for a base namespace.

■ **KRZYSZTOF CWALINA** In the initial design of the .NET Framework namespaces, all types related to a given feature area were in the same namespace. Prior to the first release, we moved design-related types to sub-namespaces with the “.Design” suffix. Unfortunately, we did not have time to do it for the Permission types. This is a problem in several parts of the Framework. For example, a large portion of the types in the System.Diagnostics namespace are types needed for the security infrastructure and very rarely used by the end users of the API.

B.4.1.1.3 The `.Interop` Subnamespace

Many frameworks need to support interoperability with legacy components. Due diligence should be used in designing interoperability from the ground up. However, the nature of the problem often requires that the shape and style of such interoperability APIs are often quite different from good managed framework design. Thus, it makes sense to put functionality related to interoperation with legacy components in a subnamespace.

You should not put types that completely abstract unmanaged concepts into the `Interop` subnamespace and expose them as managed. It is often the case that managed APIs are implemented by calling out to unmanaged code. For example, the `System.IO.FileStream` class calls out to Win32 `CreateFile` on Windows. This is perfectly acceptable and does not imply that the `FileStream` class needs to be in the `System.IO.Interop` namespace because `FileStream` completely abstracts the Win32 concepts and publicly exposes a nice managed abstraction.

- ✓ **DO** use a namespace with the “.Interop” suffix to contain types that provide interop functionality for a base namespace.
- ✓ **DO** use a namespace with the “.Interop” suffix for all code in a Primary Interop Assembly (PIA).

B.5 Obsolete Guidance from Member Design

B.5.4 Event Design

B.5.4.1 Custom Event Handler Design

 **WHY THIS IS HERE** Custom event handlers were the only way to declare event handlers in .NET Framework 1.0 and .NET Framework 1.1. When generics were introduced in .NET Framework 2.0, the guidance was to use `EventHandler<T>` when working on “new” CLR runtimes and custom event handlers when working with .NET Framework 1.0/1.1, or for maintaining consistency with existing features.

Microsoft stopped supporting .NET Framework 1.0 in 2009, and .NET Framework 1.1 in 2015. All current .NET runtime environments support generics, and any new events are expected to use `EventHandler<T>`.

In the very rare case that you need to define a custom event handler for consistency with existing feature areas, these guidelines are still appropriate.

There are cases in which `EventHandler<T>` cannot be used, such as when the framework needs to work with earlier versions of the CLR, which did not support generics. In such cases, you might need to design and develop a custom event handler delegate.

 **DO** use a return type of void for event handlers.

An event handler can invoke multiple event handling methods, possibly on multiple objects. If event handling methods were allowed to return a value, there would be multiple return values for each event invocation.

 **DO** use object as the type of the first parameter of the event handler, and call it sender.

 **DO** use `System.EventArgs` or its subclass as the type of the second parameter of the event handler, and call it e.

 **DO NOT** have more than two parameters on event handlers.

The following event handler follows all of the preceding guidelines.

```
public delegate void ClickedEventHandler(object sender,  
ClickedEventArgs e);
```

■ **CHRIS ANDERSON** Why? People always ask this. In the end, this is just about a pattern. By having event arguments packaged in a class you get better versioning semantics. By having a common pattern (`sender`, `e`) it is easily learned as the signature for all events. I think back to how bad it was with Win32—when data was in WPARAM versus LPARAM, and so on. The pattern becomes noise and developers just assume that event handlers have scope to the sender and arguments of the event.

B.7 Obsolete Guidance from Exceptions

B.7.4 Designing Custom Exceptions

■  **WHY THIS IS HERE** Between the second and third editions of this book, Code Access Security (CAS) and partial trust execution became deprecated. You should consider whether an exception message is over-sharing, but the notion of using a CAS permission to change the amount of data to include in the message is no longer generally applicable.

Without partial trust, it's almost impossible to "ensure only trusted code" can do anything because reflection can access private fields.

✓ **DO** report security-sensitive information through an override of `ToString` only after demanding an appropriate permission.

If the permission demand fails, return a string excluding the security-sensitive information.

■ **RICO MARIANI** Do not store the results of `ToString` in any generally accessible data structure unless that data structure suitably secures the string from untrusted code. This advice applies to all strings, but because exception strings frequently contain sensitive information (such as file paths), I reiterate the advice here.

- ✓ **DO** store useful security-sensitive information in a private exception state. Ensure only trusted code can get the information.

B.8 Obsolete Guidance from Usage Guidelines

B.8.10 Serialization

■  **WHY THIS IS HERE** Users change the data that they want to serialize and the formats of their data fairly rapidly. Looking back at the number of serializers that have been offered as a part of .NET, the transition from XML to JSON, and the potential of other formats to supplant JSON, it seemed like the best answer was to say that you shouldn't tie yourself to any particular serialization technology. Instead, leave that for the application developer to decide.

Serialization is the process of converting an object into a format that can be readily persisted or transported. For example, you can serialize an object, transport it over the Internet using HTTP, and deserialize it at the destination machine.

.NET offers three main serialization technologies optimized for various serialization scenarios. Table 8-1 lists these technologies and the main framework types related to these technologies.

TABLE 8-1: .NET Serialization Technologies

Technology Name	Main Types	Scenarios
Data Contract Serialization	DataContractAttribute DataMemberAttribute DataContractSerializer NetDataContractSerializer DataContractJsonSerializer ISerializable	General persistence Web Services JSON
XML Serialization	XmlSerializer	XML format with full control over the shape of the XML
Runtime Serial- ization (Binary and SOAP)	SerializableAttribute ISerializable BinaryFormatter SoapFormatter	.NET Remoting

When you design new types, you should decide which, if any, of these technologies those types need to support. The following guidelines describe how to make that choice and how to provide such support. Please note that these guidelines are not trying to help you choose what serialization technology you should use in the implementation of your application or library. Such guidelines are not directly related to API design and thus are not within the scope of this book.

- ✓ **DO** think about serialization when you design new types.

Serialization is an important design consideration for any type, because programs might need to persist or transmit instances of the type.

B.8.10.1 Choosing the Right Serialization Technology to Support

Any given type can support none, one, or more of the serialization technologies.

- ✓ **CONSIDER** supporting Data Contract Serialization if instances of your type might need to be persisted or used in Web Services.

See section 8.10.2 for details on supporting Data Contract Serialization.

- ✓ **CONSIDER** supporting the XML Serialization instead of or in addition to Data Contract Serialization if you need more control over the XML format that is produced when the type is serialized.

This may be necessary in some interoperability scenarios where you need to use an XML construct that is not supported by Data Contract Serialization, for example, to produce XML attributes. See section 8.10.3 for details on supporting XML Serialization.

- ✓ **CONSIDER** supporting the Runtime Serialization if instances of your type need to travel across .NET Remoting boundaries.

See section 8.10.4 for details on supporting Runtime Serialization.

- ✗ **AVOID** supporting Runtime Serialization or XML Serialization just for general persistence reasons. Prefer Data Contract Serialization instead.

B.8.10.2 Supporting Data Contract Serialization

Types can support Data Contract Serialization by applying the `DataContractAttribute` to the type and the `DataMemberAttribute` to the members (fields and properties) of the type.

```
[DataContract]
class Person {

    [DataMember]string lastName;
    [DataMember]string FirstName;

    public Person(string firstName; string lastName){ ... }

    public string LastName {
        get { return lastName; }
    }
    public string FirstName {
        get { return firstName; }
    }
}
```

✓ **CONSIDER** marking data members of your type public if the type can be used in partial trust.

In full trust, Data Contract serializers can serialize and deserialize non-public types and members, but only public members can be serialized and deserialized in partial trust.

✓ **DO** implement a getter and setter on all properties that have `DataMemberAttribute`. Data Contract serializers require both the getter and the setter for the type to be considered serializable.¹ If the type won't be used in partial trust, one or both of the property accessors can be nonpublic.

```
[DataContract]
class Person {

    string lastName;
    string FirstName;
```

1. In .NET Framework 3.5 SP1, some collection properties can be get-only.

```
public Person(string firstName, string lastName){  
    this.lastName = lastName;  
    this.firstName = firstName;  
}  
  
[DataMember]  
public string LastName {  
    get { return lastName; }  
    private set { lastName = value; }  
}  
  
}  
  
[DataMember]  
public string FirstName {  
    get { return firstName; }  
    private set { firstName = value; }  
}  
}
```

- ✓ **CONSIDER** using the serialization callbacks for initialization of deserialized instances.

Constructors are not called² when objects are deserialized. Therefore, any logic that executes during normal construction needs to be implemented as one of the serialization callbacks.

```
[DataContract]  
class Person {  
  
    [DataMember] string lastName;  
    [DataMember] string firstName;  
    string fullName;  
  
    public Person(string firstName, string lastName){  
        this.lastName = lastName;  
        this.firstName = firstName;  
        fullName = firstName + " " + lastName;  
    }  
  
    public string FullName {  
        get { return fullName; }  
    }  
}
```

2. There are exceptions to the rule. Constructors of collections marked with `CollectionDataContractAttribute` are called during deserialization.

```
[OnDeserialized]
void OnDeserialized(StreamingContext context) {
    fullName = firstName + " " + lastName;
}
}
```

`OnDeserializedAttribute` is the most commonly used callback attribute. The other attributes in the family are `OnDeserializingAttribute`, `OnSerializingAttribute`, and `OnSerializedAttribute`. They can be used to mark callbacks that get executed before deserialization, before serialization, and finally, after serialization, respectively.

- ✓ **CONSIDER** using the `KnownTypeAttribute` to indicate concrete types that should be used when deserializing a complex object graph.

For example, if a type of a deserialized data member is represented by an abstract class, the serializer will need the Known Type information to decide what concrete type to instantiate and assign to the member. If the Known Type is not specified using the attribute, it will need to be passed to the serializer explicitly (you can do it by passing Known Types to the serializer constructor) or it will need to be specified in the configuration file.

```
[KnownType(typeof(USAddress))]
[DataContract]
class Person {

    [DataMember] string fullName;
    [DataMember] Address address; // Address is abstract

    public Person(string fullName, Address address){
        this.fullName = fullName;
        this.address = address;
    }

    public string FullName {
        get { return fullName; }
    }
}
```

```
[DataContract]
public abstract class Address {
    public abstract string FullAddress { get; }
}

[DataContract]
public class USAddress : Address {

    [DataMember] public string Street { get; set; }
    [DataMember] public string City { get; set; }
    [DataMember] public string State { get; set; }
    [DataMember] public string ZipCode { get; set; }

    public override string FullAddress { get {
        return Street + "\n" + City + ", " + State + " " + ZipCode; }
    }
}
```

In cases where the list of Known Types is not known statically (when the Person class is compiled), the KnownTypeAttribute can also point to a method that returns a list of Known Types at runtime.

- ✓ **DO** consider backward and forward compatibility when creating or changing serializable types.

Keep in mind that serialized streams of future versions of your type can be deserialized into the current version of the type, and vice versa.

Make sure you understand that data members, even private and internal, cannot change their names, types, or even their order in future versions of the type unless special care is taken to preserve the contract using explicit parameters to the data contract attributes.

Test compatibility of serialization when making changes to serializable types. Try deserializing the new version into an old version, and vice versa.

- ✓ **CONSIDER** implementing `IExtensibleDataObject` to allow round-tripping between different versions of the type.

The interface allows the serializer to ensure that no data is lost during round-tripping. The `IExtensibleDataObject.ExtensionData` property is used to store any data from the future version of the type that is

unknown to the current version, and so it cannot store it in its data members. When the current version is subsequently serialized and deserialized into a future version, the additional data will be available in the serialized stream.

```
[DataContract]
class Person : IExtensibleDataObject {

    [DataMember] string fullName;

    public Person(string fullName){
        this.fullName = fullName;
    }

    public string FullName {
        get { return fullName; }
    }

    ExtensionDataObject serializationData;
    ExtensionDataObject IExtensibleDataObject.ExtensionData {
        get { return serializationData; }
        set { serializationData = value; }
    }
}
```

B.8.10.3 Supporting XML Serialization

Data Contract Serialization is the main (default) serialization technology in .NET, but there are serialization scenarios that Data Contract Serialization does not support. For example, it does not give you full control over the shape of XML produced or consumed by the serializer. If such fine control is required, XML Serialization has to be used, and you need to design your types to support this serialization technology.

X AVOID designing your types specifically for XML Serialization, unless you have a very strong reason to control the shape of the XML produced. This serialization technology has been superseded by the Data Contract Serialization discussed in the previous section.

In other words, don't apply attributes from the `System.Xml.Serialization` namespace to new types, unless you know that the type will

be used with XML Serialization. The following example shows how `System.Xml.Serialization` can be used to control the shape of the XML produced.

```
public class Address {  
    [XmlAttribute] // serialize as XML attribute, instead of an element  
    public string Name { get { return "John Smith"; } set { } }  
  
    [XmlElement(ElementName = "StreetLine")] // explicitly name element  
    public string Street = "1 Some Street";  
    ...  
}
```

- ✓ **CONSIDER** implementing the `IXmlSerializable` interface if you want even more control over the shape of the serialized XML than what's offered by applying the XML Serialization attributes. Two methods of the interface, `ReadXml` and `WriteXml`, allow you to fully control the serialized XML stream. You can also control the XML schema that gets generated for the type by applying the `XmlSchemaProviderAttribute`.

B.8.10.4 Supporting Runtime Serialization

Runtime Serialization is a technology used by .NET Remoting. If you think your types will be transported using .NET Remoting, you need to make sure they support Runtime Serialization.

The basic support for Runtime Serialization can be provided by applying the `SerializableAttribute`, and more advanced scenarios involve implementing a simple Runtime Serializable Pattern (implement `ISerializable` and provide serialization constructor).

- ✓ **CONSIDER** supporting Runtime Serialization if your types will be used with .NET Remoting. For example, the `System.AddIn` namespace uses .NET Remoting, and so all types exchanged between `System.AddIn` add-ins need to support Runtime Serialization.

```
[Serializable]  
public class Person {  
    ...  
}
```

✓ **CONSIDER** implementing the Runtime Serializable Pattern if you want complete control over the serialization process. For example, if you want to transform data as it gets serialized or deserialized.

The pattern is very simple. All you need to do is implement the ISerializable interface and provide a special constructor that is used when the object is deserialized.

```
[Serializable]
public class Person : ISerializable {
    string fullName;

    public Person() { }

    protected Person(SerializationInfo info, StreamingContext context) {
        if (info == null) throw new System.ArgumentNullException("info");
        fullName = (string)info.GetValue("name", typeof(string));
    }

    [SecurityPermission(
        SecurityAction.LinkDemand,
        Flags = SecurityPermissionFlag.SerializationFormatter)
    ]
    void ISerializable.GetObjectData(SerializationInfo info,
        StreamingContext context) {
        if (info == null) throw new System.ArgumentNullException("info");
        info.AddValue("name", fullName);
    }

    public string FullName {
        get { return fullName; }
        set { fullName = value; }
    }
}
```

✓ **DO** make the serialization constructor protected and provide two parameters typed and named exactly as shown in the sample here.

```
[Serializable]
public class Person : ISerializable {
    protected Person(SerializationInfo info, StreamingContext context) {
        ...
    }
}
```

- ✓ **DO** implement the `ISerializable` members explicitly.

```
[Serializable]
public class Person : ISerializable {
    void ISerializable.GetObjectData(...) {
        ...
    }
}
```

- ✓ **DO** apply a link demand to the `ISerializable.GetObjectData` implementation. This ensures that only fully trusted code and the Runtime Serializer have access to the member.

```
[Serializable]
public class Person : ISerializable {
    [SecurityPermission(
        SecurityAction.LinkDemand,
        Flags = SecurityPermissionFlag.SerializationFormatter)
    ]
    void ISerializable.GetObjectData(...) {
        ...
    }
}
```

B.9 Obsolete Guidance from Common Design Patterns

B.9.2 The Async Patterns

■  **WHY THIS IS HERE** The Task-Based Async Pattern, combined with language support for generating tasks (e.g., C# `async/await`, F# `async/let!-/use!-/do!`), has replaced both the Classic Async Pattern and the Event-Based Async Pattern.

The .NET Framework uses two different API patterns to model asynchronous APIs: the so-called Classic Async Pattern (a.k.a. Async Pattern) and a newer Event-Based Async Pattern (a.k.a. Async Pattern for Components). This section describes details of these two patterns as well as how to choose between them when designing asynchronous APIs.

B.9.2.1 Choosing Between the Async Patterns

This section discusses the criteria for choosing the appropriate pattern for implementing asynchronous APIs. The main difference between the patterns is that the Event-Based Async Pattern is optimized for usability and integration with visual designers; the other is optimized for power and small surface area.

■ **STEVEN CLARKE** This was clearly called out during our usability studies on this pattern. Most participants in our study could use the event-based pattern successfully without reading any documentation. On the other hand, developers who are unfamiliar with the Classic Async Pattern will have a more difficult time without spending time reading documentation.

While it's not the case that usability in the context of an API means the ability to use that API without reading documentation, for some users, APIs that support learning by doing are preferable to those that don't.

The Classic Async Pattern offers a powerful and flexible programming model but is lacking when it comes to ease of use, especially when applied to components supporting graphical designers. The main reasons for usability differences between the patterns are the following:

- The Classic Async Pattern callbacks are executed on an arbitrary thread as opposed to a thread appropriate for the application model (for example, the UI thread for Windows Forms applications).
- There is currently no Visual Studio support for using the Classic Async Pattern, because it is delegate-based rather than event-based. Visual Studio provides extensive support for defining and hooking up event handlers, and the Event-Based Async Pattern takes full advantage of this support.

■ **GREG SCHECHTER** Also, even if visual designers aren't used, the VB.NET code editor combined with WithEvents provides much greater support for implementing and connecting event handlers than it does for delegates.

- ✓ **DO** implement the Event-Based Async Pattern if your type is a component supporting visual designers (i.e., if the type implements `IComponent`).
- ✓ **DO** implement the Classic Async Pattern if you must support wait handles.

The main reason to support wait handles is to be able to initiate multiple asynchronous operations simultaneously and then wait for the completion of one or all of the operations.

- ✓ **CONSIDER** implementing the Event-Based Async Pattern if you are designing higher-level APIs. For example, aggregate components (see section 9.2.4) should implement this pattern.
- ✓ **CONSIDER** implementing the Classic Async Pattern for low-level APIs where usability is less important than power, memory consumption, flexibility, and small surface area.

■ **JOE DUFFY** This decision is ultimately very simple to me. If you are developing a type meant to be consumed by other library or framework developers, you should use the Classic Async Pattern. They will appreciate the flexibility. If you are developing a type meant to be consumed by application developers, you should almost always prefer the Event-Based Async Pattern. They will appreciate the simplicity and Visual Studio integration.

- ✗ **AVOID** implementing both patterns at the same time on a single type or even on a single set of related types.

Components that implement both patterns may be confusing to some users. For example, for one synchronous method `SomeMethod`, there will be four asynchronous operations.

```
SomeMethod
SomeMethodAsync
BeginSomeMethod
EndSomeMethod
SomeMethodCompleted
```

B.9.2.2 Classic Async Pattern

The Classic Async Pattern is a naming, method signature, and behavioral convention for providing APIs that can be used to execute asynchronous operations. The main elements of the Async Pattern include the following:

- The `Begin` method, which initiates an asynchronous operation.
- The `End` method, which completes an asynchronous operation.
- The `IAsyncResult` object, which is returned from the `Begin` method and is essentially a token representing a single asynchronous operation. It contains methods and properties providing access to some basic information about the operation.
- An `async callback`, which is a user-supplied method that is passed to the `Begin` method and is called when the asynchronous operation is completed.
- The `State` object, which is a user-provided state that can be passed to the `Begin` method, then is passed to the `async callback`. This state is commonly used to pass caller-specific data to the `async callback`.

The following guidelines spell out conventions related to the API design part of the Classic Async Pattern. The guidelines do not go into the details of implementing the pattern.

Note that the guidelines assume that a method implementing a synchronous version of the operation already exists. It is quite common that an asynchronous operation is provided together with a synchronous counterpart, but it is not an absolute requirement.

✓ DO use the following convention for defining APIs for asynchronous operations. Given a synchronous method named `Operation`, provide `BeginOperation` and `EndOperation` methods with the following signatures (note that the `out` params are optional):

```
// synchronous method  
public <return> Operation(<parameters>,<out params>)  
// async pattern methods
```

```
public IAsyncResult BeginOperation(<parameters>, AsyncCallback  
callback, object state)  
public <return> EndOperation(IAsyncResult asyncResult, <out params>)
```

As an example, `System.IO.Stream` defines a synchronous `Read` method and the `BeginRead` and `EndRead` methods.

```
public int Read(byte[] buffer, int offset, int count)  
public IAsyncResult BeginRead(byte[] buffer, int offset, int count,  
    AsyncCallback callback, object state)  
public int EndRead(IAsyncResult asyncResult)
```

- ✓ **DO** ensure that the return type of the `Begin` method implements `IAsyncResult`.
- ✓ **DO** ensure that any by-value and `ref` parameters of the synchronous method are represented as by-value parameters of the `Begin` method. `Out` parameters of the synchronous methods should not show in the signature of the `Begin` method.
- ✓ **DO** ensure that the return type of the `End` method is the same as the return type of the synchronous method.

```
public abstract class Stream {  
    public int Read(byte[] buffer, int offset, int count)  
    public int EndRead(IAsyncResult asyncResult)  
}
```

- ✓ **DO** ensure that any `out` and `ref` parameters of the synchronous method are represented as `out` parameters of the `End` method. By-value parameters of the synchronous methods should not show in the signature of the `End` method.

- ✗ **DO NOT** continue the asynchronous operation if the `Begin` method throws an exception.

This method should throw if it needs to indicate the asynchronous operation could not be started. The `async` callback should not be called after this method throws.

✓ **DO** notify the caller that the asynchronous operation completed via all of the following mechanisms in this order:

- Set `IAsyncResult.IsCompleted` to true.
- Signal the wait handle returned from `IAsyncResult.AsyncWaitHandle`.
- Call the async callback.

■ **JOE DUFFY** This callback should never be transferred to a specific thread. It is all right to queue a work item to the thread pool for invocation so as not to reuse the current callstack at the time the operation completes. This can be particularly useful if the operation completed synchronously. But you should specifically not try to do anything with `SynchronizationContext`, Windows Forms' `Control.Invoke`, Windows Presentation Foundation's Dispatcher, etc. to marshal the callback to another thread. One of the great things about the Classic Async Pattern is that it is consistent in this regard. If you find yourself needing to do this kind of marshaling, it's an indication you ought to consider using the Event-Based Async Pattern instead.

✓ **DO** throw exceptions from the `End` method to indicate that the asynchronous operation could not complete successfully.

This allows a deterministic place to catch those exceptions.

✓ **DO** complete all remaining work synchronously once the `End` method is called.

In other words, the `End` method blocks until the operation completes and then returns.

✓ **CONSIDER** throwing an `InvalidOperationException` if the `End` method is called with the same `IAsyncResult` twice, or if the `IAsyncResult` was returned from an unrelated `Begin` method.

✓ **DO** set `IAsyncResult.CompletedSynchronously` to true if and only if the async callback will be run on the thread that called `Begin`.

■ **BRIAN GRUNKEMEYER** The point of the `CompletedSynchronously` property is not to report the underlying details of the asynchronous operation, but instead to help the async callback deal with potential stack overflows. Some callers might want to call the `Begin` method again from within their callback, passing in the callback again as to the nested `Begin`. This can lead to stack overflows if the underlying operation executes synchronously and the callback is called on that same thread. By checking this property, the callback can tell whether it is running at some arbitrary stack depth on a user thread or at the base of a thread pool thread.

Yes, I know, the name might have been poorly chosen. Alternate names could have been something like `CallbackRunningOnThreadpoolThread` or `CallbackCanContinueCallingBeginMethodName`.

B.9.2.3 Classic Async Pattern Basic Implementation Example

The following sample shows a basic implementation of the Async Pattern. This particular implementation is for illustrative purposes only. Although the implementation is useful for understanding the general pattern, it is unlikely to produce the optimum performance in exposing asynchronous operations because it utilizes the asynchronous functionality built into delegates. This uses the remoting layer and so is not optimum in terms of performance and resource consumption.

```
public class FiboCalculator {
    delegate void Callback(int count, ICollection<decimal> series);
    private Callback callback = new Callback(GetFibo);

    // Starts the process of generating a series and returns
    public IAsyncResult BeginGetFibo(
        int count,
        ICollection<decimal> series,
        AsyncCallback callback,
        object state)
    {
        return this.callback.BeginInvoke(count, series, callback, state);
    }

    // Blocks until the process of generating a series completes
    public void EndGetFibo(IAsyncResult asyncResult) {
        this.callback.EndInvoke(asyncResult);
    }
}
```

```
// Generate a series of the first count Fibonacci numbers
public static void GetFibo(
    int count, ICollection<decimal> series)
{
    for (int i = 0; i < count; i++) {
        decimal d = GetFiboCore(i);
        lock (series) {
            series.Add(d);
        }
    }
}

// Return the Nth Fibonacci number
static decimal GetFiboCore(int n){
    if (n < 0) throw new ArgumentException("n must be > 0");
    if (n == 0 || n == 1) return 1;
    return GetFiboCore(n-1) + GetFiboCore(n-2);
}
```

CHRISTOPHER BRUMME A better way to implement this pattern is to provide Begin and End methods that use `ThreadPool.QueueUserWorkItem` under the covers. Essentially, you are avoiding all the interpretive overhead of remoting messages by writing the methods that put your state into a work item and then getting your state back out of the work item. This is pretty easy to do, and the performance should be much better than when using asynchronous delegates.

BRIAN GRUNKEMEYER To implement the async design pattern using asynchronous I/O, look at `System.Threading.Overlapped` and `System.Threading.Threadpool.BindHandle`. `BindHandle` internally binds a handle to a Win32 I/O completion port, which is the most performant mechanism in the operating system to handle asynchronous I/O, allowing the operating system to throttle I/O threads as needed. Use the `Overlapped` class to provide a `NativeOverlapped*`, which you can then pass to Win32 methods taking an `LPOVERLAPPED`. Make sure any buffers you use for your asynchronous I/O operation are passed to `Overlapped.Pack` or `UnsafePack`, or you will corrupt memory if an application domain unload happens while your asynchronous I/O operation is in flight.

B.9.2.4 Event-Based Async Pattern

Usually, asynchronous APIs are offered in addition to synchronous APIs. The following step-by-step guide for defining asynchronous APIs assumes the synchronous versions already exist, but it is important to keep in mind that the synchronous methods need not even be present if the component just wants to expose asynchronous API.

B.9.2.4.1 Defining Asynchronous Methods

Define an asynchronous method for each synchronous method that you want to provide an asynchronous version for. The asynchronous method should return void and take the same parameters as the synchronous method (see section 5.8.3 for handling `out` and `ref` parameters). The name of the method should be built by appending the “`Async`” suffix to the name of the synchronous method.

Optionally, define an overload of the asynchronous method with an additional object parameter called “`userState`.” Do this if the API supports multiple concurrent invocations of the asynchronous operation, in which case the `userState` will be delivered back to the event handler to distinguish between the invocations.

For example, given the following synchronous method:

```
public class SomeType {  
    public SomeReturnType Method(string arg1, string arg2);  
}
```

the following asynchronous methods should be added:

```
public class SomeType {  
    public void MethodAsync(string arg1, string arg2);  
  
    // optional  
    public void MethodAsync(string arg1, string arg2, object userState);  
  
    public SomeReturnType Method(string arg1, string arg2);  
}
```

- ✓ **DO** ensure that if your component defines the asynchronous method without the `userState` parameter, any attempt to invoke the method

before the prior invocation has completed will result in an `InvalidOperationException` being raised.

For each asynchronous method, also define the following event:

```
public class SomeType {  
    public event EventHandler<MethodCompleteEventArgs> MethodCompleted;  
  
    public void MethodAsync(string arg1, string arg2);  
    public void MethodAsync(string arg1, string arg2, object userState);  
    public SomeReturnType Method(string arg1, string arg2);  
}
```

- ✓ **DO** ensure you invoke event handlers on the proper thread. This is one of the main benefits of using the Event-Based Async Pattern over the Classic Async Pattern.

It is critical that event handlers get invoked on the proper thread for the given application model. The Framework provides a mechanism, `AsyncOperationManager`, to do this easily and consistently. The mechanism allows components to be used equally well across all application models.

- ✓ **DO** ensure that you always invoke the event handler, on successful completion, on an error, or on cancellation. Applications should not be put in a situation where they indefinitely wait for something that doesn't happen. An exception to this, of course, is if the actual asynchronous operation itself never starts or completes.

```
public class MethodCompletedEventArgs : AsyncCompletedEventArgs {  
    public SomeReturnType Result { get; }  
}
```

- ✓ **DO** ensure that accessing a property³ of the event arguments class of a failed asynchronous operation results in an exception being thrown. In other words, if there was an error completing the task, the results shouldn't be accessible.

3. This refers only to properties carrying the results of the asynchronous operation, not to properties carrying the error information, for example.

X DO NOT define new event handlers or event argument types for void methods. Use `AsyncCompletedEventArgs`, `AsyncCompletedEventHandler`, or `EventHandler<AsyncCompletedEventArgs>` instead.

B.9.2.5 Supporting Out and Ref Parameters

The above examples are based on synchronous methods that take only input parameters; `out` and `ref` parameters are not dealt with. Although the use of `out` and `ref` is discouraged in general, sometimes they are unavoidable.

Given a synchronous method with `out` parameters, the asynchronous version of the method should omit the parameters from its signature. Instead, the parameters should be exposed as get-only properties of the `EventArgs` class. The names and types of the properties should be the same as the names and types of the parameters.

`Ref` parameters to the synchronous method should appear as input parameters of asynchronous version, and as get-only properties of the `EventArgs` class. The names and types of the properties should be the same as the names and types of the parameters.

For example, given the following synchronous method:

```
public string Method(object arg1, ref int arg2, out long arg3)
```

The asynchronous version would look like the following:

```
public void MethodAsync(object arg1, int arg2);  
  
public class MethodCompletedEventArgs : AsyncCompletedEventArgs {  
    public string Result { get; }  
    public int Arg2 { get; }  
    public long Arg3 { get; }  
}
```

B.9.2.6 Supporting Cancellation

The pattern can optionally support cancellation of pending operations. Cancellation should be exposed through a `CancelAsync` method.

If the asynchronous operations support multiple outstanding operations, that is, if the asynchronous methods take parameter `userState`, the cancellation method should also take a parameter `userState`. Otherwise, the method does not take any parameters.

```
public class SomeType {  
  
    public void CancelAsync(object userState);  
    // or public void CancelAsync();  
    // if multiple outstanding requests are not supported  
  
    public SomeReturnType Method(string arg1, string arg2);  
    public void MethodAsync(string arg1, string arg2);  
    public void MethodAsync(string arg1, string arg2, object userState);  
    public event MethodCompletedEventHandler MethodCompleted;  
}
```

- ✓ **DO** ensure that in the case of cancellation, you set the `Cancelled` property of the event arguments class to true, and that any attempt to access the result raises an `InvalidOperationException` stating that the operation was cancelled.
- ✓ **DO** ignore calls to the cancellation method if the particular operation cannot be cancelled, instead of raising an exception. The reason for this is that in general a component cannot know whether an operation is truly cancelable at any given time and cannot know whether a previously issued cancellation has succeeded. However, the application will always know when a cancellation succeeded, because it is indicated by the `Cancelled` property of the event arguments class.

B.9.2.7 Supporting Progress Reporting

It is frequently desired and feasible to provide progress reporting during an asynchronous operation. The following section describes APIs for such support.

If an asynchronous operation needs to support progress reporting, add an additional event, named `ProgressChanged`, to be raised by the asynchronous operation.

```
public class SomeType {  
  
    public event EventHandler<ProgressChangedEventArgs>  
        ProgressChanged;  
  
    public void CancelAsync(object userState);  
    public SomeReturnType Method(string arg1, string arg2);  
    public void MethodAsync(string arg1, string arg2);  
    public void MethodAsync(string arg1, string arg2, object userState);  
    public event MethodCompletedEventHandler MethodCompleted;  
}
```

The `ProgressChangedEventArgs` parameter, passed to the handler, carries an integer-valued progress indicator that is expected to be between 0 and 100.

```
// this is a standard type defined by the Framework  
public class ProgressChangedEventArgs : EventArgs {  
    public ProgressChangedEventArgs(int progressPercentage,  
                                object userState);  
    public object UserState { get; }  
    public int ProgressPercentage { get; }  
}
```

Note that in most cases there is only one `ProgressChanged` event for the component, regardless of the number of asynchronous operations. Clients are expected to use the `userState` object that was passed in to the operations to distinguish between progress updates on multiple concurrent operations.

There may be situations where multiple operations support progress, and they each return a different indicator for that progress. In that case, a single `ProgressChanged` event is no longer appropriate, and the implementer may consider supporting multiple `ProgressChanged` events as the situation dictates. In such case, the specific progress events should be called `<MethodName>ProgressChanged`.

The basic implementation of progress reporting uses the `ProgressChangedEventArgs` class and the `EventHandler<ProgressChangedEventArgs>` delegate. Optionally, if a more domain-specific progress indicator is desired (for example, number of bytes read), a subclass of `ProgressChangedEventArgs` can be defined and used.

■ **GREG SCHECHTER** One could argue that a floating-point value between 0.0 and 1.0 would be more appropriate as `ProgressPercentage`. An integer range was chosen because it maps well to the progress control, which also uses the integer range from 0 to 100.

✓ **DO** ensure that, if you implement a `ProgressChanged` event, there are no such events raised for a particular asynchronous operation after that operation's completed event has been raised.

■ **GREG SCHECHTER** If you're dispatching progress and completion events from the same thread, ensure that no progress events occur after the operation has completed. If they're coming from different threads, you may not be able to do this without compromising concurrency, and thus it likely shouldn't be pursued in those situations.

✓ **DO** ensure that, if the standard `ProgressChangedEventArgs` is being used, the `ProgressPercentage` can always be interpreted as a percentage (it does not need to necessarily be an accurate percentage, but it does need to represent a percentage). If your progress reporting metric must be something different, then a subclass of `ProgressChangedEventArgs` is more appropriate, and you should leave the `ProgressPercentage` property at 0.

■ **CHRIS SELLS** Please make progress reporting move forward, if for no other reason than my family makes fun of me when they see a progress report going backwards, as if it's my fault. Personally, I've implemented several progress percentage algorithms and while I often can't get the timing to be smooth through all stages of an operation, at least they always move forward. In fact, I think you'd have to work extra hard to make them move backwards.

B.9.2.8 Supporting Incremental Results

Occasionally, asynchronous operations can return incremental results periodically, prior to being complete. There are a number of different options that can be used to support this scenario, based on the constraints.

If the component supports multiple asynchronous operations, each capable of returning incremental results, these incremental results should all have the same type.

- ✓ **DO** raise this `ProgressChanged` event when there is an incremental result to report back.
- ✓ **DO** extend the `ProgressChangedEventArgs` to carry the incremental result data, and define a `ProgressChanged` event with this extended data.

If the multiple asynchronous operations return a different type of data, the following approach should be used.

- ✓ **DO** separate out your incremental result reporting from your progress reporting.
- ✓ **DO** define a separate `<MethodName>ProgressChanged` event with appropriate event arguments for each asynchronous operation to handle that operation's incremental result data.

■ **CHRIS SELLS** I find the `System.ComponentModel.BackgroundWorker` component to be a simple way to implement the Event-Based Async Pattern if you'd like help. It's also an excellent example of such an implementation.

B.9.4 Dispose Pattern

B.9.4.2 Finalizable Types

■  **WHY THIS IS HERE** During the updates to this book for the third edition, we changed the guidance from AVOID-ing finalizers on public types to DO NOT to reflect how rare finalizers had become in practice. The section was then rewritten to make a distinction between public versus internal finalizable types.

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the `Dispose(bool)` method.

Finalizers are notoriously difficult to implement correctly, primarily because you cannot make certain (normally valid) assumptions about the state of the system during their execution. The following guidelines should be taken into careful consideration.

Note that some of the guidelines apply not just to the `Finalize` method, but to any code called from a finalizer. In the case of the Basic Dispose Pattern previously defined, this means logic that executes inside `Dispose(bool disposing)` when the `disposing` parameter is false.

If the base class already is finalizable and implements the Basic Dispose Pattern, you should not override `Finalize` again. You should instead just override the `Dispose(bool)` method to provide additional resource cleanup logic.

■ **HERB SUTTER** You really don't want to write a finalizer if you can help it. Besides problems already noted earlier in this chapter, writing a finalizer on a type makes that type more expensive to use even if the finalizer is never called. For example, allocating a finalizable object is more expensive because it must also be put on a list of finalizable objects. This cost can't be avoided, even if the object immediately suppresses finalization during its construction (as when creating a managed object semantically on the stack in C++).

The following code shows an example of a finalizable type:

```
public class ComplexResourceHolder : IDisposable {  
  
    private IntPtr _buffer; // unmanaged memory buffer  
    private SafeHandle _resource; // disposable handle to a resource  
  
    public ComplexResourceHolder() {  
        _buffer = ... // allocates memory  
        _resource = ... // allocates the resource  
    }  
  
    protected virtual void Dispose(bool disposing) {  
        if (_buffer != IntPtr.Zero) {  
            ReleaseBuffer(_buffer); // release unmanaged memory  
            _buffer = IntPtr.Zero;  
        }  
        if (disposing) { // release other disposable objects  
            _resource?.Dispose();  
        }  
    }  
  
    ~ComplexResourceHolder() {  
        Dispose(false);  
    }  
  
    public void Dispose() {  
        Dispose(true);  
        GC.SuppressFinalize(this);  
    }  
}
```

X AVOID making types finalizable.

Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint. Prefer using resource wrappers such as `SafeHandle` to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own resource cleanup.

■ **CHRIS SELLS** Of course, if you're implementing your own managed wrappers around unmanaged resources, those will need to implement the finalizable type pattern.

■ **BRIAN GRUNKEMEYER** If you're writing a wrapper class for an OS resource such as handles or memory, please consider `SafeHandle`. That will handle all of the tricky reliability and security problems associated with guaranteeing resources get freed eventually and in a threadsafe manner. Additionally, it will usually mean you don't need to write a finalizer on your own type. You should still implement `IDisposable`, though.

■ **JOE DUFFY** The extra cost of a finalizer object arises in three specific places. First, at allocation time the object must be registered with the GC as being finalizable. This makes just creating an object more expensive. Second, when such an object is found to be unreachable, it must be moved to a special to-be-finalized queue. This actually causes the object to be promoted to the next GC generation. Third, the finalizer thread needs to spend time processing the to-be-finalized queue and executing finalizers on the objects within it. There is only a single finalizer thread, so having too many finalizable objects can actually cause scalability problems. On a heavily loaded server, you may find that one processor spends 100 percent of its time just running finalizers.

■ **VANCE MORRISON** Wrapping unmanaged resources is pretty much the ONLY reason to have a finalizer. If you are wrapping an unmanaged resource, ideally that should be the only thing the class does, and there is a good chance you should be subclassing `SafeHandle` to do it. Thus, "ordinary" types should not have finalizers.

X DO NOT make value types finalizable.

Only reference types can actually get finalized by the CLR, and thus any attempt to place a finalizer on a value type will be ignored. The C# and C++ compilers enforce this rule.

✓ DO make a type finalizable if the type is responsible for releasing an unmanaged resource that does not have its own finalizer.

When implementing the finalizer, simply call `Dispose(false)` and place all resource cleanup logic inside the `Dispose(bool disposing)` method.

```
public class ComplexResourceHolder : IDisposable {  
  
    ~ComplexResourceHolder() {  
        Dispose(false);  
    }  
  
    protected virtual void Dispose(bool disposing) {  
        ...  
    }  
}
```

✓ **DO** implement the Basic Dispose Pattern on every finalizable type. See section 9.4.1 for details on the basic pattern.

This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

■ **JEFFREY RICHTER** This guideline is very important and should always be followed, without exception. Without this guideline, a user of a type can't control the resource properly.

■ **HERB SUTTER** Languages ought to warn on this case. If you have a finalizer, you want a destructor (`Dispose`).

✗ **DO NOT** access any finalizable objects in the finalizer code path, because there is significant risk that they will have already been finalized.

For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A's finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object

(or calling a static method that might use values stored in static variables) might not be safe if `Environment.HasShutdownStarted` returns true.

■ **JEFFREY RICHTER** Note that it is OK to touch unboxed value type fields.

✓ **DO** make your `Finalize` method protected.

C#, C++, and VB.NET developers do not need to worry about this, because the compilers help to enforce this guideline.

✗ **DO NOT** let exceptions escape from the finalizer logic, except for system-critical failures.

If an exception is thrown from a finalizer, the CLR will shut down the entire process, preventing other finalizers from executing and resources from being released in a controlled manner.

This page intentionally left blank



Sample API Specification

Revised by Immo Landwerth

MANY OF THE guidelines described in this book are best considered up front during the initial design. This appendix contains an example of an API specification that should be written early in the process of designing a framework feature. Although such an API specification does not describe the full details of the feature, it does highlight the most important elements of the design to nail down up front. This example is heavily based on specifications we use for the development of the .NET platform on GitHub. Its content is as simple as we could find, but it's a good illustration of the parts, flow, and priorities of a specification intended to describe framework APIs that adhere to the guidelines described in this book. Please note that the specification puts lots of emphasis on code samples showing how the APIs will be used. In fact, the code samples were written before the actual feature was implemented or even prototyped. You can find more details as well as our current process at <https://aka.ms/dotnetspecs>.

■ **IMMO LANDWERTH** Specifications can contain many aspects of a feature, such as security, performance, globalization, or compatibility. Don't let perfect be the enemy of good. Keep your specifications simple so that you can focus on the problem at hand, and so that you and your team can see the larger picture as early as possible. Avoid specification templates that try to cover everything. Rather, focus on the rationale, the user experience, a few bullet points for requirements, and an open-ended design section that is tailored for the feature. However, there is value in following a similar structure for all your specifications, as this creates a joint vocabulary and enables people to read documents faster.

Specification: Stopwatch

Measuring elapsed time can be as simple as using `Environment.TickCount` with some arithmetic:

```
long before = Environment.TickCount;
Thread.Sleep(1000);
Console.WriteLine(Environment.TickCount - before);
```

However, this approach has a few downsides:

- Lack of precision. The resolution of the system timer is typically in the range of 10 to 16 milliseconds. This doesn't work well for micro-measurements.
- Obscured code flow. Keeping track of the right variables and doing the math makes the coder messier and often obscures the main flow of your code.

The Windows operating system provides a high-resolution system timer, in the form of `QueryPerformanceCounter` and `QueryPerformanceFrequency` APIs, which are commonly used for performance testing, tuning, and instrumentation.

■ **BRAD ABRAMS** Although I think it is perfectly fine to describe a managed API in terms of the unmanaged version to help readers quickly understand what this feature is, do not let the shape and features of the unmanaged API influence your design of the managed API. Build the right API for your customers, not a clone of the unmanaged API.

Stopwatch is meant to be a simple wrapper for these APIs to provide high-precision measurements of elapsed time without obscuring the code flow.

Scenarios and User Experience

■ **KRZYSZTOF CWALINA** The scenarios section is the most important part of an API specification and should be written right after the introduction. APIs that were designed by writing code samples before actually designing the API are generally successful. APIs that were designed before code samples were written to show how the resulting APIs should be used are often too complex, are not self-explanatory, and ultimately need to be fixed in subsequent releases.

Measure Time Elapsed

```
Stopwatch watch = new Stopwatch();
watch.Start();
Thread.Sleep(1000);
Console.WriteLine(watch.Elapsed);
```

Reuse Stopwatch

```
Stopwatch watch = Stopwatch.StartNew();
Thread.Sleep(1000);
Console.WriteLine(watch.Elapsed);
watch.Reset();
watch.Start();
Thread.Sleep(2000);
Console.WriteLine(watch.Elapsed);
```

Measure Cumulative Intervals

The following sample measures how long it takes to process a list of orders. It excludes the time needed to enumerate the order collection.

```
Stopwatch watch = Stopwatch();
foreach (Order order in orders)
{
    watch.Start();
    order.Process();
    watch.Stop();
}
Console.WriteLine(watch.Elapsed);
```

Requirements

Goals

- Provide APIs to measure elapsed time with very high resolution and the accuracy of `QueryPerformanceCounter`.
- Most operations should require one line of code. Time measurement functionality should not obscure the main flow of an application.
- Must work on versions of Windows that don't have `QueryPerformanceCounter`.

Non-Goals

- Allow developers to customize the granularity of the timer.

■ **IMMO LANDWERTH** It's useful to think about which problems your API will not address. The purpose of this section is to cover problems that people might think you're trying to solve but deliberately would like to scope out. You'll likely add bullets to this section based on early feedback and reviews where requirements are brought up that you want to scope out.

Design

API

■ **KRZYSZTOF CWALINA** The API section is very often used during specification reviews to find issues with naming, consistency, and complexity. It gives the reviewers a similar view of the object model that the users encounter when browsing the reference documentation to familiarize themselves with new APIs.

```
namespace System.Diagnostics
{
    public class Stopwatch
    {
        public Stopwatch();
        public static Stopwatch StartNew();

        public void Start();
        public void Stop();
        public void Reset();

        public bool IsRunning { get; }

        public TimeSpan Elapsed { get; }
        public long ElapsedMilliseconds { get; }
        public long ElapsedTicks { get; }

        public static long GetTimestamp();
        public static readonly long Frequency;
        public static readonly bool IsHighResolution;
    }
}
```

Behavior

- Stopwatch will use QueryPerformanceCounter and QueryPerformanceFrequency, if they are available. On older OS versions, this API will fall back to using Environment.TickCount.
- Start(). If the Stopwatch is already started, the operation does nothing.
- Stop(). If the Stopwatch is stopped, the operation does nothing.

- `ElapsedTicks` and `ElapsedMilliseconds` might overflow.
- `IsHighResolution` indicates whether a system-provided high-resolution timer is being used. If it is `false`, `Stopwatch` will use ticks.

Q & A

■ **IMMO LANDWERTH** Features evolve and decisions are made along the road. Add the question as a subheading and provide the explanation for the decision below. This way, you can easily link to specific questions.

When you find yourself having to explain something in a discussion, consider updating your specification and link to your answer instead. This way, you avoid having to explain the same thing repeatedly.

Will this API be supported on older versions of Windows?

Yes. The API will fall back to the (much) less precise `EnvironmentTickCount`. At the time this API will be shipped, many machines still run versions of Windows that don't support `QueryPerformanceCounter` yet. However, `Stopwatch` also offers significant usability gains over using `Environment.TickCount` directly. By letting all developers use this new API, we ensure that their measurements will automatically become more precise when they upgrade their OS to a newer version of Windows.



Breaking Changes

EVERY SINGLE CHANGE that you make in a reusable library has the potential for breaking something. If you make a method faster, you may inadvertently reveal a race condition in your caller's code. If you make a specific case slower while making the general case faster, you might exceed the maximum time your caller allowed, or reveal a different race condition. Correcting an incorrect return value is generally good, but some callers may create a dependency—knowingly or not—on the incorrect value you returned.

This appendix touches on many different kinds of changes, the kinds of things that can be broken by changes, and whether a change is one that the .NET BCL team would generally accept for the core .NET libraries. This content is presented as informative, not prescriptive. If you work on a library with a small audience, you may be significantly more accepting of breaking changes than the lowest layers of the .NET ecosystem.

There are many ways of classifying breaking changes, but this appendix will classify breaks as one or more of the following types: runtime, compilation, recompile, or reflection.

A *runtime* break is an observable change in existing programs caused by using a newer version of a changed library. Runtime breaks are indicated in this chapter by .

A *compilation* break occurs when an existing program or library fails to recompile when referencing a newer version of a changed library. Compilation breaks are indicated by .

A *recompile* break arises when an existing program or library encounters different behavior merely by recompiling against a newer version of a changed library. Recompile breaks are indicated by .

A *reflection* break is an observable change in existing programs, caused by using a newer version of a changed library, that can only be experienced when using reflection to access data. In this appendix, breaks classified as runtime breaks or compilation breaks are not also considered reflection breaks unless the reflection break has a unique aspect. Reflection will immediately be aware of any metadata changes and—depending on the caller code—may or may not behave differently. Metadata-only reflection changes are not discussed here unless there's an interesting aspect to consider. Reflection breaks are indicated by .

In .NET API Reviews, “compilation breaks” and “recompile breaks” are often grouped together as “source-breaking changes.” However, this appendix makes a distinction so as to better explain the consequences of each kind of change.

Like the majority of Framework Design Guidelines, this appendix focuses on public types, and on the public or protected members on those types (unless otherwise specified). Reflection-based access to internal or private members can always be broken by those members being removed or altered; it is generally mentioned here in conjunction with a runtime or compile-time aspect of a change.

Finally, changes that would generally be permitted by the .NET BCL team are indicated with , and changes that would generally not be permitted are indicated with . A few of the types of changes are very contextual, and are marked with .

D.1 Modifying Assemblies

D.1.1 Changing the Name of an Assembly (

Changing the name of an assembly, while keeping the filename unchanged, will result in type load failures for any process that needs a type from your

assembly. This failure can manifest in nonintuitive exceptions at runtime, such as a `FileNotFoundException`. If you change the assembly name and the filename, then you've created an entirely new assembly with a duplicate type hierarchy, and the runtime will continue to use the previous assembly unless some mechanism deleted the older file.

The compile-time behavior for a user of your library after such a change depends on multiple factors, such as whether you maintained the filename, the mechanism by which the caller gets your updates, and the mechanism by which the final application gets your updates. It can range from a failure to compile due to an unresolvable reference, to producing the same nonfunctioning library, to producing a working output.

Reflection loads of your type will fail if they use the assembly-qualified name. If something has already caused your assembly to be loaded into the process, then other reflection loads might succeed, but would fail to match an `AssemblyQualifiedName` after type resolution.

The .NET BCL team generally considers changing an assembly name to be an unacceptable breaking change. An assembly rename can be effectively accomplished by producing one final version of the existing assembly that uses `[TypeForwardedTo]` to resolve the runtime and compile-time failures and ambiguities, but you should strongly prefer to maintain the existing name.

D.2 Adding Namespaces

D.2.1 Adding a Namespace That Conflicts with an Existing Type (X)

In the .NET CLR, a namespace name is only used as a prefix to type names; namespaces aren't runtime concepts. Adding a namespace that conflicts with an existing type (e.g., if a namespace named "System.Math" was created after the type `System.Math`) will result in a compile-time break for any libraries that can see both the type and the namespace, and that have a namespace-qualified reference to the type—with an error like CS0434 (The namespace 'NamespaceName1' in 'Assembly1' conflicts with the type 'TypeName1' in 'Assembly2').

The .NET BCL team considers this an unacceptable breaking change. Following Framework Design Guidelines, namespaces should have a structured prefix like “[CompanyName].[ProductName].” Moreover, within a single product, the types and namespaces should be managed to avoid conflicts. Seek out a synonym for the existing type name to use as the new namespace name.

D.3 Modifying Namespaces

D.3.1 Changing the Name or Casing of a Namespace (X○)

The .NET CLR considers the namespace to be part of the type name, and type names are case-sensitive. Changing the name of a namespace, including the casing of the name, is logically the same as deleting all of the types from that namespace and creating new, unrelated types in the new namespace. See section D.5.1 for the impact of removing a type.

The .NET BCL team considers this an unacceptable breaking change.

D.4 Moving Types

D.4.1 Moving a Type via [TypeForwardedTo] (✓⊗-)

There are two flavors of forwarding a type: into an existing dependency and into a new dependency. Generally speaking, moving a type into an existing dependency does not create any problems, as all applications that use the origin library must already have the destination library.

Moving a type into a new library, or into an existing library that was not previously a dependency, can cause missing dependency problems depending on the update mechanisms involved. Package dependencies from systems like NuGet, as well as new versions of published shared runtimes, will usually handle the dependency adequately. Like all new dependencies, this type of change can cause the total copy size of a stand-alone application to increase.

Reflection-based code can be affected because the assembly-qualified name of the type changes. While lookup methods such as `Type.GetType(...)` will find the `[TypeForwardedTo]`, post-match analysis

on values like the `AssemblyQualifiedName` property of the `Type` object will fail.

The .NET BCL team considers moving a type in this way to be an acceptable breaking change, when there's a technical reason why it's required.

D.4.2 Moving a Type Without [TypeForwardedTo] (X X U)

Because the assembly name is part of the type identity to the .NET CLR, moving a type without using `[TypeForwardedTo]` is logically the same as deleting the existing type and creating a new type with an identical namespace-qualified name in the new assembly. See section D.5.1 for information on the impact of removing types.

If your callers have both the existing assembly and the new assembly as references for their library or executable, then no compile-time break will occur and the recompiled output will be restored to a working condition.

The .NET BCL team considers this an unacceptable breaking change.

D.5 Removing Types

D.5.1 Removing Types (X X U)

It should come as no surprise that removing a type is a runtime breaking change. The .NET CLR loads types on an as-needed basis, so an application can sometimes run to completion with no errors, depending on the data-driven execution flow. When the executing code gets "close enough" to needing the type you've deleted, then the CLR will attempt to resolve the type, fail, and throw a `TypeLoadException`. The exact definition of "close enough" is complex, but it typically means at the beginning of a method that invokes a static member from your type, has a local variable of your type, instantiates your type, or loads another type that has your type as a field, base class, or implemented interface. JIT inlining lookahead and other preload runtime features can move "close enough" to an earlier point in program execution.

As your type no longer exists, libraries and executables that reference your assembly will fail to compile if they used your type in any manner.

The .NET BCL team considers this an unacceptable breaking change.

D.6 Modifying Types

D.6.1 Sealing an Unsealed Type (✗ ✖)

Both the compilation break and the runtime break from sealing an unsealed type are dependent on someone having already inherited from the type you seal. Compilation will fail with error CS0509 ('class1' : cannot derive from sealed type 'class2'), and any use of the type at runtime—including invoking static members—will result in a System.TypeLoadException with a message like “Could not load type ... because the parent type is sealed.”

The .NET BCL team considers this an unacceptable breaking change.

D.6.2 Unsealing a Sealed Type (✓ ✎)

The only potentially breaking impact of unsealing a sealed type is in metadata such as `Type.IsSealed`. Reflection loads may misidentify your type or mishandle data based on that metadata.

The .NET BCL team considers this an acceptable breaking change.

D.6.3 Changing the Case of a Type Name (✗ ✖ ○ ✎)

The .NET CLR considers type names to be case-sensitive. Changing the casing of a type is therefore logically the same as removing the existing type and creating a new one with a similar-looking name. For information on the impact of removing a type, see section D.5.1.

The compile-time behavior of your change depends on the language that the caller uses. Some languages, such as VB.NET, are case-insensitive, so recompiling will succeed and the output assembly will function. Case-sensitive languages, such as C#, will fail to compile.

Reflection-loading your type will fail using the standard reflection loads, but reflection loads allow for a case-insensitive comparison, which will allow reflection to load your type. The case change you introduced will be reflected in `Type.Name` and other metadata with the type name in it, so reflection code may still react differently after your change.

The .NET BCL team considers this an unacceptable breaking change.

D.6.4 Changing a Type Name (X ✘)

The .NET CLR has no mechanism for a type rename, so changing the name is logically the same as creating a new type with similar members as the old type while removing the old type. For information on the impact of removing a type, see section D.5.1.

The .NET BCL team considers this an unacceptable breaking change.

D.6.5 Changing the Namespace for a Type (X ✘ Q ☠)

Because the .NET CLR considers the namespace to be part of the name of a type, changing the namespace has the same effect as changing the name of the type. See section D.6.4 for information on the impact of changing a type name.

Changing the namespace without changing the name of the type can result in a successful recompile of the assembly if the caller already has a `using-import` statement for the destination namespace, unless the caller used a namespace-qualified type reference.

Reflection will fail to find the type with a namespace-qualified name. If the type is found by the `Type.Name` value alone, then it may still be treated differently due to having a different namespace-qualified name.

The .NET BCL team considers this an unacceptable breaking change.

D.6.6 Adding `readonly` on a `struct` (✓ Q)

Adding the `readonly` modifier to a `struct` will instruct the compiler that no “defensive” copies are required when invoking members on a `readonly` field or a `struct` passed to a method as a `readonly ref (in)` parameter. Your callers only get the benefit after recompiling.

The only risk of this change is that callers may experience a significant reduction in their code execution time after recompiling, which could uncover a latent race condition.

The .NET BCL team considers this an acceptable change.

D.6.7 Removing `readonly` from a `struct` (X ✘ Q)

Like adding the `readonly` modifier, removing it only changes the behavior of the calling assembly after it is recompiled. If you remove the

`readonly` modifier, then existing binaries with `readonly` fields of your type, or methods that accept your type as a `readonly ref (in)` parameter, will not be making the required copies of the `struct` value when invoking methods. Any mutation you perform in existing methods after removing the `readonly` modifier will be made visible to those callers until they recompile, which can lead to very difficult-to-diagnose bugs.

The .NET BCL team considers this an unacceptable breaking change.

D.6.8 Adding a Base Interface to an Existing Interface (XIOC)

Adding a base interface will work when all classes implementing the existing interface would conform to the sub-interface via “duck typing.” If the type does not have a valid public member for one of the members on the base interface, that type will fail to initialize, with a `TypeLoadException` reporting that the missing member does not have an implementation. The `TypeLoadException` can still occur if you create the base interface as an exact copy of the existing interface whenever the implementing classes use explicit interface implementation.

The same situations that result in a `TypeLoadException` from the runtime will cause compile-time breaks for the affected types.

The .NET BCL team considers this an unacceptable breaking change.

D.6.9 Adding the Second Declaration of a Generic Interface (XO)

When a type implements the same generic interface with different generic type arguments—such as `IEnumerable<string>` and `IEnumerable<int>`—the type is always ambiguous for generic method type inference based on that interface. Because generic inference is a compile-time language feature, this change does not have any runtime impact.

Users of your type may fail to compile with a variety of errors, depending on the specific call pattern. If they called a generic extension method, then the error may be CS1061 (error: 'Type' does not contain a definition for 'Method' and no accessible extension method ...). They will have to either use the specified generic invocation syntax (e.g., `val.Count<string>()`) or cast the expression (e.g., `((IEnumerable<string>)val).Count()`) to replace their previously functioning code.

This change should be visible on the type you’re modifying, because any members on the duplicate generic interface that use the generic type parameter will need to be explicitly implemented due to their conflicting signature.

The .NET BCL team considers this an unacceptable breaking change. Since generic type inference is already impaired when the same generic interface was specified the second time, it’s generally not further breaking for the third declaration or beyond.

D.6.10 Changing a class to a struct (✗ ✖ ○○)

Depending on how your type is used, the runtime failures from changing a `class` to a `struct` can vary. The most common failure is one in which the .NET CLR throws a `TypeLoadException` with a message like “Could not load type ... due to value type mismatch.” Other failures can include a `MissingMethodException` if your type previously had a zero-argument constructor.

Any caller that used your `class` as a generic type argument for a generic type parameter with the `class` restriction will experience a compile-time break, as will any callers that extended your type when it was a `class`. Otherwise, compilation will likely succeed and produce working output.

The .NET BCL team considers this an unacceptable breaking change.

D.6.11 Changing a struct to a class (✗ ✖ ○○)

Changing a `struct` to a `class` will generally result in the CLR throwing a `TypeLoadException` with a message like “Could not load type ... due to value type mismatch.”

Any caller that used your `struct` as a generic type argument for a generic type parameter with the `struct` restriction will experience a compile-time break. Likewise, any caller that used your `struct` directly, or indirectly as a field of another `struct`, as a generic type argument for a generic type parameter with the `unmanaged` restriction will encounter a compile-time break. Additionally, callers that initialized your `struct` through definite assignment will be broken at compile-time because reference types do not support definite assignment.

After successfully recompiling, callers may find it difficult to debug away any `NullReferenceException` throws that result from their code changing from operating on the (valid) `default` instance of your type as a `struct` to the (invalid) `null` value for your type as a `class`.

The .NET BCL team considers this an unacceptable breaking change.

D.6.12 **Changing a struct to a ref struct (✗)**

In .NET, `ref struct` types are very restricted: They cannot be used as generic type arguments, they cannot be used as the field of a type other than another `ref struct`, and they cannot be boxed. If your callers ever boxed the `struct` that you converted to a `ref struct`, then the CLR will throw an `InvalidOperationException`. If they used it as a generic type argument to a generic method, then the CLR will throw a `BadImageFormatException`. If they used it in an `async` method, in an iterator (`yield return`) method, as the type of a field of a `class` or (non-`ref`) `struct`, or as a generic type argument, then the CLR will throw a `TypeLoadException` stating that “A `ByRef`-like type cannot be used as the type for an instance field in a non-`ByRef`-like type.”

All of the scenarios that fail at runtime will result in a compile-time break for your callers.

Reflection code that instantiates the `struct` will likely fail with an `InvalidOperationException`, due to boxing of the result. The `ByRef` nature of the `struct` is visible to reflection, so reflection code may behave differently after your change.

The .NET BCL team considers this an unacceptable breaking change.

D.6.13 **Changing a ref struct to a (Non-ref) struct (✓)**

There are no compile or runtime changes from changing a `ref struct` to a “regular” `struct`. However, undoing the change later would be a breaking change, as described in section D.6.12.

The .NET BCL team considers this an acceptable change, but would warn you that it cannot be safely undone.

D.7 Adding Members

D.7.1 Masking Base Members with new (✗〇)

Existing binaries are unaffected when you use the new modifier to declare a member with the same name and parameter list as a member from a base class. If your new method has a compatible return type to the base member, then callers will successfully recompile. However, after recompiling, they will be calling your new method instead of the method from the base class—provided they have a static type reference to your type or a more derived type.

If your return type is not compatible with the base member and the caller saved or operated on the member return value, the caller will experience a compile break from your change.

The .NET BCL team considers this an unacceptable breaking change, with the exception of static “Create” methods specializing their return type to avoid casting. If your method predicated the base member and you are adding the new modifier to suppress the compiler warning, that is also acceptable. Otherwise, the .NET BCL team would recommend choosing a new name for your method.

D.7.2 Adding abstract Members (✗Ἀ〇)

If you add a new abstract member to a type, then any derived types that were not already declared as abstract will fail to initialize; a TypeLoad Exception will display a message indicating that the new member “does not have an implementation.”

Non-abstract derived types will fail to compile due to not providing an implementation for the new member.

The .NET BCL team considers this an unacceptable breaking change.

D.7.3 Adding Members to an Unsealed Type (✓〇)

Aside from adding new abstract members (section D.7.2), adding new members on an unsealed type—or a sealed type—has no runtime impact, even if a derived type already declared a member with the same name and a compatible parameter list.

At compile-time, any derived type that has already declared a member with the same name and a compatible parameter list will receive a compiler warning (CS0108: 'member1' hides inherited member 'member2'. Use the new keyword if hiding was intended.). This is a compile-time break for callers that build with the “treat warnings as errors” option.

The .NET BCL team considers this an acceptable breaking change, unless the new member would knowingly conflict with a derived type in a confusing way.

D.7.4 Adding an override Member to an Unsealed Type (✓ ⓘ)

Due to the way the Roslyn compiler—as of 2019—emits base member invocations, a derived type that calls the member you have overridden will skip past your override until it is recompiled to see that you have overridden the member. If your library is distributed via a package management system, such as NuGet, and uses multiple target frameworks, you may need to provide the override for every framework you target, even if the implementation is just to call the base member.

The .NET BCL team considers this an acceptable change. Nevertheless, we would caution you to consider what state your object might be in if your override member is skipped.

D.7.5 Adding the First Reference Type Field to a struct (✗ ⚡ ⓘ)

Adding a reference type field to a struct that previously lacked any direct or indirect reference type fields can cause some low-level operations to throw exceptions when they previously worked, because the reference type fields make the type unsafe for operations that treat the instance as raw bytes.

Any callers that obtained a pointer to an instance of your struct with the fixed statement, or used your type as the generic type argument to a generic type parameter with the unmanaged constraint, will experience a compile-time break. The same compile-time breaks will arise with any struct that has your struct, directly or indirectly, as a field.

Note that this same concern applies to a generic struct adding the first field based on a generic type parameter, unless the generic type parameter is constrained to be an unmanaged type.

The .NET BCL team considers this an unacceptable breaking change.

D.7.6 Adding a Member to an Interface (✗ ✘ ○)

If you add a new member to an interface, then any implementing types that do not already have a public compatible member will fail to initialize. The `TypeLoadException` will display a message indicating that the new member “does not have an implementation.”

The same types that will fail to initialize at runtime will experience a compile-time break due to not conforming to the interface.

The .NET BCL team considers this an unacceptable breaking change. Theoretically, the Default Interface Methods language and runtime feature allows new members to be declared on an interface without the risk of a `TypeLoadException`. As of 2019, the .NET BCL team does not have enough experience with this feature to recommend its use as a general pattern.

D.8 Moving Members

D.8.1 Moving Members to a Base Class (✓)

Moving a member to a base class is gracefully handled by the .NET CLR and is generally supported for both static and instance members in .NET languages.

The .NET BCL team considers this an acceptable change, but would warn you that it cannot be safely undone.

D.8.2 Moving Members to a Base Interface (✗ ✘ ○)

Unlike moving members to a base class, moving members to a base interface can result in both runtime and compile-time failures for types that explicitly implemented the interface member. At runtime, the type initialization will fail with a `MissingMethodException` due to having an implementation of a nonexistent interface method. At compile-time, the implementing type will fail with errors for explicitly implementing a member that does not exist on the interface as well as for not providing an implementation of an interface member.

The .NET BCL team considers this an unacceptable breaking change.

D.8.3 Moving Members to a Derived Type (✗)

Unlike moving members to a base class, the .NET CLR does not gracefully handle the case of moving members to a derived type. For both runtime and compile-time consideration, moving a member to a derived type is logically the same as deleting the existing member and adding a new one on the derived type. See section D.9.3 for information on the impact of removing a member.

Callers that have values statically typed as the receiving derived type will succeed after a recompile. In contrast, callers with only a base class reference—or an alternate derived type—will fail to compile due to the method not existing.

The .NET BCL team considers this an unacceptable breaking change.

D.9 Removing Members

D.9.1 Removing a Finalizer from an Unsealed Type (✗)

If an unsealed type has a finalizer implemented by calling `Dispose(false)`, any derived types may have been counting on the call to `Dispose(false)` for their own component of the finalization logic. Removing the finalizer from the base class will thus result in the derived type no longer running finalization logic.

The .NET BCL team considers this an unacceptable breaking change, unless the finalizer does not use the Basic Dispose Pattern or otherwise signal derived types.

D.9.2 Removing a Finalizer from a Sealed Type (✓)

Removing a finalizer from a sealed type does have indirectly observable effects: Instances of that type will be available to be cleaned up by faster sweeps of the garbage collector—ones that do not depend on the finalizer thread—even when not properly disposed of. There’s a risk that any disposable classes between `System.Object` and the sealed type may have had side effects from running in a finalizer (`Dispose(false)`); this risk can be assessed by the developer contemplating removing the finalizer.

The .NET BCL team considers this an acceptable breaking change.

D.9.3 Removing a Non-override Member (✗ ✖️)

If you completely remove a member, then any existing callers will receive a `MissingMemberException` from the CLR when the executing code gets “close enough”—typically when entering the calling method for the first time—to the member invocation. Removing a property or method will result in a `MissingMethodException`, and removing a field will result in a `MissingFieldException`.

Any callers will experience a compile-time failure due to accessing the missing member, except for methods where a compatible overload using optional parameters is already present.

The .NET BCL team considers this an unacceptable breaking change.

D.9.4 Removing an override of a virtual Member (✓ ☑)

There is no runtime observable difference between replacing an `override` member with only a call to the base member and deleting it entirely.

The Roslyn C# compiler—as of 2019—encodes base member invocations to the level in the type hierarchy that last overrode it, so recompiling an assembly with a derived type that makes a base call to the member you deleted will now “remember” to skip your type. Adding the override back after you have removed it counts the same as adding an override fresh (see section D.7.4).

The .NET BCL team considers this an acceptable change, but would probably caution against it.

D.9.5 Removing an override of an abstract Member (✗ ✖️)

This item refers to removing the base-most override of an abstract member. If you are removing the override of an override, that counts as removing the override of a virtual member (see section D.9.4).

The runtime behavior of this change depends on the overall class structure. If the member you removed was the only implementation for the member, then derived types will fail to initialize, and a `TypeLoadException` will indicate that the member does not have an implementation. If a derived type further overrode your method, but called yours via a base invocation, then the CLR will throw a `BadImageFormatException` when

execution gets “close enough” to the override method. Finally, if a derived type overrode the method and didn’t call yours via a base invocation, everything works for that type.

Compilation-time failures will occur for any type that doesn’t override the member you deleted, without invoking yours via a base invocation. Either the compiler will report that the derived type does not implement an abstract member, or it will indicate that it is performing a base invocation to a member that does not exist.

The .NET BCL team considers this an unacceptable breaking change.

D.9.6 Removing or Renaming Private Fields on Serializable Types (?

Some serializers in .NET, such as the legacy `BinaryFormatter`, use reflection techniques to serialize both private and public data about a type. Renaming fields, including private fields, with this type of serializer will result in your older payloads not properly deserializing.

When making serializable types, be aware of what data gets serialized. If private data is serialized, you should understand how to make previous payloads compatible before adding, removing, renaming, or changing the semantic meaning of any private state.

The Framework Design Guidelines encourage you to only serialize types purpose-built to be serialized.

The .NET BCL team generally does not support serialization of BCL types. For the types where serialization is explicitly supported, the serialization breaks must be resolved along with structural changes to the type—which, in practice, means no structural changes happen to serializable types.

D.10 Overloading Members

There are two types of member overloads: simplification overloads (overloads with more or fewer parameters, but for each position the parameter is the same as the longer overload, or the parameter is missing) and alternative overloads (overloads that change the type in a given position from some other overload). The breaking changes that are possible from adding overloads depend on which type of overload is added.

D.10.1 Adding the First Overload of a Member (✓)

Reflection has accelerators, such as `Type.GetMethod(string)`, that only succeed when overloading is not being used for a member. When you add the first overload (that is, the second member with the same name), these reflection methods will start to throw an `AmbiguousMatchException`.

If your first overload is an alternative overload, then callers that pass `default` will fail at compile-time for having an ambiguous method invocation.

The .NET BCL team considers this an acceptable breaking change, unless `default` is a sensible value for your existing method and there is a reasonable expectation of callers specifying it at the call site.

D.10.2 Adding Alternative-Parameter Overloads for a Reference Type Parameter (?)

When two different overloads accept reference types, the `null` (or `default`) literal value is ambiguous across the two members. If the existing member throws an `ArgumentNullException` when the varied parameter is `null`, then the .NET BCL team considers it an acceptable breaking change to introduce a compile-time break for those callers that pass `null`. However, if the `null` value is accepted by the member, then overloading is only an acceptable change if some other portion of the parameter list forces uniqueness to remove the ambiguity of a `null` literal.

D.11 Changing Member Signatures

D.11.1 Renaming a Method Parameter (✗)

Renaming a method parameter can cause a compile-time break in languages, such as C#, that permit calling methods with named arguments.

Most callers don't specify named parameter syntax for required parameters, so the risk of introducing a break is minimal; however, the benefit of renaming the parameter is slight. The .NET BCL team considers this an unacceptable breaking change.

D.11.2 Adding or Removing a Method Parameter (✗🏃‍♂️)

Methods in the .NET CLR are identified by their signature, which is composed of the name, return type, and ordered list of parameters. Removing a parameter, adding a required parameter, or adding an optional parameter all count as changing the signature and, therefore, are logically the same as deleting the original method. See section D.9.3 for information on the runtime impact of removing a member.

Adding an optional parameter will not cause a compilation break, but adding a required parameter will. Removing an optional parameter will not cause a compilation break unless a caller specified it, but removing a required parameter will cause such a break unless a compatible overload with optional parameters is already available.

The .NET BCL team considers this an unacceptable breaking change, and would recommend adding overloads as the safer alternative to this type of change.

D.11.3 Changing a Method Parameter Type (✗🏃‍♂️⌚---

Methods in the .NET CLR are identified by their signature, which is composed of the name, return type, and ordered list of parameters. Changing a parameter type is logically the same as deleting the original method. See section D.9.3 for information on the runtime impact of removing a member.

Recompilation will fail if the parameter uses the `ref`, `out`, or `in` calling convention, unless there is an overload with optional parameters that is compatible. For simple parameters, recompilation will generally succeed if you changed to a less derived type, or a type that is implicitly convertible from the previous type.

Reflection will be broken when finding a method by signature. Reflection invocation will generally succeed if you changed the parameter to a less derived type, but will fail and throw an `ArgumentException` if a user-defined implicit conversion operator is required to convert the provided value to the new parameter type.

The .NET BCL team considers this an unacceptable breaking change, and would recommend method-adding overloads as a safer alternative to this type of change.

D.11.4 Reordering Method Parameters of Differing Types (X🏃‍♂️⚠)

At runtime, reordering method parameters is just a special case of changing a method parameter type. See section D.11.3 for more information.

Callers using the named parameter syntax will not be broken during compilation, but callers using positional syntax will.

Reflection will be broken, via an `ArgumentException`, if it's being used to invoke the method after finding it by name, or will fail to find the method by signature.

The .NET BCL team considers this an unacceptable breaking change.

D.11.5 Reordering Method Parameters of The Same Type (X🏃‍♂️⚠)

Swapping the position of two parameters with the same type and same calling type produces an identical signature as far as the runtime is concerned; thus, the CLR will not throw a `MissingMethodException`. Since parameters are passed by position, not by name, the runtime breaking behavior will be the same as if you swapped the meaning of the parameters without changing their names.

Any caller that used named parameter syntax for only one of the swapped parameters will encounter a compilation failure. Callers specifying all parameters with named syntax will encounter a recompilation break—in this case, a behavior fix. Callers using only positional argument calling will neither fail nor experience different behavior after recompilation.

The .NET BCL team considers this an unacceptable breaking change.

D.11.6 Changing a Method Return Type (X🏃‍♂️⚠)

Methods in the .NET CLR are identified by their signature, which is composed of the name, return type, and ordered list of parameters. Changing the return type is logically the same as deleting the original method. See section D.9.3 for information on the runtime impact of removing a member.

Recompilation will typically succeed if you change the return type to a more derived type. In contrast, it is likely to fail when if you change the return type to a less derived type or an unrelated type.

The .NET BCL team considers this an unacceptable breaking change, and would recommend adding a new method group (not overloading) or a new type to make this type of change.

Note that this only applies to changing the return type in the method signature. Changing the method behavior to return a value of a type that is derived from the method return type is discussed in D.12.3.

D.11.7 **Changing the Type of a Property (✗ ↵ ○)**

During compilation, property reads are translated to invocations of the property get-method, and property writes are translated to invocations of the property set-method. Changing the property type changes the return type of the get-method, and thus produces all of the breaks discussed in section D.11.6. Changing the type also changes the parameter type, and the return type, of the set-method for settable properties; thus, it produces all of the breaks discussed in sections D.11.3 and D.11.6.

The .NET BCL team considers this an unacceptable breaking change, and would recommend adding a new method or property if this functional change is required.

D.11.8 **Changing Member Visibility from public to Any Other Visibility (✗ ↵ ○)**

Reducing member visibility will cause compile-time breaks for any callers that do not meet the new visibility restrictions.

At runtime, the .NET CLR will throw a `MemberAccessException` (`MethodAccessException` or `FieldAccessException`) when attempting to invoke the method from the calls that no longer meet the new visibility restrictions. Unlike member removal, this runtime break executes all code in the calling method up to the failed member invocation.

The .NET BCL team considers this an unacceptable breaking change.

D.11.9 **Changing Member Visibility from protected to public (✓)**

No runtime or compile-time breaks happen when you increase the visibility of a nonvirtual method (see the next section for discussion of virtual methods). However, this change cannot safely be undone.

The .NET BCL team considers this an acceptable change.

D.11.10 Changing a virtual (or abstract) Member from protected to public (X ✘ O)

Whether you will cause a runtime break from changing a `protected` virtual member to a `public` virtual member depends on whether any derived types have overridden the method. If the method had no overrides, then there's no runtime impact. However, any types that did override the method will fail to initialize with a `TypeLoadException` indicating that the override cannot reduce the method access.

Similar to the runtime behavior, a compile-time break will be present for any derived type that overrode the member because the override re-asserts the visibility level.

The .NET BCL team considers this an unacceptable breaking change. If you were contemplating changing the method to `public`, you may instead want to add a `public` method on the original class that calls the `virtual` method, similar to the Template Method Pattern.

D.11.11 Adding or Removing the static Modifier (X ✘ O)

In addition to the name, return type, and ordered list of parameters, the .NET CLR considers whether a member is an instance member or a static member to be part of the signature. Adding or removing the `static` modifier on a member is logically the same as removing the previous member. See section D.9.3 for information on the runtime impact of removing a member.

Any caller that used a type-qualified static member invocation, or an instance-qualified instance member invocation, will get a compile-time break for using the wrong calling format for the member. Callers from derived types that use implicit member invocation will successfully recompile, and their libraries will return to a working state.

The .NET BCL team considers this an unacceptable breaking change.

D.11.12 Changing to or from Passing a Parameter by Reference (X ✘ O)

In the .NET CLR, parameters passed by reference (C# `ref`, `in`, `out`; VB.NET `ByRef`; F# `byref`, `inref`, `outref`) are considered to be a different type than the type under normal parameter passing. Therefore, the .NET CLR

considers changing to, or from, passing by reference to be the same as changing the type of a parameter: The previous member was removed. See section D.9.3 for information on the runtime impact of removing a member.

If you change a parameter to be a reference parameter (C# `ref`) or an output parameter (C# `out`), then all of your callers will get a compile-time break for not calling with the correct modifiers; the same is true for the reverse direction. The C# language allows callers to omit the `in` modifier on call sites, so changing to or from a read-only reference parameter is fixed by a recompile for callers that omit the modifier. Callers that specify the modifier, including callers in any languages where it is required, will get a compile-time break.

The .NET BCL team considers this an unacceptable breaking change.

D.11.13 Changing By-Reference Parameter Styles (✗ ⓘ ⓘ)

Whether a parameter is fully by-reference (C# `ref`), an output parameter (C# `out`), or a read-only reference (C# `in`) is not part of the signature for method calling purposes. Thus, changing a parameter's by-reference style in one of your members will not result in a `MissingMemberException` in the same way that changing it to or from a value parameter would. However, writing through to a parameter previously declared as a read-only reference, or depending on the value of a parameter previously declared as an output parameter, is likely to cause very difficult-to-diagnose bugs.

Whether changing the parameter by-reference style will generate a compile-time break depends on the language that the caller uses. C# uses three different modifiers at the call sites (`ref`, `out`, `in`, or `implicit-in`), and it is a compile failure to use the wrong one. F# uses the same modifier for all three styles, so a recompile for F# callers will likely succeed unless the previous declaration had the parameter as an output parameter—where the caller may not have initialized the value before calling your method.

The .NET BCL team considers this an unacceptable breaking change.

D.11.14 Adding the `readonly` Modifier to a `struct` Method (✓ ⓘ)

Just like adding the `readonly` modifier to a `struct` (section D.6.6), adding this modifier to a method only has an effect after your caller recompiles.

The .NET BCL team considers this an acceptable change.

D.11.15 Removing the `readonly` Modifier from a `struct`

Method (✗ ✘)

Just like removing the `readonly` modifier from a `struct` (section D.6.7), removing this modifier from a method only has an effect after your caller recompiles. When the caller is executing its existing assembly with your updated assembly, data mutation may be visible in ways that will no longer be reproduced after the assemblies recompile.

The .NET BCL team considers this an unacceptable breaking change.

D.11.16 Changing a Parameter from Required to Optional (✓)

Optional parameters, also known as default parameters or default arguments, are a compile-time feature, not a runtime feature. Therefore, there is no runtime impact when you change a parameter from required to optional.

When making parameters optional, be careful not to introduce ambiguity between member overloads when only some parameters are specified. The Framework Design Guidelines encourage you to have optional parameters on only one overload to avoid ambiguous member invocations.

The .NET BCL team considers this an acceptable change, provided the change does not require parameter reordering and does not introduce the possibility of an ambiguous member invocation.

D.11.17 Changing a Parameter from Optional to Required (✗ ✘)

Optional parameters, also known as default parameters or default arguments, are a compile-time feature, not a runtime feature. Therefore, there is no runtime impact when you change a parameter from optional to required.

When you change a parameter from optional to required, you need to consider callers that did not explicitly provide an argument for the parameter, callers that called the method with some optional parameters by position, and callers that called the method with some optional parameters by name. In general, the only safe way to change a parameter from required to optional is to introduce a new, longer, member overload that first gives all of the previously optional parameters as optional—with the same default values—and then places any new optional parameters at the end

of the parameter list. When you add that new overload, any callers that did not specify all of the optional parameters explicitly will fail to recompile because the target member is ambiguous, forcing you to change the previous member's parameters from optional to required.

The .NET BCL team considers this an acceptable breaking change when done as part of “moving” default values to a new overload. Otherwise, it is an unacceptable breaking change.

D.11.18 Changing the Default Value for an Optional Parameter (X[✓])

Optional parameters are a compile-time feature in which any caller that did not provide a value for a parameter gets the value from the target member's metadata. Changing the default value has no effect on callers until they recompile, which can create confusion where the compiled binary and the developer's understanding of the program from the source code do not match.

The .NET BCL team considers this an unacceptable breaking change due to the possibility of developer confusion.

D.11.19 Changing the Value of a const Field (X[✓])

Constant field values are expected to be constant for all time. When callers use your constant field in their code, the compiler copies the value, not the symbolic reference, into the caller code. Callers are therefore unaware that your constant field has a new value until recompilation, which can create confusion when the compiled binary and the developer's understanding of the program from source code do not match.

When reflection is used to access the constant field at runtime, the updated value is reflected immediately. Disagreement between reflection code and runtime code about the value of a constant field is another source of developer confusion.

The .NET BCL team considers this an unacceptable breaking change due to the possibility of developer confusion. If you have not yet released a library with the constant field, use `static readonly` instead of `const` for values you reserve the right to update in a future version. If you have already released the constant field, then we recommend introducing a new field, either `const` or `static readonly`, as appropriate.

D.11.20 Changing an abstract Member to virtual (✓)

There is no runtime observable difference when you update a member from abstract to virtual. However, you should be aware that all instantiable derived types are already providing an implementation that does not invoke your member via a base member invocation.

The .NET BCL team considers this an acceptable change.

D.11.21 Changing a virtual Member to abstract (✗)

Changing a member from virtual to abstract is breaking in the same ways that removing an immediate override to an abstract member is. See section D.9.5 for information on the full impact of this type of change.

The .NET BCL team considers this an unacceptable breaking change.

D.11.22 Changing a Non-virtual Member to virtual (✓)

Generally speaking, changing a non-virtual member to a virtual member is a safe change. However, some compiler implementations use the MSIL non-virtual call instruction when the target member is known to be non-virtual and the target instance is known to be non-null. Callers that use these compilers, in these situations, will only call the base-most member, not the most derived override implementation.

The .NET BCL team considers this an acceptable breaking change, but would encourage you to consider making the behavior virtual through the Template Method Pattern rather than altering the existing method.

D.12 Changing Behavior

D.12.1 Changing Runtime Error Exceptions to Usage Error Exceptions (✓)

If your method throws a `NullReferenceException` when a parameter is passed as `null`, or an `IndexOutOfRangeException` when a parameter is passed as an out-of-bounds index, you can change it to throw the usage error equivalent exception (e.g., `ArgumentNullException` or `ArgumentOutOfRangeException`) instead. While some callers could, conceivably, depend on the `NullReferenceException` or `IndexOutOfRangeException`, the .NET BCL team believes that the information provided via the usage

error exception about which argument value caused the method to fail is worth the risk.

The .NET BCL team generally considers changing “failed inputs” to “a detailed usage error exception” to be an acceptable breaking change.

D.12.2 Changing Usage Error Exceptions to Functioning Behavior (✓ ✖)

Even though the exceptions are part of a member’s “contract,” the .NET BCL team generally considers allowing previously disallowed inputs to function to be an acceptable breaking change.

D.12.3 Changing the Type of Values Returned from a Method (❓ ✖)

Changing the declared return type of a method is discussed in section D.11.6. The present section addresses changing the implementation of the method to return a value of a different type than the method did in a previous version of your library.

Changing the returned value to be a more derived type than in the previous version of the library is generally a safe change. Callers that interact with the return value only via the type that the method is declared to return would be broken only if the new type implements virtual methods in an incompatible manner. Callers that are “hard-casting” the return value to a more specific type also won’t be broken by your change, since the new return value can be safely cast to the intermediate type. The only sort of caller pattern that is broken is when callers do an equality comparison on `value.GetType()`—which might not be apparent to callers that use the type of the returned value as a key in a dictionary.

If you were already returning a type derived from the declared return type and you change to returning some other compatible type, then you will break both type-testing callers and “hard-casting” callers. For example, if your method is declared to return `IEnumerable<int>`, but always returns `List<int>`, then callers might have depended on the return type. If so, they will get an `InvalidOperationException` if you change the return type to be `HashSet<int>`, or a non-public type that implements `IEnumerable<int>`.

The .NET BCL team is generally supportive of returning a “more derived” type than in a previous version, unless there’s a strong reason to believe type equality checks will be routinely used for that particular method. Returning a value that would break existing “hard-casting”

callers is something the .NET BCL team generally considers an unacceptable breaking change.

D.12.4 Throwing a New Type of Error Exception (✗)

When you throw a new type of exception from your method, or when you allow a new type of exception to escape your method by changing the implementation, you will likely cause the exception to be thrown past any existing catch blocks that your caller has for handling errors in your method.

The .NET BCL team generally considers this an unacceptable breaking change, outside of two specific situations: The exception is a subtype of an exception type that is already thrown by the method (section D.12.5) or using an “uncaught” exception type is necessary for fixing a critical bug.

D.12.5 Throwing a New Type of Exception, Derived from an Existing Thrown Type (✓)

When you throw a new type of exception that derives from an exception type already thrown by your method (e.g., throwing `FileNotFoundException` from a method that already throws `IOException`), you generally won’t bypass existing catch blocks, or otherwise change the exception handling from your callers—with a few caveats.

First, callers that have a catch block for the more derived type in addition to the existing type might—depending on nesting and ordering—have a different catch block trigger for the same kind of problem. While most callers would consider that a “good” breaking change, it’s something to take into account.

The second caveat is that callers using type-equality tests in exception filters, or within catch blocks, will see a runtime behavioral difference. Type-equality testing in exception handlers is much less common than type hierarchy tests (e.g., the `is` operator), but it’s still something to take into account.

The .NET BCL team is generally supportive of throwing more specific exceptions.

D.13 A Final Note

The content of this appendix is not exhaustive. We hope that by understanding the types of impacts that certain changes have, and how the .NET BCL team feels about those impacts, you can form your own opinion on types of breaking changes that we didn't cover.

The general heuristic is: If you're changing an existing member, you're probably breaking someone; and if you're adding new things, you're *probably* not breaking them.

Happy developing!



Glossary

ABSTRACT TYPE: An abstract type can define members for which it does not provide implementations.

APPLICATION MODEL: A type of application. For example, the application models supported by the .NET Framework include console application, Windows Forms application, service application, and ASP.NET application.

ASSEMBLY: A set of code modules and other resources that together are a functional software unit. Usually seen in the form of a single .dll or .exe.

ATTACHED PROPERTY: A kind of Dependency Property modeled as static Get and Set methods representing “properties” describing relationships between objects and their containers (e.g., the position of a Button object on a Panel container).

ATTRIBUTES: The means in managed code of attaching descriptive information to assemblies, types, their members, and parameters.

BOXING: An operation that converts an instance of a value type to an instance of a reference type (object) by copying the instance and embedding it in a new object instance.

CALLBACK: User code that is called by a framework, usually through a delegate.

CLI: The Common Language Infrastructure; an ECMA standard that defines the requirements for implementing a Virtual Execution System (VES).

CLS: The Common Language Specification, which defines a set of rules for writing public APIs that allow language interoperation.

CONSTRUCTED TYPE: A generic type that has type arguments specified—for example, `List<int>`.

DEFAULT CONSTRUCTOR: A constructor that does not have any parameters.

DELEGATE: A callback mechanism in managed code that is functionally equivalent to function pointers.

DEPENDENCY PROPERTY (DP): A property that stores its value in a property store instead of storing it in a type variable (field), for example.

EVENT: The most commonly used form of callbacks (constructs that allow frameworks to call into user code). Under the covers, an event is not much more than a field whose type is a delegate plus two methods to manipulate the field. Delegates used by events have special signatures (by convention) and are referred to as event handlers.

EVENT HANDLER: Special delegate used as the type of events.

EVENT HANDLING METHOD: User code (method) that is called when an event is raised.

EXTENSION METHODS: A language feature that allows static methods to be called using instance method call syntax.

FINALIZER: A method that provides help in releasing unmanaged resources. `System.Object` declares a virtual method `Finalize` (also called the finalizer) that if overridden is called by the GC before the object's memory is reclaimed and can be overridden to release unmanaged resources.

GENERIC METHOD: A method parameterized with the type of data it stores or manipulates. For example:

```
public class Utils {  
    public static void Swap<T>(ref T ref1, ref T ref2){ ... }  
}
```

GENERICs: A feature of the CLR, similar to C++ templates, that allows classes, structures, interfaces, and methods to be parameterized by the types of data they store and manipulate.

GETTER METHOD: A method of a property designed to access the property's current value.

HIGH-LEVEL COMPONENT: A component representing a high level of abstraction—for example, a component used to manipulate windows on the screen (as opposed to manipulating individual pixels).

IMMUTABLE TYPE: Instances of immutable types cannot be modified using their public members after they are instantiated. For example, `System.String` instances cannot be modified. All methods of `System.String` used to manipulate strings leave the original instance unchanged and return a new instance.

INLINING: A performance optimization in which the compiler substitutes a method call for the body of the method.

INSTANCE METHOD: A method associated with a particular instance of a type.

JIT: Just-in-Time compiler. Intermediate language (IL) instructions are compiled to machine code by the JIT before they can be executed.

LAMBDA EXPRESSION: A programming language feature for defining anonymous delegates.

LINQ: Language-Integrated Query. LINQ allows for a uniform, language-integrated programming model for querying data sets independent of the technology used to store that data.

LOSSY AND LOSSLESS CONVERSION: Lossy conversion is a conversion in which some data might be lost. For example, converting from a signed integer to an unsigned integer is lossy. Lossless conversion is a conversion in which data cannot be lost. For example, a conversion from an unsigned 16-bit integer to a signed 32-bit integer is lossless.

LOW-LEVEL COMPONENT: A component representing a low level of abstraction—for example, a component used to manipulate individual pixels on the screen (as opposed to manipulating windows).

MANAGED CODE: Code for which an underlying virtual execution system, such as the CLR, provides a set of services, including walking the stack, handling exceptions, and storage and retrieval of security information.

MEMBER: Methods, properties, nested types, events, constructors, and fields of types.

METADATA: Language-independent information describing the contents of an assembly.

NESTED TYPE: A type defined within the scope of another type, which is called the enclosing type. A nested type has access to all members of its enclosing type.

OBJECTS: Instances of classes.

OVERLOADING: Defining two or more members in the same scope with the same name but different signatures.

PRE-EVENTS AND POST-EVENTS: Pre-events are raised before a side effect takes place. Post-events are raised after a side effect takes place. For example, the `Form.Closing` pre-event is raised before a window closes. The `Form.Closed` event is raised after a window is closed.

PROGRESSIVE FRAMEWORK: Framework that is easy to use in basic scenarios yet powerful enough to be used in advanced scenarios.

PROPERTIES: Essentially, smart fields with the calling syntax of fields and the flexibility of methods.

REFERENCE TYPE: A type whose instances are allocated on the heap and whose variables store locations (references) of the instances rather than storing the actual instance. Object classes and interfaces are examples of reference types.

SEALING: A powerful mechanism that prevents extensibility. You can seal either the class or individual members. Sealing a class prevents users from inheriting from the class. Sealing a member prevents users from overriding a particular member.

SETTER METHOD: A property method designed to set the value of the property.

STATIC FIELD: A field associated with a type, not any particular instance of the type.

STATIC METHOD: A method associated with a type, not any particular instance of the type.

THIS: A special pointer to the object on which invocations of instance members are to operate.

TYPE ARGUMENT: An argument to a generic type. For example, in the following sample, `string` and `int` are both type arguments.

```
Dictionary<string,int> map = new Dictionary<string,int>();
```

TYPE PARAMETER: A parameter in a generic type. For example, in the following declaration, `T` is a type parameter.

```
public class List<T> {  
    ...  
}
```

UNBOXING: When a value type instance has been boxed (embedded in an object instance), unboxing converts it back to a value type instance.

UNMANAGED CODE: Code that does not depend on the CLR for execution.

VALUE TYPE: A type whose variables point directly to its instances rather than the address of the instances, as would be the case for a reference type.

XAML: An XML format used to represent object graphs.

This page intentionally left blank



Index

Symbols

{ } (braces), C# style conventions, 466–469
, (comma), C# style conventions, 475–476
- (hyphens), naming conventions, 53
(*. *), multiline syntax, comments, 482
// ., single line syntax, comments, 482
[TypeForwardedTo], moving types, 532–533
_ (underscore), naming conventions, 53,
 481–482

A

abbreviations, naming conventions, 55–56
abstract classes, type design guidelines,
 98–102
abstract members
 adding, 539
 changing
 to virtual members, 553
 virtual members to, 553
 override members, removing an override
 of an abstract member, 543–544
abstractions
 designing frameworks, 34–36
 extensibility, 239–241
 self-documenting object models, principle
 of, 34–36
AccessViolationException, 276
acronyms
 capitalization, 45–48
 naming conventions, 55–56
adding
 abstract members, 539
 base interfaces to interfaces, 536
 members
 to interfaces, 541
 to unsealed types, 539–540

method parameters, 546
readonly modifier to structs, 535, 550
reference type fields to structs, 540
second declaration to generic interfaces,
 536–537
static modifiers, 549
aggregate components
 component-oriented design, 331–334
 design patterns, 329–338
 designing, 335–338
 factored types, 334–335
alphanumeric characters, naming conventions, 53
API (Application Programming Interface)
 availability, 13–14
 consistency, 34
 designing frameworks
 low barrier to entry, principle of, 23–29
 scenario-driven framework design,
 16–23
 self-documenting object models,
 principle of, 29–36
 exceptions and API consistency, 250
 heavy API design processes, 2
 intuitive API, 33–34
 layered architecture, principle of, 36–39
 naming new versions of existing API,
 58–61
 sample specification, 523–528
 Stopwatch specification, 524–528
 strong typing, 33–34
 unification, 13–14
well-designed frameworks, qualities of, 3
 backward compatibility, 2
 borrowing from existing proven
 designs, 6–7

- consistency, 7–8
- evolution, 7
- expense, 4–5
- integration, 7
- OO design, 2
- prototyping, 2
- simplicity, 3–4
- trade-offs, 6
- ApplicationException*, 274
- applications
 - models, namespaces, 65–66
 - RAD, progressive frameworks, 13
 - architectures (layered), principle of, 36–39
- ArgumentException*, 275–276
- ArgumentNullException*, 275–276
- ArgumentOutOfRangeException*, 275–276
- arguments, validation, 207–210, 480
- arrays
 - collections versus, 302–303
 - design patterns, 430–433
 - usage guidelines, 287–291
- ASCII characters
 - code restrictions, 479
 - naming conventions, 54
 - Unicode escape sequences (`uXXXX`), 479
- assemblies
 - naming conventions, 61–62
 - renaming, 530–531
 - types and assembly metadata, 127–129
- assignment-expression-throw, C# style
 - conventions, 480
- Async Patterns, 502
 - Async method return types, 348–351
 - Async variants of existing
 - synchronous methods, 353–354
 - await foreach, 363–365
 - cancellation, 512–513
 - choosing, 339–341, 503–504
 - Classic Async Patterns, 361, 503–509
 - consistency, 355–361
 - context, 357–358
 - deadlock, 358
 - design patterns, 339–365
 - Event-Based Async Patterns, 361–362, 510–512
 - exceptions to Async methods, 359–361
 - IAsyncDisposable* interface, 362
 - IAsyncEnumerable<T>* interface, 362–365
 - implementation guidelines, 355–361
 - incremental results, 516
 - Out parameters, 512
 - progress reporting, 513–516
 - Ref parameters, 512
 - Task-Based Async Pattern, 341–347
- Task.Status* consistency, 355–357
- ValueTask* structs, 358–359
- ValueTask<TResult>* structs, 358–359
- attached DP design, 369–370
- attributes, usage guidelines, 291–294
- auto-implemented properties, C# style
 - conventions, 479
- await foreach, Async Patterns, 363–365
- await using, *Dispose* Patterns, 393–394

B

- backward compatibility, well-designed frameworks, qualities of, 2
- base classes
 - extensibility, 242–244
 - moving members to, 541
- base interfaces
 - adding to interfaces, 536
 - moving members to, 541
- base members, masking, 539
- BCL type names, C# style conventions, 476
- behaviors, changing, 553–555
- Binary serialization, 493
- boolean parameters, choosing, 205–207
- borrowing from existing proven designs, well-designed frameworks, 6–7
- braces (`{ }`), C# style conventions, 466–469
- breaking changes, 529
 - assemblies, renaming, 530–531
 - behaviors, changing, 553–555
- classes
 - changing structs to, 537–538
 - changing to structs, 537
 - moving members to base classes, 541
- compilation breaks, 530
- finalizers, removing from
 - sealed types, 542
 - unsealed types, 542
- interfaces
 - adding a second declaration to interfaces, 536–537
 - adding base interfaces to interfaces, 536
 - adding members to, 541
 - moving members to base interfaces, 541
- members
 - adding abstract members, 539
 - adding to interfaces, 541
 - adding to unsealed types, 539–540
 - changing member signatures, 545–553
 - masking base members, 539
 - moving to base classes, 541
 - moving to base interfaces, 541

- moving to derived types, 542
 overloading, 544–545
 override members, 540
 removing an override of a virtual member, 543
 removing an override of an abstract member, 543–544
 removing non-override members, 543
 namespaces, adding namespaces that conflict with existing types, 531–532
 private fields
 removing from serializable types, 544
 renaming on serializable types, 544
 recompile breaks, 530
 reflection breaks, 530
 runtime breaks, 529
 structs
 adding first reference type field to, 540
 adding readonly modifier to structs, 535–536
 changing classes to, 537
 changing to classes, 537–538
 ref structs, 538
 types
 changing namespaces, 535
 moving, 532–533
 moving members to derived types, 542
 names, case sensitivity, 534
 removing, 533
 removing finalizers from sealed types, 542
 removing finalizers from unsealed types, 542
 removing private fields on serializable types, 544
 renaming private fields on serializable types, 544
 sealing unsealed types, 534
 unsealing sealed types, 534
 brevity, naming conventions, 52
 buffer operators
 arrays, 430–433
 data transformation operations, 445–451
 design patterns, 430–445
 fixed sizes, 451–452
 OperationStatus value, 458–463
 partial writes to buffers, 458–463
 predetermined sizes, 451–452
 Spans, 431–445
 Try-Write Pattern, 452–458
- C**
- C#**
- coding style conventions, 465–466
 ASCII characters, code restrictions, 479
 assignment-expression-throw, 480
 auto-implemented properties, 479
 BCL type names, 476
 braces ({ }), 466–469
 collection initializers, 478
 commas (,), 475–476
 comments, 482–483
 expression-bodied members, 478–479
 file organization, 483–485
 if.throw, 480
 indents, 465–466
 language keywords, 476
 member modifiers, 473–475
 nameof(.) syntax, 479
 naming conventions, 480–482
 object initializers, 477–478
 readonly modifiers, 479
 spaces, 469–470
 this.476
 Unicode escape sequences (uXXXX), 479
 var keyword, 476–477
 vertical whitespace, 472–473
 language-specific names, naming conventions, 57
- C++, language-specific names, naming conventions, 57**
- callbacks, extensibility, 231–237
 camelCasing, 43
 C# style conventions, 481–482
 naming conventions, 481–482
 parameter names, 79
 cancellation, Async Patterns, 512–513
 capitalization, 42
 acronyms, 45–48
 case sensitivity, 51–52
 compound words, 48–51
 identifiers, 42–44
 camelCasing, 43
 PascalCasing, 42–44
 case sensitivity
 capitalization, 51–52
 type names, 534
 change notification events, DP, 371
 changing
 abstract members to virtual members, 553
 behaviors, 553–555
 classes to structs, 537
 constant field values, 552

- default values, in optional parameters, 552
- member
 - signatures, 545–553
 - visibility, 548–549
- method
 - parameters types, 546
 - return types, 547–548
- non-virtual members to virtual members, 553
- notification events in properties, 163–165
- optional parameters to required, 551–552
- property types, 548
- reference parameters, 549–550
- required parameters to optional, 551
- runtime error exceptions to usage error exceptions, 553–554
- structs to classes, 537–538
- type names
 - case sensitivity, 534
 - changing namespaces, 535
- usage error exceptions to functioning behavior, 554
- values returned type, from a method, 554–555
- virtual members to abstract members, 553
- choosing
 - boolean parameters, 205–207
 - enum parameters, 205–207
 - exceptions for throwing, 260–264
 - member
 - methods, 152–158
 - properties, 152–158
- classes
 - abstract classes, type design guidelines, 98–102
 - base classes
 - extensibility, 242–244
 - moving members to, 541
 - changing
 - classes to structs, 537
 - structs to classes, 537–538
 - defined, 84
 - members, moving to base classes, 541
 - naming conventions, 67–70
 - common types, 71
 - enumerations, 72–74
 - generic type parameters, 70–71
 - static classes
 - defined, 84
 - type design guidelines, 102–104
 - type design guidelines, 89–100, 102–104
 - unsealed classes, extensibility, 228–229
- Classic Async Patterns, 361, 503–509
- Close() method, Dispose Patterns, 382–383
- CLR (Common Language Runtime)
 - language-specific names and naming conventions, 57
- collections
 - arrays versus, 302–303
 - custom collections, 302–303
 - initializers, C# style conventions, 478
 - live collections, 301–302
 - parameters, 296–297
 - properties, 298–302
 - return values, 298–302
 - snapshot collections, 301–302
 - usage guidelines, 294–296
 - arrays versus collections, 302–303
 - collection properties, 298–302
 - custom collections, 302–303
 - live collections, 301–302
 - parameters, 296–297
 - return values, 298–302
 - snapshot collections, 301–302
- ComException, 278
- commas (,), C# style conventions, 475–476
- comments
 - C# style conventions, 482–483
 - "I" usage, 483
 - multiline syntax /* . */), 482
 - passive voice, 483
 - personification, 483
 - single-line syntax // .), 482
 - "we" usage, 483
- common names, naming conventions, 57–58
- compatibility (backward), well-designed frameworks, 2
- compilation breaks, 530
- component-oriented design, 331–334
- compound words, capitalization, 48–51
- ConfigureAwait modifier, await using, 393–394
- consistency
 - Async Patterns, 355–361
 - self-documenting object models, principle of, 34
 - Task.Status, 355–357
 - well-designed frameworks, qualities of, 7–8
- constant field values, changing, 552
- constructors
 - designing, 165–172
 - type constructors, 172–175
- contravariance, design patterns, 412–417
- conversion operators, 198–C05.1827
- core namespaces, 66

costs, well-designed frameworks, 4–5
 covariance, design patterns, 412–415, 417–423
 customizing
 collections, 303–305
 event handlers, obsolete guidance,
 491–492
 exceptions
 designing, 279–280
 obsolete guidance, 492–493

D

Data Contract serialization, 493, 495–499
 data transformation operations, 445–451
 DateTime struct, usage guidelines, 306–308
 DateTimeOffset struct, usage guidelines,
 306–308
 deadlock, Async Patterns, 358
 declarations, adding to interfaces, 536–537
 default values, changing in optional parameters, 552
 derived types, moving members to, 542
 design patterns
 aggregate components, 329–338
 arrays, 430–433
 Async Patterns, 339–365, 502
 cancellation, 512–513
 choosing between Async Patterns,
 503–504
 Classic Async Patterns, 503–509
 Event-Based Async Patterns, 503–504,
 510–512
 incremental results, 516
 Out parameters, 512
 progress reporting, 513–516
 Ref parameters, 512
 buffer operators, 430–445
 arrays, 430–433
 fixed sizes, 451–452
 OperationStatus value, 458–463
 partial writes to buffers, 458–463
 predetermined sizes, 451–452
 Spans, 431–445
 Try-Write Pattern, 452–458
 contravariance, 412–417
 covariance, 412–415, 417–423
 Dispose Patterns, 372–394, 511–517
 DP, 366–372
 factories, 394–399
 LINQ, 400–408
 optional features, 408–411
 Spans, 431–445
 Template Method Pattern,
 423–425

timeouts, 426–427
 Try-Write Pattern, 452–458
 XAML readable types, 427–430
 .Design subnamespaces, 489
 designing
 aggregate components, 335–338
 component-oriented design,
 331–334
 constructors, 165–175
 custom exceptions, 279–280
 error messages, 264–265
 events, 175–180
 extensibility, 227–228
 abstractions, 239–241
 base classes, 242–244
 callbacks, 231–237
 events, 231–237
 limiting, 244–246
 protected members, 230
 sealing, 244–246
 unsealed classes, 228–229
 virtual members, 237–239
 fields, 180–183
 frameworks, 3, 9–11, 15–16
 abstractions, 34–36
 backward compatibility, 2
 consistency, 7–8
 evolution, 7
 existing proven designs, borrowing
 from, 6–7
 expense, 4–5
 integration, 7
 low barrier to entry, principle of, 23–29
 multiframework platforms, 12–13
 OO design, 2
 programming languages, 11–12
 progressive frameworks, 12–15
 prototyping, 2
 scenario-driven framework design,
 16–23
 self-documenting object models,
 principle of, 29–39
 simplicity, 3–4
 trade-offs, 6
 members
 boolean parameters, 205–207
 choosing methods, 152–158
 choosing properties, 152–158
 conversion operators, 198–C05.1827
 enum parameters, 205–207
 explicit implementation of interface
 members, 148–152
 extension methods, 184–192

- fields, 180–183
- inequality operators, 200–202
- members with variable number of
 - parameters, 214–218
- operator overloads, 192–198
- overloading members, 136–148
- parameter argument validation,
 - 207–210
- parameter passing, 210–214
- parameters, 202–204
- pointer parameters, 218–219
- tuples in member signatures, 220–226
- parameters, 202–204
 - argument validation, 207–210
 - boolean parameters, 205–207
 - enum parameters, 205–207
 - members with variable number of
 - parameters, 214–218
 - passing, 210–214
- properties, 158–160
 - change notification events, 163–165
 - indexed properties, 161–163
- types, 84–85
 - assembly metadata and types, 127–129
 - classes, 89–104
 - constructors, 172–175
 - enums, 111–124
 - interfaces, 92–100, 104–106
 - namespaces, 85–88
 - nested types, 124–127
 - strings, 129–133
 - structs, 89–92, 106–111
- diacritical marks, naming
 - conventions, 55
- Dispose (bool) method, Dispose Patterns, 376–380
- Dispose (true) method, Dispose Patterns, 377–378
- Dispose Patterns
 - await using, 393–394
 - basic Dispose Patterns, 375–383
 - Close() method, 382–383
 - ConfigureAwait modifier, 393–394
 - design patterns, 372–394
 - Dispose (bool) method, 376–380
 - Dispose (true) method, 377–378
 - finalizable types, 383–387, 511–517
 - IAsyncDisposable interface, 391–392
 - IDisposable method, 382–383
 - rehydration, 381–382
 - scoped operations, 387–391
 - SuppressFinalize method, 378
- DLL (Dynamic-Link Libraries), naming conventions, 61–62
- DP (Dependency Properties), 365–366
 - attached DP design, 369–370
 - change notification events, 371
 - design patterns, 366–372
 - validation, 370
 - value coercion, 371–372
- E**
- enums
 - adding values to, 123–124
 - defined, 84
 - flag enums, type design guidelines, 119–123
 - naming conventions, 72–74
 - parameters, choosing, 205–207
 - type design guidelines, 111–118
- equality operators
 - reference types, 328
 - usage guidelines, 324–328
 - value types, 327
- error exceptions
 - runtime error exceptions, changing to usage error exceptions, 553–554
 - throwing new types of, 555
 - usage error exceptions, changing to functioning behavior, 554
 - runtime error exceptions to, 553–554
- error handling, exceptions and, 250–252
- error messages, designing, 264–265
- event handlers (custom), obsolete guidance, 491–492
- Event-Based Async Patterns, 361–362, 503–504, 510–512
- events
 - change notification events, DP, 371
 - custom event handlers, obsolete guidance, 491–492
 - designing, 175–180
 - extensibility, 231–237
 - naming conventions, 77–78
 - notification events, changing in properties, 163–165
- evolution of well-designed frameworks, qualities of, 7
- exceptions
 - AccessViolationException, 276
 - API consistency, 250
 - ApplicationException, 274
 - ArgumentException, 275–276
 - ArgumentNullException, 275–276
 - ArgumentOutOfRangeException, 275–276

- Async methods, 359–361
C
 ComException, 278
 custom exceptions
 designing, 279–280
 obsolete guidance, 492–493
 error exceptions, throwing new types of, 555
 error handling, 250–252
 error messages, 264–265
 ExecutionEngineException, 278
 FormatException, 278–279
 handling, 249–254, 265–271
 IndexOutOfRangeException, 276
 instrumentation and, 254
 InvalidOperationException, 274–275
 NullReferenceException, 276
 object-oriented languages, 249–250
 OperationCanceledException, 278
 OutOfMemoryException, 277–278
 performance and, 281
 Tester-Doer Pattern, 281–282
 Try Pattern, 282–286
 PlatformNotSupportedException, 279
 runtime error exceptions, changing to
 usage error exceptions, 553–554
 SEHException, 278
 self-documenting object models, 33
 StackOverflowException, 276–277
 SystemException, 274
 TaskCanceledException, 278
 throwing, 254–260
 choosing exceptions, 260–264
 error messages, 264–265
 from existing thrown types, 555
 new types of error exceptions, 555
 types, 273–279
 unhandled exception handlers, 253
 usage error exceptions
 changing runtime error exceptions to, 553–554
 changing to functioning behavior, 554
 wrapping, 271–273
 ExecutionEngineException, 278
 existing proven designs (well-designed frameworks), qualities of, 6–7
 expense, well-designed frameworks, 4–5
 explicit implementation of interface members, 148–152
 exposing layers
 in the same namespace, 38–39
 in separate namespaces, 38
 expression-bodied members, C# style conventions, 478–479
 expression-throw, C# style conventions, 480
 extensibility, 227–228
 abstractions, 239–241
 base classes, 242–244
 callbacks, 231–237
 events, 231–237
 limiting, 244–246
 protected members, 230
 sealing, 244–246
 unsealed classes, 228–229
 virtual members, 237–239
 extension methods, 184–192
- F**
- factored types, aggregate components, 334–335
 factories, design patterns, 394–399
 features (optional), design patterns, 408–411
 fields
 designing, 180–183
 naming conventions, 78–79
 private fields
 removing from serializable types, 544
 renaming private fields on serializable types, 544
 file organization, C# style conventions, 483–485
 finalizable types, Dispose Patterns, 383–387, 511–517
 Finalize method, Dispose Patterns, 378
 finalizers, removing
 sealed types, 542
 from unsealed types, 542
 first reference type field, adding to structs, 540
 fixed buffer sizes, 451–452
 flag enums, type design guidelines, 119–123
 FormatException, 278–279
 Framework Design Guidelines, naming conventions, 480
 frameworks
 designing, 3, 9–11, 15–16
 abstractions, 34–36
 backward compatibility, 2
 borrowing from existing proven designs, 6–7
 consistency, 7–8
 evolution, 7
 expense, 4–5
 integration, 7
 low barrier to entry, principle of, 23–29
 multiframework platforms, 12–13
 naming conventions, 480

- OO design, 2
 - programming languages, 11–12
 - progressive frameworks, 12–15
 - prototyping, 2
 - scenario-driven framework design, 16–23
 - self-documenting object models, principle of, 29–36
 - simplicity, 3–4
 - trade-offs, 6
 - development of, 1–3
 - multiproject platforms, 12–13
 - progressive frameworks, 12–15
 - usability studies, scenario-driven
 - framework design, 21–23
 - well-designed frameworks, qualities of, 3
 - backward compatibility, 2
 - borrowing from existing proven
 - designs, 6–7
 - consistency, 7–8
 - evolution, 7
 - expense, 4–5
 - integration, 7
 - OO design, 2
 - prototyping, 2
 - simplicity, 3–4
 - trade-offs, 6
- G**
- generic interfaces, adding a second declaration to, 536–537
 - generic type parameters, naming conventions, 70–71
 - guidance (obsolete), 487–488
 - Async Patterns, 502
 - cancellation, 512–513
 - choosing between Async Patterns, 503–504
 - Classic Async Patterns, 503–509
 - Event-Based Async Patterns, 503–504, 510–512
 - incremental results, 516
 - Out parameters, 512
 - progress reporting, 513–516
 - Ref parameters, 512
 - custom event handlers, 491–492
 - custom exceptions, 492–493
 - Dispose Patterns, finalizable types, 511–517
 - namespaces, 489–490
 - .Interop subnamespaces, 490
 - .Permissions subnamespaces, 489–490
 - naming conventions, 488
- H**
- heavy API design processes, 2
 - Hungarian notation
 - C# style conventions, 482
 - naming conventions, 53, 482
 - hyphens (-), naming conventions, 53
- I**
- "I" in comments, 483
 - IAsyncDisposable interface
 - Async Patterns, 362
 - Dispose Patterns, 391–392
 - IAsyncEnumerable<T> interface, Async Patterns, 362–365
 - ICloneable struct, usage guidelines, 308–309
 - IComparable<T> struct, usage guidelines, 309–311
 - identifiers
 - capitalization, 42, 44
 - camelCasing, 43
 - PascalCasing, 42–44
 - naming conventions, 54
 - type parameters (generic), naming conventions, 70–71
 - IDisposable method, Dispose Patterns, 377, 382–383
 - IEnumerable<T> method, LINQ support, 402–403
 - if.throw, C# style conventions, 480
 - implementation
 - Async Patterns, 355–361
 - auto-implemented properties, 479
 - expression-bodied members, 478–479
 - interface members, explicit implementation of, 148–152
 - System.Uri, 322–323
 - incremental results, Async Patterns, 516
 - indents, C# style conventions, 471–472
 - indexed properties, designing, 161–163
 - IndexOutOfRangeException, 276
 - inequality operators, 200–202
 - infrastructure namespaces, 66
 - initializers, C# style conventions
 - collection initializers, 478
 - object initializers, 477–478
 - instrumentation, exceptions and, 254
 - integration (well-designed frameworks), qualities of, 7
 - interfaces
 - serialization, 493
 - .NET serialization technologies, 493
 - usage guidelines, 493–502

abstractions, extensibility, 239–241
 adding, members, 541
 base interfaces
 adding to interfaces, 536
 moving members to, 541
 defined, 84
 generic interfaces, adding a second declaration to, 536–537
 members
 adding, 541
 adding to interfaces, 541
 implementing explicitly, 148–152
 moving to base interfaces, 541
 naming conventions, 67–70
 common types, 71
 enumerations, 72–74
 generic type parameters, 70–71
 type design guidelines, 92–100, 104–106
 .Interop subnamespaces, 490
 intuitive API, 33–34
 InvalidOperationException, 274–275
 IQueryable<T> method, LINQ support, 403–404

J - K

keywords (language)
 C# style conventions, 476
 var, C# style conventions, 476–477

L

language keywords, C# style conventions, 476
 languages (programming), framework design, 11–12
 language-specific names, naming conventions, 56–58
 layered architecture, principle of, 36–39
 limiting, extensibility, 244–246
 LINQ (Language-Integrated Queries)
 design patterns, 400–408
 overview of, 400–401
 support
 IEnumerable<T> method, 402–403
 implementation, 402
 IQueryable<T> method, 403–404
 query patterns, 404–408
 live collections, 301–302
 low barrier to entry, principle of, 23–29

M

masking base members, 539
 members
 abstract members
 adding, 539
 changing to virtual members, 553
 changing virtual members to, 553
 removing an override of an abstract member, 543–544
 base members, masking, 539
 changing, member signatures, 545–553
 constructors
 designing, 165–172
 type constructors, 172–175
 designing
 boolean parameters, 205–207
 conversion operators, 198–C05.1827
 enum parameters, 205–207
 events, 175–180
 extension methods, 184–192
 fields, 180–183
 inequality operators, 200–202
 members with variable number of parameters, 214–218
 operator overloads, 192–198
 parameter argument validation, 207–210
 parameter passing, 210–214
 parameters, 202–204
 pointer parameters, 218–219
 tuples in member signatures, 220–226
 events, designing, 175–180
 expression-bodied members, 478–479
 interface members, implementing explicitly, 148–152
 masking base members, 539
 methods, choosing, 152–158
 modifiers, C# style conventions, 473–475
 moving to
 base classes, 541
 base interfaces, 541
 derived types, 542
 non-override members, removing, 543
 non-virtual members, changing to virtual members, 553
 overloading, 136–148, 544–545
 override members, 540
 removing an override of a virtual member, 543
 removing an override of an abstract member, 543–544
 parameters, designing, 202–204
 properties
 change notification events, 163–165
 choosing, 152–158
 designing, 158–160
 indexed properties, 161–163
 protected members, extensibility, 230

- signatures
 - changing, 545–553
 - tuples in, 220–226
- unsealed types, adding members to, 539–540
- virtual members
 - changing abstract members to, 553
 - changing to abstract members, 553
 - extensibility, 237–239
 - removing an override of a virtual member, 543
 - visibility, changing, 548–549
- metadata, assembly metadata and types, 127–129
- methods
 - Async methods
 - exceptions to, 359–361
 - return types, 348–351
 - Close() method, 382–383
 - Dispose (bool) method, 376–380
 - Dispose (true) method, 377–378
 - extension methods, 184–192
 - IDisposable method, Dispose Patterns, 377, 382–383
 - IEnumerable<T> method, LINQ support, 402–403
 - IQueryable<T> method, IQueryable<T> method, 403–404
 - member methods, choosing, 152–158
 - naming conventions, 74–75, 196–197
 - operators and method names, 196–197
 - parameters
 - adding, 546
 - changing types, 546
 - removing, 546
 - renaming, 545
 - reordering parameters by the same type, 547
 - reordering parameters of differing types, 547
 - return types, changing, 547–548
 - static TryParse methods, 286
 - struct methods
 - adding readonly modifiers, 550
 - removing readonly modifiers, 551
 - SuppressFinalize method, 378
 - synchronous methods, Async variants of, 353–354
 - Try methods, value-producing Try methods, 284–285
 - values returned type, changing from a method, 554–555
- modifiers
 - member modifiers, C# style conventions, 473–475
 - readonly modifiers
 - adding to struct methods, 550
 - C# style conventions, 479
 - removing from struct methods, 551
 - static modifiers, adding/removing, 549
- moving
 - members to
 - base classes, 541
 - base interfaces, 541
 - derived types, 542
- types
 - via [TypeForwardedTo], 532–533
 - without [TypeForwardedTo], 533
- multiframework platforms, 12–13
- multiline syntax /* . */), comments, 482

N

- nameof(.) syntax, C# style conventions, 479
- namespaces
 - adding namespaces that conflict with existing types, 531–532
 - application models, 65–66
 - core namespaces, 66
 - infrastructure namespaces, 66
 - naming conventions, 63–67
 - obsolete guidance, 489–490
 - subnamespaces
 - .Design subnamespaces, 489
 - .Interop subnamespaces, 490
 - .Permissions subnamespaces, 489–490
 - naming conventions, 489–490
 - technology namespace groups, 66–67
 - type design guidelines, 85–88
 - type names, changing, 535
 - type names, conflicts, 65
 - application models, 65–66
 - core namespaces, 66
 - infrastructure namespaces, 66
 - technology namespace groups, 66–67
- naming conventions, 41–42
 - abbreviations, 55–56
 - acronyms, 55–56
 - alphanumeric characters, 53
 - API, naming new versions of existing API, 58–61
 - ASCII characters, 54
 - assemblies, 61–62
 - brevity, 52
 - C# style conventions, 480–482

- camelCasing, 481, 482
capitalization, 42
 acronyms, 45–48
 case sensitivity, 51–52
 compound words, 48–51
 identifiers, 42–44
classes, 67–70
 common types, 71
 enumerations, 72–74
 generic type parameters, 70–71
common names, 57–58
custom collections, 305
diacritical marks, 55
DLL, 61–62
enumerations, 72–74
events, 77–78
fields, 78–79
Framework Design Guidelines, 480
Hungarian notation, 53
hyphens (-), 53
identifiers, 54
 capitalization, 42–44
 type parameters (generic), 70–71
interfaces, 67–70
 common types, 71
 enumerations, 72–74
 generic type parameters, 70–71
language-specific names, 56–58
methods, 74–75, 196–197
namespaces, 63–67
obsolete guidance, 488
operators, 196–197
overload operator parameters, 80
packages, 61–62
parameters, 79–80
PascalCasing, 480–481
properties, 75–76
readability, 52
resources, 81
self-documenting object models, 30–33
structs, 67–70
 common types, 71
 enumerations, 72–74
 generic type parameters, 70–71
subnamespaces, 489–490
type members, 74
 events, 77–78
 fields, 78–79
 methods, 74–75
 properties, 75–76
types (common), 71
underscores (_), 53, 481–482
word choice, 52–55
- nested types, design guidelines, 124–127
.NET serialization technologies, 493
non-override members, removing, 543
non-virtual members, changing to virtual
 members, 553
notification events, changing in properties,
 163–165
Nullable<T> struct, usage guidelines, 311–312
NullReferenceException, 276
- ## O
- object initializers, C# style conventions,
 477–478
object models (self-documenting), principle
 of, 29–30
abstractions, 34–36
consistency, 34
exceptions, 33
naming, 30–32
strong typing, 33–34
Object.Equals, usage guidelines, 312–314
 reference types, 314
 value types, 314
Object.GetHashCode, usage guidelines,
 315–316
object-oriented design, 2
object-oriented languages, exceptions and,
 249–250
Object-Oriented Programming, 2
objects, usage guidelines, 312
 Object.Equals, 312–314
 Object.GetHashCode, 315–316
 Object.ToString, 316–318
obsolete guidance, 487–488
 Async Patterns, 502
 cancellation, 512–513
 choosing between Async Patterns,
 503–504
 Classic Async Patterns, 503–509
 Event-Based Async Patterns, 503–504,
 510–512
 incremental results, 516
 Out parameters, 512
 progress reporting, 513–516
 Ref parameters, 512
 custom event handlers, 491–492
 custom exceptions, 492–493
 Dispose Patterns, finalizable types,
 511–517
 namespaces, 489–490
 .Design subnamespaces, 489
 .Interop subnamespaces, 490
 .Permissions subnamespaces, 489–490

- naming conventions, 488
- serialization, 493
- usage guidelines, serialization, 493–502
- OO (Object-Oriented) design, 2
- OOP (Object-Oriented Programming), 2
- OperationCanceledException*, 278
- operations (scoped), Dispose Patterns, 387–391
- OperationStatus* value, buffer operators, 458–463
- operators
 - conversion operators, 198–C05.1827
 - equality operators
 - reference types, 328
 - usage guidelines, 324–328
 - value types, 327
 - inequality operators, 200–202
 - method names and, 196–197
 - overloading, 192–198
- optional features, design patterns, 408–411
- optional parameters, changing
 - default values, 552
 - to required, 551–552
 - required parameters to optional, 551
- organization (files), C# style conventions, 483–485
- Out parameters, 512
- OutOfMemoryException*, 277–278
- overload operator parameters, naming conventions, 80
- overloading
 - members, 136–148, 544–545
 - operators, 192–198
- override members
 - adding to unsealed types, 540
 - removing
 - removing an override of a virtual member, 543
 - removing an override of an abstract member, 543–544
- P**
- packages, naming conventions, 61–62
- parameters
 - argument validation, 207–210
 - boolean parameters, choosing, 205–207
 - collection parameters, 296–297
 - designing, 202–204
 - enum parameters, choosing, 205–207
 - members with variable number of parameters, 214–218
 - method parameters
 - adding, 546
- changing types, 546
- removing, 546
- renaming, 545
- reordering parameters by the same type, 547
- reordering parameters of differing types, 547
- naming conventions, 79–80
- optional parameters, changing
 - default values, 552
 - required parameters to optional, 551 to required, 551–552
- Out parameters, 512
- overload operator parameters, naming conventions, 80
- passing, 210–214, 549–550
- pointer parameters, 218–219
- Ref parameters, 512
- reference parameters
 - Async variants of existing synchronous methods, 352–353
 - changing, 549–550
- required parameters, changing
 - optional parameters to, 551–552
 - to optional, 551
- Pareto principle, 10
- partial writes to buffers, 458–463
- PascalCasing, 42–44
 - C# style conventions, 480–481
 - naming conventions, 480–481
- passing parameters, 210–214
- passive voice, comments, 483
- performance, exceptions, 281
 - Tester-Doer Pattern, 281–282
 - Try Pattern, 282–286
- .Permissions subnamespaces, 489–490
- personification, comments, 483
- PlatformNotSupportedException*, 279
- pointer parameters, 218–219
- predetermined buffer sizes, 451–452
- private fields
 - removing, on serializable types, 544
 - renaming, in serializable types, 544
- programming
 - languages, framework design, 11–12
 - OOP, 7–8
- progress reporting, Async Patterns, 513–516
- progressive frameworks, 12–15
- properties
 - auto-implemented properties, 479
 - change notification events, 163–165
 - designing, 158–160

- change notification events, 163–165
 - indexed properties, 161–163
 - indexed properties, designing, 161–163
 - member properties, choosing, 152–158
 - naming conventions, 75–76
 - events, 77–78
 - fields, 78–79
 - methods, 74–75
 - properties, 75–76
 - types, changing, 548
 - protected members, extensibility, 230
 - prototyping, well-designed
 - frameworks, 2
- Q - R**
- query patterns, LINQ support, 404–408
 - RAD (Rapid Application Development), progressive frameworks, 13
 - readability, naming conventions, 52
 - readonly modifiers
 - C# style conventions, 479
 - structs
 - adding to, 535, 550
 - removing from, 535–536, 551
 - recompile breaks, 530
 - Ref parameters, 512
 - ref structs, 538
 - reference parameters
 - Async variants of existing synchronous methods, 352–353
 - changing, 549–550
 - reference types
 - equality operators, 328
 - fields, adding to structs, 540
 - Object.Equals, 314
 - reflection breaks, 530
 - rehydration, Dispose Patterns, 381–382
 - removing
 - finalizers
 - from sealed types, 542
 - from unsealed types, 542
 - method parameters, 546
 - non-override members, 543
 - override members
 - removing an override of a virtual member, 543
 - removing an override of an abstract member, 543–544
 - private fields, on serializable types, 544
 - readonly modifiers from structs, 535–536, 551
 - static modifiers, 549
 - types, 533
 - renaming
- assemblies, 530–531
 - method parameters, 545
 - private fields in serializable types, 544
 - reordering method parameters
 - of differing types, 547
 - by the same type, 547
 - reporting (progress), Async Patterns, 513–516
 - required parameters, changing
 - to optional, 551
 - optional parameters to, 551–552
 - return types
 - Async methods, 348–351
 - method return types, changing, 547–548
 - return values in collections, 298–302
 - runtime breaks, 529
 - runtime error exceptions, changing to usage error exceptions, 553–554
 - runtime serialization, 493, 500–502
- S**
- scenario-driven framework design, 16–23
 - scoped operations, Dispose Patterns, 387–391
 - sealed types
 - removing finalizers from, 542
 - unsealing, 534
 - sealing
 - extensibility, 244–246
 - unsealed types, 534
 - SEHException, 278
 - self-documenting object models, principle of, 29–30
 - abstractions, 34–36
 - consistency, 34
 - exceptions, 33
 - layered architecture, principle of, 36–37
 - naming, 30–32
 - strong typing, 33–34
 - serializable types, private fields, 544
 - serialization
 - Binary serialization, 493
 - Data Contract serialization, 493, 495–499
 - .NET serialization technologies, 493
 - obsolete guidance, 493–502
 - runtime serialization, 493, 500–502
 - SOAP serialization, 493
 - usage guidelines, 319–321
 - XML serialization, 493, 499–500
 - signatures (member)
 - changing, 545–553
 - tuples in, 220–226
 - simplicity, well-designed frameworks, 3–4
 - single-line syntax (// .), comments, 482
 - snapshot collections, 301–302
 - SOAP serialization, 493

- source-breaking changes
 - compilation breaks, 530
 - recompile breaks, 530
- spaces, C# style conventions, 469–470
- Spans, 11–12, 431–445
- specifications
 - sample specification, 523–528
 - Stopwatch specification, 524–528
- StackOverflowException, 276–277
- static classes
 - defined, 84
 - type design guidelines, 102–104
- static modifiers, adding/removing, 549
- static TryParse methods, 286
- Stopwatch specification, 524–528
- strings (strongly typed), 129–133
- strong typing, self-documenting object
 - models (principle of), 33–34
- structs
 - adding first reference type field to, 540
 - changing
 - changing classes to, 537
 - to classes, 537–538
 - DateTime struct, 306–308
 - DateTimeOffset struct, 306–308
 - defined, 84
 - ICloneable struct, 308–309
 - IComparable<T> struct, 309–311
 - naming conventions, 67–70
 - common types, 71
 - enumerations, 72–74
 - generic type parameters, 70–71
 - Nullable<T> struct, 311–312
 - readonly modifiers
 - adding to structs, 535, 550
 - removing from structs, 535–536, 551
 - ref structs, 538
 - type design guidelines, 89–92, 106–111
 - usage guidelines
 - DateTime struct, 306–308
 - DateTimeOffset struct, 306–308
 - ICloneable struct, 308–309
 - IComparable<T> struct, 309–311
 - Nullable<T> struct, 311–312
 - ValueTask structs, Async Patterns, 358–359
 - ValueTask<TResult> structs, Async Patterns, 358–359
- style conventions, C#465–466
 - ASCII characters, code restrictions, 479
 - assignment-expression-throw, 480
 - auto-implemented properties, 479
 - BCL type names, 476
 - braces ({ }), 466–469
 - collection initializers, 478
- commas (,), 475–476
- comments, 482–483
- expression-bodied members, 478–479
- file organization, 483–485
- if.throw, 480
- indents, 465–466
- language keywords, 476
- member modifiers, 473–475
- nameof () syntax, 479
- naming conventions, 480–482
- object initializers, 477–478
- readonly modifiers, 479
- spaces, 469–470
- this.476
- Unicode escape sequences (uXXXX), 479
- var keyword, 476–477
- vertical whitespace, 472–473
- subclassing, 13
- subnamespaces
 - .Design subnamespaces, 489
 - .Interop subnamespaces, 490
 - naming conventions, 489–490
 - .Permissions subnamespaces, 489–490
- SuppressFinalize method, Dispose Patterns, 378
- synchronous methods, Async variants of, 353–354
- SystemException, 274
- System.Object, usage guidelines, 312
 - Object.Equals, 312–314
 - Object.GetHashCode, 315–316
 - Object.ToString, 316–318
- System.Uri
 - implementation guidelines, 322–323
 - usage guidelines, 321–323
- System.Xml, usage guidelines, 323–324

T

- Task-Based Aync Pattern, 341–347
- TaskCanceledException, 278
- Task.Status, consistency, 355–357
- technology namespace groups, 66–67
- Template Method Pattern, 423–425
- Tester-Doer Pattern, exceptions and performance, 281–282
- this., C# style conventions, 476
- throwing exceptions, 254–260
 - choosing exceptions, 260–264
 - error messages, 264–265
 - from existing thrown types, 555
 - new types of error exceptions, 555
- timeouts, 426–427
- trade-offs, well-designed frameworks, 6
- Try Pattern, exceptions and performance, 282–284

- static TryParse methods, 286
- value-producing Try methods, 284–285
- Try-Write Pattern**, 452–458
- tuples in member signatures, 220–226
- types**
 - abstractions, extensibility, 239–241
 - assemblies
 - metadata and types, 127–129
 - renaming, 530–531
 - classes
 - abstract classes, 98–102
 - defined, 84
 - static classes, 102–104
 - type design guidelines, 89–104
 - common types, naming conventions, 71
 - constructors, 172–175
 - derived types, moving members to, 542
 - design guidelines, 84–85
 - assembly metadata and types, 127–129
 - classes, 89–104
 - enums, 111–124
 - interfaces, 92–100, 104–106
 - namespaces, 85–88
 - nested types, 124–127
 - strongly typed strings, 129–133
 - structs, 89–92, 106–111
 - enums
 - adding values to, 123–124
 - defined, 84
 - flag enums, 119–123
 - type design guidelines, 111–124
 - exception types, 273–279
 - factored types, aggregate components, 334–335
 - finalizable types, Dispose Patterns, 383–387, 511–517
 - interfaces
 - defined, 84
 - type design guidelines, 92–100, 104–106
 - logical groupings, 83
 - members
 - moving to derived types, 542
 - naming conventions, 79
 - method parameter types
 - changing, 546
 - reordering parameters of differing types, 547
 - moving
 - via [TypeForwardedTo], 532–533
 - without [TypeForwardedTo], 533
 - names
 - case sensitivity, 534
 - changing namespaces, 535
 - namespaces
 - adding namespaces that conflict with existing types, 531–532
 - name conflicts, 65–67
 - obsolete guidance, 489–490
 - type design guidelines, 85–88
 - nested types, design guidelines, 124–127
 - parameters (generic), naming conventions, 70–71
 - property types, changing, 548
 - reference types
 - equality operators, 328
 - Object.Equals, 314
 - removing, 533
 - return types
 - Async methods, 348–351
 - changing method return types, 547–548
 - sealed types
 - removing finalizers from, 542
 - unsealing, 534
 - sealing unsealed types, 534
 - serializable types, private fields
 - removing, 544
 - renaming, 544
 - static classes, defined, 84
 - strings (strongly typed), 129–133
 - structs
 - defined, 84
 - type design guidelines, 89–92, 106–111
 - unsealed types
 - adding members to, 539–540
 - removing finalizers from, 542
 - value types, 99
 - equality operators, 327
 - Object.Equals, 314
 - values returned type, changing from a method, 554–555
 - typing (strong), self-documenting object models, 33–34

U

- underscores (_), naming conventions, 53, 481–482
- unhandled exception handlers, 253
- Unicode escape sequences (uXXXX), C# style conventions, 479
- unsealed classes, extensibility, 228–229
- unsealed types
 - adding members to, 539–540
 - removing finalizers from, 542
 - sealing, 534
- Uri
 - implementation guidelines, 322–323
 - usage guidelines, 321–323

- usability studies, designing frameworks, scenario-driven framework design, 21–23
- usage error exceptions, changing to functioning behavior, 554
- runtime error exceptions to, 553–554
- usage guidelines
 - arrays, 287–291
 - attributes, 291–294
 - collections, 294–296
 - arrays versus collections, 302–303
 - custom collections, 302–303
 - live collections, 301–302
 - parameters, 296–297
 - properties, 298–302
 - return values, 298–302
 - snapshot collections, 301–302
 - DateTime struct, 306–308
 - DateTimeOffset struct, 306–308
 - equality operators, 324–328
 - ICloneable struct, 308–309
 - IComparable<T> struct, 309–311
 - Nullable<T> struct, 311–312
 - Object.Equals, 312–314
 - reference types, 314
 - value types, 314
 - Object.GetHashCode, 315–316
 - objects, 312
 - Object.Equals, 312–314
 - Object.GetHashCode, 315–316
 - Object.ToString, 316–318
 - Object.ToString, 316–318
 - serialization, 319–321, 493–502
 - structs
 - DateTime struct, 306–308
 - DateTimeOffset struct, 306–308
 - ICloneable struct, 308–309
 - IComparable<T> struct, 309–311
 - Nullable<T> struct, 311–312
 - System.Object, 312
 - Object.Equals, 312–314
 - Object.GetHashCode, 315–316
 - Object.ToString, 316–318
 - System.Uri, 321–323
 - System.Xml, 323–324
 - Uri, 321–323
 - uXXXXX (Unicode escape sequences), C# style conventions, 479
- V**
 - validation
 - arguments, 207–210, 480
 - assignment-expression-throw, 480
- DP**, 370
- value coercion, DP, 371–372
- value types, 99
 - equality operators, 327
 - Object.Equals, 314
- value-producing Try methods, 284–285
- values
 - constant field values, changing, 552
 - default values, changing in optional parameters, 552
- values returned type, changing from a method, 554–555
- ValueTask structs, Async Patterns, 358–359
- ValueTask<TResult> structs, 358–359
- var keyword, C# style conventions, 476–477
- vertical whitespace, C# style conventions, 472–473
- virtual members
 - abstract members, changing to virtual members, 553
 - changing
 - to abstract members, 553
 - non-virtual members to, 553
 - extensibility, 237–239
 - override members, removing an override of a virtual member, 543
- visibility of members, changing, 548–549
- Visual Basic, language-specific names, 57
- W**
 - “we” in comments, 483
 - well-designed frameworks, qualities of, 3
 - backward compatibility, 2
 - consistency, 7–8
 - evolution, 7
 - existing proven designs, borrowing from, 6–7
 - expense, 4–5
 - integration, 7
 - OO design, 2
 - prototyping, 2
 - simplicity, 3–4
 - trade-offs, 6
 - whitespace (vertical), C# style conventions, 472–473
 - word choice, naming conventions, 52–55
 - wrapping exceptions, 271–273
- X - Y - Z**
 - XAML readable types, 427–430
 - XML serialization, 493, 499–500

Credits

Chapter 5, page 186: "With great power comes great responsibility." Quote said by Benjamin Parker, a character more commonly known as "Uncle Ben" in the Marvel comic series "Spider-Man".

Chapter 7, page 265: "Microsoft Word has encountered a problem and needs to close. We are sorry for the inconvenience." Quote from Microsoft Office application © Microsoft 2020.

Appendix D: Running icon by miceking/123RF.

Appendix D: Arrow icons by Tul Chalothonrangsee/123RF.

Appendix D: Eyeglasses illustration by vasya/123RF.

Appendix D: Question mark art by Hong Li/123RF.



Pearson

livelessonsTM ▶

Photo by Marvent/Shutterstock

VIDEO TRAINING FOR THE **IT PROFESSIONAL**



LEARN QUICKLY

Learn a new technology in just hours. Video training can teach more in less time, and material is generally easier to absorb and remember.



WATCH AND LEARN

Instructors demonstrate concepts so you see technology in action.



TEST YOURSELF

Our Complete Video Courses offer self-assessment quizzes throughout.



CONVENIENT

Most videos are streaming with an option to download lessons for offline viewing.

Learn more, browse our store, and watch free, sample lessons at
informit.com/video

Save 50%* off the list price of video courses with discount code **VIDBOB**



Pearson

*Discount code VIDBOB confers a 50% discount off the list price of eligible titles purchased on informit.com. Eligible titles include most full-course video titles. Book + eBook bundles, book/eBook + video bundles, individual video lessons, Rough Cuts, Safari Books Online, non-discountable titles, titles on promotion with our retail partners, and any title featured as eBook Deal of the Day or Video Deal of the Week is not eligible for discount. Discount may not be combined with any other offer and is not redeemable for cash. Offer subject to change.

informIT[®]
the trusted technology learning source

From the Library of ABE BARBERENA



Photo by izusek/gettyimages

Register Your Product at informit.com/register

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

**Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions (informit.com/promotions)
- Sign up for special offers and content newsletter (informit.com/newsletters)
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit informit.com/community



Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Que • Sams • Peachpit Press



From the Library of ABE BARBERENA