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

FREE SAMPLE CHAPTER
SHARE WITH OTHERS
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
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

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

1 Introduction 1

1.1 Qualities of a Well-Designed Framework 3
  1.1.1 Well-Designed Frameworks Are Simple 3
  1.1.2 Well-Designed Frameworks Are Expensive to Design 4
  1.1.3 Well-Designed Frameworks Are Full of Trade-Offs 6
  1.1.4 Well-Designed Frameworks Borrow from the Past 6
  1.1.5 Well-Designed Frameworks Are Designed to Evolve 7
  1.1.6 Well-Designed Frameworks Are Integrated 7
  1.1.7 Well-Designed Frameworks Are Consistent 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 Designing Flag Enums 119
  4.8.2 Adding Values to Enums 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 Member Overloading 136
  5.1.2 Implementing Interface Members Explicitly 148
  5.1.3 Choosing Between Properties and Methods 152
5.2 Property Design 158
  5.2.1 Indexed Property Design 161
  5.2.2 Property Change Notification Events 163
5.3 Constructor Design 165
  5.3.1 Type Constructor Guidelines 172
5.4 Event Design 175
5.5 Field Design 180
5.6 Extension Methods 184
5.7 Operator Overloads 192
  5.7.1 Overloading operator == 198
  5.7.2 Conversion Operators 198
  5.7.3 Inequality Operators 200
5.8 Parameter Design 202
  5.8.1 Choosing Between Enum and Boolean Parameters 205
  5.8.2 Validating Arguments 207
  5.8.3 Parameter Passing 210
  5.8.4 Members with Variable Number of Parameters 214
  5.8.5 Pointer Parameters 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
      7.3.10FormatException 278
      7.3.11PlatformNotSupportedException 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 Collection Parameters 296
  8.3.2 Collection Properties and Return Values 298
  8.3.3 Choosing Between Arrays and Collections 302
  8.3.4 Implementing Custom Collections 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 Object.Equals 312
  8.9.2 Object.GetHashCode 315
  8.9.3 Object.ToString 316
8.10 Serialization 319
8.11 Uri 321
  8.11.1 System.Uri Implementation Guidelines 322
8.12 System.Xml Usage 323
8.13 Equality Operators 324
  8.13.1 Equality Operators on Value Types 327
  8.13.2 Equality Operators on Reference Types 328

9 Common Design Patterns 329
9.1 Aggregate Components 329
  9.1.1 Component-Oriented Design 331
  9.1.2 Factored Types 334
  9.1.3 Aggregate Component Guidelines 335
9.2 The Async Patterns 339
  9.2.1 Choosing Between the Async Patterns 339
  9.2.2 Task-Based Async Pattern 341
  9.2.3 Async Method Return Types 348
9.2.4 Making an Async Variant of an Existing Synchronous Method 351
9.2.5 Implementation Guidelines for Async Pattern Consistency 355
9.2.7 Classic Async Pattern 361
9.2.8 Event-Based Async Pattern 361
9.2.9 IAsyncDisposable 362
9.2.10 IAsyncEnumerable<T> 362
9.3 Dependency Properties 365
  9.3.1 Dependency Property Design 366
  9.3.2 Attached Dependency Property Design 369
  9.3.3 Dependency Property Validation 370
  9.3.4 Dependency Property Change Notifications 371
  9.3.5 Dependency Property Value Coercion 371
9.4 Dispose Pattern 372
  9.4.1 Basic Dispose Pattern 375
  9.4.2 Finalizable Types 383
  9.4.3 Scoped Operations 387
  9.4.4 IAsyncDisposable 391
9.5 Factories 394
9.6 LINQ Support 400
  9.6.1 Overview of LINQ 400
  9.6.2 Ways of Implementing LINQ Support 402
  9.6.3 Supporting LINQ through IEnumerable<T> 402
  9.6.4 Supporting LINQ through IQueryable<T> 403
  9.6.5 Supporting LINQ through the Query Pattern 404
9.7 Optional Feature Pattern 408
9.8 Covariance and Contravariance 412
  9.8.1 Contravariance 415
  9.8.2 Covariance 417
  9.8.3 Simulated Covariance Pattern 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 Data Transformation Operations 445
  9.12.2 Writing Fixed or Predetermined Sizes to a Buffer 451
  9.12.3 Writing Values to Buffers with the Try-Write Pattern 452
  9.12.4 Partial Writes to Buffers and OperationStatus 458

9.13 And in the End... 464

A C# Coding Style Conventions 465
  A.1 General Style Conventions 466
    A.1.1 Brace Usage 466
    A.1.2 Space Usage 469
    A.1.3 Indent Usage 470
    A.1.4 Vertical Whitespace 472
    A.1.5 Member Modifiers 473
    A.1.6 Other 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 Naming Resources 488
  B.4 Obsolete Guidance from Type Design Guidelines 489
    B.4.1 Types and Namespaces 489
  B.5 Obsolete Guidance from Member Design 491
    B.5.4 Event Design 491
  B.7 Obsolete Guidance from Exceptions 492
    B.7.4 Designing Custom Exceptions 492
  B.8 Obsolete Guidance from Usage Guidelines 493
    B.8.10 Serialization 493
  B.9 Obsolete Guidance from Common Design Patterns 502
    B.9.2 The Async Patterns 502
    B.9.4 Dispose Pattern 517
C Sample API Specification  523

D Breaking Changes  529
   D.1 Modifying Assemblies  530
      D.1.1 Changing the Name of an Assembly  530
   D.2 Adding Namespaces  531
      D.2.1 Adding a Namespace That Conflicts with an Existing Type  531
   D.3 Modifying Namespaces  532
      D.3.1 Changing the Name or Casing of a Namespace  532
   D.4 Moving Types  532
      D.4.1 Moving a Type via [TypeForwardedTo]  532
      D.4.2 Moving a Type Without [TypeForwardedTo]  533
   D.5 Removing Types  533
      D.5.1 Removing Types  533
   D.6 Modifying Types  534
      D.6.1 Sealing an Unsealed Type  534
      D.6.2 Unsealing a Sealed Type  534
      D.6.3 Changing the Case of a Type Name  534
      D.6.4 Changing a Type Name  535
      D.6.5 Changing the Namespace for a Type  535
      D.6.6 Adding readonly on a struct  535
      D.6.7 Removing readonly from a struct  535
      D.6.8 Adding a Base Interface to an Existing Interface  536
      D.6.9 Adding the Second Declaration of a Generic Interface  536
      D.6.10 Changing a class to a struct  537
      D.6.11 Changing a struct to a class  537
      D.6.12 Changing a struct to a ref struct  538
      D.6.13 Changing a ref struct to a (Non-ref) struct  538
   D.7 Adding Members  539
      D.7.1 Masking Base Members with new  539
      D.7.2 Adding abstract Members  539
      D.7.3 Adding Members to an Unsealed Type  539
      D.7.4 Adding an override Member to an Unsealed Type  540
      D.7.5 Adding the First Reference Type Field to a struct  540
      D.7.6 Adding a Member to an Interface  541
D.8 Moving Members  541
D.8.1 Moving Members to a Base Class  541
D.8.2 Moving Members to a Base Interface  541
D.8.3 Moving Members to a Derived Type  542
D.9 Removing Members  542
D.9.1 Removing a Finalizer from an Unsealed Type  542
D.9.2 Removing a Finalizer from a Sealed Type  542
D.9.3 Removing a Non-override Member  543
D.9.4 Removing an override of a virtual Member  543
D.9.5 Removing an override of an abstract Member  543
D.9.6 Removing or Renaming Private Fields on Serializable Types  544
D.10 Overloading Members  544
D.10.1 Adding the First Overload of a Member  545
D.10.2 Adding Alternative-Parameter Overloads for a Reference Type Parameter  545
D.11 Changing Member Signatures  545
D.11.1 Renaming a Method Parameter  545
D.11.2 Adding or Removing a Method Parameter  546
D.11.3 Changing a Method Parameter Type  546
D.11.4 Reordering Method Parameters of Differing Types  547
D.11.5 Reordering Method Parameters of The Same Type  547
D.11.6 Changing a Method Return Type  547
D.11.7 Changing the Type of a Property  548
D.11.8 Changing Member Visibility from public to Any Other Visibility  548
D.11.9 Changing Member Visibility from protected to public  548
D.11.10 Changing a virtual (or abstract) Member from protected to public  549
D.11.11 Adding or Removing the static Modifier  549
D.11.12 Changing to or from Passing a Parameter by Reference  549
D.11.13 Changing By-Reference Parameter Styles  550
D.11.14 Adding the readonly Modifier to a struct Method  550
D.11.15 Removing the readonly Modifier from a struct Method 551
D.11.16 Changing a Parameter from Required to Optional 551
D.11.17 Changing a Parameter from Optional to Required 551
D.11.18 Changing the Default Value for an Optional Parameter 552
D.11.19 Changing the Value of a const Field 552
D.11.20 Changing an abstract Member to virtual 553
D.11.21 Changing a virtual Member to abstract 553
D.11.22 Changing a Non-virtual Member to virtual 553

D.12 Changing Behavior 553
D.12.1 Changing Runtime Error Exceptions to Usage Error Exceptions 553
D.12.2 Changing Usage Error Exceptions to Functioning Behavior 554
D.12.3 Changing the Type of Values Returned from a Method 554
D.12.4 Throwing a New Type of Error Exception 555
D.12.5 Throwing a New Type of Exception, Derived from an Existing Thrown Type 555

D.13 A Final Note 556

Glossary 557
Index 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
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 APIs\(^1\)). 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 always\(^2\) 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.

```csharp
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 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.
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 Schecter, 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 Schecter, 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 Health-Vault 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/RichterCloudApps 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.
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.

```csharp
// string cannot be inherited from
class 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. 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);
    }
}

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.

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.

```csharp
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 = FlowSwitchSetting.Entering; break;
            case "Exiting": Level = FlowSwitchSetting.Exiting; 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.

```csharp
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.

```csharp
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, ininlineable 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.
6.1 Extensibility Mechanisms

Func<...> and Action<...> represent generic delegates. The following is how .NET defines them:

```csharp
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:

```csharp
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:

```csharp
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<...>).

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.

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.

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:

```csharp
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:

```csharp
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:

```csharp
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.

```csharp
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:

```csharp
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.
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.

**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.

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.

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.

```csharp
public class 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.

```csharp
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.

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 mainline 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.

```csharp
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`.

```csharp
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.

```csharp
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.

**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.
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.

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).

**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.

```csharp
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
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

ArgumentException, 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

assign-expression-throw, C# style conventions

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

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
epression-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
“T” 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
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
OperationResult 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
multiframe framework 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
decimal 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, 85
values 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
Async methods, 359–361
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
  multiframework 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
    serialization, 493
      .NET serialization technologies, 493
      usage guidelines, 493–502

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
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
    IQueryable<T> method, 403–404
    implementation, 402
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
derived types, 542
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
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, 487–488
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, 30–32
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
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, 478–480
file organization, 483–479
if.throw, 480
indents, 465–466
language keywords, 476
member modifiers, 473–475
nameof (.) syntax, 479
naming conventions, 480–482
object initializers, 477–480
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
  .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 Async 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
data type coercion, 271–273
data type coercion, 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