Effective SOFTWARE DEVELOPMENT SERIES
Scott Meyers, Consulting Editor

MORE *Effective* C#

Second Edition

COVERS C# 7.0

50 Specific Ways to Improve Your C#

Bill Wagner

More Effective C#

Second Edition

More Effective C#

50 Specific Ways to Improve Your C#

Second Edition

Bill Wagner

Addison-Wesley

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

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2017942600

ISBN-13: 978-0-672-33788-8
ISBN-10: 0-672-33788-6

1 17

To Marlene, Lara, Sarah, and Scott, who provide the inspiration for everything I do.

This page intentionally left blank

Contents at a Glance

	Introduction	xi
Chapter 1	Working with Data Types	1
Chapter 2	API Design	61
Chapter 3	Task-Based Asynchronous Programming	139
Chapter 4	Parallel Processing	177
Chapter 5	Dynamic Programming	229
Chapter 6	Participate in the Global C# Community	267
	Index	273

This page intentionally left blank

Contents

Introduction xi

Chapter 1 Working with Data Types 1

Item 1: Use Properties Instead of Accessible Data Members 1
Item 2: Prefer Implicit Properties for Mutable Data 8
Item 3: Prefer Immutability for Value Types 12
Item 4: Distinguish Between Value Types and Reference Types 18
Item 5: Ensure That 0 Is a Valid State for Value Types 24
Item 6: Ensure That Properties Behave Like Data 28
Item 7: Limit Type Scope by Using Tuples 34
Item 8: Define Local Functions on Anonymous Types 39
Item 9: Understand the Relationships Among the Many Different Concepts of Equality 45
Item 10: Understand the Pitfalls of GetHashCode() 54

Chapter 2 API Design 61

Item 11: Avoid Conversion Operators in Your APIs 61
Item 12: Use Optional Parameters to Minimize Method Overloads 65
Item 13: Limit Visibility of Your Types 69
Item 14: Prefer Defining and Implementing Interfaces to Inheritance 73
Item 15: Understand How Interface Methods Differ from Virtual Methods 82
Item 16: Implement the Event Pattern for Notifications 86
Item 17: Avoid Returning References to Internal Class Objects 93
Item 18: Prefer Overrides to Event Handlers 97
Item 19: Avoid Overloading Methods Defined in Base Classes 100
Item 20: Understand How Events Increase Runtime Coupling Among Objects 104
Item 21: Declare Only Nonvirtual Events 107
Item 22: Create Method Groups That Are Clear, Minimal, and Complete 113
Item 23: Give Partial Classes Partial Methods for Constructors, Mutators, and Event Handlers 120
Item 24: Avoid ICloneable Because It Limits Your Design Choices 125
Item 25: Limit Array Parameters to params Arrays 129
Item 26: Enable Immediate Error Reporting in Iterators and Async Methods Using Local Functions 134

Chapter 3 Task-Based Asynchronous Programming 139
Item 27: Use Async Methods for Async Work 139
Item 28: Never Write async void Methods 143
Item 29: Avoid Composing Synchronous and Asynchronous Methods 149
Item 30: Use Async Methods to Avoid Thread Allocations and Context Switches 154
Item 31: Avoid Marshalling Context Unnecessarily 156
Item 32: Compose Asynchronous Work Using Task Objects 160
Item 33: Consider Implementing the Task Cancellation Protocol 166
Item 34: Cache Generalized Async Return Types 173

Chapter 4 Parallel Processing 177
Item 35: Learn How PLINQ Implements Parallel Algorithms 177
Item 36: Construct Parallel Algorithms with Exceptions in Mind 189
Item 37: Use the Thread Pool Instead of Creating Threads 195
Item 38: Use BackgroundWorker for Cross-Thread Communication 201
Item 39: Understand Cross-Thread Calls in XAML Environments 205
Item 40: Use lock() as Your First Choice for Synchronization 214
Item 41: Use the Smallest Possible Scope for Lock Handles 221
Item 42: Avoid Calling Unknown Code in Locked Sections 225

Chapter 5 Dynamic Programming 229
Item 43: Understand the Pros and Cons of Dynamic Typing 229
Item 44: Use Dynamic Typing to Leverage the Runtime Type of Generic Type Parameters 238
Item 45: Use DynamicObject or IDynamicMetaObjectProvider for Data-Driven Dynamic Types 242
Item 46: Understand How to Use the Expression API 253
Item 47: Minimize Dynamic Objects in Public APIs 259

Chapter 6 Participate in the Global C# Community 267
Item 48: Seek the Best Answer, Not the Most Popular Answer 267
Item 49: Participate in Specs and Code 269
Item 50: Consider Automating Practices with Analyzers 271

Index 273

Introduction

C# continues to evolve and change. As it does so, the community surrounding it is also changing. More developers are now approaching the C# language as their first professional programming language. These members of our community don't have the preconceptions common among those of us who started using C# after years of experience with another C-based language. Even for those developers who have been using C# for years, the recent pace of change has brought the need to adopt many new habits. The C# language has especially seen an increased pace of innovation since the compiler became open source. The review of proposed features to the C# language now includes the entire community, rather than just a small set of language experts. The community can also participate in the design of the new features.

Changes in recommended architectures and deployments are also changing the language idioms we use as C# developers. Building applications by composing microservices, distributed programs, and data separation from algorithms are all part of modern application development. The C# language has begun taking steps toward embracing these different idioms.

I organized this second edition of *More Effective C#* by taking into account both the changes in the language and the changes in the C# community. *More Effective C#* does not take you on a historical journey through the changes in the language, but rather provides advice on how to use the current C# language. The items that have been removed from this edition are those that aren't as relevant in today's C# language, or to today's applications. The new items cover the new language and framework features, and those practices the community has learned from building several versions of software products using C#. Readers of earlier editions will note that content from the previous edition of *Effective C#* is included in this edition, and that a larger number of items have been removed. With the current editions, I've reorganized both books. Overall, these 50 items are a set of recommendations that will help you use C# more effectively as a professional developer.

This book assumes you are using C# 7, but it is not an exhaustive treatment of the new language features. Like all books in the Effective Software Development Series, it offers practical advice on how to use these features to solve problems that you're likely to encounter every day. It specifically covers C# 7 features when new language features introduce new and better ways to write common idioms. Internet searches may still turn up earlier solutions that have years of history. This book specifically points out these older recommendations and explains why language enhancements enable better ways.

Many of the recommendations in this book can be validated by Roslyn-based analyzers and code fixes. I maintain a repository of these resources here: https://github.com/BillWagner/MoreEffectiveCSharpAnalyzers. If you have ideas or want to contribute this repository, write an issue or send me a pull request.

Who Should Read This Book?

More Effective C# was written for professional developers for whom C# is their primary programming language. It assumes you are familiar with the C# syntax and the language's features, and are generally proficient in C#. This book does not include tutorial instruction on language features. Instead, it discusses how you can integrate all the features of the current version of the C# language into your everyday development.

In addition to familiarity with the C# language features, this book assumes you have some knowledge of the Common Language Runtime (CLR) and just-in-time (JIT) compiler.

About the Content

In today's world, data is ubiquitous. An object-oriented approach treats data and code as part of a type and its responsibilities. A functional approach treats methods as data. Service-oriented approaches separate data from the code that manipulates it. C# has evolved to contain language idioms that are common in all these paradigms—which can complicate your design choices. Chapter 1 discusses these choices and provides guidance on when to pick different language idioms for different uses.

Programming is essentially API design. It's how you communicate to your users your expectations about using your code. It also speaks volumes about your understanding of other developers' needs and expectations. In Chapter 2, you'll learn the best way to express your intent using the rich palette of C# language features. You'll see how to leverage lazy evaluation, create composable interfaces, and avoid confusion among the various language elements in your public interfaces.

Task-based asynchronous programming provides new idioms for composing applications from asynchronous building blocks. Mastering these features means you can create APIs for asynchronous operations that clearly reflect how that code will execute, and are easy to use. In Chapter 3, you'll learn how to use the task-based asynchronous language support to express how your code executes across multiple services and using different resources.

Chapter 4 looks at one specific subset of asynchronous programming: multithreaded parallel execution. You'll see how PLINQ enables easier decomposition of complex algorithms across multiple cores and multiple CPUs.

Chapter 5 discusses the use of C# as a dynamic language. C# is a strongly typed, statically typed language. Today, however, an increasing number of programs contain both dynamic and static typing. C# provides ways for you to leverage dynamic programming idioms without losing the benefits of static typing throughout your entire program. In Chapter 5, you'll learn how to use dynamic features and how to avoid having dynamic types leak through your entire program.

Chapter 6 closes the book with suggestions on how to get involved in the global C# community. There are many ways to participate in this community and to help shape the language you use every day.

Code Conventions

Showing code in a book still requires making some compromises for space and clarity. I've tried to distill the samples down to illustrate the particular point of the sample. Often that means eliding other portions of a class or a method. Sometimes it means include eliding error recovery code for space. Public methods should validate their parameters

and other inputs, but that code is usually elided here owing to space constraints. Similar space considerations have prompted the removal of validation of method calls and `try/finally` clauses that would often be included in complicated algorithms.

I also usually assume that most developers can find the appropriate namespace when examples use one of the common namespaces. You can safely assume that every sample implicitly includes the following `using` statements:

```
using System;
using static System.Console;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

Providing Feedback

Despite my best efforts, and the efforts of the people who have reviewed the text, errors may have crept into the text or examples. If you believe you have found an error, please contact me at bill@thebillwagner.com, or on Twitter @billwagner. Errata will be posted at http://thebillwagner.com/Resources/MoreEffectiveCS. Many of the items in this book were inspired by email and Twitter conversations with other C# developers. If you have questions or comments about the recommendations, please contact me. Discussions of general interest will be covered on my blog at http://thebillwagner.com/blog.

Register your copy of *More Effective C#, Second Edition*, on the InformIT site for convenient access to updates and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780672337888) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

There are many people to whom I owe thanks for their contributions to this book. I've been privileged to be part of an amazing C# community over the years. Everyone on the C# Insiders mailing list (whether inside or outside Microsoft) has contributed ideas and conversations that made this a better book.

I must single out a few members of the C# community who directly helped me with ideas, and with turning ideas into concrete recommendations. Conversations with Jon Skeet, Dustin Campbell, Kevin Pilch, Jared Parsons, Scott Allen, and, most importantly, Mads Torgersen are the basis for many new ideas in this edition.

I had a wonderful team of technical reviewers for this edition. Jason Bock, Mark Michaelis, and Eric Lippert pored over the text and the examples and greatly improved the quality of the book you now hold. Their reviews were thorough and complete, which is the best anyone can hope for. Beyond that, they added recommendations that helped me explain many of the topics better.

The team at Addison-Wesley is a dream to work with. Trina Macdonald is a fantastic editor, taskmaster, and the driving force behind anything that gets done. She leans on Mark Renfrow and Olivia Basegio heavily, and so do I. Their contributions ensured the finished manuscript was a high-quality endeavor from the front cover to the back cover, and everything in between. Curt Johnson continues to do an incredible job marketing technical content. No matter which format you chose, Curt has had something to do with its existence for this book.

It's an honor, once again, to be part of Scott Meyers's series. He goes over every manuscript and offers suggestions and comments for improvement. Scott is incredibly thorough, and his experience in software, although not in C#, means he finds any areas where I haven't explained an item clearly or fully justified a recommendation. His feedback, as always, has been invaluable in the preparation of this edition.

As always, my family gave up time with me so that I could finish this manuscript. My wife, Marlene, waited patiently for countless hours while I went off to write or create samples. Without her support, I never would have finished this or any other book, nor would it be as satisfying to complete these projects.

About the Author

Bill Wagner is one of the world's foremost C# developers and a member of the ECMA C# Standards Committee. He is President of the Humanitarian Toolbox, has been awarded Microsoft Regional Director and .NET MVP for 11 years, and was recently appointed to the .NET Foundation Advisory Council. Bill has worked with companies ranging from start-ups to enterprises improving the software development process and growing their software development teams. He is currently with Microsoft, working on the .NET Core content team. He creates learning materials for developers interested in the C# language and .NET Core. Bill earned a B.S. in computer science from the University of Illinois at Champaign-Urbana.

2 API Design

You communicate with your users when you design the APIs you create for your types. The constructors, properties, and methods you expose publicly should make it easier for developers who want to use your types to do so correctly. Robust API design takes into account many aspects of the types you create. It includes how developers can create instances of a type. It includes how you choose to expose the type's capabilities through methods and properties. It includes how an object reports changes through events or outbound method calls. Finally, it includes how you express commonality among different types.

Item 11: Avoid Conversion Operators in Your APIs

Conversion operators introduce a kind of substitutability between classes. Substitutability means that one class can be substituted for another. This flexibility can be a benefit: An object of a derived class can be substituted for an object of its base class, as in the classic example of the shape hierarchy. Suppose you create a Shape base class and derive a variety of customizations: Rectangle, Ellipse, Circle, and so on. You can substitute a Circle anywhere a Shape is expected—that's using polymorphism for substitutability. This substitution works because a Circle is a specific type of a Shape.

When you create a class, certain conversions are allowed automatically. Any object can be substituted for an instance of `System.Object`, the root of the .NET class hierarchy. In the same fashion, any object of a class that you create will be substituted implicitly for an interface that it implements, any of its base interfaces, or any of its base classes. The C# language also supports a variety of numeric conversions.

When you define a conversion operator for your type, you tell the compiler that your type may be substituted for the target type. These substitutions often result in subtle errors because your type probably isn't a perfect substitute for the target type. Side effects that modify the state

of the target type won't have the same effect on your type. Even worse, if your conversion operator returns a temporary object, the side effects will modify the temporary object and be lost forever to the garbage collector. Finally, the rules for invoking conversion operators are based on the compile-time type of an object, rather than the runtime type of an object. As a consequence, users of your type might need to perform multiple casts to invoke the conversion operators—a practice that leads to unmaintainable code.

If you want to convert another type into your type, use a constructor. This practice more clearly reflects the action of creating a new object. Conversion operators can introduce hard-to-find problems in your code. Suppose that you inherit the code for a library shown in Figure 2.1. Both the Circle class and the Ellipse class are derived from the Shape class. You decide to leave that hierarchy in place because, although the Circle and the Ellipse are related, you don't want to have nonabstract leaf classes in your hierarchy, and several implementation problems can occur when you try to derive the Circle class from the Ellipse class. However, you realize that, in the world of geometry, every circle could be an ellipse. In addition, some ellipses could be substituted for circles.

That realization leads you to add two conversion operators. Every circle is an ellipse, so you add an implicit conversion to create a new Ellipse object from a Circle. An implicit conversion operator will be called whenever one type needs to be converted to another type. By contrast,

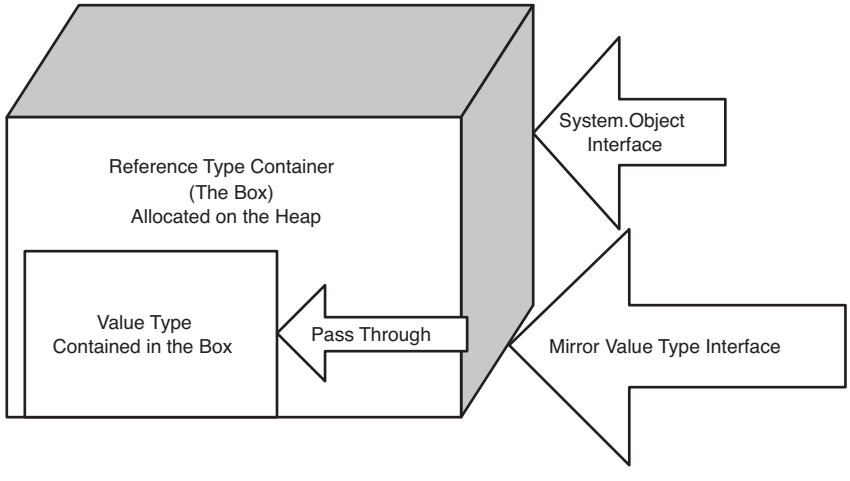

Figure 2.1 Basic shape hierarchy

an explicit conversion will be called only when the programmer puts a cast operator in the source code.

```
public class Circle : Shape
{
    private Point center;
    private double radius;

    public Circle() :
        this(new Point(), 0)
    {
    }

    public Circle(Point c, double r)
    {
        center = c;
        radius = r;
    }

    public override void Draw()
    {
        //...
    }

    static public implicit operator Ellipse(Circle c)
    {
        return new Ellipse(c.center, c.center,
            c.radius, c.radius);
    }
}
```

Now that you've got the implicit conversion operator, you can use a Circle anywhere an Ellipse is expected. Furthermore, the conversion happens automatically:

```
public static double ComputeArea(Ellipse e) =>
    e.R1 * e.R2 * Math.PI;

// Call it:
Circle c1 = new Circle(new Point(3.0, 0), 5.0f);
ComputeArea(c1);
```

This example shows what we mean by substitutability: A circle has been substituted for an ellipse. The `ComputeArea` function works even with the substitution. You got lucky. But consider this function:

```
public static void Flatten(Ellipse e)
{
    e.R1 /= 2;
    e.R2 *= 2;
}

// Call it using a circle:
Circle c = new Circle(new Point(3.0, 0), 5.0);
Flatten(c);
```

This won't work. The Flatten() method takes an ellipse as an argument, so the compiler must somehow convert a circle to an ellipse. You've created an implicit conversion that does exactly that. Your conversion gets called, and the Flatten() function receives as its parameter—that is, the ellipse created by your implicit conversion. This temporary object is modified by the Flatten() function and immediately becomes garbage. The side effects expected from your Flatten() function occur, but only on a temporary object. The end result is that nothing happens to the circle, c.

Changing the conversion from implicit to explicit merely forces users to add a cast to the call:

```
Circle c = new Circle(new Point(3.0, 0), 5.0);
Flatten((Ellipse)c);
```

The original problem remains—you just forced your users to add a cast to cause the problem. You still create a temporary object, flatten the temporary object, and throw it away. The circle, c, is not modified at all. Instead, if you create a constructor to convert the Circle to an Ellipse, the actions are clearer:

```
Circle c = new Circle(new Point(3.0, 0), 5.0);
Flatten(new Ellipse(c));
```

Most programmers would see the previous two lines and immediately realize that any modifications to the ellipse passed to Flatten() will be lost. They would fix the problem by keeping track of the new object:

```
Circle c = new Circle(new Point(3.0, 0), 5.0);
Flatten(c);
```

```
// Work with the circle.
// ...

// Convert to an ellipse.
Ellipse e = new Ellipse(c);
Flatten(e);
```

The variable e holds the flattened ellipse. By replacing the conversion operator with a constructor, you haven't lost any functionality; you've merely made it clearer when new objects are created. (Veteran C++ programmers should note that C# does not call constructors for implicit or explicit conversions. You create new objects only when you explicitly use the `new` operator, and at no other time. There is no need for the `explicit` keyword on constructors in C#.)

Conversion operators that return fields inside your objects will not exhibit this behavior, but they have other problems. With this approach, you've poked a serious hole in the encapsulation of your class. By casting your type to some other object, clients of your class can access an internal variable. That possibility is best avoided for all the reasons discussed in Item 17.

Conversion operators introduce a form of substitutability that causes problems in your code. Their use signals that, in all cases, users can reasonably expect that another class can be used in place of the one you created. When this substituted object is accessed, clients will work with temporary objects or internal fields in place of the class you created. In that case, they modify temporary objects and discard the results. These subtle bugs are difficult to find because the compiler generates code to convert these objects. Avoid conversion operators in your APIs.

Item 12: Use Optional Parameters to Minimize Method Overloads

C# enables you to specify method arguments by position or by name. In turn, the names of formal parameters are part of the public interface for your type. Changing the name of a public parameter could break calling code. To avoid this problem, you should avoid using named parameters in many situations, and you should avoid changing the names of the formal parameters for public or protected methods.

Of course, no language designer adds features just to make your life difficult. Named parameters were added for a reason, and they have

positive uses. Named parameters work with optional parameters to limit the noisiness around many APIs, especially COM APIs for Microsoft Office. The following snippet of code creates a Word document and inserts a small amount of text, using the classic COM methods:

```
var wasted = Type.Missing;
var wordApp = new
    Microsoft.Office.Interop.Word.Application();
wordApp.Visible = true;
Documents docs = wordApp.Documents;

Document doc = docs.Add(ref wasted,
    ref wasted, ref wasted, ref wasted);

Range range = doc.Range(0, 0);

range.InsertAfter("Testing, testing, testing. . .");
```

This small—and arguably useless—snippet uses the Type.Missing object four times. Any Office interop application will use a much larger number of Type.Missing objects in the application. Those instances clutter up your application and hide the actual logic of the software you're building.

That extra noise was the primary driver behind adding optional and named parameters in the C# language. By using optional parameters, the Office APIs can create default values for all those locations where Type.Missing would be used. That simplifies even this small snippet and greatly improves its readability:

```
var wordApp = new
    Microsoft.Office.Interop.Word.Application();
wordApp.Visible = true;
Documents docs = wordApp.Documents;

Document doc = docs.Add();

Range range = doc.Range(0, 0);

range.InsertAfter("Testing, testing, testing. . .");
```

Of course, you may not always want to use all the defaults, but you also may not want to add all the Type.Missing parameters in the middle. Suppose you wanted to create a new Web page instead of a new Word document. That

choice is made with the last of the four parameters in the Add() method. Using named parameters, you can specify just that last parameter:

```
var wordApp = new
    Microsoft.Office.Interop.Word.Application();
wordApp.Visible = true;
Documents docs = wordApp.Documents;

object docType = WdNewDocumentType.wdNewWebPage;
Document doc = docs.Add(DocumentType: ref docType);

Range range = doc.Range(0, 0);

range.InsertAfter("Testing, testing, testing. . .");
```

Named parameters mean that in any API with default parameters, you need to specify only those parameters you intend to use. This approach is much simpler than working with multiple overloads. In fact, with four different parameters, you would need to create 16 different overloads of the Add() method to achieve the same level of flexibility that the named and optional parameters provide. Given that some of the Office APIs have as many as 16 parameters, optional and named parameters are a big help in simplifying their use.

The preceding example included the `ref` decorator in the parameter list, but another change in C# 4.0 makes that optional in COM scenarios. In fact, the Range() call passes the values (0, 0) by reference. The `ref` modifier is omitted there in the example, because that would be clearly misleading. In fact, in most production code, the `ref` modifier should not be included in the call to Add(). (It was included in the example just so you could see the actual API signature.)

The justification for named and optional parameters was COM and the Office APIs in our examples, but you shouldn't limit their use to Office interop applications. In fact, you can't. Developers calling your API can decorate calling locations using named parameters whether you want them to or not.

For example, the method

```
private void SetName(string lastName, string firstName)
{
    // Elided
}
```

can be called using named parameters to avoid any confusion about the order of the names:

```
SetName(lastName: "Wagner", firstName: "Bill");
```

Annotating the names of the parameters ensures that people reading this code later won't wonder if the parameters appear in the right order. Developers tend to use named parameters whenever adding the names will increase the clarity of the code that someone is trying to read. Whenever you use methods that contain multiple parameters of the same type, naming the parameters at the callsite will make your code more readable.

Changes in parameter names are breaking changes. The parameter names are stored in the MSIL only at the method definition, not where the method is called. You can change parameter names and release the component without breaking any users of that component in the field. The developers who use your component will see a breaking change when they compile their assemblies against the updated version, but any earlier client assemblies will continue to run correctly—so at least you won't break existing applications in the field. The developers who use your work will still be upset, but they won't blame you for problems in the field. For example, suppose you modify SetName() by changing the parameter names:

```
public void SetName(string last, string first)
```

You could compile and release this assembly as a patch into the field. Any assemblies that called this method would then continue to run, even if they contain calls to SetName() that specify named parameters. However, when client developers try to build updates to their assemblies, any code like this will no longer compile:

```
SetName(lastName: "Wagner", firstName: "Bill");
```

The parameter names have changed.

Changing the default value also requires callers to recompile their code so as to pick up those changes. If you compile your assembly and release it as a patch, all existing callers would continue to use the previous default parameter.

Of course, you don't really want to upset the developers who use your components. For that reason, you must consider the names of your

parameters to be part of the public interface to your component. Changing the names of parameters will break client code at compile time.

In addition, adding parameters (even if they have default values) will cause code to break at runtime. Optional parameters are implemented in a similar fashion to named parameters. The callsite will contain annotations in the MSIL that reflect whether the default values exist, and what those default values are. The calling site substitutes those values for any optional parameters the caller did not explicitly specify.

Thus, adding parameters, even if they are optional parameters, is a breaking change at runtime. If those parameters have default values, it's not a breaking change at compile time.

After this explanation, the guidance in this item's title should be clearer. For your initial release, use optional and named parameters to create whatever combination of overloads your users may want to use. However, once you start creating future releases, you must create overloads for additional parameters. That way, existing client applications will still function. Furthermore, in any future release, avoid changing parameter names, because they are now part of your public interface.

Item 13: Limit Visibility of Your Types

Everyone doesn't always need to see everything; that is, not every type you create needs to be public. You should give each type the least visibility necessary to accomplish your purpose. That's often less visibility than you think. Internal or private classes can implement public interfaces. All clients can access the functionality defined in the public interfaces declared in a private type.

It's just too easy to create public types, and it's often expedient to do just that. Many stand-alone classes that you create should be internal. You can further limit visibility by creating protected or private classes nested inside your original class. The less visibility there is, the less the entire system changes when you make updates later. The fewer places that can access a piece of code, the fewer places you must change when you modify it.

Expose only what needs to be exposed. Try implementing public interfaces with less visible classes. You'll find examples using the

Enumerator pattern throughout the .NET Framework library. `System.Collections.Generic.List<T>` contains a private class, `Enumerator<T>`, that implements the `IEnumerator<T>` interface:

```
// For illustration; not the complete source
public class List<T>  : IEnumerable<T>
{
    public struct Enumerator : IEnumerator<T>
    {
        // Contains specific implementation of
        // MoveNext(), Reset(), and Current

        public Enumerator(List<T> storage)
        {
            // Elided
        }
    }

    public Enumerator GetEnumerator()
    {
        return new Enumerator(this);
    }

    // Other List members
}
```

Client code, written by you, never needs to know about the struct `Enumerator<T>`. All you need to know is that you get an object that implements the `IEnumerator<T>` interface when you call the `GetEnumerator` function on a `List<T>` object. The specific type is an implementation detail. The .NET Framework designers followed this same pattern with the other collection classes: `Dictionary<T>` contains `DictionaryEnumerator`, `Queue<T>` contains `QueueEnumerator`, and so on.

Keeping the enumerator class private offers many advantages. First, the `List<T>` class can completely replace the type implementing `IEnumerator<T>`, and you'd be none the wiser, because nothing breaks. You can use the enumerator because you know that it follows a contract, not because you have detailed knowledge about the type that implements it. The types that implement these interfaces in the framework are public structs for performance reasons, not because you need to work directly with the type.

Creating internal classes is an often-overlooked method of limiting the scope of types. Most programmers create public classes by default, without giving any thought to the alternatives. Instead of unthinkingly following this routine, you should give careful thought to where your new type will be used. Is it useful to all clients, or is it primarily used internally in this one assembly?

Exposing your functionality using interfaces enables you to more easily create internal classes without limiting their usefulness outside the assembly (see Item 17). Does the type need to be public, or is an aggregation of interfaces a better way to describe its functionality? Internal classes allow you to replace the class with a different version, as long as it implements the same interfaces. As an example, consider a class that validates phone numbers:

```
public class PhoneValidator
{
    public bool ValidateNumber(PhoneNumber ph)
    {
        // Perform validation.
        // Check for valid area code and exchange.
        return true;
    }
}
```

Months pass, and this class works well—at least until you get a request to handle international phone numbers. Now PhoneValidator fails, because it was coded to handle only U.S. phone numbers. You still need the U.S. phone validator, but you also need to include an international version of the phone validator in one installation. Rather than stick the extra functionality in this one class, you would be better advised to reduce the coupling between the different items. You create an interface to validate any phone number:

```
public interface IPhoneValidator
{
    bool ValidateNumber(PhoneNumber ph);
}
```

Next, change the existing phone validator to implement that interface, and make it an internal class:

```
internal class USPhoneValidator : IPhoneValidator
{
    public bool ValidateNumber(PhoneNumber ph)
    {
```

```
        // Perform validation.
        // Check for valid area code and exchange.
        return true;
    }
}
```

Finally, create a class for international phone validators:

```
internal class InternationalPhoneValidator : IPhoneValidator
{
    public bool ValidateNumber(PhoneNumber ph)
    {
        // Perform validation.
        // Check international code.
        // Check specific phone number rules.
        return true;
    }
}
```

To finish this implementation, you need to create the proper class based on the type of the phone number. You can use the factory pattern for this purpose. Outside the assembly, only the interface is visible. The classes, which are specific for different regions in the world, are visible only inside the assembly. You can add different validation classes for different regions without disturbing any other assemblies in the system. By limiting the scope of the classes, you have limited the code you need to change to update and extend the entire system.

```
public static IPhoneValidator CreateValidator(PhoneTypes type)
{
    switch (type)
    {
        case PhoneTypes.UnitedStates:
            return new USPhoneValidator();
        case PhoneTypes.UnitedKingdom:
            return new UKPhoneValidator();
        case PhoneTypes.Unknown:
        default:
            return new InternationalPhoneValidator();
    }
}
```

You could also create a public abstract base class for PhoneValidator, which could contain common implementation algorithms. The consumers could access the public functionality through the accessible base class. In this example, the implementation using public interfaces is an excellent choice because there is little, if any, shared functionality. Other uses would be better served with public abstract base classes. With either option, fewer classes are publicly accessible.

If there are fewer public types, there are fewer publicly accessible methods for which you need to create tests. Also, if more of the public APIs are exposed through interfaces, you have automatically created a system whereby you can replace those types using mock-ups or stubs for unit test purposes.

Those classes and interfaces that you expose publicly to the outside world are your contract: You must live up to them. The more cluttered that public contract is, the more constrained your future direction is. The fewer public types you expose, the more options you have to extend and modify any implementation in the future.

Item 14: Prefer Defining and Implementing Interfaces to Inheritance

Abstract base classes provide a common ancestor for a class hierarchy. An interface describes related methods comprising functionality that can be implemented by a type. Each of these strategies has its place, but it is a different place. Interfaces offer a way to declare the signature of a design contract: A type that implements an interface must supply an implementation for expected methods. Abstract base classes provide a common abstraction for a set of related types. It's a cliché, but it's one that works: Inheritance means "is a," whereas interface means "behaves like." These clichés have lived so long because they provide a means to describe the differences in both constructs: Base classes describe what an object is; interfaces describe one way in which an object behaves.

Interfaces describe a set of functionality, which represents a contract. You can create placeholders for anything in an interface: methods, properties, indexers, and events. Any non-abstract type that implements the interface must supply concrete implementations of all elements defined in the interface. You must implement all methods, supply any and all property accessors and indexers, and define all events defined in the interface. You

can identify and factor reusable behavior into interfaces. You can also use interfaces as parameters and return values. In addition, interfaces offer more opportunities to reuse code because unrelated types can implement interfaces. What's more, it's easier for other developers to implement an interface than it is to derive types from a base class you've created.

What you *can't* do in an interface is provide implementation for any of these members. Interfaces contain no implementation whatsoever, and they cannot contain any concrete data members. With an interface, you declare the binary contract that must be supported by all types that implement the interface. If you like, you can then create extension methods on those interfaces to give the illusion of an implementation for interfaces. For example, the `System.Linq.Enumerable` class contains more than 30 extension methods declared on `IEnumerable<T>`. Those methods appear to be part of any type that implements `IEnumerable<T>` by virtue of being extension methods (see Item 27 in *Effective C#, Third Edition*):

```
public static class Extensions
{
    public static void ForAll<T>(
        this IEnumerable<T> sequence,
        Action<T> action)
    {
        foreach (T item in sequence)
            action(item);
    }
}
// Usage
foo.ForAll((n) => Console.WriteLine(n.ToString()));
```

Abstract base classes can supply some implementation for derived types, in addition to describing the common behavior. You can specify data members, concrete methods, implementation for virtual methods, properties, events, and indexers. A base class can provide implementation for some of the methods, thereby providing common implementation reuse. Any of the elements can be virtual, abstract, or nonvirtual. An abstract base class can provide an implementation for any concrete behavior; interfaces cannot.

This implementation reuse provides another benefit: If you add a method to the base class, all derived classes are automatically and

implicitly enhanced. In that sense, base classes provide a way to extend the behavior of several types efficiently over time. When you add and implement functionality in the base class, all derived classes immediately incorporate that behavior. Adding a member to an interface, however, breaks all the classes that implement that interface. They will not contain the new method and will no longer compile. Each implementer must update that type to include the new member. Alternatively, if you need to add functionality to an interface without breaking the existing code, you can create a new interface and have it inherit from the existing interface.

Choosing between an abstract base class and an interface is a question of how best to support your abstractions over time. Interfaces are fixed: You release an interface as a contract for a set of functionality that any type can implement. In contrast, base classes can be extended over time, and those extensions then become part of every derived class.

The two models can be mixed to reuse implementation code while supporting multiple interfaces. One obvious example in the .NET Framework is the `IEnumerable<T>` interface and the `System.Linq.Enumerable` class. The `System.Linq.Enumerable` class contains a large number of extension methods defined on the `System.Collections.Generic.IEnumerable<T>` interface. That separation enables very important benefits. Any class that implements `IEnumerable<T>` appears to include all those extension methods, but those additional methods are not formally defined in the `IEnumerable<T>` interface. As a consequence, class developers do not need to create their own implementation of all those methods.

As an example, consider the following class, which implements `IEnumerable<T>` for weather observations:

```
public enum Direction
{
    North,
    NorthEast,
    East,
    SouthEast,
    South,
    SouthWest,
    West,
    NorthWest
}
```

```
public class WeatherData
{
    public WeatherData(double temp, int speed,
        Direction direction)
    {
        Temperature = temp;
        WindSpeed = speed;
        WindDirection = direction;
    }
    public double Temperature { get;  }
    public int WindSpeed { get;  }
    public Direction WindDirection { get;  }
    public override string ToString() =>
        @$"Temperature = {Temperature}, Wind is {WindSpeed}
mph from the {WindDirection}";
}

public class WeatherDataStream : IEnumerable<WeatherData>
{
    private Random generator = new Random();

    public WeatherDataStream(string location)
    {
        // Elided
    }

    private IEnumerator<WeatherData> getElements()
    {
        // Real implementation would read from
        // a weather station.
        for (int i = 0; i < 100; i++)
        yield return new WeatherData(
            temp: generator.NextDouble() * 90,
            speed: generator.Next(70),
            direction: (Direction)generator.Next(7)
        );
    }

    public IEnumerator<WeatherData> GetEnumerator() =>
        getElements();
```

```
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() =>
        getElements();
}
```

To model a sequence of weather observations, the WeatherStream class implements `IEnumerable<WeatherData>`. That means creating two methods: the `GetEnumerator<T>` method and the classic `GetEnumerator` method. The latter interface is explicitly implemented so that client code will naturally be drawn to the generic interface rather than the version typed as `System.Object`.

The implementation of those two methods means that the WeatherStream class supports all the extension methods defined in `System.Linq.Enumerable`. That means WeatherStream can be a source for LINQ queries:

```
var warmDays = from item in
                    new WeatherDataStream("Ann Arbor")
               where item.Temperature > 80
               select item;
```

LINQ query syntax compiles to method calls. For example, the preceding query translates to the following calls:

```
var warmDays2 = new WeatherDataStream("Ann Arbor").
    Where(item => item.Temperature > 80);
```

In this code, the Where and Select calls might seem to belong to IEnumerable<WeatherData>, but they do not. That is, those methods appear to belong to IEnumerable<WeatherData> because they are extension methods, but they are actually static methods in `System.Linq.Enumerable`. The compiler translates those calls into the following static calls:

```
// Don't write this; presented for explanatory purposes only
var warmDays3 = Enumerable.Select(
                    Enumerable.Where(
                    new WeatherDataStream("Ann Arbor"),
                    item => item.Temperature > 80),
                    item => item);
```

The preceding code illustrates that interfaces really can't contain implementation. You can emulate that state by using extension methods. LINQ does so by creating several extension methods on `IEnumerable<T>` in the class.

That brings us to the topic of using interfaces as parameters and return values. An interface can be implemented by any number of unrelated types. Coding to interfaces provides greater flexibility for other developers than coding to base class types. That's important because of the single inheritance hierarchy enforced by the .NET type system.

The following three methods perform the same task:

```
public static void PrintCollection<T>(
    IEnumerable<T> collection)
{
    foreach (T o in collection)
        Console.WriteLine($"Collection contains {o}");
}

public static void PrintCollection(
    System.Collections.IEnumerable collection)
{
    foreach (object o in collection)
        Console.WriteLine($"Collection contains {o}");
}

public static void PrintCollection(
    WeatherDataStream collection)
{
    foreach (object o in collection)
        Console.WriteLine($"Collection contains {o}");
}
```

The first method is most reusable. Any type that supports `IEnumerable<T>` can use that method. In addition to WeatherDataStream, you can use `List<T>`, `SortedList<T>`, any array, and the results of any LINQ query. The second method will also work with many types, but uses the less preferable nongeneric `IEnumerable`. The third method is far less reusable; it cannot be used with `Arrays`, `ArrayLists`, `DataTables`, `Hashtables`, `ImageLists`, or many other collection classes. Coding the method using interfaces as its parameter types is far more generic and far easier to reuse.

Using interfaces to define the APIs for a class also provides greater flexibility. The WeatherDataStream class could implement a method that returned a collection of WeatherData objects. That would look something like this:

```
public List<WeatherData> DataSequence => sequence;
private List<WeatherData> sequence = new List<WeatherData>();
```

Unfortunately, this code leaves you vulnerable to future problems. At some point, you might change from using a List<WeatherData> to exposing an array, a `SortedList<T>`. Any of those changes will break the code. Sure, you can change the parameter type, but that's changing the public interface to your class. Changing the public interface to a class causes you to make many more changes to a large system; you would need to change all the locations where the public property was accessed.

Another problem with this code is more immediate and more troubling: The `List<T>` class provides numerous methods to change the data it contains. Users of your class could delete, modify, or even replace every object in the sequence—which is almost certainly not your intent. Luckily, you can limit the capabilities of the users of your class. Instead of returning a reference to some internal object, you should return the interface that you intend clients to use—in this case, IEnumerable<WeatherData>.

When your type exposes properties as class types, it exposes the entire interface to that class. Using interfaces, you can choose to expose only those methods and properties you want clients to use. The class used to implement the interface is an implementation detail that can change over time (see Item 17).

Furthermore, unrelated types can implement the same interface. Suppose you're building an application that manages employees, customers, and vendors. Those entities are unrelated, at least in terms of the class hierarchy. Nevertheless, they share some common functionality. They all have names, and you will likely display those names in controls in your applications.

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string Name => $"{LastName}, {FirstName}";
    // Other details elided
}

public class Customer
{
    public string Name => customerName;
```

```
    // Other details elided
    private string customerName;
}

public class Vendor
{
    public string Name => vendorName;

    // Other details elided
    private string vendorName;
}
```

The Employee, Customer, and Vendor classes should not share a common base class, but they do share some properties: names (as shown earlier), addresses, and contact phone numbers. You could factor out those properties into an interface:

```
public interface IContactInfo
{
    string Name { get; }
    PhoneNumber PrimaryContact { get; }
    PhoneNumber Fax { get; }
    Address PrimaryAddress { get; }
}

public class Employee : IContactInfo
{
    // Implementation elided
}
```

This new interface can simplify your programming tasks by letting you build common routines for unrelated types:

```
public void PrintMailingLabel(IContactInfo ic)
{
    // Implementation deleted
}
```

This single routine works for all entities that implement the IContactInfo interface. Now Customer, Employee, and Vendor all use the same routine—but only because you factored them into interfaces.

Using interfaces also means that you can occasionally save an unboxing penalty for `structs`. When you place a `struct` in a box, the box supports all interfaces that the `struct` supports. When you access the `struct` through the interface reference, you don't have to unbox the `struct` to access that object. To illustrate, imagine this `struct` that defines a link and a description:

```
public struct URLInfo : IComparable<URLInfo>, IComparable
{
    private Uri URL;
    private string description;

    // Compare the string representation of
    // the URL:
    public int CompareTo(URLInfo other) =>
        URL.ToString().CompareTo(other.URL.ToString());

    int IComparable.CompareTo(object obj) =>
        (obj is URLInfo other) ?
            CompareTo(other) :
            throw new ArgumentException(
                message: "Compared object is not URLInfo",
                paramName: nameof(obj));
}
```

This example makes use of two new features in C# 7. The initial condition is a pattern-matching expression. It tests whether obj is a URLInfo; if it is, it assigns obj to the variable other. The other new feature is a throw expression. In cases where obj is not a URLInfo, an exception is thrown. The throw expression no longer needs to be a separate statement.

You can create a sorted list of URLInfo objects easily because URLInfo implements `IComparable<T>` and `IComparable`. Even code that relies on the classic `IComparable` will need boxing and unboxing less often because the client can call `IComparable.CompareTo()` without unboxing the object.

Base classes describe and implement common behaviors across related concrete types. Interfaces describe atomic pieces of functionality that unrelated concrete types can implement. Both have their place. Classes

define the types you create; interfaces describe the behavior of those types as pieces of functionality. When you understand the differences, you can create more expressive designs that are more resilient in the face of change. Use class hierarchies to define related types. Expose functionality using interfaces implemented across those types.

Item 15: Understand How Interface Methods Differ from Virtual Methods

At first glance, implementing an interface might seem to be the same as overriding an abstract function; that is, in both cases you provide a definition for a member that has been declared in another type. That first glance is very deceiving, however: Implementing an interface is very different from overriding a virtual function. An implementation of an abstract (or virtual) base class member is required to be virtual; an implementation of an interface member is not. The implementation of an interface member may be, and often is, virtual. Interfaces can be explicitly implemented, which hides them from a class's public interface. In short, implementing an interface and overriding a virtual function are different concepts with different uses.

Even so, you can implement interfaces in such a way that derived classes can modify your implementation. You just have to create hooks for derived classes.

To illustrate the differences, examine a simple interface and implementation of it in one class:

```
interface IMessage
{
    void Message();
}

public class MyClass : IMessage
{
    public void Message() =>
        WriteLine(nameof(MyClass));
}
```

The Message() method is part of MyClass's public interface. Message can also be accessed through the IMessage point that is part of the MyClass type. Now let's complicate the situation a little by adding a derived class:

```
public class MyDerivedClass : MyClass
{
    public new void Message() =>
        WriteLine(nameof(MyDerivedClass));
}
```

Notice that the `new` keyword was added to the definition of the previous Message method (see Item 10 in *Effective C#, Third Edition*). MyClass.Message() is not virtual. Derived classes cannot provide an overridden version of Message. The MyDerived class creates a new Message method, but that method does not override MyClass.Message; instead, it hides it. Furthermore, MyClass.Message is still available through the IMsg reference:

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // Prints "MyDerivedClass"
IMsg m = d as IMsg;
m.Message(); // Prints "MyClass"
```

When you implement an interface, you are declaring a concrete implementation of a particular contract in that type. You, as the class author, decide whether that method is virtual.

Let's review the C# language rules for implementing interfaces. When a class declaration contains interfaces in its base types, the compiler determines which member of the class corresponds to each member of the interface. An explicit interface implementation is a better match than an implicit implementation. If an interface member cannot be found in that class definition, accessible members of base types are considered. Recall that virtual and abstract members are considered to be members of the type that declares them, not the type that overrides them.

In many cases, you will want to create interfaces, implement them in base classes, and modify the behavior in derived classes. You can, and you have two options for doing so. If you do not have access to the base class, you can reimplement the interface in the derived class:

```
public class MyDerivedClass : MyClass
{
    public new void Message() =>
        WriteLine("MyDerivedClass");
}
```

The addition of the IMessage interface changes the behavior of your derived class so that IMessage.Message() now uses the derived class version:

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // Prints "MyDerivedClass"
IMessagem = d as IMessage;
m.Message(); // Prints " MyDerivedClass "
```

You still need the new keyword in the definition of the MyDerivedClass .Message() method. That's your clue that there are still problems (see Item 33). The base class version is still accessible through a reference to the base class:

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // Prints "MyDerivedClass"
IMessagem = d as IM IMessagesg;
m.Message(); // Prints "MyDerivedClass"
MyClass b = d;
b.Message(); // Prints "MyClass"
```

One way to fix this problem is to modify the base class, declaring that the interface methods should be virtual:

```
public class MyClass : IMessage
{
    public virtual void Message() =>
        WriteLine(nameof(MyClass));
}

public class MyDerivedClass : MyClass
{
    public override void Message() =>
        WriteLine(nameof(MyDerivedClass));
}
```

MyDerivedClass—and all classes derived from MyClass—can declare their own methods for Message(). The overridden version will be called every time: through the MyDerivedClass reference, through the IMsg reference, and through the MyClass reference.

If you dislike the concept of impure virtual functions, just make one small change to the definition of MyClass:

```
public abstract class MyClass : IMessage
{
    public abstract void Message();
}
```

Yes, you can implement an interface without actually implementing the methods in that interface. By declaring abstract versions of the methods in the interface, you declare that all concrete types derived from your type must override those interface members and define their own implementations. The IMessage interface is part of the declaration of MyClass, but defining the methods is deferred to each concrete derived class.

Another partial solution is to implement the interface, and include a call to a virtual method that enables derived classes to participate in the interface contract. You would do that in MyClass as follows:

```
public class MyClass : IMessage
{
    protected virtual void OnMessage()
    {
    }

    public void Message()
    {
        OnMessage();
        WriteLine(nameof(MyClass));
    }
}
```

Any derived class can override OnMessage() and add its own work to the Message() method declared in MyClass. You've seen this pattern elsewhere, when classes implement `IDisposable` (see Item 26).

Explicit interface implementation (see Item 26 in *Effective C#, Third Edition*) enables you to implement an interface, yet hide its members from the public interface of your type. Its use throws a few other twists into the relationships between implementing interfaces and overriding virtual functions. Explicit interface implementation allows you to prevent client code from using the interface methods when a more appropriate version is available. The `IComparable` idiom in Item 20 in *Effective C#, Third Edition*, shows this behavior in detail.

There is one last wrinkle when you're working with interfaces and base classes: A base class can provide a default implementation for methods in an interface, and a derived class can then declare that it implements this interface. The derived class will inherit the implementation from the base class, as the next example shows:

```
public class DefaultMessageGenerator
{
    public void Message() =>
        WriteLine("This is a default message");
}

public class AnotherMessageGenerator :
    DefaultMessageGenerator, IMessage
{
    // No explicit Message() method needed
}
```

Notice that the derived class can declare the interface as part of its contract, because its base class provides an implementation. As long as it has a publicly accessible method with the proper signature, the interface contract is satisfied.

Implementing interfaces provides more options than just creating and overriding virtual functions. You can create sealed implementations, virtual implementations, or abstract contracts for class hierarchies. You can also create a sealed implementation and include virtual method calls in the methods that implement interfaces. You can decide exactly how and when derived classes can modify the default behavior of members of any interface that your class implements. Interface methods are not virtual methods, but rather a separate contract.

Item 16: Implement the Event Pattern for Notifications

The .NET Event Pattern is nothing more than syntax conventions on the Observer Pattern (see *Design Patterns* by Gamma, Helm, Johnson, and Vlissides, pp. 293–303). Events define the notifications for your type. They are built on delegates to provide type-safe function signatures for event handlers. Add to this the fact that most examples that use delegates are events, and it is not surprising that developers start thinking

that events and delegates are the same things. Item 7 in *Effective C#, Third Edition*, offers examples of when you can use delegates without defining events. You should raise events when your type must communicate with multiple clients to inform them of actions in the system. Events are how objects notify observers.

Consider a simple example. Suppose you're building a log class that acts as a dispatcher of all messages in an application. It will accept all messages from sources in your application and will dispatch those messages to any interested listeners. These listeners might be attached to the console, a database, the system log, or some other mechanism. You define the class as follows, to raise one event whenever a message arrives:

```
public class Logger
{
    static Logger()
    {
        Singleton = new Logger();
    }

    private Logger()
    {
    }

    public static Logger Singleton { get;  }

    // Define the event:
    public event EventHandler<LoggerEventArgs> Log;

    // Add a message, and log it.
    public void AddMsg(int priority, string msg) =>
        Log?.Invoke(this, new LoggerEventArgs(priority, msg));
}
```

The `AddMsg` method shows the proper way to raise events. The `?.` operator ensures that the event is raised only when listeners are attached to the event.

In the preceding example, LoggerEventArgs holds the priority of an event and the message. The delegate defines the signature for the event

handler. Inside the Logger class, the event field defines the event handler. The compiler sees the public event field definition and creates the add and remove operators for you. The generated code is similar to the following:

```
public class Logger
{
    private EventHandler<LoggerEventArgs> log;

    public event EventHandler<LoggerEventArgs> Log
    {
        add { log = log + value; }
        remove { log = log - value; }
    }

    public void AddMsg(int priority, string msg) =>
        log?.Invoke(this, new LoggerEventArgs(priority, msg));
}
```

The versions of the add and remove accessors that the C# compiler creates for the event use a different add and assign construct that is guaranteed to be thread safe. In general, the public event declaration language is more concise and easier to read and maintain than the add/remove syntax. When you create events in your class, you should declare public events and let the compiler automatically create the add and remove properties for you. Writing your own add and remove handlers lets you do more work in the add and remove handlers.

Events do not need to have any knowledge about the potential listeners. The following class automatically routes all messages to the Standard Error console:

```
class ConsoleLogger
{
    static ConsoleLogger() =>
        Logger.Singleton.Log += (sender, msg) =>
            Console.Error.WriteLine("{0}:\t{1}",
                msg.Priority.ToString(),
                msg.Message);
}
```

Another class could direct output to the system event log:

```
class EventLogger
{
    private static Logger logger = Logger.Singleton;
    private static string eventSource;
    private static EventLog logDest;

    static EventLogger() =>
        logger.Log += (sender, msg) =>
        {
        logDest?.WriteEntry(msg.Message,
            EventLogEntryType.Information,
            msg.Priority);
        };

    public static string EventSource
    {
        get { return eventSource; }

        set
        {
            eventSource = value;
            if (!EventLog.SourceExists(eventSource))
                EventLog.CreateEventSource(eventSource,
                    "ApplicationEventLogger");

            logDest?.Dispose();
            logDest = new EventLog();
            logDest.Source = eventSource;
        }
    }
}
```

Events notify any number of interested clients that something happened. The Logger class does not need any prior knowledge of which objects are interested in logging events.

The Logger class contains only one event, but some other classes (mostly Windows controls) have very large numbers of events. In those cases, the idea of using one field per event might be unacceptable. Sometimes,

only a small number of the defined events are actually used in any one application. If you encounter that situation, you can modify the design to create the event objects only when they are needed at runtime.

The core framework contains examples of how to achieve this goal in the Windows control subsystem. To do so in our example, you would add subsystems to the Logger class, then create an event for each subsystem. Clients would subsequently register on the event that is pertinent to their subsystems.

The extended Logger class has a `System.ComponentModel.EventHandlerList` container that stores all the event objects that should be raised for a given system. The updated AddMsg() method now takes a string parameter specifying the subsystem that generated the log message. If the subsystem has any listeners, the event is raised. Also, if an event listener has registered an interest in all messages, its event is raised:

```
public sealed class Logger
{
    private static EventHandlerList
        Handlers = new EventHandlerList();

    static public void AddLogger(
        string system, EventHandler<LoggerEventArgs> ev) =>
        Handlers.AddHandler(system, ev);

    static public void RemoveLogger(string system,
        EventHandler<LoggerEventArgs> ev) =>
        Handlers.RemoveHandler(system, ev);

    static public void AddMsg(string system,
        int priority, string msg)
    {
        if (!string.IsNullOrEmpty(system))
        {
            EventHandler<LoggerEventArgs> handler =
                Handlers[system] as
                EventHandler<LoggerEventArgs>;

            LoggerEventArgs args = new LoggerEventArgs(
                priority, msg);
            handler?.Invoke(null, args);
```

```
            // The empty string means receive all messages:
            l = Handlers[""] as
                EventHandler<LoggerEventArgs>;
            handler?.Invoke(null, args);
        }
    }
}
```

The preceding code stores the individual event handlers in the `EventHandlerList` collection. Sadly, there is no generic version of `EventHandlerList`, so you'll see a lot more casts and conversions in this block of code than you'll see in many of the samples in this book. In the example, client code attaches to a specific subsystem, and a new event object is created. Subsequent requests for the same subsystem retrieve the same event object.

If you develop a class that contains a large number of events in its interface, you should consider using this collection of event handlers. You create event members when clients attach to the event handler of their choice. Inside the .NET Framework, the `System.Windows.Forms.Control` class uses a more complicated variation of this implementation to hide the complexity of its event fields. Each event field internally accesses a collection of objects to add and remove the particular handlers. You can find more information about this idiom in the C# language specification.

The `EventHandlerList` class is one of the classes that have not been updated with a new generic version. It's not hard to construct your own from the `Dictionary` class:

```
public sealed class Logger
{
    private static Dictionary<string,
        EventHandler<LoggerEventArgs>>
        Handlers = new Dictionary<string,
            EventHandler<LoggerEventArgs>>();

    static public void AddLogger(
        string system, EventHandler<LoggerEventArgs> ev)
    {
        if (Handlers.ContainsKey(system))
            Handlers[system] += ev;
```

```
            else
                Handlers.Add(system, ev);
        }

        // Will throw an exception if the system
        // does not contain a handler.
        static public void RemoveLogger(string system,
            EventHandler<LoggerEventArgs> ev) =>
            Handlers[system] -= ev;

        static public void AddMsg(string system,
            int priority, string msg)
        {
            if (string.IsNullOrEmpty(system))
            {
                EventHandler<LoggerEventArgs> handler = null;
                Handlers.TryGetValue(system, out l);

                LoggerEventArgs args = new LoggerEventArgs(
                    priority, msg);
                handler?.Invoke(null, args);

                // The empty string means receive all messages:
                handler = Handlers[""] as
                    EventHandler<LoggerEventArgs>;
                handler?.Invoke(null, args);
            }
        }
    }
```

The generic version trades casts and type conversions for increased code to handle event maps. You might prefer the generic version, but it's a close tradeoff.

Events provide a standard syntax for notifying listeners. The .NET Event Pattern follows the event syntax to implement the Observer Pattern. Any number of clients can attach handlers to the events and process them, and those clients need not be known at compile time. Events don't need subscribers for the system to function properly. Using events in C# decouples the sender and the possible receivers of notifications.

The sender can be developed completely independently of any receivers. Events are the standard way to broadcast information about actions that your type has taken.

Item 17: Avoid Returning References to Internal Class Objects

You'd like to think that a read-only property is read-only and that callers can't modify it. Unfortunately, that's not always the way it works. If you create a property that returns a reference type, the caller can access any public member of that object, including those that modify the state of the property. For example:

```
public class MyBusinessObject
{
    public MyBusinessObject()
    {
        // Read-only property providing access to a
        // private data member:
        Data = new BindingList<ImportantData>();
    }

    public BindingList<ImportantData> Data { get; }
    // Other details elided
}
// Access the collection:
BindingList<ImportantData> stuff = bizObj.Data;
// Not intended, but allowed:
stuff.Clear(); // Deletes all data.
```

Any public client of MyBusinessObject can modify your internal data set. You created properties to hide your internal data structures, and you provided methods to let clients manipulate the data only through known methods, so your class can manage any changes to internal state. And then a read-only property opens a gaping hole in your class encapsulation! It's not even a read-write property, for which you would routinely consider these issues, but a true read-only property.

Welcome to the wonderful world of reference-based systems. Any member that returns a reference type returns a handle to that object. You gave the caller a handle to your internal structures, so the caller no longer needs to go through your object to modify that contained reference.

Clearly, you want to prevent this kind of behavior. You built the interface to your class, and you want users to follow it. You don't want users to access or modify the internal state of your objects without your knowledge. Naïve developers may innocently misuse your APIs and create bugs for which they later blame you. More sinister developers may maliciously probe your libraries for ways to exploit them. Don't provide functionality that you did not intend to offer. It won't be tested or hardened against malicious use.

Four different strategies can be used to protect your internal data structures from unintended modifications: value types, immutable types, interfaces, and wrappers.

Value types are copied when clients access them through a property. Any changes to the copy retrieved by the clients of your class do not affect your object's internal state. Clients can change these copies as much as necessary to achieve their purposes, and their changes will not affect your internal state.

Immutable types, such as `System.String`, are also safe (see Item 2). You can return strings, or any immutable type, while remaining safe in the knowledge that no client of your class can modify the string. Your internal state is safe.

Another option is to define interfaces that allow clients to access a subset of your internal member's functionality (see Item 14). When you create your own classes, you can create sets of interfaces that support subsets of the functionality of your class. By exposing the functionality through those interfaces, you minimize the possibility that your internal data will change in ways you did not intend. Clients can access the internal object through the interface you supplied, which will not include the full functionality of the class. Exposing the `IEnumerable<T>` interface reference in the `List<T>` is one example of this strategy. Machiavellian programmers may be able to defeat that strategy by using debugger tools or simply calling `GetType()` on a returned object to learn the type of the object that implements the interface and using a cast. Even so, you should take any steps that you can to make it harder for these developers to misuse your work to exploit end users.

Note that one strange twist in the `BindingList` class may cause some problems. A generic version of `IBindingList` isn't available, so you may want to create two different API methods for accessing the data:

one that supports data binding via the IBindingList interface, and one that supports programming through ICollection<T> or a similar interface.

```
public class MyBusinessObject
{
    // Read-only property providing access to a
    // private data member:
    private BindingList<ImportantData> listOfData = new
        BindingList<ImportantData>();
    public IBindingList BindingData =>
        listOfData;

    public ICollection<ImportantData> CollectionOfData =>
        listOfData;
    // Other details elided
}
```

Before we talk about how to create a completely read-only view of the data, let's briefly consider how you can respond to changes in your data when you allow public clients to modify it. This point is important because you'll often want to export an IBindingList to UI controls so that the user can edit the data. At some point, you've undoubtedly used Windows forms data binding to provide the means for your users to edit private data in your objects. The BindingList<T> class supports the IBindingList interface so that you can respond to any additions, updates, or deletions of items in the collection being shown to the user.

You can generalize this technique whenever you want to expose internal data elements for modification by public clients, but you need to validate and respond to those clients' changes. Your class subscribes to events generated by your internal data structure. Event handlers validate changes or respond to those changes by updating other internal state (see Item 16).

Returning to the original problem, you want to let clients view your data but not make any changes. When your data is stored in a BindingList<T>, you can enforce that constraint by setting various properties on the BindingList object (e.g., AddEdit, AllowNew, AllowRemove). The values of these properties are honored by UI controls; that is, the UI controls enable and disable different actions based on the values.

Because these properties are public, you can use them to modify the behavior of your collection. Of course, that also means you should not expose the `BindingList<T>` object as a public property. If you did, clients could modify those properties and circumvent your intent to create a read-only binding collection. Once again, exposing the internal storage through an interface type rather than the class type will limit what client code can do with that object.

The final choice for protecting your internal data structures from modification is to provide a wrapper object and export an instance of the wrapper, which minimizes access to the contained object. The .NET Framework immutable collections provide different collection types that support this approach. The `System.Collections.ObjectModel.ReadOnlyCollection<T>` type is the standard way to wrap a collection and export a read-only version of that data:

```
public class MyBusinessObject
{
    // Read-only property providing access to a
    // private data member:
    private BindingList<ImportantData> listOfData = new
        BindingList<ImportantData>();

    public IBindingList BindingData =>
        listOfData;
    public ReadOnlyCollection<ImportantData> CollectionOfData
=>
        new ReadOnlyCollection<ImportantData>(listOfData);
    // Other details elided
}
```

Exposing reference types through your public interface allows users of your object to modify its internal content without going through the methods and properties you've defined. That seems counterintuitive, which leads to a common mistake. You need to modify your class's interfaces to account for the fact that you are exporting references rather than values. If you simply return internal data, you've given access to those contained members. Your clients can then call any method that is available in your members. To limit that access, you should expose private internal data using interfaces, wrapper objects, or value types.

Item 18: Prefer Overrides to Event Handlers

Many .NET classes provide two different ways to handle events from the system: by attaching an event handler or by overriding a virtual function in the base class. Why provide two ways of doing the same thing? Because different situations call for different methods. Inside derived classes, you should always override the virtual function. Event handlers should be used only to respond to events in unrelated objects.

Suppose you write a nifty Windows Presentation Foundation (WPF) application that needs to respond to mouse down events. In your form class, you can choose to override the OnMouseDown() method:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    protected override void OnMouseDown(MouseButtonEventArgs e)
    {
        DoMouseThings(e);
        base.OnMouseDown(e);
    }
}
```

Alternatively, you could attach an event handler (which requires both C# and XAML):

```
<!-- XAML Description -->
    <Window x:Class="WpfApp1.MainWindow"
        xmlns:local="clr-namespace:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525"
        MouseDown="OnMouseDownHandler">
    <Grid >

    </Grid>
</Window>
```

```
// C# file
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void OnMouseDownHandler(object sender,
        MouseButtonEventArgs e)
    {
        DoMouseThings(e);
    }
}
```

You should prefer the first solution. This preference might seem surprising given the emphasis on declarative code in WPF applications. Even so, if the logic must be implemented in code, you should use the virtual method. If an event handler throws an exception, no other handlers in the chain for that event will be called (see Item 7 in *Effective C#, Third Edition*, and Item 16 earlier in this chapter). Some other ill-formed code may prevent the system from calling your event handler. By overriding the protected virtual function, you ensure that your handler is called first. The base class version of the virtual function is responsible for calling any event handlers attached to the particular event. Thus, if you want to call the event handlers (and you almost always do), you must call the base class. In some rare cases, you will want to replace the default behavior instead of calling the base class version so that the event handlers aren't called. You can't guarantee that all the event handlers will be called—because some ill-formed event handler might throw an exception—but you can guarantee that your derived class's behavior is correct.

If that explanation doesn't convince you of the superiority of the virtual function, examine the first listing in this item again and compare it to the second listing. Which is clearer? Overriding a virtual function means that there is only one function to examine and modify if you need to maintain the form. By comparison, the event mechanism has two points to maintain: the event handler function and the code that wires up the event. Either of these could be the point of failure. One function is simpler.

Of course, the .NET Framework designers must have added events for a reason, right? Yes, they did. Like the rest of us, they're too busy to write code that no one uses. The overrides are intended for derived classes; every other class must use the event mechanism. That also means declarative actions defined in the XAML file will be accessed through the event handlers.

In our example, the designer may want certain actions to occur on a mouse down event. The designer will create XAML declarations for those behaviors, and the behaviors will be accessed using events on the form. You could redefine all that behavior in your code, but that's far too much work to handle one event. In addition, it just moves the problem from the designer's hands to yours. Clearly, you would prefer that designers do the design work instead of you. The obvious way to handle this situation is to create an event and access the XAML declarations created by a design tool. In the end, you will have created a new class to send an event to the form class. It would be simpler to just attach the form's event handler to the form in the first place. After all, that's why the .NET Framework designers put those events in the forms.

Another reason for using the event mechanism is that events are wired up at runtime. As a consequence, events offer more flexibility. You can wire up different event handlers, depending on the circumstances of the program. As an example, suppose that you write a drawing program. Depending on the state of the program, a mouse down event might start drawing a line, or it might select an object. When the user switches modes, you can switch event handlers. Different classes, with different event handlers, can handle the event in different ways depending on the state of the application.

Finally, with events, you can hook up multiple event handlers to the same event. Imagine the same drawing program again. Multiple event handlers might be hooked up on the mouse down event. The first might perform the particular action. The second might update the status bar or update the accessibility of different commands. In this way, you can ensure that multiple actions take place in response to the same event.

When one function handles one event in a derived class, an override is the better approach. It is easier to maintain, more likely to remain correct over time, and more efficient. Reserve the event handlers for other uses. Prefer overriding the `base` class implementation to attaching an event handler.

Item 19: Avoid Overloading Methods Defined in Base Classes

When a base class chooses the name of a member, it assigns the semantics to that name. Under no circumstances may the derived class use the same name for different purposes. Yet, there are many other reasons why a derived class may want to use the same name. For example, it may want to implement the same semantics in a different way, or with different parameters. Sometimes that's naturally supported by the language: Class designers declare virtual functions so that derived classes can implement semantics differently. Item 10 in *Effective C#, Third Edition*, explains why using the `new` modifier can lead to hard-to-find bugs in your code. In this item, you'll learn why creating overloads of methods that are defined in a base class leads to similar issues. You should not overload methods declared in a base class.

The rules for overload resolution are necessarily complicated. Possible candidate methods might be declared in the target class, any of its base classes, any extension method using the class, and interfaces it implements. Add generic methods and generic extension methods, and it gets very complicated. Throw in optional parameters, and you might not know exactly what the results will be. Do you really want to add more complexity to this situation? Creating overloads for methods declared in your base class adds more possibilities to the best overload match, which in turn increases the chance of ambiguity. It also increases the chance that your interpretation of the specification will differ from the compiler's interpretations, and it will certainly confuse your users. The solution is simple: Pick a different method name. It's your class, and you certainly have enough brilliance to come up with a different name for a method, especially if the alternative is confusion for everyone using your types.

The guidance here is straightforward, yet people question whether it really needs to be so strict. Perhaps that's because overloading sounds very much like overriding. Overriding virtual methods is a core principle of C-based object-oriented languages; that's obviously not what is meant here. Overloading means creating multiple methods with the same name and different parameter lists. Does overloading base class methods dramatically affect overload resolution? To explore this question, let's look at the different ways in which overloading methods in the base class can cause issues.

This problem has a lot of permutations, so let's start simple. The interplay between overloads in base classes reflects the base and derived classes used for parameters. The examples use this class hierarchy for parameters:

```
public class Fruit { }
public class Apple : Fruit { }
```

Here's a class with one method, using the derived parameter (Apple):

```
public class Animal
{
    public void Foo(Apple parm) =>
        WriteLine("In Animal.Foo");
}
```

Obviously, this snippet of code writes "In Animal.Foo":

```
var obj1 = new Animal();
obj1.Foo(new Apple());
```

Now let's add a new derived class with an overloaded method:

```
public class Tiger : Animal
{
    public void Foo(Fruit parm) =>
        WriteLine("In Fruit.Foo");
}
```

What happens when you execute this code?

```
var obj2 = new Timer();
obj2.Foo(new Apple());
obj2.Foo(new Fruit());
```

Both lines print "in Tiger.Foo." You always call the method in the derived class. Any number of developers would figure that the first call would print "in Animal.Foo," but even the simple overload rules can be surprising. Both calls resolve to Tiger.Foo because, when a candidate method appears in the most derived compile-time type, that method is the better method. That's still true when an even better match is found in a base class. The principle at work is that the derived class author has more knowledge about the specific scenario. The argument that is most heavily weighted in overload resolution is the receiver, `this`. What do you suppose the following code snippet does?

```
Animal obj3 = new Tiger();
obj3.Foo(new Apple());
```

In this snippet, obj3 has the compile-time type of Animal (the base class), even though the runtime type is Tiger (the derived class). Foo isn't virtual; therefore, obj3.Foo() must resolve to Animal.Foo.

If your perplexed users actually want to get the resolution rules they might expect, they will need to use casts:

```
var obj4 = new Tiger();
((Animal)obj4).Foo(new Apple());
obj4.Foo(new Fruit());
```

If your API forces this kind of construct on your poor users, you've failed. In fact, you can easily add a bit more confusion. Add one method to your base class, B:

```
public class Animal
{
    public void Foo(Apple parm) =>
        WriteLine("In Animal.Foo");

    public void Bar(Fruit parm) =>
        WriteLine("In Animal.Bar");
}
```

Clearly, the following code prints "In Animal.Bar":

```
var obj1 = new Tiger();
obj1.Bar(new Apple());
```

Now add a different overload, and include an optional parameter:

```
public class Tiger : Animal
{
    public void Foo(Apple parm) =>
        WriteLine("In Tiger.Foo");

    public void Bar(Fruit parm1, Fruit parm2 = null) =>
        WriteLine("In Tiger.Bar");
}
```

You've already seen what will happen here. The same snippet of code now prints "In Tiger.Bar" (you're calling your derived class again):

```
var obj1 = new Tiger();
obj1.Bar(new Apple());
```

The only way to get at the method in the base class (again) is to provide a cast in the calling code.

These examples illustrate the kinds of problems you can encounter with a single parameter method. The issues become increasingly more confusing as you add parameters based on generics. Suppose you add this method:

```
public class Animal
{
    public void Foo(Apple parm) =>
        WriteLine("In Animal.Foo");

    public void Bar(Fruit parm) =>
        WriteLine("In Animal.Bar");

    public void Baz(IEnumerable<Apple> parm) =>
        WriteLine("In Animal.Foo2");
}
```

Now provide a different overload in the derived class:

```
public class Tiger : Animal
{
    public void Foo(Fruit parm) =>
        WriteLine("In Tiger.Foo");

    public void Bar(Fruit parm1, Fruit parm2 = null) =>
        WriteLine("In Tiger.Bar");

        public void Baz(IEnumerable<Fruit> parm) =>
            WriteLine("In Tiger.Foo2");
}
```

Call Baz in a manner similar to the earlier calls:

```
var sequence = new List<Apple> { new Apple(), new Apple() };
var obj2 = new Tiger();

obj2.Baz(sequence);
```

What do you suppose is printed this time? If you've been paying attention, you might assume that "In Tiger.Foo2" is printed. That answer gets you partial credit, because that output is what happens in C# 4.0. In C# 4.0 and later, generic interfaces support covariance and contravariance, which means Tiger.Foo2 is a candidate method for an IEnumerable<Apple> when its formal parameter type is an IEnumerable<Apple>. In contrast, earlier versions of C# do not support generic variance; that is, generic parameters are invariant. In those versions, Tiger.Foo2 is not a candidate method when the parameter is an IEnumerable<Apple>. The only candidate method is Animal.Foo2, which is the correct answer in those versions.

The code examples illustrated that you sometimes need casts to help the compiler pick the method you want in many complicated situations. In the real world, you'll undoubtedly run into situations where you need to use casts because class hierarchies, implemented interfaces, and extension methods have conspired to decide the method you want, rather than the compiler picking the "best" method. Of course, just because real-world situations are occasionally ugly, that does not mean you should add to the problem by creating more overloads yourself.

Now you can amaze your friends at programmer cocktail parties with a more in-depth knowledge of overload resolution in C#. This can be useful information to have, and the more you know about your chosen language, the better you'll be as a developer. But don't expect your users to have the same level of knowledge. More importantly, don't rely on everyone having that kind of detailed knowledge of how overload resolution works as a precondition for using your API. Instead, do your users a favor and don't overload methods declared in a base class. It doesn't provide any value, and it will just lead to confusion among your users.

Item 20: Understand How Events Increase Runtime Coupling Among Objects

Events seem to provide a way to completely decouple your class from those types it needs to notify. Thus, you'll often provide outgoing event definitions. Let subscribers, whatever type they might be, subscribe to those events. Inside your class, you raise the events. Your class knows nothing about the subscribers, and it places no restrictions on the classes that can implement those interfaces. Any code can be extended to subscribe to those events and create whatever behavior they need when those events are raised.

And yet, it's not that simple. Coupling issues arise related to event-based APIs. To begin with, some event argument types contain status flags that direct your class to perform certain operations.

```
public class WorkerEngine
{
    public event EventHandler<WorkerEventArgs> OnProgress;
    public void DoLotsOfStuff()
    {
        for (int i = 0; i < 100; i++)
        {
            SomeWork();
            WorkerEventArgs args = new WorkerEventArgs();
            args.Percent = i;
            OnProgress?.Invoke(this, args);
            if (args.Cancel)
                return;
        }
    }
    private void SomeWork()
    {
        // Elided
    }
}
```

This code ensures that every subscriber to that event is coupled. Suppose you have multiple subscribers on a single event. One subscriber might submit a cancel request, and a second subscriber might reverse that request. The foregoing definition does not guarantee that this behavior can't happen. Having multiple subscribers and a mutable event argument means that the last subscriber in the chain can override every other subscriber. There's no way to enforce having only one subscriber, and there is no way to guarantee that you're the last subscriber. You could modify the event arguments to ensure that once the cancel flag is set, no subscriber can turn it off:

```
public class WorkerEventArgs : EventArgs
{
    public int Percent { get; set; }
    public bool Cancel { get; private set; }
```

```
    public void RequestCancel()
    {
        Cancel = true;
    }
}
```

Changing the public interface works correctly here, but it might not work as intended in some other cases. If you need to ensure that there is exactly one subscriber, you must choose another way of communicating with any interested code. For example, you can define an interface and call that one method. Alternatively, you can ask for a delegate that defines the outgoing method. Then your single subscriber can decide whether it wants to support multiple subscribers and how to orchestrate the semantics of cancel requests.

At runtime, another form of coupling exists between event sources and event subscribers. Your event source holds a reference to the delegate that represents the event subscriber. The event subscriber's object lifetime now will match the event source's object lifetime. The event source will call the subscriber's handler whenever the event occurs. That behavior must not continue after the event subscriber is disposed. (Recall that the contract of `IDisposable` states that no other methods should be called after an object is disposed; see Item 17 in *Effective C#, Third Edition*.)

Event subscribers need to modify their implementation of the dispose pattern to unhook event handlers as part of the `Dispose()` method. Otherwise, subscriber objects will continue to live because reachable delegates exist in the event source object. This scenario is another case in which runtime coupling can cost you. Even though coupling appears to be looser because the compile-time dependencies are minimized, runtime coupling does have costs.

Event-based communication loosens the static coupling between types, but that outcome comes at the cost of tighter runtime coupling between the event generator and the event subscribers. The multicast nature of events means that all subscribers must agree on a protocol for responding to the event source. The event model, in which the event source holds a reference to all subscribers, means that all subscribers must either (1) remove event handlers when the subscriber wants to be disposed of or (2) simply cease to exist. Also, the event source must unhook

all event handlers when the source should cease to exist. You must factor those issues into your design decision to use events.

Item 21: Declare Only Nonvirtual Events

Like many other class members in C#, events can be declared as virtual. It would be nice to think that this process is as easy as declaring any other C# language element as virtual. Unfortunately, because you can declare events using field-like syntax as well as the `add` and `remove` syntax, it's not that simple. It's remarkably easy to create event handlers across base and derived classes that don't work the way you expect. Even worse, you can create hard-to-diagnose crashes.

Let's modify the worker engine from Item 20 to provide a base class that defines the basic event mechanism:

```
public abstract class WorkerEngineBase
{
    public virtual event
        EventHandler<WorkerEventArgs> OnProgress;

    public void DoLotsOfStuff()
    {
        for (int i = 0; i < 100; i++)
        {
            SomeWork();
            WorkerEventArgs args = new WorkerEventArgs();
            args.Percent = i;
            OnProgress?.Invoke(this, args);
            if (args.Cancel)
                return;
        }
    }

    protected abstract void SomeWork();
}
```

The compiler creates a private backing field, along with public `add` and `remove` methods.

Because that private backing field is compiler generated, you can't write code to access it directly. Instead, you can invoke it only through the

publicly accessible event declaration. That restriction obviously applies to derived events as well. Although you can't manually write code that accesses the private backing field of the base class, the compiler can access its own generated fields; in this way, the compiler can create the proper code to override the events in the correct manner. In effect, creating a derived event hides the event declaration in the base class. This derived class does exactly the same work as in the original example:

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        // Elided
    }
}
```

The addition of an `override` event breaks the code:

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        Thread.Sleep(50);
    }
    // Broken. This hides the private event field in
    // the base class.
    public override event
        EventHandler<WorkerEventArgs> OnProgress;
}
```

The declaration of the overridden event means that the hidden backing field in the base class is not assigned when user code subscribes to the event. The user code subscribes to the derived event, and there is no code in the derived class to raise the event.

In turn, when the base class uses a field-like event, overriding that event definition hides the event field defined in the base class. Code in the base class that raises the event doesn't do anything, because all subscribers have attached to the derived class. It doesn't matter whether the derived class uses a field-like event definition or a property-like event definition: The derived class version hides the base class event. No events raised in the base class code actually call a subscriber's code.

Derived classes work only if they use the `add` and `remove` accessors:

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        Thread.Sleep(50);
    }
    public override event
        EventHandler<WorkerEventArgs> OnProgress
    {
        add { base.OnProgress += value; }
        remove { base.OnProgress -= value; }
    }
    // Important: Only the base class can raise the event.
    // Derived classes cannot raise the events directly.
    // If derived classes should raise events, the base
    // class must provide a protected method to
    // raise the events.
}
```

You can also make this idiom work if the base class declares a property-like event.

The base class needs to be modified so that it contains a protected event field, and the derived class property can then modify the base class variable:

```
public abstract class WorkerEngineBase
{
    protected EventHandler<WorkerEventArgs> progressEvent;

    public virtual event
        EventHandler<WorkerEventArgs> OnProgress
    {
        [MethodImpl(MethodImplOptions.Synchronized)]
        add
        {
            progressEvent += value;
        }
```

```
            [MethodImpl(MethodImplOptions.Synchronized)]
            remove
            {
                progressEvent -= value;
            }
        }

        public void DoLotsOfStuff()
        {
            for (int i = 0; i < 100; i++)
            {
                SomeWork();
                WorkerEventArgs args = new WorkerEventArgs();
                args.Percent = i;
                progressEvent?.Invoke(this, args);

                if (args.Cancel)
                    return;
            }
        }

        protected abstract void SomeWork();
    }
    public class WorkerEngineDerived : WorkerEngineBase
    {
        protected override void SomeWork()
        {
            // Elided
        }
        // Works. Access base class event field.
        public override event
            EventHandler<WorkerEventArgs> OnProgress
        {
            [MethodImpl(MethodImplOptions.Synchronized)]
            add
            {
                progressEvent += value;
            }
```

```
        [MethodImpl(MethodImplOptions.Synchronized)]
        remove
        {
            progressEvent -= value;
        }
    }
}
```

However, this code still constrains your derived class's implementations. The derived class cannot use the field-like event syntax:

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        // Elided
    }
    // Broken. Private field hides the base class.
    public override event
        EventHandler<WorkerEventArgs> OnProgress;
}
```

You are left with two options here to fix the problem. First, whenever you create a virtual event, don't use field-like syntax—not in the base class nor in any derived classes. The other solution is to create a virtual method that raises the event whenever you create a virtual event definition. Any derived class must override the raise event method as well as override the virtual event definition.

```
public abstract class WorkerEngineBase
{
    public virtual event
        EventHandler<WorkerEventArgs> OnProgress;

    protected virtual WorkerEventArgs
        RaiseEvent(WorkerEventArgs args)
    {
        OnProgress?.Invoke(this, args);
        return args;
    }
```

```
    public void DoLotsOfStuff()
    {
        for (int i = 0; i < 100; i++)
        {
            SomeWork();
            WorkerEventArgs args = new WorkerEventArgs();
            args.Percent = i;
            RaiseEvent(args);
            if (args.Cancel)
                return;
        }
    }

    protected abstract void SomeWork();
}

public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        Thread.Sleep(50);
    }

    public override event
        EventHandler<WorkerEventArgs> OnProgress;

    protected override WorkerEventArgs
        RaiseEvent(WorkerEventArgs args)
    {
        OnProgress?.Invoke(this, args);
        return args;
    }
}
```

An examination of this code reveals that you really don't gain anything by declaring the event as virtual. The existence of the virtual method to raise the event is all you need to customize the event-raising behavior in the derived class. There really isn't anything you can do by overriding the event itself that you can't do by overriding the method that raises the event: You can iterate all the delegates by hand, and you can provide different semantics for handling how event arguments are changed by each subscriber. You can even suppress events by not raising anything.

At first glance, events seem to provide a loose-coupling interface between your class and those other pieces of code that are interested in communicating with your class. If you've created virtual events, both compile-time and runtime coupling occurs between your event sources and those classes that subscribe to your events. The fixes you need to add to your code to make virtual events work usually mean you don't need a virtual event anyway.

Item 22: Create Method Groups That Are Clear, Minimal, and Complete

The more possible overloads you create for a method, the more often you'll run into ambiguity. Even worse, when you make seemingly innocent changes to your code, you can cause different methods to be called and, in turn, unexpected results to be generated.

In many cases, it's easier to work with fewer overloaded methods than with more overloads. Your goal should be to create precisely the right number of overloads: enough of them that your type is easy for client developers to use, but not so many that you complicate the API and make it harder for the compiler to create exactly the one best method.

The more the ambiguity you create, the more difficult it is for other developers to create code that uses new C# features such as type inference. The more ambiguous methods you have in place, the more likely it is that the compiler cannot conclude that exactly one method is best.

The C# language specification describes all the rules that determine which method will be interpreted as the best match. As a C# developer, you should have some understanding of these rules. More importantly, as an API writer, you should have a *solid* understanding of the rules. It is your responsibility to create an API that minimizes the risk of compilation errors caused by the compiler's attempt to resolve ambiguity. It's even more important that you don't lead your users down the path of misunderstanding which of your methods the compiler will choose in reasonable situations.

The C# compiler can follow quite a lengthy path as it determines whether there is one best method to call and, if there is, what that one best method is. When a class has only nongeneric methods, it's reasonably easy to follow the logic and identify which methods will be called. The more possible variations you add, however, the worse the situation gets, and the more likely it is that you will create ambiguity.

Several conditions may change the way the compiler resolves these methods. Specifically, this process is affected by the number and the type of parameters, whether generic methods are potential candidates, whether any interface methods are possible, and whether any extension methods are candidates and are imported into the current context.

The compiler can look in numerous locations for candidate methods. Then, after it finds all candidate methods, it must try to pick the one best method. If there are no candidate methods or if there is no unique best candidate among the multiple candidate methods, a compiler error is generated. Of course, those are the easy cases: You can't ship code that has compiler errors. More challenging problems occur when you and the compiler disagree about which method is best. In those cases, the compiler always wins, and you may get undesired behavior.

Any methods that have the same name should perform essentially the same function. For example, two methods in the same class named Add() should do the same thing. If the methods do semantically different things, then they should have different names. For example, you should never write code like this:

```
public class Vector
{
    private List<double> values = new List<double>();

    // Add a value to the internal list.
    public void Add(double number) =>
        values.Add(number);

    // Add values to each item in the sequence.
    public void Add(IEnumerable<double> sequence)
    {
        int index = 0;
        foreach (double number in sequence)
        {
            if (index == values.Count)
                return;
            values[index++] += number;
        }
    }
}
```

Either of these two Add() methods is reasonable, but there is no way both should be part of the same class. Different overloaded methods should provide different parameter lists—never different actions.

That rule alone limits the possible errors caused when the compiler calls a different method from the one you expect. If both methods perform the same action, it really shouldn't matter which one gets called, right?

Of course, different methods with different parameter lists often have different performance metrics. Even when multiple methods perform the same task, you should get the method you expect. You as the class author can make that happen by minimizing the chances for ambiguity.

Ambiguity problems arise when methods have similar arguments and the compiler must make a choice. In the simplest case, there is only one parameter for any of the possible overloads:

```
public void Scale(short scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(int scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(float scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(double scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}
```

By creating all these overloads, you avoid introducing any ambiguity. Every numeric type except `decimal` is listed, so the compiler always calls

the version that is a correct match. (The `decimal` type is omitted here because converting a value from `decimal` to `double` requires an explicit conversion.) If you have a C++ programming background, you're probably wondering why I haven't recommended replacing all those overloads with a single generic method. The answer is C# generics don't support that practice in the way C++ templates do. With C# generics, you can't assume that arbitrary methods or operators are present in the type parameters. You must specify your expectations using constraints (see Item 18 in *Effective C#, Third Edition*). Of course, you might think about using delegates to define a method constraint (see Item 7 in *Effective C#, Third Edition*). Unfortunately, that technique merely moves the problem to another location in the code where both the type parameter and the delegate are specified. You're stuck with some version of this code.

But suppose you left out some of the overloads:

```
public void Scale(float scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(double scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}
```

Now it's a bit trickier for users of the class to determine which method will be called for the `short` and `double` cases. There are implicit conversions from `short` to `float`, and from `short` to `double`. Which one will the compiler pick? If it can't pick one method, you've forced coders to specify an explicit cast so that their code will compile. In this case, the compiler decides that `float` is a better match than `double`. Every float can be converted to a double, but not every double can be converted to a float. Therefore, `float` must be "more specific" than `double`, making it a better choice. However, most of your users may not come to the same conclusion. Here's how to avoid this problem: When you create multiple overloads for a method, make sure that most developers would immediately recognize which method the compiler will pick as a best match. That's best achieved by providing a complete set of method overloads.

Single-parameter methods are rather simple, but it can be difficult to understand methods that have multiple parameters. Here are two methods with two sets of parameters:

```
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public void Scale(int xScale, int yScale)
    {
        X *= xScale;
        Y *= yScale;
    }

    public void Scale(double xScale, double yScale)
    {
        X *= xScale;
        Y *= yScale;
    }
}
```

What happens if you call the methods with `int`, `float`? Or with `int`, `long`?

```
Point p = new Point { X = 5, Y = 7 };
// Note that the second parameter is a long:
p.Scale(5, 7L); // Calls Scale(double,double)
```

In both cases, only one of the parameters is an exact match to one of the overloaded method parameters. That method does not contain an implicit conversion for the other parameter, so it's not even a candidate method. Some developers would probably guess wrong when trying to determine which method gets called.

But wait—method lookup can get even more complicated. Let's throw a new wrench into the works and see what happens. What if there appears to be a better method available in a base class than exists in a derived class? (See Item 19 for details.)

```
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }
```

```
    // Earlier code elided
    public void Scale(int scale)
    {
        X *= scale;
        Y *= scale;
    }
}
public class Point3D : Point
{
    public double Z { get; set; }

    // Not override, not new. Different parameter type.
    public void Scale(double scale)
    {
        X *= scale;
        Y *= scale;
        Z *= scale;
    }
}

Point3D p2 = new Point3D { X = 1, Y = 2, Z = 3 };
p2.Scale(3);
```

There are quite a few mistakes here. Point should declare Scale() as a virtual method if the class author intends for Scale to be overridden. But the author of the overriding method—let's call her Kaitlyn—made a different mistake: By creating a new method (rather than hiding the original), Kaitlyn has ensured that the user of her type will generate code that calls the wrong method. The compiler finds both methods in scope and determines (based on the type of the parameters) that Point.Scale(int) is a better match. By creating a set of conflicting method signatures, Kaitlyn has created this ambiguity.

Adding a generic method to catch all the missing cases, using a default implementation, creates an even more sinister situation:

```
public static class Utilities
{
    // Prefer Math.Max for double:
    public static double Max(double left, double right) =>
        Math.Max(left, right);
```

```
    // Note that float, int, etc., are handled here:
    public static T Max<T>(T left, T right)
         where T : IComparable<T> =>
        (left.CompareTo(right) > 0 ? left : right);
}
double a1 = Utilities.Max(1, 3);
double a2 = Utilities.Max(5.3, 12.7f);
double a3 = Utilities.Max(5, 12.7f);
```

The first call instantiates a generic method for Max<int>; the second call goes to Max(double, double); and the third call goes to a generic method for Max<float>. That result occurs because one of the types can always be a perfect match for a generic method, and no conversion is required. A generic method becomes the best method if the compiler can perform the correct type substitution for all type parameters. Yes, even if there are obvious candidate methods that require implicit conversions, the generic method is considered a better method match whenever it is accessible.

But I'm not finished throwing complications at you: Extension methods can also be considered in the mix. What happens if an extension method appears to be a better match than an accessible member function? Thankfully, extension methods are a last resort; they are examined only if no applicable instance method is found.

As you can see, the compiler looks for candidate methods in quite a few places. As you put more methods in more places, you expand that list. The larger the list, the more likely it is that the potential methods will present an ambiguity. Even if the compiler is certain which method is the one best method, you've introduced potential ambiguity into the mix for your users. If only one in 20 developers can correctly identify which method overload is called when he or she invokes one of a series of overloaded methods, you've clearly made your API too complex. Users should immediately know which of the possible set of accessible overloads the compiler has chosen as the best. Anything less is obfuscating your library.

To provide a complete set of functionality for your users, create the minimum set of overloads—then stop. Adding methods will just increase your library's complexity without enhancing its usefulness.

Item 23: Give Partial Classes Partial Methods for Constructors, Mutators, and Event Handlers

The C# language team added partial classes so that code generators can create their part of the classes, and human developers can augment the generated code in another file. Unfortunately, that separation is not sufficient for sophisticated usage patterns. Often, the human developers need to add code in members created by the code generator. Those members might include constructors, event handlers defined in the generated code, and any mutator methods defined in the generated code.

Your purpose is to free developers who use your code generator from feeling that they should modify your generated code. If you are on the other side, using code created by a tool, you should never modify the generated code. Doing so breaks the relationship with the code generator tool and makes it much more difficult for you to continue to use it.

In some ways, writing partial classes is API design. You, as the human developer or as the author of a code generation tool, are creating code that must be used by some other developer (either the person or the code generation tool). In other ways, it's like having two developers work on the same class, but with serious restrictions. The two developers can't talk to each other, and neither developer can modify the code written by the other. To deal with these challenges, you need to provide plenty of hooks for those other developers. You should implement those hooks—which another developer may, or may not, need to implement—in the form of partial methods.

Your code generator defines partial methods for those extension points. Partial methods provide a way for you to declare methods that may be defined in another source file in a partial class. The compiler looks at the full class definition, and, if partial methods have been defined, it generates calls to those methods. If no class author has written the partial method, then the compiler removes any calls to it.

Because partial methods may or may not be part of the class, the language imposes several restrictions on the method signatures of partial methods: The return type must be `void`, partial methods cannot be abstract or virtual, and they cannot implement interface methods. The parameters cannot include any `out` parameters, because the compiler cannot initialize `out` parameters. Nor can it create the return value if the method body has not been defined. Implicitly, all partial methods are private.

For three class member types, you should add partial methods that enable users to monitor or modify the class behavior—namely, mutator methods, event handlers, and constructors.

Mutator methods are any methods that change the observable state of the class. From the standpoint of partial methods and partial classes, you should interpret that definition as involving any change in state. The other source files that make up a partial class implementation are part of the class and, therefore, have complete access to your class's internal structures.

Mutator methods should provide the other class authors with two partial methods. The first method should be called before the change that provides validation hooks and before the other class author has a chance to reject the change. The second method is called after a change in state and allows the other class author to respond to the state change.

Your tool's core code would look something like this:

```
// Consider this the portion generated by your tool
public partial class GeneratedStuff
{
    private int storage = 0;

    public void UpdateValue(int newValue) =>
        storage = newValue;
}
```

You should add hooks both before and after the state change. In this way, you let other class authors modify or respond to the change:

```
// Consider this the portion generated by your tool
public partial class GeneratedStuff
{
    private struct ReportChange
    {
        public readonly int OldValue;
        public readonly int NewValue;

        public ReportChange(int oldValue, int newValue)
        {
            OldValue = oldValue;
            NewValue = newValue;
        }
    }
```

```
    private class RequestChange
    {
        public ReportChange Values { get; set; }
        public bool Cancel { get; set; }
    }

    partial void ReportValueChanging(RequestChange args);
    partial void ReportValueChanged(ReportChange values);

    private int storage = 0;

    public void UpdateValue(int newValue)
    {
        // Precheck the change
        RequestChange updateArgs = new RequestChange
        {
            Values = new ReportChange(storage, newValue)
        };
        ReportValueChanging(updateArgs);
        if (!updateArgs.Cancel) // If OK,
        {
            storage = newValue; // change
                                // and report:
            ReportValueChanged(new ReportChange(
                storage, newValue));
        }
    }
}
```

If no one has written bodies for either partial method, then the compiled version of UpdateValue() looks like this:

```
public void UpdateValue(int newValue)
{
    RequestChange updateArgs = new RequestChange
    {
        Values = new ReportChange(this.storage, newValue)
    };
    if (!updateArgs.Cancel)
    {
        this.storage = newValue;
    }
}
```

The hooks allow the developer to validate or respond to any change:

```
public partial class GeneratedStuff
{
    partial void ReportValueChanging(
        RequestChange args)
    {
        if (args.Values.NewValue < 0)
        {
            WriteLine($@"Invalid value:
               {args.Values.NewValue}, canceling");
            args.Cancel = true;
        }
        else
            WriteLine($@"Changing
                {args.Values.OldValue} to
                {args.Values.NewValue}");
    }
    partial void ReportValueChanged(
        ReportChange values)
    {
        WriteLine($@"Changed
            {values.OldValue} to {values.NewValue}");
    }
}
```

This example shows a protocol with a cancel flag that lets developers cancel any mutator operation. When creating your class, you might prefer a protocol in which the user-defined code can throw an exception to cancel an operation. Throwing the exception is a better option if the cancel operation should be propagated up to the calling code. Otherwise, the Boolean cancel flag should be used because of its lightweight nature.

Furthermore, notice that the RequestChange object is created in this example even when ReportValueChanged() will not be called. You can have any code execute in that constructor, and the compiler cannot assume that the constructor call can be removed without changing the semantics of the UpdateValue() method. You should strive to require minimal work for client developers to create those extra objects needed for validating and requesting changes.

It's fairly easy to spot all the public mutator methods in a class, but remember to include all the public `set` accessors for properties. If you forget some of them, other class authors can't validate or respond to property changes.

Next, you need to provide hooks for user-generated code in the constructors. Neither the generated code nor the user-written code can control which constructor gets called, so your code generator must solve this problem. It should provide a hook to call user-defined code when one of the generated constructors gets called. Here is an extension to the GeneratedStuff class shown earlier:

```
// Hook for user-defined code:
partial void Initialize();

public GeneratedStuff() :
    this(0)
{
}

public GeneratedStuff(int someValue)
{
    this.storage = someValue;
    Initialize();
}
```

Notice that Initialize() is the last method called during construction. That organization enables the handwritten code to examine the current object state and possibly make any modifications or throw exceptions if developers find something invalid for their problem domains. You need to ensure that you don't call Initialize() twice, and that this method is called from every constructor defined in the generated code. Human developers must not call their own Initialize() routines from any constructors they add. Instead, they should explicitly call one of the constructors defined in the generated class to ensure that any initialization necessary in the generated code takes place.

Finally, if the generated code subscribes to any events, you should consider providing partial method hooks during the processing of that event. This consideration is especially important if the event requests status or cancel notifications from the generated class. The user-defined code may want to modify the status or change the cancel flag.

Partial classes and partial methods provide the mechanisms you need to completely separate generated code from user-written code in the same class. With the extensions shown here, you should never need to modify code generated by a tool. Most developers are probably using code generated by Visual Studio or other tools. Before you consider modifying any of the code written by such a tool, you must examine the interface provided by the generated code and determine whether it has provided partial method declarations that you can use to accomplish your goal. More importantly, if you are the author of the code generator, you must provide a complete set of hooks in the form of partial methods to support any desired extensions to your generated code. Doing anything less will lead developers down a dangerous path and will encourage them to abandon your code generator.

Item 24: Avoid ICloneable Because It Limits Your Design Choices

`ICloneable` sounds like a good idea: You implement the `ICloneable` interface for types that support copies; if you don't want to support copies, don't implement it. Of course, your type does not live in a vacuum. Your decision to support `ICloneable` affects derived types as well. Once a type supports `ICloneable`, all its derived types must do the same. All its member types must also support `ICloneable` or have some other mechanism to create a copy.

Moreover, supporting deep copies is very problematic when you create designs that contain webs of objects. `ICloneable` finesses this problem in its official definition: It supports either a deep copy or a shallow copy. A shallow copy creates a new object that contains copies of all member fields. If those member variables are reference types, the new object refers to the same object that the original does. A deep copy creates a new object that copies all member fields as well. All reference types are cloned recursively in the copy. In built-in types, such as integers, the deep and shallow copies produce the same results. Which one does a type support? That depends on the type—but recognize that mixing shallow and deep copies in the same object causes quite a few inconsistencies.

When you go wading into the `ICloneable` waters, it can be hard to escape. Most often, avoiding `ICloneable` altogether makes for a simpler class. Such a class is easier to use, and it's easier to implement.

Any value type that contains only built-in types as members does not need to support `ICloneable`; a simple assignment copies all the values of the `struct` more efficiently than `Clone()`. `Clone()` must box its return value so that it can be coerced into a `System.Object` reference. The caller must then perform another cast to extract the value from the box. You've got enough to do—don't write a `Clone()` function that replicates an assignment.

What about value types that contain reference types? The most obvious case is a value type that contains a `string`:

```
public struct ErrorMessage
{
    private int errCode;
    private int details;
    private string msg;

    // Details elided
}
```

The `string` type is a special case because this class is immutable. If you assign an error message object, both the original and the newly assigned error message objects will refer to the same `string`. This does not cause any of the problems that might happen with a general reference type. If you change the msg variable through either reference, you create a new `string` object (see Item 15 in *Effective C#, Third Edition*).

The general case of creating a `struct` that contains arbitrary reference fields is more complicated, albeit far rarer. The built-in assignment for the `struct` creates a shallow copy, with both the original and the copied `structs` referring to the same object. To create a deep copy, you need to clone the contained reference type, and you need to know that the reference type supports a deep copy with its `Clone()` method. Even then, that process will work only if the contained reference type also supports `ICloneable`, and its `Clone()` method creates a deep copy.

Now let's move on to reference types. Reference types could support the `ICloneable` interface to indicate that they support either shallow or deep copying. You should add support for `ICloneable` judiciously, because doing so mandates that all classes derived from your type must also support `ICloneable`. Consider this small hierarchy:

```
class BaseType : ICloneable
{
    private string label = "class name";
    private int[] values = new int[10];

    public object Clone()
    {
        BaseType rVal = new BaseType();
        rVal.label = label;
        for (int i = 0; i < values.Length; i++)
            rVal.values[i] = values[i];
        return rVal;
    }
}

class Derived : BaseType
{
    private double[] dValues = new double[10];

    static void Main(string[] args)
    {
        Derived d = new Derived();
        Derived d2 = d.Clone() as Derived;

        if (d2 == null)
            Console.WriteLine("null");
    }
}
```

If you run this program, you will find that the value of d2 is null. The Derived class does inherit `ICloneable.Clone()` from BaseType, but that implementation is not correct for the Derived type: It clones only the base type. BaseType.Clone() creates a BaseType object, not a Derived object. That is why d2 is null in the test program—it's not a Derived object. However, even if you could overcome this problem, BaseType.Clone() could not properly copy the dValues array that was defined in Derived.

When you implement `ICloneable`, you force all derived classes to implement it as well. In fact, you should provide a hook function to let all derived classes use your implementation (see Item 15). To support

cloning, derived classes can add only member fields that are value types or reference types that implement `ICloneable`. That is a very stringent limitation on all derived classes. Adding `ICloneable` support to base classes usually creates such a burden on derived types that you should avoid implementing `ICloneable` in nonsealed classes.

When an entire hierarchy must implement `ICloneable`, you can create an abstract `Clone()` method and force all derived classes to implement it. In those cases, you need to define a way for the derived classes to create copies of the base members. That's done by defining a protected copy constructor:

```
class BaseType
{
    private string label;
    private int[] values;

    protected BaseType()
    {
        label = "class name";
        values = new int[10];
    }

    // Used by derived values to clone
    protected BaseType(BaseType right)
    {
        label = right.label;
        values = right.values.Clone() as int[];
    }
}

sealed class Derived : BaseType, ICloneable
{
    private double[] dValues = new double[10];

    public Derived()
    {
        dValues = new double[10];
    }
```

```
    // Construct a copy
    // using the base class copy ctor
    private Derived(Derived right) :
        base(right)
    {
        dValues = right.dValues.Clone()
            as double[];
    }

    public object Clone()
    {
        Derived rVal = new Derived(this);
        return rVal;
    }
}
```

Base classes do not implement `ICloneable`; instead, they provide a protected copy constructor that enables derived classes to copy the base class parts. Leaf classes, which should all be sealed, implement `ICloneable` when necessary. The base class does not force all derived classes to implement `ICloneable`, but it does provide the necessary methods for any derived classes that want `ICloneable` support.

`ICloneable` does have its uses, but those cases are the exception rather than the rule. Notably, the .NET Framework did not add `ICloneable<T>` when it was updated with generic support. You should never add support for `ICloneable` to value types; use the assignment operation instead. You should add support for `ICloneable` to leaf classes when a copy operation is truly necessary for the type. Base classes that are likely to be used where `ICloneable` will be supported should create a protected copy constructor. In all other cases, avoid `ICloneable`.

Item 25: Limit Array Parameters to params Arrays

Using array parameters can expose your code to several unexpected problems. It's much better to create method signatures that use alternative representations to pass collections or variable-size arguments to methods.

Arrays have special properties that allow you to write methods that appear to implement strict type checking but fail at runtime. The following small program compiles without problems and passes all the compile-time type

checking. However, it throws an `ArrayTypeMismatchException` when you assign a value to the first object in the parms array in ReplaceIndices:

```
string[] labels = new string[] { "one", "two",
    "three", "four", "five" };

ReplaceIndices(labels);

static private void ReplaceIndices(object[] parms)
{
    for (int index = 0; index < parms.Length; index++)
        parms[index] = index;
}
```

This problem arises because arrays are covariant as input parameters. You don't have to pass the exact type of the array into the method. Furthermore, even though the array is passed by value, the contents of the array can be references to reference types. Your method can change the members of the array in ways that will not work with some valid types.

Of course, the foregoing example is a bit obvious, and you probably think you'll never write code like that. But consider this small class hierarchy:

```
class B
{
    public static B Factory() => new B();

    public virtual void WriteType() => WriteLine("B");
}

class D1 : B
{
    public static new B Factory() => new D1();

    public override void WriteType() => WriteLine("D1");
}

class D2 : B
{
    public static new B Factory() => new D2();

    public override void WriteType() => WriteLine("D2");
}
```

If you use this hierarchy correctly, everything is fine:

```
static private void FillArray(B[] array, Func<B> generator)
{
    for (int i = 0; i < array.Length; i++)
        array[i] = generator();
}

// Elsewhere:
B[]
storage = new B[10];
FillArray(storage, () => B.Factory());
FillArray(storage, () => D1.Factory());
FillArray(storage, () => D2.Factory());
```

Nevertheless, any mismatch between the derived types will throw the same `ArrayTypeMismatchException`:

```
B[] storage = new D1[10];
// All three calls will throw exceptions:
FillArray(storage, () => B.Factory());
FillArray(storage, () => D1.Factory());
FillArray(storage, () => D2.Factory());
```

Furthermore, because arrays don't support contravariance, when you write array members, your code will fail to compile, even though it should work:

```
static void FillArray(D1[] array)
{
    for (int i = 0; i < array.Length; i++)
        array[i] = new D1();
}

B[] storage = new B[10];
// Generates compiler error CS1503 (argument mismatch)
// even though D objects can be placed in a B array
FillArray(storage);
```

Things become even more complicated if you want to pass arrays as ref parameters. You will be able to create a derived class, but not a base class, inside the method. However, the objects in the array can still be the wrong type.

You can avoid this kind of problem by specifying parameters as interface types that create a type-safe sequence to use. Input parameters should be specified as `IEnumerable<T>` for some `T`. This strategy ensures that you can't modify the input sequence, because `IEnumerable<T>` does not provide any methods to modify the collection. Another alternative is to pass types as base classes—a practice that may also avoid the creation of APIs that support modifying the collection. When you write a method for which one of the arguments is an array, the caller must expect that you may replace any or all the elements of that array. There's no way to limit that usage. If you don't intend to make modifications to the collection, indicate that fact in your API signature. (See the other items in this chapter for many examples.)

When you need to modify the sequence, it's best to use an input parameter of one sequence and return the modified sequence (see Item 31 in *Effective C#, Third Edition*). When you want to generate the sequence, return the sequence as an `IEnumerable<T>` for some `T`.

On some occasions, you may want to pass arbitrary options in methods—and that's when you can reach for an array of arguments. When you do so, make sure to use a `params` array. The `params` array allows the user of your method to simply place those elements as other parameters. Contrast these two methods:

```
// Regular array
private static void WriteOutput1(object[] stuffToWrite)
{
    foreach (object o in stuffToWrite)
        Console.WriteLine(o);
}
// params array
private static void WriteOutput2(
    params object[] stuffToWrite)
{
    foreach (object o in stuffToWrite)
        Console.WriteLine(o);
}
```

As you can see, there is very little difference in how you create the method or how you test for the members of the array. However, note the difference in the calling sequence:

```
WriteOutput1(new string[]
    { "one", "two", "three", "four", "five" });
WriteOutput2("one", "two", "three", "four", "five");
```

The trouble for your users gets worse if they don't want to specify any of the optional parameters. The `params` array version can be called with no parameters:

```
WriteOutput2();
```

The version with a regular array presents your users with some painful options. This version won't compile:

```
WriteOutput1(); // Won't compile
```

Trying `null` as the argument will throw a null exception:

```
WriteOutput1(null); // Throws a null argument exception
```

Your users are stuck with all this extra typing:

```
WriteOutput1(new object[] { });
```

This alternative is still not perfect. Even `params` arrays can have the same problems with covariant argument types, although you're less likely to run into these difficulties. First, the compiler generates the storage for the array passed to your method. It doesn't make sense to try to change the elements of a compiler-generated array, and the calling method won't see any of the changes anyway. Furthermore, the compiler automatically generates the correct type of array. To create the exception, the developers using your code need to write truly pathological constructs. They would need to create an actual array of a different type. Then they would have to use that array as the argument in place of the `params` array. Although it is possible, the system has already done quite a bit to protect against this kind of error.

Arrays are not always the wrong method parameters, but they can cause two types of errors. The array's covariance behavior can cause runtime errors, and array aliasing can mean the callee can replace the callers' objects. Even when your method doesn't exhibit those problems, the method signature implies that it might. That possibility will raise concerns among developers using your code: Is it safe? Should they create temporary storage? Whenever you use an array as a parameter to a method, there is almost always a better alternative. If the parameter

represents a sequence, use `IEnumerable<T>` or a constructed `IEnumerable<T>` for the proper type. If the parameter represents a mutable collection, then rework the signature to mutate an input sequence and create the output sequence. If the parameter represents a set of options, use a `params` array. In all those cases, you'll end up with a better, safer interface.

Item 26: Enable Immediate Error Reporting in Iterators and Async Methods Using Local Functions

Modern C# includes some very high-level language constructs that generate a large amount of machine code. Among these are iterator methods and async methods. Major advantages of these constructs are less source code and clearer source code. Of course, nothing is ever truly free: Both iterator methods and async methods delay execution of the code you write in those methods. This initial code often takes the form of argument checking and object validation code that should throw exceptions immediately if a method was called incorrectly or at the wrong time. Those outcomes won't happen, however, because the compiler-generated code has restructured your algorithm. Consider this example:

```
public IEnumerable<T> GenerateSample<T>(
    IEnumerable<T> sequence, int sampleFrequency)
{
    if (sequence == null)
        throw new ArgumentException(
            "Source sequence cannot be null",
            paramName: nameof(sequence));
    if (sampleFrequency < 1)
        throw new ArgumentException(
            "Sample frequency must be a positive integer",
            paramName: nameof(sampleFrequency));

    int index = 0;
    foreach(T item in sequence)
    {
        if (index % sampleFrequency == 0)
            yield return item;
    }
}
```

```
var samples = processor.GenerateSample(fullSequence, -8);
Console.WriteLine("Exception not thrown yet!");
foreach (var item in samples) // Exception thrown here
{
    Console.WriteLine(item);
}
```

The argument exception is not thrown when the iterator method is called. Instead, it's thrown when the sequence returned by the iterator is enumerated. In this simplified example, you can likely see where the error is, and fix it quickly. In contrast, in large-scale programs, the code that creates the iterator and the code that enumerates the sequence might not be in the same method, or even in the same class. That can make it much more difficult to find and diagnose the problem, because the exception is thrown in code that's unrelated to the code that actually has the problem.

The same situation happens with async methods. Consider this example:

```
public async Task<string> LoadMessage(string userName)
{
    if (string.IsNullOrWhiteSpace(userName))
        throw new ArgumentException(
            message: "This must be a valid user",
            paramName: nameof(userName));
    var settings = await context.LoadUser(userName);
    var message = settings.Message ?? "No message";
    return message;
}
```

The `async` modifier instructs the compiler to rearrange the code in the method, and return a Task that manages the status of the asynchronous work. The returned Task object stores the state of that asynchronous work. Only when that Task is awaited will any exceptions thrown during that method be observed. (See items in Chapter 3 for details.) As with iterator methods, the exception may be thrown in code that isn't near the code that generated the initial problem.

Ideally, you'd like to report those errors as soon as they are found. Developers who are using your library incorrectly should see mistakes reported when they are made, thereby ensuring that they can fix those mistakes more easily. The way to achieve that goal is to separate these methods into two different methods. Let's start with iterator methods.

An iterator method is a method that uses `yield return` statements to return a sequence as that sequence is enumerated. These methods must return an `IEnumerable<T>` or an `IEnumerable`. In fact, many methods can return those types. The technique you use to ensure that programming errors are reported eagerly is to split the iterator method into two methods: an implementation method that uses `yield return`, and a wrapper method that does all the validation. You can split the first example into two methods as follows. Here's the wrapper method:

```
public IEnumerable<T> GenerateSample<T>(
    IEnumerable<T> sequence, int sampleFrequency)
{
    if (sequence == null)
        throw new ArgumentNullException(
            paramName: nameof(sequence),
            message: "Source sequence cannot be null",
            );
    if (sampleFrequency < 1)
        throw new ArgumentException(
        message: "Sample frequency must be a positive integer",
        paramName: nameof(sampleFrequency));

    return generateSampleImpl();
}
```

This wrapper method handles all the argument validation and any other state validation. Then, it calls the implementation method that does the work. Here's the implementation method as a local function nested inside GenerateSample:

```
IEnumerable<T> generateSampleImpl()
{
    int index = 0;
    foreach (T item in sequence)
    {
        if (index % sampleFrequency == 0)
            yield return item;
    }
}
```

This second method does not have any error checking, so you should limit its accessibility as much as possible. At a minimum, it should be a private method. Starting with C# 7, you can make this implementation

method a local function, defined inside the wrapper method. This technique offers several advantages. Here's the full code, using a local function for the implementation iterator method:

```
public IEnumerable<T> GenerateSampleFinal<T>(
    IEnumerable<T> sequence, int sampleFrequency)
{
    if (sequence == null)
        throw new ArgumentException(
            message: "Source sequence cannot be null",
            paramName: nameof(sequence));
    if (sampleFrequency < 1)
        throw new ArgumentException(
        message: "Sample frequency must be a positive integer",
        paramName: nameof(sampleFrequency));

    return generateSampleImpl();

    IEnumerable<T> generateSampleImpl()
    {
        int index = 0;
        foreach (T item in sequence)
        {
            if (index % sampleFrequency == 0)
                yield return item;
        }
    }
}
```

The most important advantage of using a local function in this way is that this implementation method can be called only from the wrapper method. Thus, there is no way to bypass the validation code and call the implementation method directly. Notice also that the implementation method does have access to all the local variables and all the arguments to the wrapper method. None of them need to be explicitly passed as arguments to the implementation method.

You can use the same technique for async methods. In that case, the public method is a Task or ValueTask returning method that does not include the `async` modifier. This wrapper method does all the validation and eagerly reports any errors. The implementation method includes the `async` modifier and performs the asynchronous work.

The implementation method should have the most limited scope possible. You should use a local function whenever possible:

```
public Task<string> LoadMessageFinal(string userName)
{
    if (string.IsNullOrWhiteSpace(userName))
        throw new ArgumentException(
            message: "This must be a valid user",
            paramName: nameof(userName));

    return loadMessageImpl();

    async Task<string> loadMessageImpl()
    {
        var settings = await context.LoadUser(userName);
        var message = settings.Message ?? "No message";
        return message;
    }
}
```

The advantages are the same as those provided by the other approaches: Programming errors in calling the method are reported eagerly, and should be easier to fix. The implementation method is hidden inside the wrapper method. The wrapper method's validation code cannot be bypassed.

Let's make one final observation before leaving this topic: The technique of using local functions may look very similar to using lambda functions for the implementation method. The implementation is different, and local functions are the better choice. The compiler must generate more complex structures for a lambda expression than for a local function. Lambda expressions require instantiation of a delegate object, whereas local functions can often be implemented as private methods.

High-level constructs such as iterator methods and async methods rearrange your code and change when errors are reported. That's how those methods work. You can create the behavior you want by splitting the methods in two. When you choose this approach, make sure you limit the accessibility of the implementation method that lacks error checking.

Index

Numbers

0 (zero), ensuring 0 is valid state for value type, 24–28

A

Abstract base classes
inheritance and, 73
when to use inheritance vs. interfaces, 74–75

Abstract `Clone()` method, 128–129

Abstract properties, 4

Accessors
add/remove, 88
implicit properties supporting, 8–10
as inexpensive operation, 29–31
providing synchronized access to data, 3–4
source objects, 256
specifying, 4–5

`add` accessor
derived classes using, 108–109
overview of, 88

`Add()` method
costs of dynamic typing, 261–262
dynamic methods at runtime, 238
rewriting using lambda expressions, 231–232

`Aggregate()`, of `Enumerable` class, 232

`AggregateException`, handling exceptions in parallel operations, 190–194

Algorithms
anonymous types and, 43–45
for computing square root, 197–201
constructing parallel algorithms with exceptions in mind, 189–195
constructing using async methods, 139
PLINQ implementation of parallel algorithms, 177–189

Amdahl's law, 188

Analyzers, automating practices with, 271–272

Anonymous types
creating user-defined types, 34–35
defining local functions, 41–45
drawbacks of, 35–36
limiting type scope using tuples, 38–39
scope of, 36
tuples compared with, 39

API design
avoiding conversion operators, 61–65
avoiding `ICloneable` interface, 125–129
avoiding overloading methods defined in base classes, 100–104
avoiding returning references to internal class objects, 93–96
creating clear, minimal, and complete method groups, 113–119
deadlock due to poor API design, 150
declaring only nonvirtual events, 107–113
enabling error reporting in iterators and async methods, 134–138

API design (*continued*)
Event Pattern use for notifications, 86–93
giving partial classes partial methods for constructors, mutators, and event handlers, 120–125
interface methods compared with virtual methods, 82–86
interfaces preferred over inheritance, 73–82
limiting array parameters to `params` arrays, 129–134
limiting type visibility, 69–73
overrides preferred to event handlers, 97–99
overview of, 61
parameter use in minimizing method overloads, 65–69
understanding how events increase runtime coupling among objects, 104–107

APIs
minimizing dynamic objects in public APIs, 259–265
Roslyn repository, 271–272
using Expression API, 253–259

Arrays, limiting array parameters to `params` arrays, 129–134

`AsParallel()`
adding parallel execution to queries, 178
adding to loops, 177
impact on execution model when added to query, 182–183

Async methods
avoiding composing synchronous and asynchronous methods, 149–154
cache generalized async return types, 173–176
download method, 190
enabling error reporting, 134–138
not using async void methods, 143–149
using, 139–143

Asynchronous programming
avoiding composing synchronous and asynchronous methods, 149–154
avoiding marshalling context unnecessarily, 156–160
avoiding thread allocations and context switches, 154–156
cache generalized async return types, 173–176
composing using task objects, 160–166
exception to rule for avoiding composing synchronous and asynchronous methods, 152
implementing task cancellation protocol, 166–173
not using async void methods, 143–149
overriding, 139
using async methods, 139–143

Atomic types, mutability of, 12–13

Automating practices, with analyzers, 271–272

`await`
flow control and, 140–141
processing methods with multiple `await` expressions, 142–143

B

`BackgroundWorker` class, for cross-thread communication, 201–205

Backing field
creating private, 107–108
implicit properties supporting access specifiers, 8–10
read-only properties surrounding, 34

Base classes
abstract, 73–75
accessing methods, 103–104
avoiding overloading methods defined, 100–104
declaring property-like event, 109–111

implementing common behavior across related types, 81–82
implanting `ICloneable` interface and, 129
when to use inheritance vs. interfaces, 74–75

`Bind` methods, use with `DynamicDictionary`, 250–252

Bugs, due to poor API design, 150

C

C# community
automating practices with analyzers, 271–272
overview of, 267
participating in specs and code, 269–270
seeking the best answer, 267–269

C++ language, C# contrasted with, 18–19

C language, overriding virtual methods in C-based languages, 100

Callbacks, sources of unknown code, 228

`CallInterface()` method, 254

Cast operator
accessing methods in base class, 103–104
converting dynamic types to static, 231
for explicit conversion, 63

`Cast<T>`, `System.Linq.Enumerable`, 238–240

`catch`. *See* `try/catch` blocks

Chunk partitioning, in PLINQ, 179

Class hierarchies. *See also* Inheritance
defining related types, 82
implanting `ICloneable` interface and, 128
limiting array parameters to params arrays, 130–131

`Class` keyword, defining value types and reference types, 19

Class Library, .NET Framework, 48

Classes. *See also* by individual types
array class as reference type, 17
avoiding `ICloneable` interface, 125–129
avoiding overloading methods defined in base classes, 100–104
avoiding returning references to internal class objects, 93–96
conversion operators used for substitutability between, 61
creating user-defined types, 34
data binding classes supporting properties no public data fields, 2
declaring property-like event in base class, 109–111
derived classes, 82–86, 127–128
giving partial classes partial methods for constructors, mutators, and event handlers, 120–125
implementing common behavior across related types, 81–82
limiting visibility of types, 69–73
understanding how event increase runtime coupling among objects, 104–107
when to use, 18–19
when to use inheritance vs. interfaces, 74–75

CLR (Common Language Runtime), 50

Code
code reuse, 75
improving, 268
participating in C# community, 269–270

Code Cracker library, 271

Collaboration, 269

Collection types, protecting internal data structures using wrappers, 96

COM (Component Object Model)
COM APIs using named parameters, 66–67
STA (single-threaded apartment) model, 205–206

Common Language Runtime (CLR), 50

Community. *See* C# community

`CompareExchange()`, in locking threads, 221

Compilers
accessing methods in base class, 104
costs of dynamic typing, 261–262
creating clear, minimal, and complete method groups, 113–119
creating user-defined types, 35
processing async methods, 141–142
runtime costs of dynamic typing and building expressions at, 237
`try/catch` blocks, 143

Component Object Model (COM)
COM APIs using named parameters, 66–67
STA (single-threaded apartment) model, 205–206

Constraints, specifying, 229–230

Constructors
giving partial classes partial methods for, 120–125
monitoring/modifying class behavior, 121
type conversion and, 62
when to replace conversion operator with, 65

Context, avoiding marshalling context unnecessarily, 156–160

Context-aware code, 156–160

Context-free code, 156–160

Context switches, 154–156

Conversion operators
compiler selection of methods, 116
converting dynamic types to static, 231
converting value types to reference types, 240
implicit vs. explicit type conversion, 62–64
side effects in type conversion, 61–62
substitutability between classes, 61
when to replace with constructor, 65

Coupling, understanding how events increase runtime coupling among objects, 104–107

Covariance behavior, arrays, 133

CPUs
avoiding thread allocations and context switches, 154–156
running CPU work synchronously vs. asynchronously, 153–154

CSharpLang repository, 269–270

CSV data, 263–265

D

Data
data-driven dynamic types, 242–252
ensuring properties behave like data, 28–34
implicit properties preferred for mutable data, 8–12
protecting internal data structures, 94–95
providing synchronized access to data, 3–4
separating storage from manipulation, 1
storing, 21
working with CSV data, 263–265

Data binding, support for properties, 2

Data members
ensuring properties behave like data, 28–34
property use instead of accessible data members, 4–7

Data types
creating user-defined types, 34–35
data binding support for properties, 2
defining shape of tuple as return type, 39–41
distinguishing between value types and reference types, 18–24
drawbacks of using anonymous types, 35–36
dynamic types. *See* Dynamic programming
enhancements to properties, 1
ensuring 0 is valid state for value type, 24–28
ensuring properties behave like data, 28–34
event-based communication loosening static coupling between types, 106–107
immutable types preferred for value types, 12–18
implicit properties preferred for mutable data, 8–12
implicit property syntax, 4
indexers, 5–8
leveraging runtime type of generic type parameters, 238–241
limiting type scope using tuples, 38–39
limiting type visibility, 69–73
multithreaded support implemented via properties, 3
overview of, 1
pitfalls of `GetHashCode()`, 54–60
property use instead of accessible data members, 4–7
pros/cons of dynamic typing, 229–238
scope of anonymous types, 36–38
type conversion and, 62
type safety, 231
understanding equality relationships, 45–54
virtual properties, 4

Databases, pulling data from remote, 32–33

Deadlocks
causes of, 225
due to poor API design, 150
locking strategy and, 222
problems from composing synchronous code on top of asynchronous methods, 150–152

Debugging
difficulty of debugging expression trees, 252
walking through code with `await` expressions, 141

Deep copies, `ICloneable` interface supporting, 125

Delegates. *See also* Events
abstract, 206–207
anonymous, 209
blocking thread until completed, 227
built on events, 86–87
caching compiled, 234–235
callbacks, 228
lambda expressions compiled into, 232–233, 255
outdated practices, 268
processing, 211–214

Derived classes
implanting `ICloneable` interface and, 127–128
understanding difference between interface and virtual function, 82–86

Derived events, declaring only nonvirtual events, 107–113

Derived types, value types not supporting, 16

Developers
C# community, 267
collaboration between, 269
forcing use of error-handling mechanism, 144–145
giving partial classes partial methods for constructors, mutators, and event handlers, 120–125
meeting expectations of, 28–32

`Dispatcher` class, WPF (Windows Presentation Foundation), 210

Download method, async methods, 190

Dynamic programming
- `DynamicObject` or `IDynamicMetaObjectProvider` for data-driven dynamic types, 242–252
- leveraging runtime type of generic type parameters, 238–241
- minimizing dynamic objects in public APIs, 259–265
- overview of, 229
- pros/cons of dynamic typing, 229–238
- using Expression API, 253–259

`DynamicObject`, for data-driven dynamic types, 242–252

`DynamicPropertyBag`, creating dynamic types, 242–244

E

Encapsulation, property use and, 1

End-of-task cycle, thread pools managing, 196–197

Entity Framework, lazy evaluation of queries, 182

`enum`
- as flag, 27–28
- modifying start value of 0, 24–26

`Enumerable` class
- `Aggregate()`, 232
- benefits of private classes in limiting visibility, 70

`Enumerator` pattern, .NET Framework, 70

Equality
- checking if two object references are same, 50–52
- `Equals()` not throwing exceptions, 49–50
- `ISStructuralEquality`, 53–54
- `operator==()` expression, 53
- overriding `Equals()`, 53
- specifying equality, 46–48
- summary of C# options for testing, 54

`Equals()`
- not throwing exceptions, 49–50
- returning hash values from two equal objects, 57
- specifying equality, 45–49

Errors. *See also* Exceptions
- `async void` methods in error recovery, 149
- enabling error reporting in iterators and async methods, 134–138
- forcing use of mechanism for, 144–145

Event handlers
- creating async event handlers, 146–148
- event-based communication loosening static coupling between types, 106–107
- Event Pattern use for notifications, 86–93
- giving partial classes partial methods for, 120–125
- monitoring/modifying class behavior, 121
- overrides preferred over, 97–99

Event Pattern, 86–93

Events
- declaring only nonvirtual, 107–113
- Event Pattern use for notifications, 86–93
- understanding how they increase runtime coupling among objects, 104–107

Exception handling, PLINQ, 194–195

Exceptions. *See also* Errors
- async methods handling, 143
- asynchronous methods reporting through `Task` object, 143–144
- constructing parallel algorithms with exceptions in mind, 189–195
- `Equals()` not throwing, 49–50

problems from composing synchronous code on top of asynchronous methods, 150–151
throwing, 123
Expression API, 253–259
Expression trees
avoiding repeated code, 232–233
difficulty of debugging, 252
Expression API, 253
Expressions
deciding when to use expressions or dynamic typing, 235–238
for executing pseudocode, 256–258
Expression API, 253
Extension methods, creating clear, minimal, and complete method groups, 119

F

Fixes, improving code, 268
Flag, `enum` as, 27–28
Flow control, `await` and async and, 140–141
`for` loops. *See also* Loops, 29
Functions
defining local functions on anonymous type, 41–45
properties use instead of member functions, 4
for specifying equality, 45
understanding difference between interface and virtual function, 82–86

G

Generics
comparing C++ with C#, 116
generic methods, 118–119
`get` accessor
implicit properties supporting, 8–10
as inexpensive operation, 29–33
providing synchronized access to data, 3–4
source objects, 256
specifying accessibility modifiers, 4–5
`GetHashCode()`
defining hash value, 54
rules for overloading, 55–60
specific requirements for applying, 60
`GetMetaObject`, `IDynamicMetaObjectProvider`, 249
GitHub, 269–272
Groups, method, 113–119
GUI applications
avoiding marshalling context unnecessarily, 156–160
avoiding thread allocations and context switches, 154–156

H

Handles, specifying smallest possible scope for lock handles, 223–225
Hash-based collections, immutable types used with, 12
Hash code, generating integer value, 55
Hash partitioning, in PLINQ, 180
Hashes, defining hash value, 54
Hero of Alexandria algorithm, computing square root, 197–201
Higher-order functions, defining local functions on anonymous type, 42

I

`ICloneable` interface, avoiding, 125–129
`IDynamicMetaObjectProvider`, for data-driven dynamic types, 242–252
`IEnumerable`, LINQ queries and, 238–240
`IEnumerator <string>`, iteration with, 183

Immutable types
adding constructors to, 15–16
creating, 16–17
difficulties of using in practice, 12–14
initializing, 18
preferred for value types, 12–18
protecting internal data structures, 94
understanding, 12

Implementation method
splitting iterator into two methods, 136–137
using with async, 137–138

Implicit properties
limitation of, 12
preferred for mutable data, 8–11

Indexers
implementing as methods, 5–8
retrieving XElement, 248

Inheritance. *See also* Class hierarchies
compared with interfaces, 73–74
examples, 75–82
when to use, 74–75

`InnerExceptions`, 190–193

Interfaces
avoiding `ICloneable` interface, 125–129
`CallInterface()`, 254
compared with inheritance, 73
creating, 83–84
examples, 75–82
implementing public interfaces with less visible classes, 69–70
interface methods compared with virtual methods, 82–86
limiting visibility of types, 69–73
limitations of, 74
options in describing set of functionality (contract), 73–74
protecting internal data structures,94–95
specifying parameters as interface types, 132
when to use, 75

`Interlocked` method, `System.Threading` class, 220–221

Inverted enumeration, PLINQ, 181–182

`Invoke`, Windows Forms controls
`BeginInvoke` and `EndInvoke`, 206, 208–209, 212–214
`InvokeAsync`, 208–209
`InvokeIfNeeded`, 207–208
`InvokeRequired`, 206, 208–211, 214
processing UI threads, 211–212
`WndProc`, 213–214

`IParallelEnumerable` class, 178

IronPython, 260

`ISStructuralEquality`, 53–54

Iterator methods, enabling error reporting, 134–138

Iterators, `IEnumerator <string>` and, 183

J

Java, C# contrasted with, 18–19

L

Lambda expressions
compiling into delegate, 232–233
creating lock inside, 225
implementation methods compared with, 138
manipulating anonymous types, 45
`orderby` clause, 188
parameter expression, 255
returning destination objects, 258
rewriting `Add` method, 231–232
sources of unknown code, 228

`Lazy<T>` class, .NET Framework, 32

Library
Code Cracker library, 271
.NET Framework Class library, 48

Task Parallel Library for managing threads, 196
for working with CSV data, 263–265
LINQ
compared with PLINQ, 178
`IEnumerable` sequences, 238–240
issues due to errors in background tasks, 194–195
source sequence for LINQ query, 258
support in C#, 253
LINQ to Objects
enumeration loop, 183–187
lazy evaluation of queries, 182
overview of, 177
PLINQ implementation of queries compared with, 188
LINQ to SQL, 182, 188
LINQ to XML, 244–248
Listeners, event syntax for notifying, 92–93
`lock` keyword, 215
Locks, thread
avoiding calling unknown code in locked sections, 225–228
`CompareExchange()`, 221
`lock()` for thread synchronization, 214–220
specifying smallest possible scope for lock handles, 221–225
`System.Threading.Interlocked`, 220–221
Logs/logging
`async void` method using exception filter, 147–148
events, 88–91
Loops
`AsParallel()`, 177
developer expectations and, 29
enumeration loop, 183

M

Members functions, properties use versus, 4
`MethodImpAttribute`, protecting method from deadlock, 222–223
Methods
avoiding composing synchronous and asynchronous methods, 149–154
avoiding overloading methods defined in base classes, 100–104
creating clear, minimal, and complete method groups, 113–119
enabling error reporting in iterators and async methods, 134–138
giving partial classes partial methods for constructors, mutators, and event handlers, 120–125
indexers implemented as, 5–6
interface methods compared with virtual methods, 82–86
manipulating anonymous types, 45
mapping from type to type, 43–44
not using async void methods, 143–149
parameter use in minimizing method overloads, 65–69
properties implemented as, 1–2, 28
protecting from deadlocks, 222–223
using async methods, 139–143
Microsoft Intermediate Language (MSIL)
implicit properties supporting access specifiers, 8–9
storing parameter names, 68
Microsoft Office, 66–67
`Monitor.Enter()`, primitives for thread synchronization, 215–220
`Monitor.Exit()`, primitives for thread synchronization, 215–220
MSIL (Microsoft Intermediate Language)
implicit properties supporting access specifiers, 8–9
storing parameter names, 68

Multithreading, implementing via properties, 3
Mutable types
creating companion class for immutable type, 18
implicit properties preferred for, 8–12
Mutators
giving partial classes partial methods for, 120–125
monitoring/modifying class behavior, 121–123
`MyType`
implicit conversion operator in, 239
locks, 223
storing strings with, 240–241

N

Named parameters, when to use, 65–69
.NET Framework
avoiding `ICloneable`, 129
Class Library, 48
code reuse, 75
collection types, 96
`Color` type, 18
cross-thread calls emulating synchronous calls, 225
`Enumerator` pattern, 70
event handlers, 97–99
examining or creating code at runtime, 253
`GetHashCode()` rule, 60
hiding complexity of event fields, 91
`Lazy<T>` class, 32
`operator==()`, 53
property use for public members, 2
tools for asynchronous programming, 139
value types vs. reference types, 19
`ValueTask<T>` type, 173
WIN 32 API legacy behavior, 214
`new` keyword, creating interfaces, 83–84
Notifications, Event Pattern for, 86–93

O

Object-oriented languages, overriding virtual methods, 100
`Object.Equals()`, specifying equality, 46–49
`Object.ReferenceEquals()`, specifying equality, 46–48
Objects
avoiding returning references to internal class objects, 93–96
composing using task objects, 160–166
dynamic types as `System.Object` with runtime binding, 230
`DynamicObject` for data-driven dynamic types, 242–252
minimizing dynamic objects in public APIs, 259–265
understanding how event increase runtime coupling among objects, 104–107
Observer Pattern, 86
Open source, C# as, 269–270
`operator==()`, 45, 53
`orderby` clause, lambda expressions, 188
Overloading
avoiding overloading methods defined in base classes, 100–104
overuse creates ambiguity, 113–119
parameter use in minimizing method overloads, 65–69
Overriding
declaring only nonvirtual events, 107–113
`operator ==()`, 45
overloading compared with, 100
preferred to event handlers, 97–99
understanding difference between interface and virtual function, 82–86

P

Parallel processing
avoiding calling unknown code in locked sections, 225–228
`BackgroundWorker` for cross-thread communication, 201–205
`CompareExchange()`, 221
constructing parallel algorithms with exceptions in mind, 189–195
cross-thread calls in XAML environment, 205–214
`lock()` for thread synchronization, 214–220
overview of, 177
PLINQ implementation of parallel algorithms, 177–189
specifying smallest possible scope for lock handles, 221–225
`System.Threading.Interlocked`, 220–221
thread pools in, 195–201

`ParallelEnumerable` class, 177,188–189

Parameters
limiting array parameters to `params` arrays, 129–134
methods with multiple, 117
use in minimizing method overloads, 65–69

`params` array, limiting array parameters to, 129–134

Partial classes, 120–125

Partial methods, 120–125

Partitioning, in PLINQ, 179–180

Passing by value, value types vs. reference types, 19

Pattern matching, testing objects, 81

Pipelining, PLINQ algorithms for parallelization, 180–181

Placeholders, in interfaces, 73

PLINQ
`AsParallel()`, 178
comparing to LINQ to Objects, 182–188
controlling execution of parallel queries, 189
exception handling, 194–195
implementation of parallel algorithms, 177–189
overview of, 177
parallelization algorithms, 180–182
partitioning algorithms, 179–180

Polymorphism, value types vs. reference types, 19

Primitives, `lock()` for thread synchronization, 214–221

Programming
asynchronous. ***See*** Asynchronous programming
dynamic. ***See*** Dynamic programming

Properties
avoiding returning references to internal class objects, 93–96
data binding support for, 2
enhancements to, 1
ensuring behave like data, 28–34
implicit properties preferred for mutable data, 8–12
implicit syntax, 4
multithreaded support implemented via, 3
virtual properties, 4

Public types
avoiding returning references to internal class objects, 93–96
limiting visibility of types, 69–73

Q

Queries
enumeration loop, 183–187
lazy evaluation of, 182
LINQ to Objects. ***See*** LINQ to SQL
LINQ to SQL. ***See*** LINQ to Objects
PLINQ. ***See*** PLINQ

`QueueUserWorkItem`, creating background threads, 201–202, 205

R

raiseProgress() method, avoiding calling unknown code in locked sections, 226–228

Range partitioning, in PLINQ, 179

Read-only properties, avoiding returning references to internal class objects, 93–96

Reference types
- array class as, 17
- avoiding ICloneable interface, 126–127
- avoiding returning references to internal class objects, 93–96
- changing data type changes its behavior, 23
- considering size of, 23–24
- converting value types to, 240
- defining behavior of, 21
- distinguishing between value types and reference types, 18–24

ReferenceEquals(), specifying equality, 46–48

Remove accessor, 88, 108–109

Resource consumption, problems from composing synchronous code on top of asynchronous methods, 150

Return types, cache generalized async return types, 173–176

Roslyn repository, GitHub, 269–272

RunAsync(), 190–191

Runtime
- costs of dynamic typing and building expressions at, 237
- dynamic types as System.Object with runtime binding, 230
- events wired at, 99
- understanding how event increase runtime coupling among objects, 104–107

S

set accessor
- implicit properties supporting, 8–10
- as inexpensive operation, 29–33
- providing synchronized access to data, 3–4
- source objects, 256
- specifying accessibility modifiers, 4–5

Shallow copies, ICloneable interface, 125

Single-threaded apartment (STA) model, COM, 205–206

Source code, constructs for minimizing and making clear. *See also* Code, 134

Specs, participating in C# community, 269–270

Square roots, computing, 197–201

STA (single-threaded apartment) model, COM, 205–206

Static types
- C# as statically typed language, 265
- comparing with dynamic, 229, 260

Stop and go, PLINQ algorithms for parallelization, 181

Strings
- archaic syntax, 268
- IEnumerator <string>, 183
- storing with MyType, 240–241

Striped partitioning, in PLINQ, 180

struct keyword, defining value types and reference types, 19

Structs
- avoiding ICloneable interface, 126
- creating user-defined types, 34
- preventing unboxing penalty, 81
- when to use, 18–19

Synchronization primitives, lock() for thread synchronization, 214–221

SynchronizationContext class
implementing async methods, 141–142
throwing exceptions, 144

Synchronous methods
avoiding composing synchronous and asynchronous methods, 149–154
compared with async methods, 139
exception to rule for avoiding composing synchronous and asynchronous methods, 152

System.Collection class, protecting internal data structures using wrappers, 96

System.Dynamic.DynamicObject class, for data-driven dynamic types, 242–252

System.Linq.Enumerable.Cast<T>, 238–240

System.Object class
conversion operators used for substitutability between, 61
dynamic typing, 230
hashes, 55–57
minimizing dynamic objects in public APIs, 259–265
synchronization handle, 223–225

System.Threading class, Interlocked method, 220–221

System.ValueType class, hashes, 55–56

System.Window.Forms.Control, 88

T

Task objects, composing objects using, 160–166

Task Parallel Library, managing threads, 196

TaskCompletion Source class, 163–164

Tasks
composing using task objects, 160–166
implementing task cancellation protocol, 166–173
LINQ issues due to errors in background tasks, 194–195
task-based asynchronous programming. ***See*** Asynchronous programming
thread pools managing end-of-task cycle, 196–197

this keyword, declaring indexers, 6

Thread pools
Hero of Alexandria algorithm example, 197–201
managing thread resources, 196–197
overview of, 195
using instead of creating threads, 195–201

Threads
avoiding calling unknown code in locked sections, 225–228
avoiding thread allocations and context switches, 154–156
BackgroundWorker for cross-thread communication, 201–205
CompareExchange(), 221
cross-thread calls in XAML environment, 205–214
exception handling in PLINQ, 194–195
immutable types and thread safety, 14
lock() for thread synchronization, 214–220
specifying smallest possible scope for lock handles, 221–225
System.Threading.Interlocked, 220–221
using thread pools instead of creating, 195–201

Throw expressions
testing objects, 81
throwing exceptions, 123

`try/catch` blocks
exception handling, 143, 148
problems from composing synchronous code on top of asynchronous methods, 150–151

`TryGetIndex`, retrieving XElement with indexer, 248

`TryGetMember`, creating dynamic types, 244

`TrySetMember`, creating dynamic types, 244

Tuples
anonymous types compared with, 39
defining shape of tuple as return type, 39–41
limiting type scope using, 38–39

UI (user interface), data binding applies to classes displayed in, 2

Updates, improving code, 268

User-defined type, 34–35

Validation
immutable types and, 14
implicit properties and, 11

Value types
avoiding `ICloneable` interface, 126
changing type changes its behavior, 23
converting to reference types, 240
derived types not supported, 16
distinguishing between value types and reference types, 18–24
ensuring 0 is valid state for, 24–28
immutable types preferred for, 12–18
protecting internal data structures, 94
questions to ask in determining use of, 24
size of, 23–24
storing data, 21
`ValueTask<T>` type, 173–176

`ValueTask<T>` type, 173–176

Virtual events, 107, 111–112

Virtual methods
interface methods compared with, 82–86
overriding, 100
sources of unknown code, 228

Virtual properties, 4

Visibility, limiting type visibility, 69–73

`void` methods, not using async void methods, 143–149

Web applications, avoiding marshalling context unnecessarily, 156–160

Web Forms, 2

Web services
`BackgroundWorker` class for, 205
example of task cancellation, 166–173
workflow, 253

Windows controls, COM STA model, 205–206

Windows Forms
avoiding calling unknown code in locked sections, 227
`BackgroundWorker` class for, 205
COM STA model, 205–206
data binding classes supporting properties no public data fields, 2
extensions, 208–209

WPF (Windows Presentation Foundation)
data binding classes supporting properties no public data fields, 2
`Dispatcher` class, 210

overrides compared with event handlers, 97–98
thread controls, 209–210

Wrappers
protecting internal data structures, 94, *96*
splitting iterator into two methods, 136

X

XAML
cross-thread calls in XAML environment, 207–214
overrides compared with event handlers, 97–98

XElement, retrieving with indexer, 248

XML, LINQ to XML, 244–248